The LaTeX3 Sources

The LaTeX3 Project[*]

Released 2017/09/18

Abstract

This is the reference documentation for the expl3 programming environment. The expl3 modules set up an experimental naming scheme for LaTeX commands, which allow the LaTeX programmer to systematically name functions and variables, and specify the argument types of functions.

The TeX and ε-TeX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the expl3 modules define an independent low-level LaTeX3 programming language.

At present, the expl3 modules are designed to be loaded on top of LaTeX 2ε. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2ε packages *now* while a stand-alone LaTeX3 is developed.

While expl3 is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of expl3.

New modules will be added to the distributed version of expl3 as they reach maturity.

[*] E-mail: latex-team@latex-project.org

Contents

Part I

Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the expl3 language. A general guide to the LaTeX3 programming language is found in expl3.pdf.

1 Naming functions and variables

LaTeX3 does not use `@` as a "letter" for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* ("arg-spec"). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all expl3 function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

`D` The `D` specifier means *do not use*. All of the TeX primitives are initially `\let` to a `D` name, and some are then given a second name. Only the kernel team should use anything with a `D` specifier!

`N` and `n` These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.

`c` This means *csname*, and indicates that the argument will be turned into a csname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.

`V` and `v` These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying TeX structure containing the data. A `V` argument will be a single token (similar to `N`), for example `\foo:V \MyVariable`; on the other hand, using `v` a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

`o` This means *expansion once*. In general, the `V` and `v` specifiers are favoured over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.

`x` The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The TeX `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are in general *not* expandable, unless specifically noted.

f The f specifier stands for *full expansion*, and in contrast to x stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_mya_tl { A }
\tl_set:Nn \l_myb_tl { B }
\tl_set:Nf \l_mya_tl { \l_mya_tl \l_myb_tl }
```

will leave `\l_mya_tl` with the content `A\l_myb_tl`, as A cannot be expanded and so terminates expansion before `\l_myb_tl` is considered.

T and F For logic tests, there are the branch specifiers T (*true*) and F (*false*). Both specifiers treat the input in the same way as n (no change), but make the logic much easier to see.

p The letter p indicates TeX *parameters*. Normally this will be used for delimited functions as expl3 provides better methods for creating simple sequential arguments.

w Finally, there is the w specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

c Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module[1] name and then a descriptive part. Variables end with a short identifier to show the variable type:

bool Either true or false.

box Box register.

clist Comma separated list.

coffin a "box with handles" — a higher-level data type for carrying out box alignment operations.

dim "Rigid" lengths.

fp floating-point values;

[1] The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the int module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

int Integer-valued count register.

prop Property list.

seq "Sequence": a data-type used to implement lists (with access at both ends) and stacks.

skip "Rubber" lengths.

stream An input or output stream (for reading from or writing to, respectively).

tl Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

A word of warning. In this document, and others referring to the expl3 programming modules, we often refer to "variables" and "functions" as if they were actual constructs from a real programming language. In truth, TeX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a "function" with no arguments and a "token list variable" are in truth one and the same. On the other hand, some "variables" are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the expl3 code are designed to clearly separate the ideas of "macros that contain data" and "macros that contain code", and a consistent wrapper is applied to all forms of "data" whether they be macros or actually registers. This means that sometimes we will use phrases like "the function returns a value", when actually we just mean "the macro expands to something". Similarly, the term "execute" might be used in place of "expand" or it might refer to the more specific case of "processing in TeX's stomach" (if you are familiar with the TeXbook parlance).

If in doubt, please ask; chances are we've been hasty in writing certain definitions and need to be told to tighten up our terminology.

2 Documentation conventions

This document is typeset with the experimental l3doc class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a "user" name, this might read:

```
\ExplSyntaxOn
\ExplSyntaxOff
```

```
\ExplSyntaxOn ...  \ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use _ and : in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

`\seq_new:N`
`\seq_new:c`

`\seq_new:N` ⟨*sequence*⟩

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, ⟨*sequence*⟩ indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type argument (in plain TeX terms, inside an `\edef`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

`\cs_to_str:N` ⋆

`\cs_to_str:N` ⟨*cs*⟩

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a ⟨*cs*⟩, shorthand for a ⟨*control sequence*⟩.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

`\seq_map_function:NN` ☆

`\seq_map_function:NN` ⟨*seq*⟩ ⟨*function*⟩

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different "true"/"false" branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

`\sys_if_engine_xetex:`*TF* ⋆

`\sys_if_engine_xetex:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

The underlining and italic of `TF` indicates that `\sys_if_engine_xetex:T`, `\sys_if_engine_xetex:F` and `\sys_if_engine_xetex:TF` are all available. Usually, the illustration will use the `TF` variant, and so both ⟨*true code*⟩ and ⟨*false code*⟩ will be shown. The two variant forms `T` and `F` take only ⟨*true code*⟩ and ⟨*false code*⟩, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the expl3 modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

`\l_tmpa_tl`

A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in LaTeX 2ε or plain TeX. In these cases, the text will include an extra "**TeXhackers note**" section:

`\token_to_str:N` ⋆

`\token_to_str:N` ⟨*token*⟩

The normal description text.

TeXhackers note: Detail for the experienced TeX or LaTeX 2ε programmer. In this case, it would point out that this function is the TeX primitive `\string`.

Changes to behaviour When new functions are added to expl3, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of expl3 after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

3 Formal language conventions which apply generally

As this is a formal reference guide for LaTeX3 programming, the descriptions of functions are intended to be reasonably "complete". However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a `TF` argument specification, the test if evaluated to give a logically `TRUE` or `FALSE` result. Depending on this result, either the ⟨*true code*⟩ or the ⟨*false code*⟩ will be left in the input stream. In the case where the test is expandable, and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

4 TeX concepts not supported by LaTeX3

The TeX concept of an "`\outer`" macro is *not supported* at all by LaTeX3. As such, the functions provided here may break when used on top of LaTeX 2$_\varepsilon$ if `\outer` tokens are used in the arguments.

Part II
The l3bootstrap package Bootstrap code

1 Using the LaTeX3 modules

The modules documented in `source3` are designed to be used on top of LaTeX 2ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the LaTeX3 format, but work in this area is incomplete and not included in this documentation at present.

As the modules use a coding syntax different from standard LaTeX 2ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\ExplSyntaxOn` ⟨*code*⟩ `\ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code régime in which spaces are ignored and in which the colon (`:`) and underscore (`_`) are treated as "letters", thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code régime.

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`

Updated: 2017-03-19

```
\RequirePackage{expl3}
\ProvidesExplPackage {⟨package⟩} {⟨date⟩} {⟨version⟩} {⟨description⟩}
```

These functions act broadly in the same way as the corresponding LaTeX 2ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as LaTeX 2ε provides in turning on `\makeatletter` within package and class code.) The ⟨*date*⟩ should be given in the format ⟨*year*⟩/⟨*month*⟩/⟨*day*⟩. If the ⟨*version*⟩ is given then it will be prefixed with `v` in the package identifier line.

`\GetIdInfo`

Updated: 2012-06-04

```
\RequirePackage{l3bootstrap}
\GetIdInfo $Id: ⟨SVN info field⟩ $ {⟨description⟩}
```

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual LaTeX 2ε category codes and the LaTeX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

1.1 Internal functions and variables

`\l__kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn`/`\ExplSyntaxOff`.

Part III

The **l3names** package Namespace for primitives

1 Setting up the LaTeX3 programming language

This module is at the core of the LaTeX3 programming language. It performs the following tasks:

- defines new names for all TeX primitives;
- switches to the category code régime for programming;
- provides support settings for building the code as a TeX format.

This module is entirely dedicated to primitives, which should not be used directly within LaTeX3 code (outside of "kernel-level" code). As such, the primitives are not documented here: *The TeXbook*, *TeX by Topic* and the manuals for pdfTeX, XeTeX and LuaTeX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by TeX itself;

`\etex_...` Introduced by the ε-TeX extensions;

`\pdftex_...` Introduced by pdfTeX;

`\xetex_...` Introduced by XeTeX;

`\luatex_...` Introduced by LuaTeX;

`\utex_...` Introduced by XeTeX and LuaTeX;

`\ptex_...` Introduced by pTeX;

`\uptex_...` Introduced by upTeX.

Part IV

The l3basics package
Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

1 No operation functions

`\prg_do_nothing:` ⋆

`\prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`

`\scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

2 Grouping material

`\group_begin:`
`\group_end:`

`\group_begin:`
`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`\group_insert_after:N`

`\group_insert_after:N` ⟨*token*⟩

Adds ⟨*token*⟩ to the list of ⟨*tokens*⟩ to be inserted when the current group level ends. The list of ⟨*tokens*⟩ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one ⟨*token*⟩ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

3 Control sequences and functions

As TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text ("code") in which each parameter in the code (`#1`, `#2`, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, ⟨*code*⟩ is therefore used as a shorthand for "replacement text".

Functions which are not "protected" are fully expanded inside an `x` expansion. In contrast, "protected" functions are not expanded within `x` expansions.

3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

`new` Create a new function with the `new` scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

`set` Create a new function with the `set` scope, such as `\cs_set:Npn`. The definition is restricted to the current TeX group and does not result in an error if the function is already defined.

`gset` Create a new function with the `gset` scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

`nopar` Create a new function with the `nopar` restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

`protected` Create a new function with the `protected` restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an `x`-type expansion.

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

`N` and `n` No manipulation.

`T` and `F` Functionally equivalent to `n` (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 1).

`p` and `w` These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

`\cs_new:Npn`
`\cs_new:cpn`
`\cs_new:Npx`
`\cs_new:cpx`

`\cs_new:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_new_nopar:Npn`
`\cs_new_nopar:cpn`
`\cs_new_nopar:Npx`
`\cs_new_nopar:cpx`

`\cs_new_nopar:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_new_protected:Npn`
`\cs_new_protected:cpn`
`\cs_new_protected:Npx`
`\cs_new_protected:cpx`

`\cs_new_protected:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_new_protected_nopar:Npn`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:Npx`
`\cs_new_protected_nopar:cpx`

`\cs_new_protected_nopar:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_set:Npn`
`\cs_set:cpn`
`\cs_set:Npx`
`\cs_set:cpx`

`\cs_set:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_nopar:Npn`
`\cs_set_nopar:cpn`
`\cs_set_nopar:Npx`
`\cs_set_nopar:cpx`

`\cs_set_nopar:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_protected:Npn`
`\cs_set_protected:cpn`
`\cs_set_protected:Npx`
`\cs_set_protected:cpx`

`\cs_set_protected:Npn ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level. The ⟨*function*⟩ will not expand within an `x`-type argument.

`\cs_set_protected_nopar:Npn`
`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:Npx`
`\cs_set_protected_nopar:cpx`

`\cs_set_protected_nopar:Npn` ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level. The ⟨*function*⟩ will not expand within an `x`-type argument.

`\cs_gset:Npn`
`\cs_gset:cpn`
`\cs_gset:Npx`
`\cs_gset:cpx`

`\cs_gset:Npn` ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TeX group level: the assignment is global.

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:Npx`
`\cs_gset_nopar:cpx`

`\cs_gset_nopar:Npn` ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TeX group level: the assignment is global.

`\cs_gset_protected:Npn`
`\cs_gset_protected:cpn`
`\cs_gset_protected:Npx`
`\cs_gset_protected:cpx`

`\cs_gset_protected:Npn` ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TeX group level: the assignment is global. The ⟨*function*⟩ will not expand within an `x`-type argument.

`\cs_gset_protected_nopar:Npn`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:Npx`
`\cs_gset_protected_nopar:cpx`

`\cs_gset_protected_nopar:Npn` ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TeX group level: the assignment is global. The ⟨*function*⟩ will not expand within an `x`-type argument.

3.3 Defining new functions using the signature

`\cs_new:Nn`
`\cs_new:(cn|Nx|cx)`

`\cs_new:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error results if the ⟨*function*⟩ is already defined.

```
\cs_new_nopar:Nn
\cs_new_nopar:(cn|Nx|cx)
```

```
\cs_new_nopar:Nn ⟨function⟩ {⟨code⟩}
```

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The definition is global and an error results if the ⟨*function*⟩ is already defined.

```
\cs_new_protected:Nn
\cs_new_protected:(cn|Nx|cx)
```

```
\cs_new_protected:Nn ⟨function⟩ {⟨code⟩}
```

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

```
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:(cn|Nx|cx)
```

```
\cs_new_protected_nopar:Nn ⟨function⟩ {⟨code⟩}
```

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

```
\cs_set:Nn
\cs_set:(cn|Nx|cx)
```

```
\cs_set:Nn ⟨function⟩ {⟨code⟩}
```

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

```
\cs_set_nopar:Nn
\cs_set_nopar:(cn|Nx|cx)
```

```
\cs_set_nopar:Nn ⟨function⟩ {⟨code⟩}
```

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

```
\cs_set_protected:Nn
\cs_set_protected:(cn|Nx|cx)
```

```
\cs_set_protected:Nn ⟨function⟩ {⟨code⟩}
```

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_protected_nopar:Nn`
`\cs_set_protected_nopar:(cn|Nx|cx)`

`\cs_set_protected_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TEX group level.

`\cs_gset:Nn`
`\cs_gset:(cn|Nx|cx)`

`\cs_gset:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_nopar:Nn`
`\cs_gset_nopar:(cn|Nx|cx)`

`\cs_gset_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_protected:Nn`
`\cs_gset_protected:(cn|Nx|cx)`

`\cs_gset_protected:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_protected_nopar:Nn`
`\cs_gset_protected_nopar:(cn|Nx|cx)`

`\cs_gset_protected_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:(cNnn|Ncnn)`

Updated: 2012-01-14

`\cs_generate_from_arg_count:NNnn` ⟨*function*⟩ ⟨*creator*⟩ ⟨*number*⟩ ⟨*code*⟩

Uses the ⟨*creator*⟩ function (which should have signature `Npn`, for example `\cs_new:Npn`) to define a ⟨*function*⟩ which takes ⟨*number*⟩ arguments and has ⟨*code*⟩ as replacement text. The ⟨*number*⟩ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text "cs" is used as an abbreviation for "control sequence".

`\cs_new_eq:NN`
`\cs_new_eq:(Nc|cN|cc)`

`\cs_new_eq:NN` ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
`\cs_new_eq:NN` ⟨*cs*$_1$⟩ ⟨*token*⟩

Globally creates ⟨*control sequence*$_1$⟩ and sets it to have the same meaning as ⟨*control sequence*$_2$⟩ or ⟨*token*⟩. The second control sequence may subsequently be altered without affecting the copy.

`\cs_set_eq:NN`
`\cs_set_eq:(Nc|cN|cc)`

`\cs_set_eq:NN` ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
`\cs_set_eq:NN` ⟨*cs*$_1$⟩ ⟨*token*⟩

Sets ⟨*control sequence*$_1$⟩ to have the same meaning as ⟨*control sequence*$_2$⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence*$_1$⟩ is restricted to the current TEX group level.

`\cs_gset_eq:NN`
`\cs_gset_eq:(Nc|cN|cc)`

`\cs_gset_eq:NN` ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
`\cs_gset_eq:NN` ⟨*cs*$_1$⟩ ⟨*token*⟩

Globally sets ⟨*control sequence*$_1$⟩ to have the same meaning as ⟨*control sequence*$_2$⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence*$_1$⟩ is *not* restricted to the current TEX group level: the assignment is global.

3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

`\cs_undefine:N`
`\cs_undefine:c`
Updated: 2011-09-15

`\cs_undefine:N` ⟨*control sequence*⟩

Sets ⟨*control sequence*⟩ to be globally undefined.

3.6 Showing control sequences

`\cs_meaning:N` ⋆
`\cs_meaning:c` ⋆
Updated: 2011-12-22

`\cs_meaning:N` ⟨*control sequence*⟩

This function expands to the *meaning* of the ⟨*control sequence*⟩ control sequence. For a macro, this includes the ⟨*replacement text*⟩.

TEXhackers note: This is TEX's `\meaning` primitive. The `c` variant correctly reports undefined arguments.

\cs_show:N
\cs_show:c

Updated: 2017-02-14

`\cs_show:N` ⟨*control sequence*⟩

Displays the definition of the ⟨*control sequence*⟩ on the terminal.

TEXhackers note: This is similar to the TEX primitive `\show`, wrapped to a fixed number of characters per line.

\cs_log:N
\cs_log:c

New: 2014-08-22
Updated: 2017-02-14

`\cs_log:N` ⟨*control sequence*⟩

Writes the definition of the ⟨*control sequence*⟩ in the log file. See also `\cs_show:N` which displays the result in the terminal.

3.7 Converting to and from control sequences

\use:c ⋆

`\use:c {`⟨*control sequence name*⟩`}`

Converts the given ⟨*control sequence name*⟩ into a single control sequence token. This process requires two expansions. The content for ⟨*control sequence name*⟩ may be literal material or from other expandable functions. The ⟨*control sequence name*⟩ must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

As an example of the `\use:c` function, both

```
\use:c { a b c }
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

\cs_if_exist_use:N ⋆
\cs_if_exist_use:c ⋆
\cs_if_exist_use:N*TF* ⋆
\cs_if_exist_use:c*TF* ⋆

New: 2012-11-10

`\cs_if_exist_use:N` ⟨*control sequence*⟩
`\cs_if_exist_use:NTF` ⟨*control sequence*⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type), and if it is inserts the ⟨*control sequence*⟩ into the input stream followed by the ⟨*true code*⟩. Otherwise the ⟨*false code*⟩ is used.

\cs:w ⋆
\cs_end: ⋆

`\cs:w` ⟨*control sequence name*⟩ `\cs_end:`

Converts the given ⟨*control sequence name*⟩ into a single control sequence token. This process requires one expansion. The content for ⟨*control sequence name*⟩ may be literal material or from other expandable functions. The ⟨*control sequence name*⟩ must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TEXhackers note: These are the TEX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

```
\cs:w a b c \cs_end:
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

```
\cs_to_str:N ⋆    \cs_to_str:N ⟨control sequence⟩
```

Converts the given ⟨*control sequence*⟩ into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an `x`-type expansion, or two `o`-type expansions are required to convert the ⟨*control sequence*⟩ to a sequence of characters in the input stream. In most cases, an `f`-expansion is correct as well, but this loses a space at the start of the result.

4 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

```
\use:n    ⋆    \use:n    {⟨group1⟩}
\use:nn   ⋆    \use:nn   {⟨group1⟩} {⟨group2⟩}
\use:nnn  ⋆    \use:nnn  {⟨group1⟩} {⟨group2⟩} {⟨group3⟩}
\use:nnnn ⋆    \use:nnnn {⟨group1⟩} {⟨group2⟩} {⟨group3⟩} {⟨group4⟩}
```

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

`\use_i:nn` ★
`\use_ii:nn` ★

`\use_i:nn` {⟨arg_1⟩} {⟨arg_2⟩}

These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i:nnn` ★
`\use_ii:nnn` ★
`\use_iii:nnn` ★

`\use_i:nnn` {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩}

These functions absorb three arguments from the input stream. The function `\use_i:nnn` discards the second and third arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnn` and `\use_iii:nnn` work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i:nnnn` ★
`\use_ii:nnnn` ★
`\use_iii:nnnn` ★
`\use_iv:nnnn` ★

`\use_i:nnnn` {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩}

These functions absorb four arguments from the input stream. The function `\use_i:nnnn` discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnnn`, `\use_iii:nnnn` and `\use_iv:nnnn` work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i_ii:nnn` ★

`\use_i_ii:nnn` {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩}

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

`\use_none:n` ★
`\use_none:nn` ★
`\use_none:nnn` ★
`\use_none:nnnn` ★
`\use_none:nnnnn` ★
`\use_none:nnnnnn` ★
`\use_none:nnnnnnn` ★
`\use_none:nnnnnnnn` ★
`\use_none:nnnnnnnnn` ★

`\use_none:n` {⟨$group_1$⟩}

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (*i.e.* an `N` argument).

`\use:x`

Updated: 2011-12-31

`\use:x {⟨expandable tokens⟩}`

Fully expands the ⟨*expandable tokens*⟩ and inserts the result into the input stream at the current location. Any hash characters (`#`) in the argument must be doubled.

4.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

`\use_none_delimit_by_q_nil:w` ⋆
`\use_none_delimit_by_q_stop:w` ⋆
`\use_none_delimit_by_q_recursion_stop:w` ⋆

```
\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil
\use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop
\use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩
\q_recursion_stop
```

Absorb the ⟨*balanced text*⟩ form the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

`\use_i_delimit_by_q_nil:nw` ⋆
`\use_i_delimit_by_q_stop:nw` ⋆
`\use_i_delimit_by_q_recursion_stop:nw` ⋆

```
\use_i_delimit_by_q_nil:nw {⟨inserted tokens⟩} ⟨balanced text⟩
\q_nil
\use_i_delimit_by_q_stop:nw {⟨inserted tokens⟩} ⟨balanced
text⟩ \q_stop
\use_i_delimit_by_q_recursion_stop:nw {⟨inserted tokens⟩}
⟨balanced text⟩ \q_recursion_stop
```

Absorb the ⟨*balanced text*⟩ form the input stream delimited by the marker given in the function name, leaving ⟨*inserted tokens*⟩ in the input stream for further processing.

5 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the ⟨*true code*⟩ or the ⟨*false code*⟩. These arguments are denoted with `T` and `F`, respectively. An example would be

```
\cs_if_free:cTF {abc} {⟨true code⟩} {⟨false code⟩}
```

a function that turns the first argument into a control sequence (since it's marked as `c`) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as "conditionals"; whenever a `TF` function is defined it is usually accompanied by `T` and `F` functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with ⟨*true code*⟩ and/or ⟨*false code*⟩ are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a "predicate" for the same test as described below.

Predicates "Predicates" are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return "true" if its argument (a single token denoted by `N`) is still free for definition. It would be used in constructions like

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {⟨true code⟩} {⟨false code⟩}
```

For each predicate defined, a "branching conditional" also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain TeX and LaTeX 2$_\varepsilon$. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

`\c_true_bool`
`\c_false_bool`

Constants that represent `true` and `false`, respectively. Used to implement predicates.

5.1 Tests on control sequences

`\cs_if_eq_p:NN` ★
`\cs_if_eq:NN`*TF* ★

```
\cs_if_eq_p:NN {⟨cs_1⟩} {⟨cs_2⟩}
\cs_if_eq:NNTF {⟨cs_1⟩} {⟨cs_2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the definition of two ⟨*control sequences*⟩ and is logically `true` if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

`\cs_if_exist_p:N` ★
`\cs_if_exist_p:c` ★
`\cs_if_exist:N`*TF* ★
`\cs_if_exist:c`*TF* ★

```
\cs_if_exist_p:N ⟨control sequence⟩
\cs_if_exist:NTF ⟨control sequence⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type). Any valid definition of ⟨*control sequence*⟩ evaluates as `true`.

`\cs_if_free_p:N` ★
`\cs_if_free_p:c` ★
`\cs_if_free:N`*TF* ★
`\cs_if_free:c`*TF* ★

```
\cs_if_free_p:N ⟨control sequence⟩
\cs_if_free:NTF ⟨control sequence⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*control sequence*⟩ is currently free to be defined. This test is `false` if the ⟨*control sequence*⟩ currently exists (as defined by `\cs_if_exist:N`).

5.2 Primitive conditionals

The ε-TEX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a :w part but higher level functions are often available. See for instance \int_compare_p:nNn which is a wrapper for \if_int_compare:w.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with \if_.

\if_true: ★
\if_false: ★
\else: ★
\fi: ★
\reverse_if:N ★

\if_true: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_false: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\reverse_if:N ⟨*primitive conditional*⟩

\if_true: always executes ⟨*true code*⟩, while \if_false: always executes ⟨*false code*⟩. \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in l3int and used in case switches.

> **TEXhackers note:** These are equivalent to their corresponding TEX primitive conditionals; \reverse_if:N is ε-TEX's \unless.

\if_meaning:w ★

\if_meaning:w ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

\if_meaning:w executes ⟨*true code*⟩ when ⟨*arg*$_1$⟩ and ⟨*arg*$_2$⟩ are the same, otherwise it executes ⟨*false code*⟩. ⟨*arg*$_1$⟩ and ⟨*arg*$_2$⟩ could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

> **TEXhackers note:** This is TEX's \ifx.

\if:w ★
\if_charcode:w ★
\if_catcode:w ★

\if:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_catcode:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

These conditionals expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with \exp_not:N. \if_catcode:w tests if the category codes of the two tokens are the same whereas \if:w tests if the character codes are identical. \if_charcode:w is an alternative name for \if:w.

\if_cs_exist:N ★
\if_cs_exist:w ★

\if_cs_exist:N ⟨*cs*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_cs_exist:w ⟨*tokens*⟩ \cs_end: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

Check if ⟨*cs*⟩ appears in the hash table or if the control sequence that can be formed from ⟨*tokens*⟩ appears in the hash table. The latter function does not turn the control sequence in question into \scan_stop:! This can be useful when dealing with control sequences which cannot be entered as a single token.

\if_mode_horizontal: ★
\if_mode_vertical: ★
\if_mode_math: ★
\if_mode_inner: ★

\if_mode_horizontal: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

Execute ⟨*true code*⟩ if currently in horizontal mode, otherwise execute ⟨*false code*⟩. Similar for the other functions.

6 Internal kernel functions

`\__chk_if_free_cs:N`
`\__chk_if_free_cs:c`

`\__chk_if_free_cs:N` ⟨*cs*⟩

This function checks that ⟨*cs*⟩ is free according to the criteria for `\cs_if_free_p:N`, and if not raises a kernel-level error.

`\__cs_count_signature:N` ★
`\__cs_count_signature:c` ★

`\__cs_count_signature:N` ⟨*function*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). The ⟨*number*⟩ of tokens in the ⟨*signature*⟩ is then left in the input stream. If there was no ⟨*signature*⟩ then the result is the marker value -1.

`\__cs_split_function:NN` ★

`\__cs_split_function:NN` ⟨*function*⟩ ⟨*processor*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). This information is then placed in the input stream after the ⟨*processor*⟩ function in three parts: the ⟨*name*⟩, the ⟨*signature*⟩ and a logic token indicating if a colon was found (to differentiate variables from function names). The ⟨*name*⟩ does not include the escape character, and both the ⟨*name*⟩ and ⟨*signature*⟩ are made up of tokens with category code 12 (other). The ⟨*processor*⟩ should be a function with argument specification `:nnN` (plus any trailing arguments needed).

`\__cs_get_function_name:N` ★ `\__cs_get_function_name:N` ⟨*function*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). The ⟨*name*⟩ is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

`\__cs_get_function_signature:N` ★ `\__cs_get_function_signature:N` ⟨*function*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). The ⟨*signature*⟩ is then left in the input stream made up of tokens with category code 12 (other).

`\__cs_tmp:w`

Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\__debug:TF`

`\__debug:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Runs the ⟨*true code*⟩ if debugging is enabled, namely only in LaTeX 2_ε package mode with one of the options `check-declarations`, `enable-debug`, or `log-functions`. Otherwise runs the ⟨*false code*⟩. The `T` and `F` variants are not provided for this low-level conditional.

`\__debug_chk_cs_exist:N`
`\__debug_chk_cs_exist:c`

`\__debug_chk_cs_exist:N` ⟨*cs*⟩

This function is only created if debugging is enabled. It checks that ⟨*cs*⟩ exists according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`\__debug_chk_expr:nNnN`

`\__debug_chk_expr:nNnN {⟨expr⟩} ⟨eval⟩ {⟨convert⟩} ⟨caller⟩`

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nnnn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D` ⟨*eval*⟩ ⟨*expr*⟩ `\tex_relax:D` after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the ⟨*caller*⟩. For instance ⟨*eval*⟩ can be `\__int_eval:w` and ⟨*caller*⟩ can be `\int_eval:n` or `\int_set:Nn`. The argument ⟨*convert*⟩ is empty except for mu expressions where it is `\etex_mutoglue:D`, used for internal purposes.

`\__debug_chk_var_exist:N`

`\__debug_chk_var_exist:N ⟨var⟩`

This function is only created if debugging is enabled. It checks that ⟨*var*⟩ is defined according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`\__debug_log:x`

`\__debug_log:x {⟨message text⟩}`

If the `log-functions` option is active, this function writes the ⟨*message text*⟩ to the log file using `\iow_log:x`. Otherwise, the ⟨*message text*⟩ is ignored using `\use_none:n`. This function is only created if debugging is enabled.

`\__debug_suspend_log:`
`\__debug_resume_log:`

`\__debug_suspend_log: ... \__debug_log:x ... \__debug_resume_log:`

Any `\__debug_log:x` command between `\__debug_suspend_log:` and `\__debug_resume_log:` is suppressed. These two commands can be nested. These functions are only created if debugging is enabled.

`\__debug_patch:nnNNpn`

`\__debug_patch:nnNNpn {⟨before⟩} {⟨after⟩}`
`⟨definition⟩ ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

If debugging is not enabled, this function ignores the ⟨*before*⟩ and ⟨*after*⟩ code and performs the ⟨*definition*⟩ with no patching. Otherwise it replaces ⟨*code*⟩ by ⟨*before*⟩ ⟨*code*⟩ ⟨*after*⟩ (which can involve `#1` and so on) in the ⟨*definition*⟩ that follows. The ⟨*definition*⟩ must start with `\cs_new:Npn` or `\cs_set:Npn` or `\cs_gset:Npn` or their `_protected` counterparts. Other cases can be added as needed.

`\__debug_patch_conditional:nNNpnn`

`\__debug_patch_conditional:nNNpn {⟨before⟩}`
`⟨definition⟩ ⟨conditional⟩ ⟨parameters⟩ {⟨type⟩} {⟨code⟩}`

Similar to `\__debug_patch:nnNNpn` for conditionals, namely ⟨*definition*⟩ must be `\prg_new_conditional:Npnn` or its `_protected` counterpart. There is no ⟨*after*⟩ code because that would interfere with the action of the conditional.

`\__debug_patch_args:nNNpn`
`\__debug_patch_conditional_args:nNNpnn`

`\__debug_patch_args:nNNpn {⟨arguments⟩}`
`⟨definition⟩ ⟨function⟩ ⟨parameters⟩ {⟨code⟩}`

Like `\__debug_patch:nnNNpn`, this tweaks the following definition, but from the "inside out" (and if debugging is not enabled, the ⟨*arguments*⟩ are ignored). It replaces `#1`, `#2` and so on in the ⟨*code*⟩ of the definition as indicated by the ⟨*arguments*⟩. More precisely, a temporary function is defined using the ⟨*definition*⟩ with the ⟨*parameters*⟩ and ⟨*code*⟩, then the result of expanding that function once in front of the ⟨*arguments*⟩ is used instead of the ⟨*code*⟩ when defining the actual function. For instance,

```
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \int_eval:n #1
{ \__int_value:w \__int_eval:w #1 \__int_eval_end: }
```

would replace `#1` by `(#1)` in the definition of `\int_eval:n` when debugging is enabled. This fails if the ⟨*code*⟩ contains `##`. The `\__debug_patch_conditional_args:nNNpnn` function is for use before `\prg_new_conditional:Npnn` or its `_protected` counterpart.

`\__kernel_register_show:N`
`\__kernel_register_show:c`

`\__kernel_register_show:N ⟨register⟩`

Used to show the contents of a TeX register at the terminal, formatted such that internal parts of the mechanism are not visible.

`\__kernel_register_log:N`
`\__kernel_register_log:c`

Updated: 2015-08-03

`\__kernel_register_log:N ⟨register⟩`

Used to write the contents of a TeX register to the log file in a form similar to `\__kernel_register_show:N`.

`\__prg_case_end:nw` ⋆

`\__prg_case_end:nw {⟨code⟩} ⟨tokens⟩ \q_mark {⟨true code⟩} \q_mark {⟨false code⟩}`
`\q_stop`

Used to terminate case statements (`\int_case:nnTF`, *etc.*) by removing trailing ⟨*tokens*⟩ and the end marker `\q_stop`, inserting the ⟨*code*⟩ for the successful case (if one is found) and either the `true code` or `false code` for the over all outcome, as appropriate.

Part V

The l3expan package Argument expansion

This module provides generic methods for expanding TeX arguments in a systematic manner. The functions in this module all have prefix exp.

Not all possible variations are implemented for every base function. Instead only those that are used within the LaTeX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions from the \exp_ module. They all look alike, an example would be \exp_args:NNo. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying \exp_args:NNo expands the content of third argument once before any expansion of the first and second arguments. If \seq_gpush:No was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
   \g_file_name_stack
   \l_tmpa_tl
```

In other words, the first argument to \exp_args:NNo is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate \exp_ function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of \seq_gpush:No

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is uncritical as the \cs_generate_variant:Nn function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function \cs_generate_variant:Nn, described next.

2 Methods for defining variants

We recall the set of available argument specifiers.

- N is used for single-token arguments while c constructs a control sequence from its name and passes it to a parent function as an N-type argument.
- Many argument types extract or expand some tokens and provide it as an n-type argument, namely a braced multiple-token argument: V extracts the value of a variable, v extracts the value from the name of a variable, n uses the argument as it is, o expands once, f expands fully the first token, x expands fully all tokens at the price of being non-expandable.
- A few odd argument types remain: T and F for conditional processing, otherwise identical to n, p for the parameter text in definitions, w for arguments with a specific syntax, and D to denote primitives that should not be used directly.

\cs_generate_variant:Nn

Updated: 2015-08-06

\cs_generate_variant:Nn ⟨*parent control sequence*⟩ {⟨*variant argument specifiers*⟩}

This function is used to define argument-specifier variants of the ⟨*parent control sequence*⟩ for LaTeX3 code-level macros. The ⟨*parent control sequence*⟩ is first separated into the ⟨*base name*⟩ and ⟨*original argument specifier*⟩. The comma-separated list of ⟨*variant argument specifiers*⟩ is then used to define variants of the ⟨*original argument specifier*⟩ where these are not already defined. For each ⟨*variant*⟩ given, a function is created which expands its arguments as detailed and passes them to the ⟨*parent control sequence*⟩. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function \foo:cn which expands its first argument into a control sequence name and passes the result to \foo:Nn. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions \foo:NV and \foo:cV in the same way. The \cs_generate_variant:Nn function can only be applied if the ⟨*parent control sequence*⟩ is already defined. Only n and N arguments can be changed to other types. If the ⟨*parent control sequence*⟩ is protected or if the ⟨*variant*⟩ involves x arguments, then the ⟨*variant control sequence*⟩ is also protected. The ⟨*variant*⟩ is created globally, as is any \exp_args:N⟨*variant*⟩ function needed to carry out the expansion.

While \cs_generate_variant:Nn \foo:N { o } is currently allowed, one must know that it will break if the result of the expansion is more than one token or if \foo:N requires its argument not to be braced.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster then others. Therefore (when speed is important) it is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms:

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- Arguments that need full expansion (*i.e.*, are denoted with x) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type cannot work correctly in arguments that are themselves subject to x expansion.
- In general, unless in the last position, multi-token arguments n, f, and o need special processing when more than one argument is being expanded. This special processing is not fast. Therefore it is best to use the optimized functions, namely those that contain only N, c, V, and v, and, in the last position, o, f, with possible trailing N or n, which are not expanded.

The V type returns the value of a register, which can be one of tl, int, skip, dim, toks, or built-in TeX registers. The v type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a V specifier should be used. For those referred to by (cs)name, the v specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with o specifiers be employed.

The f type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then x-expansion cannot be used, and f-expansion provides an alternative that expands as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression $3+4$ and pass the result 7 as an argument to an expandable function \example:n. For this, one should define a variant using \cs_generate_variant:Nn \example:n { f }, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that x-expansion would also expand \int_eval:n fully to its result 7, but the variant \example:x cannot be expandable. Note also that o-expansion would not expand \int_eval:n fully to its result since that function requires several expansions. Besides the fact that x-expansion is protected rather than expandable, another difference between f-expansion and x-expansion is that f-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while x-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call \example:n { 3 , \int_eval:n { 3 + 4 } } while using \example:x instead results in \example:n { 3 , 7 } at the cost of being protected. If you use this type of expansion in conditional processing then you should stick to using TF type functions only as it does not try to finish any \if... \fi: itself!

If is important to note that both f- and o-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, o-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, f-type expansion stops at the emphfirst non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable.

4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

`\exp_args:No` ⋆

`\exp_args:No ⟨function⟩ {⟨tokens⟩} ...`

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nc` ⋆
`\exp_args:cc` ⋆

`\exp_args:Nc ⟨function⟩ {⟨tokens⟩}`

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded until only characters remain, and are then turned into a control sequence. (An internal error occurs if such a conversion is not possible). The result is inserted into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the ⟨*function*⟩ name in the same manner as described for the ⟨*tokens*⟩.

`\exp_args:NV` ⋆

`\exp_args:NV ⟨function⟩ ⟨variable⟩`

This function absorbs two arguments (the names of the ⟨*function*⟩ and the ⟨*variable*⟩). The content of the ⟨*variable*⟩ are recovered and placed inside braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ⋆

`\exp_args:Nv ⟨function⟩ {⟨tokens⟩}`

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded until only characters remain, and are then turned into a control sequence. (An internal error occurs if such a conversion is not possible). This control sequence should be the name of a ⟨*variable*⟩. The content of the ⟨*variable*⟩ are recovered and placed inside braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nf` ⋆

`\exp_args:Nf ⟨function⟩ {⟨tokens⟩}`

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

\exp_args:Nx

\exp_args:Nx ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩) and exhaustively expands the ⟨*tokens*⟩ second. The result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

5 Manipulating two arguments

\exp_args:NNo ★
\exp_args:NNc ★
\exp_args:NNv ★
\exp_args:NNV ★
\exp_args:NNf ★
\exp_args:Nco ★
\exp_args:Ncf ★
\exp_args:Ncc ★
\exp_args:NVV ★

\exp_args:NNc ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*tokens*⟩}

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

\exp_args:Nno ★
\exp_args:NnV ★
\exp_args:Nnf ★
\exp_args:Noo ★
\exp_args:Nof ★
\exp_args:Noc ★
\exp_args:Nff ★
\exp_args:Nfo ★
\exp_args:Nnc ★

Updated: 2012-01-14

\exp_args:Noo ⟨*token*⟩ {⟨*tokens*$_1$⟩} {⟨*tokens*$_2$⟩}

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions need special (slower) processing.

\exp_args:NNx
\exp_args:Nnx
\exp_args:Ncx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

\exp_args:NNx ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*tokens*⟩}

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable.

6 Manipulating three arguments

\exp_args:NNNo ★
\exp_args:NNNV ★
\exp_args:Nccc ★
\exp_args:NcNc ★
\exp_args:NcNo ★
\exp_args:Ncco ★

\exp_args:NNNo ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*token*$_3$⟩ {⟨*tokens*⟩}

These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

\exp_args:NNoo ★
\exp_args:NNno ★
\exp_args:Nnno ★
\exp_args:Nnnc ★
\exp_args:Nooo ★

`\exp_args:NNoo` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*token*$_3$⟩`}` `{`⟨*tokens*⟩`}`

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.* These functions need special (slower) processing.

\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox
\exp_args:Ncnx
\exp_args:Nccx

New: 2015-08-12

`\exp_args:NNnx` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*tokens*$_1$⟩`}` `{`⟨*tokens*$_2$⟩`}`

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

7 Unbraced expansion

\exp_last_unbraced:NV ★
\exp_last_unbraced:(Nf|No|Nv) ★
\exp_last_unbraced:Nco ★
\exp_last_unbraced:(NcV|NNV|NNo) ★
\exp_last_unbraced:Nno ★
\exp_last_unbraced:(Noo|Nfo) ★
\exp_last_unbraced:NNNV ★
\exp_last_unbraced:NNNo ★
\exp_last_unbraced:NnNo ★

Updated: 2012-02-12

`\exp_last_unbraced:Nno` ⟨*token*⟩ ⟨*tokens*$_1$⟩ ⟨*tokens*$_2$⟩

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the `:Nno`, `:Noo`, and `:Nfo` variants need special (slower) processing.

TeXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is `f`-expanded.

\exp_last_unbraced:Nx

`\exp_last_unbraced:Nx` ⟨*function*⟩ `{`⟨*tokens*⟩`}`

This functions fully expands the ⟨*tokens*⟩ and leaves the result in the input stream after reinsertion of ⟨*function*⟩. This function is not expandable.

\exp_last_two_unbraced:Noo ★

`\exp_last_two_unbraced:Noo` ⟨*token*⟩ ⟨*tokens*$_1$⟩ `{`⟨*tokens*$_2$⟩`}`

This function absorbs three arguments and expand the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

`\exp_after:wN` ★

`\exp_after:wN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

Carries out a single expansion of ⟨*token*$_2$⟩ (which may consume arguments) prior to the expansion of ⟨*token*$_1$⟩. If ⟨*token*$_2$⟩ has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that ⟨*token*$_1$⟩ may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal TEX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

TEXhackers note: This is the TEX primitive `\expandafter` renamed.

8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

`\exp_not:N` ★

`\exp_not:N` ⟨*token*⟩

Prevents expansion of the ⟨*token*⟩ in a context where it would otherwise be expanded, for example an `x`-type argument.

TEXhackers note: This is the TEX `\noexpand` primitive.

`\exp_not:c` ★

`\exp_not:c {`⟨*tokens*⟩`}`

Expands the ⟨*tokens*⟩ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

`\exp_not:n` ★

`\exp_not:n {`⟨*tokens*⟩`}`

Prevents expansion of the ⟨*tokens*⟩ in a context where they would otherwise be expanded, for example an `x`-type argument.

TEXhackers note: This is the ε-TEX `\unexpanded` primitive. Hence its argument *must* be surrounded by braces.

`\exp_not:V` ★

`\exp_not:V` ⟨*variable*⟩

Recovers the content of the ⟨*variable*⟩, then prevents expansion of this material in a context where it would otherwise be expanded, for example an `x`-type argument.

`\exp_not:v` ★

`\exp_not:v {`⟨*tokens*⟩`}`

Expands the ⟨*tokens*⟩ until only unexpandable content remains, and then converts this into a control sequence (which should be a ⟨*variable*⟩ name). The content of the ⟨*variable*⟩ is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an `x`-type argument.

`\exp_not:o` ★

`\exp_not:o {`⟨*tokens*⟩`}`

Expands the ⟨*tokens*⟩ once, then prevents any further expansion in a context where they would otherwise be expanded, for example an `x`-type argument.

\exp_not:f ★

```
\exp_not:f {⟨tokens⟩}
```

Expands ⟨*tokens*⟩ fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.

\exp_stop_f: ★

Updated: 2011-06-03

```
\foo_bar:f { ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩ }
```

This function terminates an `f`-type expansion. Thus if a function `\foo_bar:f` starts an `f`-type expansion and all of ⟨*tokens*⟩ are expandable `\exp_stop_f:` terminates the expansion of tokens even if ⟨*more tokens*⟩ are also expandable. The function itself is an implicit space token. Inside an `x`-type expansion, it retains its form, but when typeset it produces the underlying space (␣).

9 Controlled expansion

The expl3 language makes all efforts to hide the complexity of TEX expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the "base" functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down TEX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of ⟨*expandable-tokens*⟩ as that will break badly if unexpandable tokens are encountered in that place!

\exp:w ★
\exp_end: ★

New: 2015-08-23

```
\exp:w ⟨expandable-tokens⟩ \exp_end:
```

Expands ⟨*expandable-tokens*⟩ until reaching `\exp_end:` at which point expansion stops. The full expansion of ⟨*expandable-tokens*⟩ has to be empty. If any token in ⟨*expandable-tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.[2]

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of ⟨*expandable-tokens*⟩ rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

[2] Due to the implementation you might get the character in position 0 in the current font (typically "`") in the output without any error message!

\exp:w ⋆
\exp_end_continue_f:w ⋆

New: 2015-08-23

\exp:w ⟨*expandable-tokens*⟩ \exp_end_continue_f:w ⟨*further-tokens*⟩

Expands ⟨*expandable-tokens*⟩ until reaching \exp_end_continue_f:w at which point expansion continues as an f-type expansion expanding ⟨*further-tokens*⟩ until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by \exp_stop_f:). As with all f-type expansions a space ending the expansion gets removed.

The full expansion of ⟨*expandable-tokens*⟩ has to be empty. If any token in ⟨*expandable-tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result \exp_end_continue_f:w will be misinterpreted later on.[3]

In typical use cases ⟨*expandable-tokens*⟩ contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the \exp_after:wN triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command \exp_after:wN would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

\exp:w ⟨*expandable-tokens*⟩ \exp_end:

can be alternatively achieved through an f-type expansion by using \exp_stop_f:, i.e.

\exp:w \exp_end_continue_f:w ⟨*expandable-tokens*⟩ \exp_stop_f:

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

\exp:w ⋆
\exp_end_continue_f:nw ⋆

New: 2015-08-23

\exp:w ⟨*expandable-tokens*⟩ \exp_end_continue_f:nw ⟨*further-tokens*⟩

The difference to \exp_end_continue_f:w is that we first we pick up an argument which is then returned to the input stream. If ⟨*further-tokens*⟩ starts with a brace group then the braces are removed. If on the other hand it starts with space tokens then these space tokens are removed while searching for the argument. Thus such space tokens will not terminate the f-type expansion.

10 Internal functions and variables

\l__exp_internal_tl

The \exp_ module has its private variables to temporarily store results of the argument expansion. This is done to avoid interference with other functions using temporary variables.

[3] In this particular case you may get a character into the output as well as an error message.

```
\::n
\::N
\::p
\::c
\::o
\::f
\::x
\::v
\::V
\:::
```

```
\cs_set:Npn \exp_args:Ncof { \::c \::o \::f \::: }
```

Internal forms for the base expansion types. These names do *not* conform to the general LaTeX3 approach as this makes them more readily visible in the log and so forth.

Part VI
The l3tl package
Token lists

TeX works with tokens, and LaTeX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called "token list variable", which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test an manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, function which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two "views" of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of "items", or a list of "tokens". An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or ␣, `{`, or `}` (assuming normal TeX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, ␣, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

1 Creating and initialising token list variables

```
\tl_new:N
\tl_new:c
```

`\tl_new:N` ⟨*tl var*⟩

Creates a new ⟨*tl var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*tl var*⟩ is initially empty.

```
\tl_const:Nn
\tl_const:(Nx|cn|cx)
```

`\tl_const:Nn` ⟨*tl var*⟩ `{`⟨*token list*⟩`}`

Creates a new constant ⟨*tl var*⟩ or raises an error if the name is already taken. The value of the ⟨*tl var*⟩ is set globally to the ⟨*token list*⟩.

```
\tl_clear:N
\tl_clear:c
\tl_gclear:N
\tl_gclear:c
```

`\tl_clear:N` ⟨*tl var*⟩

Clears all entries from the ⟨*tl var*⟩.

`\tl_clear_new:N`
`\tl_clear_new:c`
`\tl_gclear_new:N`
`\tl_gclear_new:c`

`\tl_clear_new:N ⟨tl var⟩`

Ensures that the ⟨*tl var*⟩ exists globally by applying `\tl_new:N` if necessary, then applies `\tl_(g)clear:N` to leave the ⟨*tl var*⟩ empty.

`\tl_set_eq:NN`
`\tl_set_eq:(cN|Nc|cc)`
`\tl_gset_eq:NN`
`\tl_gset_eq:(cN|Nc|cc)`

`\tl_set_eq:NN ⟨tl var1⟩ ⟨tl var2⟩`

Sets the content of ⟨*tl var*$_1$⟩ equal to that of ⟨*tl var*$_2$⟩.

`\tl_concat:NNN`
`\tl_concat:ccc`
`\tl_gconcat:NNN`
`\tl_gconcat:ccc`

New: 2012-05-18

`\tl_concat:NNN ⟨tl var1⟩ ⟨tl var2⟩ ⟨tl var3⟩`

Concatenates the content of ⟨*tl var*$_2$⟩ and ⟨*tl var*$_3$⟩ together and saves the result in ⟨*tl var*$_1$⟩. The ⟨*tl var*$_2$⟩ is placed at the left side of the new token list.

`\tl_if_exist_p:N` ★
`\tl_if_exist_p:c` ★
`\tl_if_exist:NTF` ★
`\tl_if_exist:cTF` ★

New: 2012-03-03

`\tl_if_exist_p:N ⟨tl var⟩`
`\tl_if_exist:NTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*tl var*⟩ is currently defined. This does not check that the ⟨*tl var*⟩ really is a token list variable.

2 Adding data to token list variables

`\tl_set:Nn`
`\tl_set:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)`
`\tl_gset:Nn`
`\tl_gset:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)`

`\tl_set:Nn ⟨tl var⟩ {⟨tokens⟩}`

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, removing any previous content from the variable.

`\tl_put_left:Nn`
`\tl_put_left:(NV|No|Nx|cn|cV|co|cx)`
`\tl_gput_left:Nn`
`\tl_gput_left:(NV|No|Nx|cn|cV|co|cx)`

`\tl_put_left:Nn ⟨tl var⟩ {⟨tokens⟩}`

Appends ⟨*tokens*⟩ to the left side of the current content of ⟨*tl var*⟩.

`\tl_put_right:Nn`
`\tl_put_right:(NV|No|Nx|cn|cV|co|cx)`
`\tl_gput_right:Nn`
`\tl_gput_right:(NV|No|Nx|cn|cV|co|cx)`

`\tl_put_right:Nn ⟨tl var⟩ {⟨tokens⟩}`

Appends ⟨*tokens*⟩ to the right side of the current content of ⟨*tl var*⟩.

3 Modifying token list variables

`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`

Updated: 2011-08-11

`\tl_replace_once:Nnn ⟨tl var⟩ {⟨old tokens⟩} {⟨new tokens⟩}`

Replaces the first (leftmost) occurrence of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\tl_replace_all:Nnn`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`

Updated: 2011-08-11

`\tl_replace_all:Nnn ⟨tl var⟩ {⟨old tokens⟩} {⟨new tokens⟩}`

Replaces all occurrences of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*old tokens*⟩ may remain after the replacement (see `\tl_remove_all:Nn` for an example).

`\tl_remove_once:Nn`
`\tl_remove_once:cn`
`\tl_gremove_once:Nn`
`\tl_gremove_once:cn`

Updated: 2011-08-11

`\tl_remove_once:Nn ⟨tl var⟩ {⟨tokens⟩}`

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\tl_remove_all:Nn`
`\tl_remove_all:cn`
`\tl_gremove_all:Nn`
`\tl_gremove_all:cn`

Updated: 2011-08-11

`\tl_remove_all:Nn ⟨tl var⟩ {⟨tokens⟩}`

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*tokens*⟩ may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply TeX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes).

```
\tl_set_rescan:Nnn
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2015-08-11

`\tl_set_rescan:Nnn ⟨tl var⟩ {⟨setup⟩} {⟨tokens⟩}`

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, applying the category code régime specified in the ⟨*setup*⟩ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_rescan:nn`.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

```
\tl_rescan:nn
```

Updated: 2015-08-11

`\tl_rescan:nn {⟨setup⟩} {⟨tokens⟩}`

Rescans ⟨*tokens*⟩ applying the category code régime specified in the ⟨*setup*⟩, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of `\tl_rescan:nn`.) The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the ⟨*tokens*⟩ argument of `\tl_rescan:nn`.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

5 Token list conditionals

```
\tl_if_blank_p:n        ⋆
\tl_if_blank_p:(V|o)    ⋆
\tl_if_blank:nTF        ⋆
\tl_if_blank:(V|o)TF    ⋆
```

`\tl_if_blank_p:n {⟨token list⟩}`
`\tl_if_blank:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*token list*⟩ consists only of blank spaces (*i.e.* contains no item). The test is `true` if ⟨*token list*⟩ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

\tl_if_empty_p:N ★
\tl_if_empty_p:c ★
\tl_if_empty:NTF ★
\tl_if_empty:cTF ★

\tl_if_empty_p:N ⟨*tl var*⟩
\tl_if_empty:NTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list variable*⟩ is entirely empty (*i.e.* contains no tokens at all).

\tl_if_empty_p:n ★
\tl_if_empty_p:(V|o) ★
\tl_if_empty:nTF ★
\tl_if_empty:(V|o)TF ★

New: 2012-05-24
Updated: 2012-06-05

\tl_if_empty_p:n {⟨*token list*⟩}
\tl_if_empty:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ is entirely empty (*i.e.* contains no tokens at all).

\tl_if_eq_p:NN ★
\tl_if_eq_p:(Nc|cN|cc) ★
\tl_if_eq:NNTF ★
\tl_if_eq:(Nc|cN|cc)TF ★

\tl_if_eq_p:NN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩
\tl_if_eq:NNTF ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*token list variables*⟩ and is logically `true` if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields `false`.

\tl_if_eq:nnTF

\tl_if_eq:nnTF {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_1$⟩ and ⟨*token list*$_2$⟩ contain the same list of tokens, both in respect of character codes and category codes.

\tl_if_in:NnTF
\tl_if_in:cnTF

\tl_if_in:NnTF ⟨*tl var*⟩ {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ is found in the content of the ⟨*tl var*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_in:nnTF
\tl_if_in:(Vn|on|no)TF

\tl_if_in:nnTF {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_2$⟩ is found inside ⟨*token list*$_1$⟩. The ⟨*token list*$_2$⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_single_p:N ★
\tl_if_single_p:c ★
\tl_if_single:NTF ★
\tl_if_single:cTF ★

Updated: 2011-08-13

\tl_if_single_p:N ⟨*tl var*⟩
\tl_if_single:NTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the content of the ⟨*tl var*⟩ consists of a single item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

\tl_if_single_p:n ★
\tl_if_single:nTF ★

Updated: 2011-08-13

```
\tl_if_single_p:n {⟨token list⟩}
\tl_if_single:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token list*⟩ has exactly one item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \tl_count:n.

\tl_case:Nn ★
\tl_case:cn ★
\tl_case:NnTF ★
\tl_case:cnTF ★

New: 2013-07-24

```
\tl_case:NnTF ⟨test token list variable⟩
  {
    ⟨token list variable case1⟩ {⟨code case1⟩}
    ⟨token list variable case2⟩ {⟨code case2⟩}
    ...
    ⟨token list variable casen⟩ {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for \tl_if_eq:NNTF) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function \tl_case:Nn, which does nothing if there is no match, is also available.

6 Mapping to token lists

\tl_map_function:NN ☆
\tl_map_function:cN ☆

Updated: 2012-06-29

```
\tl_map_function:NN ⟨tl var⟩ ⟨function⟩
```

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*tl var*⟩. The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving n-type arguments. See also \tl_map_function:nN.

\tl_map_function:nN ☆

Updated: 2012-06-29

```
\tl_map_function:nN ⟨token list⟩ ⟨function⟩
```

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*token list*⟩, The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving n-type arguments. See also \tl_map_function:NN.

\tl_map_inline:Nn
\tl_map_inline:cn

Updated: 2012-06-29

```
\tl_map_inline:Nn ⟨tl var⟩ {⟨inline function⟩}
```

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. See also \tl_map_function:NN.

\tl_map_inline:nn

Updated: 2012-06-29

```
\tl_map_inline:nn ⟨token list⟩ {⟨inline function⟩}
```

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. See also \tl_map_function:nN.

`\tl_map_variable:NNn`
`\tl_map_variable:cNn`

Updated: 2012-06-29

`\tl_map_variable:NNn` ⟨*tl var*⟩ ⟨*variable*⟩ {⟨*function*⟩}

Applies the ⟨*function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*function*⟩ should consist of code which receives the ⟨*item*⟩ stored in the ⟨*variable*⟩. One variable mapping can be nested inside another. See also `\tl_map_inline:Nn`.

`\tl_map_variable:nNn`

Updated: 2012-06-29

`\tl_map_variable:nNn` ⟨*token list*⟩ ⟨*variable*⟩ {⟨*function*⟩}

Applies the ⟨*function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*function*⟩ should consist of code which receives the ⟨*item*⟩ stored in the ⟨*variable*⟩. One variable mapping can be nested inside another. See also `\tl_map_inline:nn`.

`\tl_map_break:` ☆

Updated: 2012-06-29

`\tl_map_break:`

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list variable*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
    % Do something useful
  }
```

See also `\tl_map_break:n`. Use outside of a `\tl_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This depends on the design of the mapping function.

`\tl_map_break:n` ☆

Updated: 2012-06-29

`\tl_map_break:n` {⟨*tokens*⟩}

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list variable*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \tl_map_break:n { <tokens> } }
    % Do something useful
  }
```

Use outside of a `\tl_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This depends on the design of the mapping function.

7 Using token lists

\tl_to_str:n ⋆
\tl_to_str:V ⋆

`\tl_to_str:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting character tokens in the input stream. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

TEXhackers note: Converting a ⟨*token list*⟩ to a ⟨*string*⟩ yields a concatenation of the string representations of every token in the ⟨*token list*⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not "letter".

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character "backslash", "lower-case a", "space", but it may also produce a single "lower-case a" if the escape character is negative and `a` is currently not a letter.

\tl_to_str:N ⋆
\tl_to_str:c ⋆

`\tl_to_str:N ⟨tl var⟩`

Converts the content of the ⟨*tl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

\tl_use:N ⋆
\tl_use:c ⋆

`\tl_use:N ⟨tl var⟩`

Recovers the content of a ⟨*tl var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*tl var*⟩ directly without an accessor function.

8 Working with the content of token lists

\tl_count:n ⋆
\tl_count:(V|o) ⋆

New: 2012-05-13

`\tl_count:n {⟨tokens⟩}`

Counts the number of ⟨*items*⟩ in ⟨*tokens*⟩ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group (`{`...`}`). This process ignores any unprotected spaces within ⟨*tokens*⟩. See also `\tl_count:N`. This function requires three expansions, giving an ⟨*integer denotation*⟩.

`\tl_count:N` ★
`\tl_count:c` ★
New: 2012-05-13

`\tl_count:N` ⟨*tl var*⟩

Counts the number of token groups in the ⟨*tl var*⟩ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group (`{`...`}`). This process ignores any unprotected spaces within the ⟨*tl var*⟩. See also `\tl_count:n`. This function requires three expansions, giving an ⟨*integer denotation*⟩.

`\tl_reverse:n` ★
`\tl_reverse:(V|o)` ★
Updated: 2012-01-08

`\tl_reverse:n {`⟨*token list*⟩`}`

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨*item*$_1$⟩⟨*item*$_2$⟩⟨*item*$_3$⟩ ... ⟨*item*$_n$⟩ becomes ⟨*item*$_n$⟩... ⟨*item*$_3$⟩⟨*item*$_2$⟩⟨*item*$_1$⟩. This process preserves unprotected space within the ⟨*token list*⟩. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

TEXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_reverse:N`
`\tl_reverse:c`
`\tl_greverse:N`
`\tl_greverse:c`
Updated: 2012-01-08

`\tl_reverse:N` ⟨*tl var*⟩

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that ⟨*item*$_1$⟩⟨*item*$_2$⟩⟨*item*$_3$⟩ ... ⟨*item*$_n$⟩ becomes ⟨*item*$_n$⟩... ⟨*item*$_3$⟩⟨*item*$_2$⟩⟨*item*$_1$⟩. This process preserves unprotected spaces within the ⟨*token list variable*⟩. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also `\tl_reverse:n`, and, for improved performance, `\tl_reverse_items:n`.

`\tl_reverse_items:n` ★
New: 2012-01-08

`\tl_reverse_items:n {`⟨*token list*⟩`}`

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that `{`⟨*item*$_1$⟩`}{`⟨*item*$_2$⟩`}{`⟨*item*$_3$⟩`}` ... `{`⟨*item*$_n$⟩`}` becomes `{`⟨*item*$_n$⟩`}` ... `{`⟨*item*$_3$⟩`}{`⟨*item*$_2$⟩`}{`⟨*item*$_1$⟩`}`. This process removes any unprotected space within the ⟨*token list*⟩. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

TEXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_trim_spaces:n` ★
New: 2011-07-09
Updated: 2012-06-25

`\tl_trim_spaces:n {`⟨*token list*⟩`}`

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and leaves the result in the input stream.

TEXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_trim_spaces:N`
`\tl_trim_spaces:c`
`\tl_gtrim_spaces:N`
`\tl_gtrim_spaces:c`
New: 2011-07-09

`\tl_trim_spaces:N` ⟨*tl var*⟩

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the ⟨*tl var*⟩. Note that this therefore *resets* the content of the variable.

`\tl_sort:Nn`
`\tl_sort:cn`
`\tl_gsort:Nn`
`\tl_gsort:cn`

New: 2017-02-06

`\tl_sort:Nn ⟨tl var⟩ {⟨comparison code⟩}`

Sorts the items in the ⟨*tl var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*tl var*⟩. The details of sorting comparison are described in Section 1.

`\tl_sort:nN` ⋆

New: 2017-02-06

`\tl_sort:nN {⟨token list⟩} ⟨conditional⟩`

Sorts the items in the ⟨*token list*⟩, using the ⟨*conditional*⟩ to compare items, and leaves the result in the input stream. The ⟨*conditional*⟩ should have signature `:nnTF`, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped. The details of sorting comparison are described in Section 1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

9 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

`\tl_head:N` ⋆
`\tl_head:n` ⋆
`\tl_head:(V|v|f)` ⋆

Updated: 2012-09-09

`\tl_head:n {⟨token list⟩}`

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

both leave `a` in the input stream. If the "head" is a brace group, rather than a single token, the braces are removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields `␣ab`. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_head:w` ⋆

`\tl_head:w ⟨token list⟩ { } \q_stop`

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank ⟨*token list*⟩ (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a "blank" argument. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

`\tl_tail:N` ⋆
`\tl_tail:n` ⋆
`\tl_tail:(V|v|f)` ⋆

Updated: 2012-09-01

`\tl_tail:n {⟨token list⟩}`

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*token list*⟩, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

both leave `␣{bc}d` in the input stream. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_if_head_eq_catcode_p:nN` ⋆
`\tl_if_head_eq_catcode:nNTF` ⋆

Updated: 2012-07-09

`\tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩`
  `{⟨true code⟩} {⟨false code⟩}`

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same category code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test is always `false`.

`\tl_if_head_eq_charcode_p:nN` ⋆
`\tl_if_head_eq_charcode_p:fN` ⋆
`\tl_if_head_eq_charcode:nNTF` ⋆
`\tl_if_head_eq_charcode:fNTF` ⋆

Updated: 2012-07-09

`\tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩`
  `{⟨true code⟩} {⟨false code⟩}`

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same character code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test is always `false`.

`\tl_if_head_eq_meaning_p:nN` ⋆
`\tl_if_head_eq_meaning:nNTF` ⋆

Updated: 2012-07-09

`\tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩`
  `{⟨true code⟩} {⟨false code⟩}`

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same meaning as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, the test is always `false`.

```
\tl_if_head_is_group_p:n ⋆
\tl_if_head_is_group:nTF ⋆
```

New: 2012-07-08

```
\tl_if_head_is_group_p:n {⟨token list⟩}
\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit begin-group character (with category code 1 and any character code), in other words, if the ⟨*token list*⟩ starts with a brace group. In particular, the test is `false` if the ⟨*token list*⟩ starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_N_type_p:n ⋆
\tl_if_head_is_N_type:nTF ⋆
```

New: 2012-07-08

```
\tl_if_head_is_N_type_p:n {⟨token list⟩}
\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is a normal `N`-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields `false`, as it does not have a "normal" first token. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_space_p:n ⋆
\tl_if_head_is_space:nTF ⋆
```

Updated: 2012-07-08

```
\tl_if_head_is_space_p:n {⟨token list⟩}
\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is `false` if the ⟨*token list*⟩ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

10 Using a single item

```
\tl_item:nn ⋆
\tl_item:Nn ⋆
\tl_item:cn ⋆
```

New: 2014-07-17

```
\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}
```

Indexing items in the ⟨*token list*⟩ from 1 on the left, this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the ⟨*token list*⟩ in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then thr function expands to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

11 Viewing token lists

```
\tl_show:N
\tl_show:c
```

Updated: 2015-08-01

```
\tl_show:N ⟨tl var⟩
```

Displays the content of the ⟨*tl var*⟩ on the terminal.

TEXhackers note: This is similar to the TEX primitive `\show`, wrapped to a fixed number of characters per line.

\tl_show:n

Updated: 2015-08-07

\tl_show:n ⟨*token list*⟩

Displays the ⟨*token list*⟩ on the terminal.

TeXhackers note: This is similar to the ε-TeX primitive \showtokens, wrapped to a fixed number of characters per line.

\tl_log:N
\tl_log:c

New: 2014-08-22
Updated: 2015-08-01

\tl_log:N ⟨*tl var*⟩

Writes the content of the ⟨*tl var*⟩ in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n

New: 2014-08-22
Updated: 2015-08-07

\tl_log:n {⟨*token list*⟩}

Writes the ⟨*token list*⟩ in the log file. See also \tl_show:n which displays the result in the terminal.

12 Constant token lists

\c_empty_tl

Constant that is always empty.

\c_space_tl

An explicit space character contained in a token list (compare this with \c_space_token). For use where an explicit space is required.

13 Scratch token lists

\l_tmpa_tl
\l_tmpb_tl

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_tl
\g_tmpb_tl

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

14 Internal functions

__tl_trim_spaces:nn

__tl_trim_spaces:nn { \q_mark ⟨*token list*⟩ } {⟨*continuation*⟩}

This function removes all leading and trailing explicit space characters from the ⟨*token list*⟩, and expands to the ⟨*continuation*⟩, followed by a brace group containing \use_none:n \q_mark ⟨*trimmed token list*⟩. For instance, \tl_trim_spaces:n is implemented by taking the ⟨*continuation*⟩ to be \exp_not:o, and the o-type expansion removes the \q_mark. This function is also used in l3clist and l3candidates.

Part VII
The l3str package
Strings

TeX associates each character with a category code: as such, there is no concept of a "string" as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense "ignoring" category codes: this is done by treating token lists as strings in a TeX sense.

A TeX string (and thus an expl3 string) is a series of characters which have category code 12 ("other") with the exception of space characters which have category code 10 ("space"). Thus at a technical level, a TeX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn't primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

Note that as string variables are a special case of token list variables the coverage of `\str_...:N` functions is somewhat smaller than `\tl_...:N`.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in l3basics, l3tl and l3token, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N`
`\str_new:c`

New: 2015-09-18

`\str_new:N` ⟨*str var*⟩

Creates a new ⟨*str var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*str var*⟩ is initially empty.

`\str_const:Nn`
`\str_const:(Nx|cn|cx)`

New: 2015-09-18

`\str_const:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ is set globally to the ⟨*token list*⟩, converted to a string.

`\str_clear:N`
`\str_clear:c`
`\str_gclear:N`
`\str_gclear:c`

New: 2015-09-18

`\str_clear:N` ⟨*str var*⟩

Clears the content of the ⟨*str var*⟩.

`\str_clear_new:N`
`\str_clear_new:c`

New: 2015-09-18

`\str_clear_new:N` ⟨*str var*⟩

Ensures that the ⟨*str var*⟩ exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the ⟨*str var*⟩ empty.

`\str_set_eq:NN`
`\str_set_eq:(cN|Nc|cc)`
`\str_gset_eq:NN`
`\str_gset_eq:(cN|Nc|cc)`

New: 2015-09-18

`\str_set_eq:NN` ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩

Sets the content of ⟨*str var*$_1$⟩ equal to that of ⟨*str var*$_2$⟩.

2 Adding data to string variables

`\str_set:Nn`
`\str_set:(Nx|cn|cx)`
`\str_gset:Nn`
`\str_gset:(Nx|cn|cx)`

New: 2015-09-18

`\str_set:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

`\str_put_left:Nn`
`\str_put_left:(Nx|cn|cx)`
`\str_gput_left:Nn`
`\str_gput_left:(Nx|cn|cx)`

New: 2015-09-18

`\str_put_left:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

`\str_put_right:Nn`
`\str_put_right:(Nx|cn|cx)`
`\str_gput_right:Nn`
`\str_gput_right:(Nx|cn|cx)`

New: 2015-09-18

`\str_put_right:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

2.1 String conditionals

\str_if_exist_p:N ★
\str_if_exist_p:c ★
\str_if_exist:NTF ★
\str_if_exist:cTF ★

New: 2015-09-18

\str_if_exist_p:N ⟨*str var*⟩
\str_if_exist:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ is currently defined. This does not check that the ⟨*str var*⟩ really is a string.

\str_if_empty_p:N ★
\str_if_empty_p:c ★
\str_if_empty:NTF ★
\str_if_empty:cTF ★

New: 2015-09-18

\str_if_empty_p:N ⟨*str var*⟩
\str_if_empty:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*string variable*⟩ is entirely empty (*i.e.* contains no characters at all).

\str_if_eq_p:NN ★
\str_if_eq_p:(Nc|cN|cc) ★
\str_if_eq:NNTF ★
\str_if_eq:(Nc|cN|cc)TF ★

New: 2015-09-18

\str_if_eq_p:NN ⟨*str var$_1$*⟩ ⟨*str var$_2$*⟩
\str_if_eq:NNTF ⟨*str var$_1$*⟩ ⟨*str var$_2$*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*str variables*⟩ and is logically `true` if the two contain the same characters.

\str_if_eq_p:nn ★
\str_if_eq_p:(Vn|on|no|nV|VV) ★
\str_if_eq:nnTF ★
\str_if_eq:(Vn|on|no|nV|VV)TF ★

\str_if_eq_p:nn {⟨*tl$_1$*⟩} {⟨*tl$_2$*⟩}
\str_if_eq:nnTF {⟨*tl$_1$*⟩} {⟨*tl$_2$*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_if_eq_x_p:nn ★
\str_if_eq_x:nnTF ★

New: 2012-06-05

\str_if_eq_x_p:nn {⟨*tl$_1$*⟩} {⟨*tl$_2$*⟩}
\str_if_eq_x:nnTF {⟨*tl$_1$*⟩} {⟨*tl$_2$*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_case:nn ★
\str_case:(on|nV|nv) ★
\str_case:nnTF ★
\str_case:(on|nV|nv)TF ★

New: 2013-07-24
Updated: 2015-02-28

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩. If the two are equal (as described for `\str_if_eq:nnTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

\str_case_x:nn ★
\str_case_x:nnTF ★

New: 2013-07-24

```
\str_case_x:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string cases*⟩. If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The ⟨*test string*⟩ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

3 Working with the content of strings

\str_use:N ★
\str_use:c ★

New: 2015-09-18

```
\str_use:N ⟨str var⟩
```

Recovers the content of a ⟨*str var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*str*⟩ directly without an accessor function.

`\str_count:N` ★
`\str_count:c` ★
`\str_count:n` ★
`\str_count_ignore_spaces:n` ★

New: 2015-09-18

```
\str_count:n {⟨token list⟩}
```

Leaves in the input stream the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ★
`\str_count_spaces:c` ★
`\str_count_spaces:n` ★

New: 2015-09-18

```
\str_count_spaces:n {⟨token list⟩}
```

Leaves in the input stream the number of space characters in the string representation of ⟨*token list*⟩, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

`\str_head:N` ★
`\str_head:c` ★
`\str_head:n` ★
`\str_head_ignore_spaces:n` ★

New: 2015-09-18

```
\str_head:n {⟨token list⟩}
```

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the ⟨*string*⟩ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

`\str_tail:N` ★
`\str_tail:c` ★
`\str_tail:n` ★
`\str_tail_ignore_spaces:n` ★

New: 2015-09-18

```
\str_tail:n {⟨token list⟩}
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

```
\str_item:Nn                ★
\str_item:nn                ★
\str_item_ignore_spaces:nn  ★
```

New: 2015-09-18

`\str_item:nn {⟨token list⟩} {⟨integer expression⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, *etc.*

```
\str_range:Nnn                ★
\str_range:cnn                ★
\str_range:nnn                ★
\str_range_ignore_spaces:nnn  ★
```

New: 2015-09-18

`\str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Positive ⟨*indices*⟩ are counted from the start of the string, 1 being the first character, and negative ⟨*indices*⟩ are counted from the end of the string, -1 being the last character. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is 0, the result is empty. For instance,

```
\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }
```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The ⟨*start index*⟩ must always be smaller than or equal to the ⟨*end index*⟩: if this is not the case then no output is generated. Thus

```
\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

4 String manipulation

\str_lower_case:n ★
\str_lower_case:f ★
\str_upper_case:n ★
\str_upper_case:f ★

New: 2015-03-01

\str_lower_case:n {⟨*tokens*⟩}
\str_upper_case:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
  {
    \cs_set_protected:cpn
      {
        user
        \str_upper_case:f { \tl_head:n {#1} }
        \str_lower_case:f { \tl_tail:n {#1} }
      }
      { #2 }
  }
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_fold_case:n` for this situation (case folding is district from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

TEXhackers note: As with all expl3 functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XETEX and LuaTEX, subject only to the fact that XETEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

\str_fold_case:n ⋆
\str_fold_case:V ⋆

New: 2014-06-19
Updated: 2016-03-07

`\str_fold_case:n {⟨tokens⟩}`

Converts the input ⟨*tokens*⟩ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting ⟨*string*⟩ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for "text". The folding provided by `\str_fold_case:n` follows the mappings provided by the Unicode Consortium, who state:

> Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_fold_case:n` follows the "full" scheme defined by the Unicode Consortium (*e.g.* SSfolds to `SS`). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* `I` always folds to `i` and not to `ı`).

TEXhackers note: As with all expl3 functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XƎTEX and LuaTEX, subject only to the fact that XƎTEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

5 Viewing strings

\str_show:N
\str_show:c
\str_show:n

New: 2015-09-18

`\str_show:N ⟨str var⟩`

Displays the content of the ⟨*str var*⟩ on the terminal.

6 Constant token lists

\c_ampersand_str
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str

New: 2015-09-19

Constant strings, containing a single character token, with category code 12.

7 Scratch strings

\l_tmpa_str
\l_tmpb_str

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_str
\g_tmpb_str

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7.1 Internal string functions

__str_if_eq_x:nn ★

`\__str_if_eq_x:nn {⟨tl₁⟩} {⟨tl₂⟩}`

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Leaves `0` in the input stream if the condition is true, and `+1` or `-1` otherwise.

__str_if_eq_x_return:nn

`\__str_if_eq_x_return:nn {⟨tl₁⟩} {⟨tl₂⟩}`

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Either `\prg_return_true:` or `\prg_return_false:` is then left in the input stream. This is a version of `\str_if_eq_x:nnTF` coded for speed.

__str_to_other:n ★

`\__str_to_other:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code "other". This function can be `f`-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

`\__str_to_other_fast:n` ☆

`\__str_to_other_fast:n {⟨token list⟩}`

Same behaviour `\__str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string. It is used for `\iow_wrap:nnnN`.

`\__str_count:n` ★

`\__str_count:n {⟨other string⟩}`

This function expects an argument that is entirely made of characters with category "other", as produced by `\__str_to_other:n`. It leaves in the input stream the number of character tokens in the ⟨*other string*⟩, faster than the analogous `\str_count:n` function.

`\__str_range:nnn` ★

`\__str_range:nnn {⟨other string⟩} {⟨start index⟩} {⟨end index⟩}`

Identical to `\str_range:nnn` except that the first argument is expected to be entirely made of characters with category "other", as produced by `\__str_to_other:n`, and the result is also an ⟨*other string*⟩.

Part VIII
The l3seq package
Sequences and stacks

LaTeX3 implements a "sequence" data type, which contain an ordered list of entries which may contain any ⟨*balanced text*⟩. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in LaTeX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

`\seq_new:N`
`\seq_new:c`

`\seq_new:N ⟨sequence⟩`

Creates a new ⟨*sequence*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*sequence*⟩ initially contains no items.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

`\seq_clear:N ⟨sequence⟩`

Clears all items from the ⟨*sequence*⟩.

`\seq_clear_new:N`
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

`\seq_clear_new:N ⟨sequence⟩`

Ensures that the ⟨*sequence*⟩ exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the ⟨*sequence*⟩ empty.

`\seq_set_eq:NN`
`\seq_set_eq:(cN|Nc|cc)`
`\seq_gset_eq:NN`
`\seq_gset_eq:(cN|Nc|cc)`

`\seq_set_eq:NN ⟨sequence1⟩ ⟨sequence2⟩`

Sets the content of ⟨*sequence*$_1$⟩ equal to that of ⟨*sequence*$_2$⟩.

`\seq_set_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc)`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc)`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

New: 2014-07-17

`\seq_set_from_clist:NN ⟨sequence⟩ ⟨comma-list⟩`

Converts the data in the ⟨*comma list*⟩ into a ⟨*sequence*⟩: the original ⟨*comma list*⟩ is unchanged.

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV

New: 2011-08-15
Updated: 2012-07-02

\seq_set_split:Nnn ⟨*sequence*⟩ {⟨*delimiter*⟩} {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*sequence*⟩. Spaces on both sides of each ⟨*item*⟩ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of l3clist functions. Empty ⟨*items*⟩ are preserved by \seq_set_split:Nnn, and can be removed afterwards using \seq_remove_all:Nn ⟨*sequence*⟩ {⟨⟩}. The ⟨*delimiter*⟩ may not contain {, } or # (assuming TEX's normal category code régime). If the ⟨*delimiter*⟩ is empty, the ⟨*token list*⟩ is split into ⟨*items*⟩ as a ⟨*token list*⟩.

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

\seq_concat:NNN ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ ⟨*sequence*$_3$⟩

Concatenates the content of ⟨*sequence*$_2$⟩ and ⟨*sequence*$_3$⟩ together and saves the result in ⟨*sequence*$_1$⟩. The items in ⟨*sequence*$_2$⟩ are placed at the left side of the new sequence.

\seq_if_exist_p:N ★
\seq_if_exist_p:c ★
\seq_if_exist:N*TF* ★
\seq_if_exist:c*TF* ★

New: 2012-03-03

\seq_if_exist_p:N ⟨*sequence*⟩
\seq_if_exist:NTF ⟨*sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*sequence*⟩ is currently defined. This does not check that the ⟨*sequence*⟩ really is a sequence variable.

2 Appending data to sequences

\seq_put_left:Nn
\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

\seq_put_left:Nn ⟨*sequence*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the left of the ⟨*sequence*⟩.

\seq_put_right:Nn
\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

\seq_put_right:Nn ⟨*sequence*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the right of the ⟨*sequence*⟩.

3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the ⟨*token list variable*⟩ used with \tl_set:Nn and *never* \tl_gset:Nn.

\seq_get_left:NN
\seq_get_left:cN

Updated: 2012-05-14

\seq_get_left:NN ⟨*sequence*⟩ ⟨*token list variable*⟩

Stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker \q_no_value.

\seq_get_right:NN
\seq_get_right:cN

Updated: 2012-05-19

`\seq_get_right:NN ⟨sequence⟩ ⟨token list variable⟩`

Stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_pop_left:NN
\seq_pop_left:cN

Updated: 2012-05-14

`\seq_pop_left:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the left-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_gpop_left:NN
\seq_gpop_left:cN

Updated: 2012-05-14

`\seq_gpop_left:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the left-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_pop_right:NN
\seq_pop_right:cN

Updated: 2012-05-19

`\seq_pop_right:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the right-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_gpop_right:NN
\seq_gpop_right:cN

Updated: 2012-05-19

`\seq_gpop_right:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the right-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_item:Nn ⋆
\seq_item:cn ⋆

New: 2014-07-17

`\seq_item:Nn ⟨sequence⟩ {⟨integer expression⟩}`

Indexing items in the ⟨*sequence*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the sequence in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the sequence. If the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*sequence*⟩ (as calculated by `\seq_count:N`) then the function expands to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

`\seq_get_left:NNTF`
`\seq_get_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get_left:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

New: 2012-05-19

`\seq_get_right:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

`\seq_pop_left:NNTF`
`\seq_pop_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop_left:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

`\seq_gpop_left:NNTF`
`\seq_gpop_left:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop_left:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

`\seq_pop_right:NNTF`
`\seq_pop_right:cNTF`

New: 2012-05-19

`\seq_pop_right:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

`\seq_gpop_right:NNTF`
`\seq_gpop_right:cNTF`

New: 2012-05-19

`\seq_gpop_right:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

`\seq_remove_duplicates:N`
`\seq_remove_duplicates:c`
`\seq_gremove_duplicates:N`
`\seq_gremove_duplicates:c`

`\seq_remove_duplicates:N` ⟨*sequence*⟩

Removes duplicate items from the ⟨*sequence*⟩, leaving the left most copy of each item in the ⟨*sequence*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TEXhackers note: This function iterates through every item in the ⟨*sequence*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large sequences.

`\seq_remove_all:Nn`
`\seq_remove_all:cn`
`\seq_gremove_all:Nn`
`\seq_gremove_all:cn`

`\seq_remove_all:Nn` ⟨*sequence*⟩ `{`⟨*item*⟩`}`

Removes every occurrence of ⟨*item*⟩ from the ⟨*sequence*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

`\seq_reverse:N`
`\seq_reverse:c`
`\seq_greverse:N`
`\seq_greverse:c`

New: 2014-07-18

`\seq_reverse:N` ⟨*sequence*⟩

Reverses the order of the items stored in the ⟨*sequence*⟩.

`\seq_sort:Nn`
`\seq_sort:cn`
`\seq_gsort:Nn`
`\seq_gsort:cn`

New: 2017-02-06

`\seq_sort:Nn` ⟨*sequence*⟩ `{`⟨*comparison code*⟩`}`

Sorts the items in the ⟨*sequence*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*sequence*⟩. The details of sorting comparison are described in Section 1.

6 Sequence conditionals

`\seq_if_empty_p:N` ★
`\seq_if_empty_p:c` ★
`\seq_if_empty:NTF` ★
`\seq_if_empty:cTF` ★

`\seq_if_empty_p:N` ⟨*sequence*⟩
`\seq_if_empty:NTF` ⟨*sequence*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*sequence*⟩ is empty (containing no items).

`\seq_if_in:NnTF`
`\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF`

`\seq_if_in:NnTF` ⟨*sequence*⟩ `{`⟨*item*⟩`}` `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*item*⟩ is present in the ⟨*sequence*⟩.

7 Mapping to sequences

`\seq_map_function:NN` ☆
`\seq_map_function:cN` ☆

Updated: 2012-06-29

`\seq_map_function:NN` ⟨*sequence*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*sequence*⟩. The ⟨*function*⟩ will receive one argument for each iteration. The ⟨*items*⟩ are returned from left to right. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items. One mapping may be nested inside another.

\seq_map_inline:Nn
\seq_map_inline:cn

Updated: 2012-06-29

\seq_map_inline:Nn ⟨*sequence*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

\seq_map_variable:NNn
\seq_map_variable:(Ncn|cNn|ccn)

Updated: 2012-06-29

\seq_map_variable:NNn ⟨*sequence*⟩ ⟨*tl var.*⟩ {⟨*function using tl var.*⟩}

Stores each entry in the ⟨*sequence*⟩ in turn in the ⟨*tl var.*⟩ and applies the ⟨*function using tl var.*⟩ The ⟨*function*⟩ will usually consist of code making use of the ⟨*tl var.*⟩, but this is not enforced. One variable mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

\seq_map_break: ☆

Updated: 2012-06-29

\seq_map_break:

Used to terminate a \seq_map_... function before all entries in the ⟨*sequence*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \seq_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {⟨tokens⟩}`

Used to terminate a `\seq_map_...` function before all entries in the ⟨*sequence*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This depends on the design of the mapping function.

`\seq_count:N` ★
`\seq_count:c` ★

New: 2012-07-13

`\seq_count:N ⟨sequence⟩`

Leaves the number of items in the ⟨*sequence*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*sequence*⟩ includes those which are empty and duplicates, *i.e.* every item in a ⟨*sequence*⟩ is unique.

8 Using the content of sequences directly

`\seq_use:Nnnn` ★
`\seq_use:cnnn` ★

New: 2013-05-26

`\seq_use:Nnnn ⟨seq var⟩ {⟨separator between two⟩}`
`{⟨separator between more than two⟩} {⟨separator between final two⟩}`

Places the contents of the ⟨*seq var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the sequence has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `x`-type argument expansion.

\seq_use:Nn ★
\seq_use:cn ★

New: 2013-05-26

`\seq_use:Nn ⟨seq var⟩ {⟨separator⟩}`

Places the contents of the ⟨*seq var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the sequence has a single item, it is placed in the input stream with no ⟨*separator*⟩, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts "`a and b and c and de and f`" in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `x`-type argument expansion.

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

\seq_get:NN
\seq_get:cN

Updated: 2012-05-14

`\seq_get:NN ⟨sequence⟩ ⟨token list variable⟩`

Reads the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_pop:NN
\seq_pop:cN

Updated: 2012-05-14

`\seq_pop:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_gpop:NN
\seq_gpop:cN

Updated: 2012-05-14

`\seq_gpop:NN ⟨sequence⟩ ⟨token list variable⟩`

Pops the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ is set to the special marker `\q_no_value`.

\seq_get:NN*TF*
\seq_get:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

`\seq_get:NNTF ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the top item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_pop:NN*TF*
\seq_pop:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the top item from the ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop:NN*TF*
\seq_gpop:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the top item from the ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\seq_push:Nn
\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gpush:Nn
\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

`\seq_push:Nn` ⟨*sequence*⟩ {⟨*item*⟩}

Adds the {⟨*item*⟩} to the top of the ⟨*sequence*⟩.

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurences of a given item. To make sure that a ⟨*sequence variable*⟩ only has distinct items, use `\seq_remove_duplicates:N` ⟨*sequence variable*⟩. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set ⟨*seq var*⟩ are straightforward. For instance, `\seq_count:N` ⟨*seq var*⟩ expands to the number of items, while `\seq_if_in:NnTF` ⟨*seq var*⟩ {⟨*item*⟩} tests if the ⟨*item*⟩ is in the set.

Adding an ⟨*item*⟩ to a set ⟨*seq var*⟩ can be done by appending it to the ⟨*seq var*⟩ if it is not already in the ⟨*seq var*⟩:

```
\seq_if_in:NnF ⟨seq var⟩ {⟨item⟩}
{ \seq_put_right:Nn ⟨seq var⟩ {⟨item⟩} }
```

Removing an ⟨*item*⟩ from a set ⟨*seq var*⟩ can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}
```

The intersection of two sets ⟨*seq* var_1⟩ and ⟨*seq* var_2⟩ can be stored into ⟨*seq* var_3⟩ by collecting items of ⟨*seq* var_1⟩ which are in ⟨*seq* var_2⟩.

```
\seq_clear:N ⟨seq var$_3$⟩
\seq_map_inline:Nn ⟨seq var$_1$⟩
{
\seq_if_in:NnT ⟨seq var$_2$⟩ {#1}
{ \seq_put_right:Nn ⟨seq var$_3$⟩ {#1} }
}
```

The code as written here only works if ⟨*seq var*$_3$⟩ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence `\l__`⟨*pkg*⟩`_internal_seq`, then ⟨*seq var*$_3$⟩ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ through

```
\seq_concat:NNN ⟨seq var$_3$⟩ ⟨seq var$_1$⟩ ⟨seq var$_2$⟩
\seq_remove_duplicates:N ⟨seq var$_3$⟩
```

or by adding items to (a copy of) ⟨*seq var*$_1$⟩ one by one

```
\seq_set_eq:NN ⟨seq var$_3$⟩ ⟨seq var$_1$⟩
\seq_map_inline:Nn ⟨seq var$_2$⟩
{
\seq_if_in:NnF ⟨seq var$_3$⟩ {#1}
{ \seq_put_right:Nn ⟨seq var$_3$⟩ {#1} }
}
```

The second approach is faster than the first when the ⟨*seq var*$_2$⟩ is short compared to ⟨*seq var*$_1$⟩.

The difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by removing items of the ⟨*seq var*$_2$⟩ from (a copy of) the ⟨*seq var*$_1$⟩ one by one.

```
\seq_set_eq:NN ⟨seq var$_3$⟩ ⟨seq var$_1$⟩
\seq_map_inline:Nn ⟨seq var$_2$⟩
{ \seq_remove_all:Nn ⟨seq var$_3$⟩ {#1} }
```

The symmetric difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by computing the difference between ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ and storing the result as `\l__`⟨*pkg*⟩`_internal_seq`, then the difference between ⟨*seq var*$_2$⟩ and ⟨*seq var*$_1$⟩, and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l__⟨pkg⟩_internal_seq ⟨seq var$_1$⟩
\seq_map_inline:Nn ⟨seq var$_2$⟩
{ \seq_remove_all:Nn \l__⟨pkg⟩_internal_seq {#1} }
\seq_set_eq:NN ⟨seq var$_3$⟩ ⟨seq var$_2$⟩
\seq_map_inline:Nn ⟨seq var$_1$⟩
{ \seq_remove_all:Nn ⟨seq var$_3$⟩ {#1} }
\seq_concat:NNN ⟨seq var$_3$⟩ ⟨seq var$_3$⟩ \l__⟨pkg⟩_internal_seq
```

11 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

`New: 2012-07-02`

\l_tmpa_seq
\l_tmpb_seq

New: 2012-04-26

Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_seq
\g_tmpb_seq

New: 2012-04-26

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12 Viewing sequences

\seq_show:N
\seq_show:c

Updated: 2015-08-01

\seq_show:N ⟨*sequence*⟩

Displays the entries in the ⟨*sequence*⟩ in the terminal.

\seq_log:N
\seq_log:c

New: 2014-08-12
Updated: 2015-08-01

\seq_log:N ⟨*sequence*⟩

Writes the entries in the ⟨*sequence*⟩ in the log file.

13 Internal sequence functions

\s__seq

This scan mark (equal to \scan_stop:) marks the beginning of a sequence variable.

__seq_item:n ⋆

__seq_item:n {⟨*item*⟩}

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

__seq_push_item_def:n
__seq_push_item_def:x

__seq_push_item_def:n {⟨*code*⟩}

Saves the definition of __seq_item:n and redefines it to accept one parameter and expand to ⟨*code*⟩. This function should always be balanced by use of __seq_pop_item_def:.

__seq_pop_item_def:

__seq_pop_item_def:

Restores the definition of __seq_item:n most recently saved by __seq_push_item_def:n. This function should always be used in a balanced pair with __seq_push_item_def:n.

Part IX
The **l3int** package
Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* ("`intexpr`").

1 Integer expressions

`\int_eval:n` ⋆

`\int_eval:n {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩, expanding any integer and token list variables within the ⟨*expression*⟩ to their content (without requiring `\int_use:N`/`\tl_use:N`) and applying the standard mathematical rules. For example both

```
    \int_eval:n { 5 +  4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
    \tl_new:N  \l_my_tl
    \tl_set:Nn \l_my_tl { 5 }
    \int_new:N  \l_my_int
    \int_set:Nn \l_my_int { 4 }
  \int_eval:n { \l_my_tl +  \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

both evaluate to -6. The `{`⟨*integer expression*⟩`}` may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an ⟨*integer denotation*⟩: a sequence of a sign and digits matching the regex `\-?[0-9]+`). After expansion `\int_eval:n` yields an ⟨*integer denotation*⟩ which is left in the input stream.

TEXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an ⟨*internal integer*⟩, and therefore requires suitable termination if used in a TEX-style integer assignment.

`\int_abs:n` ⋆

Updated: 2012-09-26

`\int_abs:n {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_round:nn` ⋆

Updated: 2012-09-26

`\int_div_round:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an ⟨*integer expression*⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_truncate:nn` ★

Updated: 2012-02-09

`\int_div_truncate:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_max:nn` ★
`\int_min:nn` ★

Updated: 2012-09-26

`\int_max:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`
`\int_min:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

Evaluates the ⟨*integer expressions*⟩ as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_mod:nn` ★

Updated: 2012-09-26

`\int_mod:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}` times ⟨*intexpr*$_2$⟩ from ⟨*intexpr*$_1$⟩. Thus, the result has the same sign as ⟨*intexpr*$_1$⟩ and its absolute value is strictly less than that of ⟨*intexpr*$_2$⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

`\int_new:N` ⟨*integer*⟩

Creates a new ⟨*integer*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*integer*⟩ is initially equal to 0.

`\int_const:Nn`
`\int_const:cn`

Updated: 2011-10-22

`\int_const:Nn` ⟨*integer*⟩ `{`⟨*integer expression*⟩`}`

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ is set globally to the ⟨*integer expression*⟩.

`\int_zero:N`
`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

`\int_zero:N` ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0.

`\int_zero_new:N`
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

New: 2011-12-13

`\int_zero_new:N` ⟨*integer*⟩

Ensures that the ⟨*integer*⟩ exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the ⟨*integer*⟩ set to zero.

`\int_set_eq:NN`
`\int_set_eq:(cN|Nc|cc)`
`\int_gset_eq:NN`
`\int_gset_eq:(cN|Nc|cc)`

`\int_set_eq:NN` ⟨*integer*$_1$⟩ ⟨*integer*$_2$⟩

Sets the content of ⟨*integer*$_1$⟩ equal to that of ⟨*integer*$_2$⟩.

`\int_if_exist_p:N` ★
`\int_if_exist_p:c` ★
`\int_if_exist:NTF` ★
`\int_if_exist:cTF` ★

New: 2012-03-03

`\int_if_exist_p:N ⟨int⟩`
`\int_if_exist:NTF ⟨int⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*int*⟩ is currently defined. This does not check that the ⟨*int*⟩ really is an integer variable.

3 Setting and incrementing integers

`\int_add:Nn`
`\int_add:cn`
`\int_gadd:Nn`
`\int_gadd:cn`

Updated: 2011-10-22

`\int_add:Nn ⟨integer⟩ {⟨integer expression⟩}`

Adds the result of the ⟨*integer expression*⟩ to the current content of the ⟨*integer*⟩.

`\int_decr:N`
`\int_decr:c`
`\int_gdecr:N`
`\int_gdecr:c`

`\int_decr:N ⟨integer⟩`

Decreases the value stored in ⟨*integer*⟩ by 1.

`\int_incr:N`
`\int_incr:c`
`\int_gincr:N`
`\int_gincr:c`

`\int_incr:N ⟨integer⟩`

Increases the value stored in ⟨*integer*⟩ by 1.

`\int_set:Nn`
`\int_set:cn`
`\int_gset:Nn`
`\int_gset:cn`

Updated: 2011-10-22

`\int_set:Nn ⟨integer⟩ {⟨integer expression⟩}`

Sets ⟨*integer*⟩ to the value of ⟨*integer expression*⟩, which must evaluate to an integer (as described for `\int_eval:n`).

`\int_sub:Nn`
`\int_sub:cn`
`\int_gsub:Nn`
`\int_gsub:cn`

Updated: 2011-10-22

`\int_sub:Nn ⟨integer⟩ {⟨integer expression⟩}`

Subtracts the result of the ⟨*integer expression*⟩ from the current content of the ⟨*integer*⟩.

4 Using integers

`\int_use:N` ★
`\int_use:c` ★

Updated: 2011-10-22

`\int_use:N ⟨integer⟩`

Recovers the content of an ⟨*integer*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an ⟨*integer*⟩ is required (such as in the first and third arguments of `\int_compare:nNnTF`).

TEXhackers note: `\int_use:N` is the TEX primitive `\the`: this is one of several LATEX3 names for this primitive.

5 Integer expression conditionals

\int_compare_p:nNn ★
\int_compare:nNnTF ★

```
\int_compare_p:nNn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩}
\int_compare:nNnTF
  {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*integer expressions*⟩ as described for `\int_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	`=`
Greater than	`>`
Less than	`<`

\int_compare_p:n ★
\int_compare:nTF ★

Updated: 2013-01-13

```
\int_compare_p:n
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+1⟩
  }
\int_compare:nTF
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*integer expressions*⟩ as described for `\int_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*intexpr*$_1$⟩ and ⟨*intexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*intexpr*$_2$⟩ and ⟨*intexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*intexpr*$_N$⟩ and ⟨*intexpr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*integer expression*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*integer expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	`=` or `==`
Greater than or equal to	`>=`
Greater than	`>`
Less than or equal to	`<=`
Less than	`<`
Not equal	`!=`

\int_case:nn ★
\int_case:nnTF ★

New: 2013-07-24

```
\int_case:nnTF {⟨test integer expression⟩}
  {
    {⟨intexpr case_1⟩} {⟨code case_1⟩}
    {⟨intexpr case_2⟩} {⟨code case_2⟩}
    ...
    {⟨intexpr case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
  { 2 * 5 }
  {
    { 5 }       { Small }
    { 4 + 6 }   { Medium }
    { -2 * 10 } { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream.

\int_if_even_p:n ★
\int_if_even:nTF ★
\int_if_odd_p:n ★
\int_if_odd:nTF ★

```
\int_if_odd_p:n {⟨integer expression⟩}
\int_if_odd:nTF {⟨integer expression⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*integer expression*⟩ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

\int_do_until:nNnn ☆

```
\int_do_until:nNnn {⟨intexpr_1⟩} ⟨relation⟩ {⟨intexpr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

\int_do_while:nNnn ☆

```
\int_do_while:nNnn {⟨intexpr_1⟩} ⟨relation⟩ {⟨intexpr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

\int_until_do:nNnn ☆

`\int_until_do:nNnn {⟨intexpr_1⟩} ⟨relation⟩ {⟨intexpr_2⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

\int_while_do:nNnn ☆

`\int_while_do:nNnn {⟨intexpr_1⟩} ⟨relation⟩ {⟨intexpr_2⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

\int_do_until:nn ☆

Updated: 2013-01-13

`\int_do_until:nn {⟨integer relation⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

\int_do_while:nn ☆

Updated: 2013-01-13

`\int_do_while:nn {⟨integer relation⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

\int_until_do:nn ☆

Updated: 2013-01-13

`\int_until_do:nn {⟨integer relation⟩} {⟨code⟩}`

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

\int_while_do:nn ☆

Updated: 2013-01-13

`\int_while_do:nn {⟨integer relation⟩} {⟨code⟩}`

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

7 Integer step functions

`\int_step_function:nnnN` ☆

New: 2012-06-04
Updated: 2014-05-30

`\int_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

[I saw 1]   [I saw 2]   [I saw 3]   [I saw 4]   [I saw 5]

`\int_step_inline:nnnn`

New: 2012-06-04
Updated: 2014-05-30

`\int_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`).

`\int_step_variable:nnnNn`

New: 2012-06-04
Updated: 2014-05-30

```
\int_step_variable:nnnNn
  {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}
```

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

`\int_to_arabic:n` ★

Updated: 2011-10-22

`\int_to_arabic:n {⟨integer expression⟩}`

Places the value of the ⟨*integer expression*⟩ in the input stream as digits, with category code 12 (other).

\int_to_alph:n ⋆
\int_to_Alph:n ⋆

Updated: 2011-09-17

\int_to_alph:n {⟨*integer expression*⟩}

Evaluates the ⟨*integer expression*⟩ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alph:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_symbols:nnn ⋆

Updated: 2011-09-17

\int_to_symbols:nnn
  {⟨*integer expression*⟩} {⟨*total symbols*⟩}
  ⟨*value to symbol mapping*⟩

This is the low-level function for conversion of an ⟨*integer expression*⟩ into a symbolic form (often letters). The ⟨*total symbols*⟩ available should be given as an integer expression. Values are actually converted to symbols according to the ⟨*value to symbol mapping*⟩. This should be given as ⟨*total symbols*⟩ pairs of entries, a number and the appropriate symbol. Thus the \int_to_alph:n function is defined as

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        ...
        { 26 } { z }
      }
  }
```

\int_to_bin:n ⋆

New: 2014-02-11

\int_to_bin:n {⟨*integer expression*⟩}

Calculates the value of the ⟨*integer expression*⟩ and places the binary representation of the result in the input stream.

\int_to_hex:n ★
\int_to_Hex:n ★

New: 2014-02-11

`\int_to_hex:n {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_oct:n ★

New: 2014-02-11

`\int_to_oct:n {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_base:nn ★
\int_to_Base:nn ★

Updated: 2014-02-11

`\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}`

Calculates the value of the ⟨*integer expression*⟩ and converts it into the appropriate representation in the ⟨*base*⟩; the later may be given as an integer expression. For bases greater than 10 the higher "digits" are represented by letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum ⟨*base*⟩ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

> **TEXhackers note:** This is a generic version of `\int_to_bin:n`, *etc.*

\int_to_roman:n ☆
\int_to_Roman:n ☆

Updated: 2011-10-22

`\int_to_roman:n {⟨integer expression⟩}`

Places the value of the ⟨*integer expression*⟩ in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). The Roman numerals are letters with category code 11 (letter).

9 Converting from other formats to integers

\int_from_alph:n ★

Updated: 2014-08-25

`\int_from_alph:n {⟨letters⟩}`

Converts the ⟨*letters*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*letters*⟩ are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with "a" equal to 1 through to "z" equal to 26. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_alph:n` and `\int_to_Alph:n`.

\int_from_bin:n ★

New: 2014-02-11
Updated: 2014-08-25

`\int_from_bin:n {⟨binary number⟩}`

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*binary number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

`\int_from_hex:n` ★

New: 2014-02-11
Updated: 2014-08-25

`\int_from_hex:n {`⟨*hexadecimal number*⟩`}`

Converts the ⟨*hexadecimal number*⟩ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the ⟨*hexadecimal number*⟩ by upper or lower case letters. The ⟨*hexadecimal number*⟩ is first converted to a string, with no expansion. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

`\int_from_oct:n` ★

New: 2014-02-11
Updated: 2014-08-25

`\int_from_oct:n {`⟨*octal number*⟩`}`

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*octal number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

`\int_from_roman:n` ★

Updated: 2014-08-25

`\int_from_roman:n {`⟨*roman numeral*⟩`}`

Converts the ⟨*roman numeral*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*roman numeral*⟩ is first converted to a string, with no expansion. The ⟨*roman numeral*⟩ may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value is -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ★

Updated: 2014-08-25

`\int_from_base:nn {`⟨*number*⟩`} {`⟨*base*⟩`}`

Converts the ⟨*number*⟩ expressed in ⟨*base*⟩ into the appropriate value in base 10. The ⟨*number*⟩ is first converted to a string, with no expansion. The ⟨*number*⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨*base*⟩ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

10 Viewing integers

`\int_show:N`
`\int_show:c`

`\int_show:N` ⟨*integer*⟩

Displays the value of the ⟨*integer*⟩ on the terminal.

`\int_show:n`

New: 2011-11-22
Updated: 2015-08-07

`\int_show:n {`⟨*integer expression*⟩`}`

Displays the result of evaluating the ⟨*integer expression*⟩ on the terminal.

`\int_log:N`
`\int_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\int_log:N` ⟨*integer*⟩

Writes the value of the ⟨*integer*⟩ in the log file.

`\int_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\int_log:n {`⟨*integer expression*⟩`}`

Writes the result of evaluating the ⟨*integer expression*⟩ in the log file.

11 Constant integers

```
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand
```

Integer values used with primitive tests and assignments: self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_int`

The maximum value that can be stored as an integer.

`\c_max_register_int`

Maximum number of registers.

`\c_max_char_int`

Maximum character code completely supported by the engine.

12 Scratch integers

```
\l_tmpa_int
\l_tmpb_int
```

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

```
\g_tmpa_int
\g_tmpb_int
```

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

13 Primitive conditionals

\if_int_compare:w ★

```
\if_int_compare:w ⟨integer1⟩ ⟨relation⟩ ⟨integer2⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Compare two integers using ⟨*relation*⟩, which must be one of =, < or > with category code 12. The \else: branch is optional.

> **TEXhackers note:** These are both names for the TEX primitive \ifnum.

\if_case:w ★
\or: ★

```
\if_case:w ⟨integer⟩ ⟨case0⟩
  \or: ⟨case1⟩
  \or: ...
  \else: ⟨default⟩
\fi:
```

Selects a case to execute based on the value of the ⟨*integer*⟩. The first case (⟨*case*$_0$⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨*case*$_1$⟩) if the ⟨*integer*⟩ is 1, *etc.* The ⟨*integer*⟩ may be a literal, a constant or an integer expression (*e.g.* using \int_eval:n).

> **TEXhackers note:** These are the TEX primitives \ifcase and \or.

\if_int_odd:w ★

```
\if_int_odd:w ⟨tokens⟩  ⟨optional space⟩
  ⟨true code⟩
\else:
  ⟨true code⟩
\fi:
```

Expands ⟨*tokens*⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨*integer*⟩ is odd. If so, ⟨*true code*⟩ is executed. The \else: branch is optional.

> **TEXhackers note:** This is the TEX primitive \ifodd.

14 Internal functions

__int_to_roman:w ★

```
\__int_to_roman:w ⟨integer⟩ ⟨space⟩ or ⟨non-expandable token⟩
```

Converts ⟨*integer*⟩ to it lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *are* expanded by this process. Negative ⟨*integer*⟩ values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

> **TEXhackers note:** This is the TEX primitive \romannumeral renamed.

`\__int_value:w` ★

`\__int_value:w` ⟨*integer*⟩
`\__int_value:w` ⟨*tokens*⟩ ⟨*optional space*⟩

Expands ⟨*tokens*⟩ until an ⟨*integer*⟩ is formed. One space may be gobbled in the process.

> **TEXhackers note:** This is the TEX primitive `\number`.

`\__int_eval:w` ★
`\__int_eval_end:` ★

`\__int_eval:w` ⟨*intexpr*⟩ `\__int_eval_end:`

Evaluates ⟨*integer expression*⟩ as described for `\int_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when `\__int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\__int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

> **TEXhackers note:** This is the ε-TEX primitive `\numexpr`.

`\__prg_compare_error:`
`\__prg_compare_error:Nw`

`\__prg_compare_error:`
`\__prg_compare_error:Nw` ⟨*token*⟩

These are used within `\int_compare:nTF`, `\dim_compare:nTF` and so on to recover correctly if the `n`-type argument does not contain a properly-formed relation.

Part X
The l3intarray package: low-level arrays of small integers

1 l3intarray documentation

This module provides no user function: at present it is meant for kernel use only.

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to l3seq sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the l3intarray package transparently converts these from/to integers. Assignments are always global.

While LuaTEX's memory is extensible, other engines can "only" deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TEXLive settings).

1.1 Internal functions

`\__intarray_new:Nn`

`\__intarray_new:Nn ⟨intarray var⟩ {⟨size⟩}`

Evaluates the integer expression ⟨*size*⟩ and allocates an ⟨*integer array variable*⟩ with that number of (zero) entries.

`\__intarray_count:N` ⋆

`\__intarray_count:N ⟨intarray var⟩`

Expands to the number of entries in the ⟨*integer array variable*⟩. Contrarily to `\seq_count:N` this is performed in constant time.

`\__intarray_gset:Nnn`
`\__intarray_gset_fast:Nnn`

`\__intarray_gset:Nnn ⟨intarray var⟩ {⟨position⟩} {⟨value⟩}`
`\__intarray_gset_fast:Nnn ⟨intarray var⟩ {⟨position⟩} {⟨value⟩}`

Stores the result of evaluating the integer expression ⟨*value*⟩ into the ⟨*integer array variable*⟩ at the (integer expression) ⟨*position*⟩. While `\__intarray_gset:Nnn` checks that the ⟨*position*⟩ is between 1 and the `\__intarray_count:N` and that the ⟨*value*⟩'s absolute value is at most $2^{30} - 1$, the "fast" function performs no such bound check. Assignments are always global.

`\__intarray_item:Nn` ⋆
`\__intarray_item_fast:Nn` ⋆

`\__intarray_item:Nn ⟨intarray var⟩ {⟨position⟩}`
`\__intarray_item_fast:Nn ⟨intarray var⟩ {⟨position⟩}`

Expands to the integer entry stored at the (integer expression) ⟨*position*⟩ in the ⟨*integer array variable*⟩. While `\__intarray_item:Nn` checks that the ⟨*position*⟩ is between 1 and the `\__intarray_count:N`, the "fast" function performs no such bound check.

Part XI
The l3flag package: expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its ⟨*height*⟩. In expansion-only contexts, a flag can only be "raised": this increases the ⟨*height*⟩ by 1. The ⟨*height*⟩ can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a ⟨*flag name*⟩ such as `str_missing`. The ⟨*flag name*⟩ is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occured during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by l3str-convert, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

1 Setting up flags

`\flag_new:n`

`\flag_new:n {⟨flag name⟩}`

Creates a new flag with a name given by ⟨*flag name*⟩, or raises an error if the name is already taken. The ⟨*flag name*⟩ may not contain spaces. The declaration is global, but flags are always local variables. The ⟨*flag*⟩ initially has zero height.

`\flag_clear:n`

`\flag_clear:n {⟨flag name⟩}`

The ⟨*flag*⟩'s height is set to zero. The assignment is local.

`\flag_clear_new:n`

`\flag_clear_new:n {⟨flag name⟩}`

Ensures that the ⟨*flag*⟩ exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

`\flag_show:n`

`\flag_show:n {⟨flag name⟩}`

Displays the ⟨*flag*⟩'s height in the terminal.

`\flag_log:n`

`\flag_log:n {⟨flag name⟩}`

Writes the ⟨*flag*⟩'s height to the log file.

2 Expandable flag commands

`\flag_if_exist_p:n` ⋆
`\flag_if_exist:nTF` ⋆

`\flag_if_exist:n {⟨flag name⟩}`

This function returns `true` if the ⟨*flag name*⟩ references a flag that has been defined previously, and `false` otherwise.

`\flag_if_raised_p:n` ⋆
`\flag_if_raised:nTF` ⋆

`\flag_if_raised:n {⟨flag name⟩}`

This function returns `true` if the ⟨*flag*⟩ has non-zero height, and `false` if the ⟨*flag*⟩ has zero height.

`\flag_height:n` ⋆

`\flag_height:n {⟨flag name⟩}`

Expands to the height of the ⟨*flag*⟩ as an integer denotation.

`\flag_raise:n` ⋆

`\flag_raise:n {⟨flag name⟩}`

The ⟨*flag*⟩'s height is increased by 1 locally.

Part XII
The **l3quark** package Quarks

1 Introduction to quarks and scan marks

Two special types of constants in LaTeX3 are "quarks" and "scan marks". By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, with the most command use case as the 'stop token' (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
  { <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N`

`\quark_new:N` ⟨*quark*⟩

Creates a new ⟨*quark*⟩ which expands only to ⟨*quark*⟩. The ⟨*quark*⟩ is defined globally, and an error message is raised if the name was already taken.

\q_stop

Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

\q_mark

Used as a marker for delimited arguments when \q_stop is already in use.

\q_nil

Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to \q_stop, which is only ever used as a delimiter).

\q_no_value

A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a "return" value by functions such as \prop_get:NnN if there is no data to return.

3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The later should therefore only be used when the argument can definitely take more than a single token.

\quark_if_nil_p:N ★
\quark_if_nil:NTF ★

```
\quark_if_nil_p:N ⟨token⟩
\quark_if_nil:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is equal to \q_nil.

\quark_if_nil_p:n ★
\quark_if_nil_p:(o|V) ★
\quark_if_nil:nTF ★
\quark_if_nil:(o|V)TF ★

```
\quark_if_nil_p:n {⟨token list⟩}
\quark_if_nil:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token list*⟩ contains only \q_nil (distinct from ⟨*token list*⟩ being empty or containing \q_nil plus one or more other tokens).

\quark_if_no_value_p:N ★
\quark_if_no_value_p:c ★
\quark_if_no_value:NTF ★
\quark_if_no_value:cTF ★

```
\quark_if_no_value_p:N ⟨token⟩
\quark_if_no_value:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is equal to \q_no_value.

\quark_if_no_value_p:n ★
\quark_if_no_value:nTF ★

```
\quark_if_no_value_p:n {⟨token list⟩}
\quark_if_no_value:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token list*⟩ contains only \q_no_value (distinct from ⟨*token list*⟩ being empty or containing \q_no_value plus one or more other tokens).

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 5.

`\q_recursion_tail`

This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop`

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N`

`\quark_if_recursion_tail_stop:N ⟨token⟩`

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`

Updated: 2011-09-06

`\quark_if_recursion_tail_stop:n {⟨token list⟩}`

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn`

`\quark_if_recursion_tail_stop_do:Nn ⟨token⟩ {⟨insertion⟩}`

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:on`

Updated: 2011-09-06

`\quark_if_recursion_tail_stop_do:nn {⟨token list⟩} {⟨insertion⟩}`

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

5 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce "`[-a-b-] [-c-d-] `". Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
 {
   \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
   \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
   \q_recursion_stop
 }
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
 {
   \quark_if_recursion_tail_stop:n {#1}
   \quark_if_recursion_tail_stop:n {#2}
   \__my_map_dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
   \__my_map_dbl:nn
 }
```

Note that contrarily to LaTeX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `\__my_map_dbl_fn:nn`.

6 Internal quark functions

```
\__quark_if_recursion_tail_break:NN   \__quark_if_recursion_tail_break:nN {⟨token list⟩}
\__quark_if_recursion_tail_break:nN   \⟨type⟩_map_break:
```

Tests if ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so terminates the recursion using `\⟨type⟩_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \⟨type⟩_map_break:`.

7 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

`\__scan_new:N`

`\__scan_new:N` ⟨*scan mark*⟩

Creates a new ⟨*scan mark*⟩ which is set equal to `\scan_stop:`. The ⟨*scan mark*⟩ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s__stop`

Used at the end of a set of instructions, as a marker that can be jumped to using `\__use_none_delimit_by_s__stop:w`.

`\__use_none_delimit_by_s__stop:w`

`\__use_none_delimit_by_s__stop:w` ⟨*tokens*⟩ `\s__stop`

Removes the ⟨*tokens*⟩ and `\s__stop` from the input stream. This leads to a low-level TeX error if `\s__stop` is absent.

Part XIII
The **l3prg** package Control structures

Conditional processing in LaTeX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are ⟨*true*⟩ and ⟨*false*⟩.

LaTeX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean ⟨*true*⟩ or ⟨*false*⟩. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean ⟨*true*⟩ or ⟨*false*⟩ values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result.

TEXhackers note: The arguments are executed after exiting the underlying `\if...\fi:` structure.

1 Defining a set of conditional functions

`\prg_new_conditional:Npnn`
`\prg_set_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\prg_set_conditional:Nnn`

Updated: 2012-02-06

`\prg_new_conditional:Npnn \`⟨*name*⟩`:`⟨*arg spec*⟩ ⟨*parameters*⟩ `{`⟨*conditions*⟩`} {`⟨*code*⟩`}`
`\prg_new_conditional:Nnn \`⟨*name*⟩`:`⟨*arg spec*⟩ `{`⟨*conditions*⟩`} {`⟨*code*⟩`}`

These functions create a family of conditionals using the same `{`⟨*code*⟩`}` to perform the test created. Those conditionals are expandable if ⟨*code*⟩ is. The `new` versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

`\prg_new_protected_conditional:Npnn`
`\prg_set_protected_conditional:Npnn`
`\prg_new_protected_conditional:Nnn`
`\prg_set_protected_conditional:Nnn`

Updated: 2012-02-06

`\prg_new_protected_conditional:Npnn \`⟨*name*⟩`:`⟨*arg spec*⟩ ⟨*parameters*⟩ `{`⟨*conditions*⟩`} {`⟨*code*⟩`}`
`\prg_new_protected_conditional:Nnn \`⟨*name*⟩`:`⟨*arg spec*⟩ `{`⟨*conditions*⟩`} {`⟨*code*⟩`}`

These functions create a family of protected conditionals using the same `{`⟨*code*⟩`}` to perform the test created. The ⟨*code*⟩ does not need to be expandable. The `new` version check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version do not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `T`, `F` and `TF` (not `p`).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\⟨name⟩_p:⟨arg spec⟩` — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for `protected` conditionals.
- `\⟨name⟩:⟨arg spec⟩T` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in this additional argument will be left on the input stream only if the test is `true`.
- `\⟨name⟩:⟨arg spec⟩F` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*false branch*⟩ code in this additional argument will be left on the input stream only if the test is `false`.
- `\⟨name⟩:⟨arg spec⟩TF` — a function with two more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in the first additional argument will be left on the input stream if the test is `true`, while the ⟨*false branch*⟩ code in the second argument will be left on the input stream if the test is `false`.

The ⟨*code*⟩ of the test may use ⟨*parameters*⟩ as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the ⟨*argument specification*⟩ but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (*cf.* `\cs_new:Nn`, *etc.*). Within the ⟨*code*⟩, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
  {
    \if_meaning:w \l_tmpa_tl #1
      \prg_return_true:
    \else:
      \if_meaning:w \l_tmpa_tl #2
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the ⟨*conditions*⟩ list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

`\prg_new_eq_conditional:NNn`
`\prg_set_eq_conditional:NNn`

`\prg_new_eq_conditional:NNn \⟨name₁⟩:⟨arg spec₁⟩ \⟨name₂⟩:⟨arg spec₂⟩ {⟨conditions⟩}`

These functions copy a family of conditionals. The `new` version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the `set` version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

`\prg_return_true:` ★
`\prg_return_false:` ★

`\prg_return_true:`
`\prg_return_false:`

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an `f`-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

TEXhackers note: The `bool` data type is not implemented using the `\iffalse`/`\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain TEX, LATEX 2ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

`\bool_new:N`
`\bool_new:c`

`\bool_new:N` ⟨*boolean*⟩

Creates a new ⟨*boolean*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*boolean*⟩ is initially `false`.

`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

`\bool_set_false:N` ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `false`.

`\bool_set_true:N`
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`

`\bool_set_true:N` ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `true`.

\bool_set_eq:NN
\bool_set_eq:(cN|Nc|cc)
\bool_gset_eq:NN
\bool_gset_eq:(cN|Nc|cc)

\bool_set_eq:NN ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩

Sets ⟨*boolean*$_1$⟩ to the current value of ⟨*boolean*$_2$⟩.

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

Updated: 2017-07-15

\bool_set:Nn ⟨*boolean*⟩ {⟨*boolexpr*⟩}

Evaluates the ⟨*boolean expression*⟩ as described for \bool_if:nTF, and sets the ⟨*boolean*⟩ variable to the logical truth of this evaluation.

\bool_if_p:N ★
\bool_if_p:c ★
\bool_if:N*TF* ★
\bool_if:c*TF* ★

Updated: 2017-07-15

\bool_if_p:N ⟨*boolean*⟩
\bool_if:NTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests the current truth of ⟨*boolean*⟩, and continues expansion based on this result.

\bool_show:N
\bool_show:c

New: 2012-02-09
Updated: 2015-08-01

\bool_show:N ⟨*boolean*⟩

Displays the logical truth of the ⟨*boolean*⟩ on the terminal.

\bool_show:n

New: 2012-02-09
Updated: 2017-07-15

\bool_show:n {⟨*boolean expression*⟩}

Displays the logical truth of the ⟨*boolean expression*⟩ on the terminal.

\bool_log:N
\bool_log:c

New: 2014-08-22
Updated: 2015-08-03

\bool_log:N ⟨*boolean*⟩

Writes the logical truth of the ⟨*boolean*⟩ in the log file.

\bool_log:n

New: 2014-08-22
Updated: 2017-07-15

\bool_log:n {⟨*boolean expression*⟩}

Writes the logical truth of the ⟨*boolean expression*⟩ in the log file.

\bool_if_exist_p:N ★
\bool_if_exist_p:c ★
\bool_if_exist:N*TF* ★
\bool_if_exist:c*TF* ★

New: 2012-03-03

\bool_if_exist_p:N ⟨*boolean*⟩
\bool_if_exist:NTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*boolean*⟩ is currently defined. This does not check that the ⟨*boolean*⟩ really is a boolean variable.

\l_tmpa_bool
\l_tmpb_bool

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_bool
\g_tmpb_bool

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean ⟨*true*⟩ or ⟨*false*⟩ values, it seems only fitting that we also provide a parser for ⟨*boolean expressions*⟩.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean ⟨*true*⟩ or ⟨*false*⟩. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
  (
    \int_compare_p:n { 2 = 3 } ||
    \int_compare_p:n { 4 <= 4 } ||
    \str_if_eq_p:nn { abc } { def }
  ) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This "eager" evaluation should be contrasted with the "lazy" evaluation of `\bool_lazy_...` functions.

TeXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in TeX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
  {
    \bool_lazy_any_p:n
      {
        { \int_compare_p:n { 2 = 3 } }
        { \int_compare_p:n { 4 <= 4 } }
        { \int_compare_p:n { 1 = \error } } % skipped
      }
  }
  { ! \int_compare_p:n { 2 = 4 } }
```

the line marked with skipped is not expanded because the result of \bool_lazy_any_p:n is known once the second boolean expression is found to be logically true. On the other hand, the last line is expanded because its logical value is needed to determine the result of \bool_lazy_and_p:nn.

\bool_if_p:n ★
\bool_if:n*TF* ★
Updated: 2017-07-15

\bool_if_p:n {⟨*boolean expression*⟩}
\bool_if:nTF {⟨*boolean expression*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests the current truth of ⟨*boolean expression*⟩, and continues expansion based on this result. The ⟨*boolean expression*⟩ should consist of a series of predicates or boolean variables with the logical relationship between these defined using && ("And"), || ("Or"), ! ("Not") and parentheses. The logical Not applies to the next predicate or group.

\bool_lazy_all_p:n ★
\bool_lazy_all:n*TF* ★
New: 2015-11-15
Updated: 2017-07-15

\bool_lazy_all_p:n { {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} ··· {⟨*boolexpr*$_N$⟩} }
\bool_lazy_all:nTF { {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} ··· {⟨*boolexpr*$_N$⟩} } {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "And" operation on the ⟨*boolean expressions*⟩, hence is true if all of them are true and false if any of them is false. Contrarily to the infix operator &&, only the ⟨*boolean expressions*⟩ which are needed to determine the result of \bool_lazy_all:nTF are evaluated. See also \bool_lazy_and:nnTF when there are only two ⟨*boolean expressions*⟩.

\bool_lazy_and_p:nn ★
\bool_lazy_and:nn*TF* ★
New: 2015-11-15
Updated: 2017-07-15

\bool_lazy_and_p:nn {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩}
\bool_lazy_and:nnTF {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "And" operation between two boolean expressions, hence is true if both are true. Contrarily to the infix operator &&, the ⟨*boolexpr*$_2$⟩ is only evaluated if it is needed to determine the result of \bool_lazy_and:nnTF. See also \bool_lazy_all:nTF when there are more than two ⟨*boolean expressions*⟩.

\bool_lazy_any_p:n ★
\bool_lazy_any:n*TF* ★
New: 2015-11-15
Updated: 2017-07-15

\bool_lazy_any_p:n { {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} ··· {⟨*boolexpr*$_N$⟩} }
\bool_lazy_any:nTF { {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} ··· {⟨*boolexpr*$_N$⟩} } {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "Or" operation on the ⟨*boolean expressions*⟩, hence is true if any of them is true and false if all of them are false. Contrarily to the infix operator ||, only the ⟨*boolean expressions*⟩ which are needed to determine the result of \bool_lazy_any:nTF are evaluated. See also \bool_lazy_or:nnTF when there are only two ⟨*boolean expressions*⟩.

\bool_lazy_or_p:nn ★
\bool_lazy_or:nn*TF* ★
New: 2015-11-15
Updated: 2017-07-15

\bool_lazy_or_p:nn {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩}
\bool_lazy_or:nnTF {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "Or" operation between two boolean expressions, hence is true if either one is true. Contrarily to the infix operator ||, the ⟨*boolexpr*$_2$⟩ is only evaluated if it is needed to determine the result of \bool_lazy_or:nnTF. See also \bool_lazy_any:nTF when there are more than two ⟨*boolean expressions*⟩.

\bool_not_p:n ★
Updated: 2017-07-15

\bool_not_p:n {⟨*boolean expression*⟩}

Function version of !(⟨*boolean expression*⟩) within a boolean expression.

\bool_xor_p:nn ★

Updated: 2017-07-15

\bool_xor_p:nn {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩}

Implements an "exclusive or" operation between two boolean expressions. There is no infix operation for this logical operator.

4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn ☆
\bool_do_until:cn ☆

Updated: 2017-07-15

\bool_do_until:Nn ⟨*boolean*⟩ {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean*⟩. If it is `false` then the ⟨*code*⟩ is inserted into the input stream again and the process loops until the ⟨*boolean*⟩ is `true`.

\bool_do_while:Nn ☆
\bool_do_while:cn ☆

Updated: 2017-07-15

\bool_do_while:Nn ⟨*boolean*⟩ {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean*⟩. If it is `true` then the ⟨*code*⟩ is inserted into the input stream again and the process loops until the ⟨*boolean*⟩ is `false`.

\bool_until_do:Nn ☆
\bool_until_do:cn ☆

Updated: 2017-07-15

\bool_until_do:Nn ⟨*boolean*⟩ {⟨*code*⟩}

This function firsts checks the logical value of the ⟨*boolean*⟩. If it is `false` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean*⟩ is re-evaluated. The process then loops until the ⟨*boolean*⟩ is `true`.

\bool_while_do:Nn ☆
\bool_while_do:cn ☆

Updated: 2017-07-15

\bool_while_do:Nn ⟨*boolean*⟩ {⟨*code*⟩}

This function firsts checks the logical value of the ⟨*boolean*⟩. If it is `true` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean*⟩ is re-evaluated. The process then loops until the ⟨*boolean*⟩ is `false`.

\bool_do_until:nn ☆

Updated: 2017-07-15

\bool_do_until:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean expression*⟩ as described for `\bool_if:nTF`. If it is `false` then the ⟨*code*⟩ is inserted into the input stream again and the process loops until the ⟨*boolean expression*⟩ evaluates to `true`.

\bool_do_while:nn ☆

Updated: 2017-07-15

\bool_do_while:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean expression*⟩ as described for `\bool_if:nTF`. If it is `true` then the ⟨*code*⟩ is inserted into the input stream again and the process loops until the ⟨*boolean expression*⟩ evaluates to `false`.

\bool_until_do:nn ☆

Updated: 2017-07-15

\bool_until_do:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

This function firsts checks the logical value of the ⟨*boolean expression*⟩ (as described for `\bool_if:nTF`). If it is `false` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean expression*⟩ is re-evaluated. The process then loops until the ⟨*boolean expression*⟩ is `true`.

`\bool_while_do:nn` ☆

Updated: 2017-07-15

```
\bool_while_do:nn {⟨boolean expression⟩} {⟨code⟩}
```

This function firsts checks the logical value of the ⟨*boolean expression*⟩ (as described for `\bool_if:nTF`). If it is `true` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean expression*⟩ is re-evaluated. The process then loops until the ⟨*boolean expression*⟩ is `false`.

5 Producing multiple copies

`\prg_replicate:nn` ★

Updated: 2011-07-04

```
\prg_replicate:nn {⟨integer expression⟩} {⟨tokens⟩}
```

Evaluates the ⟨*integer expression*⟩ (which should be zero or positive) and creates the resulting number of copies of the ⟨*tokens*⟩. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting TeX's mode

`\mode_if_horizontal_p:` ★
`\mode_if_horizontal:TF` ★

```
\mode_if_horizontal_p:
\mode_if_horizontal:TF {⟨true code⟩} {⟨false code⟩}
```

Detects if TeX is currently in horizontal mode.

`\mode_if_inner_p:` ★
`\mode_if_inner:TF` ★

```
\mode_if_inner_p:
\mode_if_inner:TF {⟨true code⟩} {⟨false code⟩}
```

Detects if TeX is currently in inner mode.

`\mode_if_math_p:` ★
`\mode_if_math:TF` ★

Updated: 2011-09-05

```
\mode_if_math:TF {⟨true code⟩} {⟨false code⟩}
```

Detects if TeX is currently in maths mode.

`\mode_if_vertical_p:` ★
`\mode_if_vertical:TF` ★

```
\mode_if_vertical_p:
\mode_if_vertical:TF {⟨true code⟩} {⟨false code⟩}
```

Detects if TeX is currently in vertical mode.

7 Primitive conditionals

`\if_predicate:w` ★

```
\if_predicate:w ⟨predicate⟩ ⟨true code⟩ \else: ⟨false code⟩ \fi:
```

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the ⟨*predicate*⟩ but to make the coding clearer this should be done through `\if_bool:N`.)

`\if_bool:N` ★

```
\if_bool:N ⟨boolean⟩ ⟨true code⟩ \else: ⟨false code⟩ \fi:
```

This function takes a boolean variable and branches according to the result.

8 Internal programming functions

\group_align_safe_begin: ★
\group_align_safe_end: ★

Updated: 2011-08-11

```
\group_align_safe_begin:
...
\group_align_safe_end:
```

These functions are used to enclose material in a TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the & token inside \halign. This is necessary to allow grabbing of tokens for testing purposes, as TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as \peek_after:Nw would result in a forbidden comparison of the internal \endtemplate token, yielding a fatal error. Each \group_align_safe_begin: must be matched by a \group_align_safe_end:, although this does not have to occur within the same function.

__prg_break_point:Nn ★

```
\__prg_break_point:Nn \⟨type⟩_map_break: ⟨tokens⟩
```

Used to mark the end of a recursion or mapping: the functions \⟨*type*⟩_map_break: and \⟨*type*⟩_map_break:n use this to break out of the loop. After the loop ends, the ⟨*tokens*⟩ are inserted into the input stream. This occurs even if the break functions are *not* applied: __prg_break_point:Nn is functionally-equivalent in these cases to \use_ii:nn.

__prg_map_break:Nn ★

```
\__prg_map_break:Nn \⟨type⟩_map_break: {⟨user code⟩}
...
\__prg_break_point:Nn \⟨type⟩_map_break: {⟨ending code⟩}
```

Breaks a recursion in mapping contexts, inserting in the input stream the ⟨*user code*⟩ after the ⟨*ending code*⟩ for the loop. The function breaks loops, inserting their ⟨*ending code*⟩, until reaching a loop with the same ⟨*type*⟩ as its first argument. This \⟨*type*⟩_map_break: argument is simply used as a recognizable marker for the ⟨*type*⟩.

\g__prg_map_int

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions __prg_map_1:w, __prg_map_2:w, *etc.*, labelled by \g__prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.

__prg_break_point: ★

This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursion: the function __prg_break:n uses this to break out of the loop.

__prg_break: ★
__prg_break:n ★

```
\__prg_break:n {⟨tokens⟩} ... \__prg_break_point:
```

Breaks a recursion which has no ⟨*ending code*⟩ and which is not a user-breakable mapping (see for instance \prop_get:Nn), and inserts ⟨*tokens*⟩ in the input stream.

Part XIV
The **l3clist** package
Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

leaves `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual TEX category codes apply).

1 Creating and initialising comma lists

`\clist_new:N`
`\clist_new:c`

`\clist_new:N` ⟨*comma list*⟩

Creates a new ⟨*comma list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*comma list*⟩ initially contains no items.

`\clist_const:Nn`
`\clist_const:(Nx|cn|cx)`
New: 2014-07-05

`\clist_const:Nn` ⟨*clist var*⟩ `{`⟨*comma list*⟩`}`

Creates a new constant ⟨*clist var*⟩ or raises an error if the name is already taken. The value of the ⟨*clist var*⟩ is set globally to the ⟨*comma list*⟩.

`\clist_clear:N`
`\clist_clear:c`
`\clist_gclear:N`
`\clist_gclear:c`

`\clist_clear:N` ⟨*comma list*⟩

Clears all items from the ⟨*comma list*⟩.

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

`\clist_clear_new:N` ⟨*comma list*⟩

Ensures that the ⟨*comma list*⟩ exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)

`\clist_set_eq:NN` ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩

Sets the content of ⟨*comma list*$_1$⟩ equal to that of ⟨*comma list*$_2$⟩.

\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

New: 2014-07-17

`\clist_set_from_seq:NN` ⟨*comma list*⟩ ⟨*sequence*⟩

Converts the data in the ⟨*sequence*⟩ into a ⟨*comma list*⟩: the original ⟨*sequence*⟩ is unchanged. Items which contain either spaces or commas are surrounded by braces.

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

`\clist_concat:NNN` ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩ ⟨*comma list*$_3$⟩

Concatenates the content of ⟨*comma list*$_2$⟩ and ⟨*comma list*$_3$⟩ together and saves the result in ⟨*comma list*$_1$⟩. The items in ⟨*comma list*$_2$⟩ are placed at the left side of the new comma list.

\clist_if_exist_p:N ★
\clist_if_exist_p:c ★
\clist_if_exist:N*TF* ★
\clist_if_exist:c*TF* ★

New: 2012-03-03

`\clist_if_exist_p:N` ⟨*comma list*⟩
`\clist_if_exist:NTF` ⟨*comma list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*comma list*⟩ is currently defined. This does not check that the ⟨*comma list*⟩ really is a comma list.

2 Adding data to comma lists

\clist_set:Nn
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)

New: 2011-09-06

`\clist_set:Nn` ⟨*comma list*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Sets ⟨*comma list*⟩ to contain the ⟨*items*⟩, removing any previous content from the variable. Spaces are removed from both sides of each item.

\clist_put_left:Nn
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)

Updated: 2011-09-05

`\clist_put_left:Nn` ⟨*comma list*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the left of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

\clist_put_right:Nn
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)

Updated: 2011-09-05

`\clist_put_right:Nn ⟨comma list⟩ {⟨item_1⟩,...,⟨item_n⟩}`

Appends the ⟨*items*⟩ to the right of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

\clist_remove_duplicates:N
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

`\clist_remove_duplicates:N ⟨comma list⟩`

Removes duplicate items from the ⟨*comma list*⟩, leaving the left most copy of each item in the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: This function iterates through every item in the ⟨*comma list*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it does not work if any of the items in the ⟨*comma list*⟩ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn

Updated: 2011-09-06

`\clist_remove_all:Nn ⟨comma list⟩ {⟨item⟩}`

Removes every occurrence of ⟨*item*⟩ from the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TeX category codes apply).

\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

New: 2014-07-18

`\clist_reverse:N ⟨comma list⟩`

Reverses the order of items stored in the ⟨*comma list*⟩.

\clist_reverse:n

New: 2014-07-18

`\clist_reverse:n {⟨comma list⟩}`

Leaves the items in the ⟨*comma list*⟩ in the input stream in reverse order. Braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an `x`-type argument expansion.

\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn

New: 2017-02-06

`\clist_sort:Nn ⟨clist var⟩ {⟨comparison code⟩}`

Sorts the items in the ⟨*clist var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*clist var*⟩. The details of sorting comparison are described in Section 1.

4 Comma list conditionals

\clist_if_empty_p:N ★
\clist_if_empty_p:c ★
\clist_if_empty:N*TF* ★
\clist_if_empty:c*TF* ★

```
\clist_if_empty_p:N ⟨comma list⟩
\clist_if_empty:NTF ⟨comma list⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*comma list*⟩ is empty (containing no items).

\clist_if_empty_p:n ★
\clist_if_empty:n*TF* ★

New: 2014-07-05

```
\clist_if_empty_p:n {⟨comma list⟩}
\clist_if_empty:nTF {⟨comma list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*comma list*⟩ is empty (containing no items). The rules for space trimming are as for other `n`-type comma-list functions, hence the comma list `{~,~,,~}` (without outer braces) is empty, while `{~,{},}` (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

\clist_if_in:Nn*TF*
\clist_if_in:(NV|No|cn|cV|co)*TF*
\clist_if_in:nn*TF*
\clist_if_in:(nV|no)*TF*

Updated: 2011-09-06

```
\clist_if_in:NnTF ⟨comma list⟩ {⟨item⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*item*⟩ is present in the ⟨*comma list*⟩. In the case of an `n`-type ⟨*comma list*⟩, spaces are stripped from each item, but braces are not removed. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields `false`.

TEXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TEX category codes apply), and should not contain `,` nor start or end with a space.

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

When the comma list is given explicitly, as an `n`-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if your comma list that is being mapped is `{a␣,␣{{b}␣},␣,{},␣{c},}` then the arguments passed to the mapped function are '`a`', '`{b}␣`', an empty argument, and '`c`'.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

\clist_map_function:NN ☆
\clist_map_function:cN ☆
\clist_map_function:nN ☆

Updated: 2012-06-29

\clist_map_function:NN ⟨*comma list*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*comma list*⟩. The ⟨*function*⟩ receives one argument for each iteration. The ⟨*items*⟩ are returned from left to right. The function \clist_map_inline:Nn is in general more efficient than \clist_map_function:NN. One mapping may be nested inside another.

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

Updated: 2012-06-29

\clist_map_inline:Nn ⟨*comma list*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*comma list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn

Updated: 2012-06-29

\clist_map_variable:NNn ⟨*comma list*⟩ ⟨*tl var.*⟩ {⟨*function using tl var.*⟩}

Stores each entry in the ⟨*comma list*⟩ in turn in the ⟨*tl var.*⟩ and applies the ⟨*function using tl var.*⟩ The ⟨*function*⟩ usually consists of code making use of the ⟨*tl var.*⟩, but this is not enforced. One variable mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

\clist_map_break: ☆

Updated: 2012-06-29

\clist_map_break:

Used to terminate a \clist_map_... function before all entries in the ⟨*comma list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n {`⟨*tokens*⟩`}`

Used to terminate a `\clist_map_...` function before all entries in the ⟨*comma list*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\clist_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ★
`\clist_count:c` ★
`\clist_count:n` ★

New: 2012-07-13

`\clist_count:N` ⟨*comma list*⟩

Leaves the number of items in the ⟨*comma list*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*comma list*⟩ includes those which are duplicates, *i.e.* every item in a ⟨*comma list*⟩ is unique.

6 Using the content of comma lists directly

`\clist_use:Nnnn` ★
`\clist_use:cnnn` ★

New: 2013-05-26

`\clist_use:Nnnn` ⟨*clist var*⟩ `{`⟨*separator between two*⟩`}` `{`⟨*separator between more than two*⟩`}` `{`⟨*separator between final two*⟩`}`

Places the contents of the ⟨*clist var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `x`-type argument expansion.

\clist_use:Nn ★
\clist_use:cn ★

New: 2013-05-26

\clist_use:Nn ⟨clist var⟩ {⟨separator⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts "a and b and c and de and f" in the input stream.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an x-type argument expansion.

7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

\clist_get:NN
\clist_get:cN

Updated: 2012-05-14

\clist_get:NN ⟨comma list⟩ ⟨token list variable⟩

Stores the left-most item from a ⟨*comma list*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*comma list*⟩. The ⟨*token list variable*⟩ is assigned locally. If the ⟨*comma list*⟩ is empty the ⟨*token list variable*⟩ is set to the marker value \q_no_value.

\clist_get:NN*TF*
\clist_get:cN*TF*

New: 2012-05-14

\clist_get:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, stores the top item from the ⟨*comma list*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*comma list*⟩. The ⟨*token list variable*⟩ is assigned locally.

\clist_pop:NN
\clist_pop:cN

Updated: 2011-09-06

\clist_pop:NN ⟨comma list⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨*comma list*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*token list variable*⟩. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN ⟨comma list⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨*comma list*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*token list variable*⟩. The ⟨*comma list*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local.

\clist_pop:NNTF
\clist_pop:cNTF

New: 2012-05-14

`\clist_pop:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, pops the top item from the ⟨*comma list*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*comma list*⟩. Both the ⟨*comma list*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\clist_gpop:NNTF
\clist_gpop:cNTF

New: 2012-05-14

`\clist_gpop:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, pops the top item from the ⟨*comma list*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*comma list*⟩. The ⟨*comma list*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\clist_push:Nn
\clist_push:(NV|No|Nx|cn|cV|co|cx)
\clist_gpush:Nn
\clist_gpush:(NV|No|Nx|cn|cV|co|cx)

`\clist_push:Nn ⟨comma list⟩ {⟨items⟩}`

Adds the {⟨*items*⟩} to the top of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

8 Using a single item

\clist_item:Nn ★
\clist_item:cn ★
\clist_item:nn ★

New: 2014-07-17

`\clist_item:Nn ⟨comma list⟩ {⟨integer expression⟩}`

Indexing items in the ⟨*comma list*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the comma list in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*comma list*⟩ (as calculated by `\clist_count:N`) then the function expands to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

9 Viewing comma lists

\clist_show:N
\clist_show:c

Updated: 2015-08-03

`\clist_show:N ⟨comma list⟩`

Displays the entries in the ⟨*comma list*⟩ in the terminal.

\clist_show:n

Updated: 2013-08-03

`\clist_show:n {⟨tokens⟩}`

Displays the entries in the comma list in the terminal.

`\clist_log:N`
`\clist_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\clist_log:N ⟨comma list⟩`

Writes the entries in the ⟨*comma list*⟩ in the log file. See also `\clist_show:N` which displays the result in the terminal.

`\clist_log:n`

New: 2014-08-22

`\clist_log:n {⟨tokens⟩}`

Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

10 Constant and scratch comma lists

`\c_empty_clist`

New: 2012-07-02

Constant that is always empty.

`\l_tmpa_clist`
`\l_tmpb_clist`

New: 2011-09-06

Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_clist`
`\g_tmpb_clist`

New: 2011-09-06

Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part XV
The **l3token** package Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its "shape" (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its "meaning", which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of TeX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, TeX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 8.

1 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`
`\char_gset_active_eq:NN`
`\char_gset_active_eq:Nc`

Updated: 2015-11-12

`\char_set_active_eq:NN` ⟨*char*⟩ ⟨*function*⟩

Sets the behaviour of the ⟨*char*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

`\char_set_active_eq:nN`
`\char_set_active_eq:nc`
`\char_gset_active_eq:nN`
`\char_gset_active_eq:nc`

New: 2015-11-12

`\char_set_active_eq:nN {`⟨*integer expression*⟩`}` ⟨*function*⟩

Sets the behaviour of the ⟨*char*⟩ which has character code as given by the ⟨*integer expression*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

`\char_generate:nn` ⋆

New: 2015-09-09

`\char_generate:nn {⟨charcode⟩} {⟨catcode⟩}`

Generates a character token of the given ⟨*charcode*⟩ and ⟨*catcode*⟩ (both of which may be integer expressions). The ⟨*catcode*⟩ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)

and other values raise an error.

The ⟨*charcode*⟩ may be any one valid for the engine in use. Note however that for XeTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

Updated: 2015-11-11

`\char_set_catcode_letter:N ⟨character⟩`

Sets the category code of the ⟨*character*⟩ to that indicated in the function name. Depending on the current category code of the ⟨*token*⟩ the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```
\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
```

Updated: 2015-11-11

`\char_set_catcode_letter:n {⟨integer expression⟩}`

Sets the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. This version can be used to set up characters which cannot otherwise be given (*cf.* the `N`-type variants). The assignment is local.

`\char_set_catcode:nn`

Updated: 2015-11-11

`\char_set_catcode:nn` {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

These functions set the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. The first ⟨*integer expression*⟩ is the character code and the second is the category code to apply. The setting applies within the current TEX group. In general, the symbolic functions `\char_set_catcode_`⟨*type*⟩ should be preferred, but there are cases where these lower-level functions may be useful.

`\char_value_catcode:n` ★

`\char_value_catcode:n` {⟨*integer expression*⟩}

Expands to the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_catcode:n`

`\char_show_value_catcode:n` {⟨*integer expression*⟩}

Displays the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_lccode:nn`

Updated: 2015-08-06

`\char_set_lccode:nn` {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Sets up the behaviour of the ⟨*character*⟩ when found inside `\tl_lower_case:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TEX ‘⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_lccode:nn { `\A } { `\a } % Standard behaviour
\char_set_lccode:nn { `\A } { `\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TEX group.

`\char_value_lccode:n` ★

`\char_value_lccode:n` {⟨*integer expression*⟩}

Expands to the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_lccode:n`

`\char_show_value_lccode:n` {⟨*integer expression*⟩}

Displays the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_uccode:nn`

Updated: 2015-08-06

`\char_set_uccode:nn` {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Sets up the behaviour of the ⟨*character*⟩ when found inside `\tl_upper_case:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TEX ‘⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_uccode:nn { `\a } { `\A } % Standard behaviour
\char_set_uccode:nn { `\A } { `\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TEX group.

`\char_value_uccode:n` ★

`\char_value_uccode:n {⟨integer expression⟩}`

Expands to the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_uccode:n`

`\char_show_value_uccode:n {⟨integer expression⟩}`

Displays the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_mathcode:nn`

Updated: 2015-08-06

`\char_set_mathcode:nn {⟨intexpr`$_1$`⟩} {⟨intexpr`$_2$`⟩}`

This function sets up the math code of ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TEX group.

`\char_value_mathcode:n` ★

`\char_value_mathcode:n {⟨integer expression⟩}`

Expands to the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_mathcode:n`

`\char_show_value_mathcode:n {⟨integer expression⟩}`

Displays the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_sfcode:nn`

Updated: 2015-08-06

`\char_set_sfcode:nn {⟨intexpr`$_1$`⟩} {⟨intexpr`$_2$`⟩}`

This function sets up the space factor for the ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TEX group.

`\char_value_sfcode:n` ★

`\char_value_sfcode:n {⟨integer expression⟩}`

Expands to the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_sfcode:n`

`\char_show_value_sfcode:n {⟨integer expression⟩}`

Displays the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\l_char_active_seq`

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.

`\l_char_special_seq`

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories ⟨*letter*⟩ (catcode 11) or ⟨*other*⟩ (catcode 12). Each entry in the sequence consists of a single escaped token, for example `\\` for the backslash or `\{` for an opening brace.Escaped tokens should be added to the sequence when they are defined for general document use.

3 Generic tokens

`\token_new:Nn`

`\token_new:Nn` ⟨*token*$_1$⟩ {⟨*token*$_2$⟩}

Defines ⟨*token*$_1$⟩ to globally be a snapshot of ⟨*token*$_2$⟩. This is an implicit representation of ⟨*token*$_2$⟩.

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

`\c_catcode_letter_token`
`\c_catcode_other_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

`\c_catcode_active_tl`

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

4 Converting tokens

`\token_to_meaning:N` ⋆
`\token_to_meaning:c` ⋆

`\token_to_meaning:N` ⟨*token*⟩

Inserts the current meaning of the ⟨*token*⟩ into the input stream as a series of characters of category code 12 (other). This is the primitive TEX description of the ⟨*token*⟩, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as `macros`.

TEXhackers note: This is the TEX primitive `\meaning`.

`\token_to_str:N` ⋆
`\token_to_str:c` ⋆

`\token_to_str:N` ⟨*token*⟩

Converts the given ⟨*token*⟩ into a series of characters with category code 12 (other). If the ⟨*token*⟩ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the ⟨*token*⟩). This function requires only a single expansion.

TEXhackers note: `\token_to_str:N` is the TEX primitive `\string` renamed.

5 Token conditionals

```
\token_if_group_begin_p:N ★
\token_if_group_begin:NTF ★
```

```
\token_if_group_begin_p:N ⟨token⟩
\token_if_group_begin:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a begin group token (`{` when normal TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid `N`-type argument.

```
\token_if_group_end_p:N ★
\token_if_group_end:NTF ★
```

```
\token_if_group_end_p:N ⟨token⟩
\token_if_group_end:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of an end group token (`}` when normal TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid `N`-type argument.

```
\token_if_math_toggle_p:N ★
\token_if_math_toggle:NTF ★
```

```
\token_if_math_toggle_p:N ⟨token⟩
\token_if_math_toggle:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a math shift token (`$` when normal TeX category codes are in force).

```
\token_if_alignment_p:N ★
\token_if_alignment:NTF ★
```

```
\token_if_alignment_p:N ⟨token⟩
\token_if_alignment:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of an alignment token (`&` when normal TeX category codes are in force).

```
\token_if_parameter_p:N ★
\token_if_parameter:NTF ★
```

```
\token_if_parameter_p:N ⟨token⟩
\token_if_alignment:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a macro parameter token (`#` when normal TeX category codes are in force).

```
\token_if_math_superscript_p:N ★
\token_if_math_superscript:NTF ★
```

```
\token_if_math_superscript_p:N ⟨token⟩
\token_if_math_superscript:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a superscript token (`^` when normal TeX category codes are in force).

```
\token_if_math_subscript_p:N ★
\token_if_math_subscript:NTF ★
```

```
\token_if_math_subscript_p:N ⟨token⟩
\token_if_math_subscript:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a subscript token (`_` when normal TeX category codes are in force).

```
\token_if_space_p:N ★
\token_if_space:NTF ★
```

```
\token_if_space_p:N ⟨token⟩
\token_if_space:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*token*⟩ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid `N`-type argument.

`\token_if_letter_p:N` ★
`\token_if_letter:NTF` ★

`\token_if_letter_p:N` ⟨*token*⟩
`\token_if_letter:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if ⟨*token*⟩ has the category code of a letter token.

`\token_if_other_p:N` ★
`\token_if_other:NTF` ★

`\token_if_other_p:N` ⟨*token*⟩
`\token_if_other:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if ⟨*token*⟩ has the category code of an “other” token.

`\token_if_active_p:N` ★
`\token_if_active:NTF` ★

`\token_if_active_p:N` ⟨*token*⟩
`\token_if_active:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if ⟨*token*⟩ has the category code of an active character.

`\token_if_eq_catcode_p:NN` ★
`\token_if_eq_catcode:NNTF` ★

`\token_if_eq_catcode_p:NN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
`\token_if_eq_catcode:NNTF` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the two ⟨*tokens*⟩ have the same category code.

`\token_if_eq_charcode_p:NN` ★
`\token_if_eq_charcode:NNTF` ★

`\token_if_eq_charcode_p:NN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
`\token_if_eq_charcode:NNTF` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the two ⟨*tokens*⟩ have the same character code.

`\token_if_eq_meaning_p:NN` ★
`\token_if_eq_meaning:NNTF` ★

`\token_if_eq_meaning_p:NN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
`\token_if_eq_meaning:NNTF` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the two ⟨*tokens*⟩ have the same meaning when expanded.

`\token_if_macro_p:N` ★
`\token_if_macro:NTF` ★

Updated: 2011-05-23

`\token_if_macro_p:N` ⟨*token*⟩
`\token_if_macro:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*token*⟩ is a TeX macro.

`\token_if_cs_p:N` ★
`\token_if_cs:NTF` ★

`\token_if_cs_p:N` ⟨*token*⟩
`\token_if_cs:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*token*⟩ is a control sequence.

`\token_if_expandable_p:N` ★
`\token_if_expandable:NTF` ★

`\token_if_expandable_p:N` ⟨*token*⟩
`\token_if_expandable:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*token*⟩ is expandable. This test returns ⟨*false*⟩ for an undefined token.

`\token_if_long_macro_p:N` ★
`\token_if_long_macro:NTF` ★

Updated: 2012-01-20

`\token_if_long_macro_p:N` ⟨*token*⟩
`\token_if_long_macro:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*token*⟩ is a long macro.

`\token_if_protected_macro_p:N` ★
`\token_if_protected_macro:NTF` ★

Updated: 2012-01-20

`\token_if_protected_macro_p:N` ⟨*token*⟩
`\token_if_protected_macro:NTF` ⟨*token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*token*⟩ is a protected macro: for a macro which is both protected and long this returns `false`.

```
\token_if_protected_long_macro_p:N ★
\token_if_protected_long_macro:NTF ★
```

Updated: 2012-01-20

```
\token_if_protected_long_macro_p:N ⟨token⟩
\token_if_protected_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false
code⟩}
```

Tests if the ⟨*token*⟩ is a protected long macro.

```
\token_if_chardef_p:N ★
\token_if_chardef:NTF ★
```

Updated: 2012-01-20

```
\token_if_chardef_p:N ⟨token⟩
\token_if_chardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a chardef.

TEXhackers note: Booleans, boxes and small integer constants are implemented as `\chardef`s.

```
\token_if_mathchardef_p:N ★
\token_if_mathchardef:NTF ★
```

Updated: 2012-01-20

```
\token_if_mathchardef_p:N ⟨token⟩
\token_if_mathchardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a mathchardef.

```
\token_if_dim_register_p:N ★
\token_if_dim_register:NTF ★
```

Updated: 2012-01-20

```
\token_if_dim_register_p:N ⟨token⟩
\token_if_dim_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a dimension register.

```
\token_if_int_register_p:N ★
\token_if_int_register:NTF ★
```

Updated: 2012-01-20

```
\token_if_int_register_p:N ⟨token⟩
\token_if_int_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a integer register.

TEXhackers note: Constant integers may be implemented as integer registers, `\chardef`s, or `\mathchardef`s depending on their value.

```
\token_if_muskip_register_p:N ★
\token_if_muskip_register:NTF ★
```

New: 2012-02-15

```
\token_if_muskip_register_p:N ⟨token⟩
\token_if_muskip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a muskip register.

```
\token_if_skip_register_p:N ★
\token_if_skip_register:NTF ★
```

Updated: 2012-01-20

```
\token_if_skip_register_p:N ⟨token⟩
\token_if_skip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a skip register.

`\token_if_toks_register_p:N` ★
`\token_if_toks_register:NTF` ★

Updated: 2012-01-20

`\token_if_toks_register_p:N` ⟨*token*⟩
`\token_if_toks_register:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a toks register (not used byLaTeX3).

`\token_if_primitive_p:N` ★
`\token_if_primitive:NTF` ★

Updated: 2011-05-23

`\token_if_primitive_p:N` ⟨*token*⟩
`\token_if_primitive:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is an engine primitive.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the "peek" functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

`\peek_after:Nw`

`\peek_after:Nw` ⟨*function*⟩ ⟨*token*⟩

Locally sets the test variable `\l_peek_token` equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\peek_gafter:Nw`

`\peek_gafter:Nw` ⟨*function*⟩ ⟨*token*⟩

Globally sets the test variable `\g_peek_token` equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\l_peek_token`

Token set by `\peek_after:Nw` and available for testing as described above.

`\g_peek_token`

Token set by `\peek_gafter:Nw` and available for testing as described above.

`\peek_catcode:NTF`

Updated: 2012-12-20

`\peek_catcode:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_catcode_ignore_spaces:NTF`

Updated: 2012-12-20

`\peek_catcode_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_catcode_remove:NTF`

Updated: 2012-12-20

`\peek_catcode_remove:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_catcode_remove_ignore_spaces:NTF`

Updated: 2012-12-20

`\peek_catcode_remove_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_charcode:NTF`

Updated: 2012-12-20

`\peek_charcode:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_charcode_ignore_spaces:NTF`

Updated: 2012-12-20

`\peek_charcode_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_charcode_remove:NTF`

Updated: 2012-12-20

`\peek_charcode_remove:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_charcode_remove_ignore_spaces:NTF`

Updated: 2012-12-20

`\peek_charcode_remove_ignore_spaces:NTF ⟨test token⟩`
  `{⟨true code⟩} {⟨false code⟩}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_meaning:NTF`

Updated: 2011-07-02

`\peek_meaning:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_meaning_ignore_spaces:NTF`

Updated: 2012-12-05

`\peek_meaning_ignore_spaces:NTF ⟨test token⟩ {⟨true code⟩} {⟨false`
`code⟩}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_meaning_remove:NTF`

Updated: 2011-07-02

`\peek_meaning_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_meaning_remove_ignore_spaces:NTF`

Updated: 2012-12-05

`\peek_meaning_remove_ignore_spaces:NTF ⟨test token⟩`
  `{⟨true code⟩} {⟨false code⟩}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

These functions decompose TeX macros into their constituent parts: if the ⟨*token*⟩ passed is not a macro then no decomposition can occur. In the later case, all three functions leave `\scan_stop:` in the input stream.

`\token_get_arg_spec:N` ★ `\token_get_arg_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function leaves the primitive TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1 y #2 }
```

leaves `#1#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the arg spec. contains the string `->`, then the `spec` function produces incorrect results.

`\token_get_replacement_spec:N` ★ `\token_get_replacement_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function leaves the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

leaves `x#1 y#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the arg spec. contains the string `->`, then the `spec` function produces incorrect results.

`\token_get_prefix_spec:N` ★ `\token_get_prefix_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function leaves the TeX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

leaves `\long` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream

8 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.

- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).[4]

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A "frozen" `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand` ⟨*token*⟩ (when the ⟨*token*⟩ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:` ⟨*token*⟩, whose shape coincides with the ⟨*token*⟩ and whose meaning differs from `\relax`.
- An `\outer endtemplate:` (expanding to another internal token, `end of alignment template`) can be encountered when peeking ahead at the next token.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their LaTeX3 names and most common example:

1 begin-group character (`group_begin`, often `{`),

2 end-group character (`group_end`, often `}`),

3 math shift character (`math_toggle`, often `$`),

4 alignment tab character (`alignment`, often `&`),

6 macro parameter character (`parameter`, often `#`),

7 superscript character (`math_superscript`, often `^`),

8 subscript character (`math_subscript`, often `_`),

10 blank space (`space`, often character code 32),

11 the letter (`letter`, such as `A`),

12 the character (`other`, such as `0`).

[4] In LuaTeX, there is also the case of "bytes", which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in LaTeX3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in LaTeX3 for some functions,
- a register such as `\count123`, used in LaTeX3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros be `\protected` or not, `\long` or not (the opposite of what LaTeX3 calls `nopar`), and `\outer` or not (unused in LaTeX3). Their `\meaning` takes the form

⟨*properties*⟩ `macro:`⟨*parameters*⟩`->`⟨*replacement*⟩

where ⟨*properties*⟩ is among `\protected\long\outer`, ⟨*parameters*⟩ describes parameters that the macro expects, such as `#1#2#3`, and ⟨*replacement*⟩ describes how the parameters are manipulated, such as `#2/#1/#3`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then TeX scans ahead to obtain an equal number of explicit character tokens with category 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

9 Internal functions

`\__char_generate:nn` ⋆

New: 2016-03-25

`\__char_generate:nn {`⟨*charcode*⟩`} {`⟨*catcode*⟩`}`

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The ⟨*catcode*⟩ must give an explicit integer when expanded (and must not absorb a space for instance).

Part XVI

The l3prop package Property lists

LaTeX3 implements a "property list" data type, which contain an unordered list of entries each of which consists of a ⟨*key*⟩ and an associated ⟨*value*⟩. The ⟨*key*⟩ and ⟨*value*⟩ may both be any ⟨*balanced text*⟩. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique ⟨*key*⟩: if an entry is added to a property list which already contains the ⟨*key*⟩ then the new entry overwrites the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the keys module.

1 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` ⟨*property list*⟩

Creates a new ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ initially contains no entries.

`\prop_clear:N`
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

`\prop_clear:N` ⟨*property list*⟩

Clears all entries from the ⟨*property list*⟩.

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

`\prop_clear_new:N` ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

`\prop_set_eq:NN`
`\prop_set_eq:(cN|Nc|cc)`
`\prop_gset_eq:NN`
`\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN` ⟨*property list*$_1$⟩ ⟨*property list*$_2$⟩

Sets the content of ⟨*property list*$_1$⟩ equal to that of ⟨*property list*$_2$⟩.

2 Adding entries to property lists

\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)

Updated: 2012-07-09

\prop_put:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with \tl_to_str:n, meaning that category codes are ignored. If the ⟨*key*⟩ is already present in the ⟨*property list*⟩, the existing entry is overwritten by the new ⟨*value*⟩.

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn

\prop_put_if_new:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then no action is taken. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then a new entry is added. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with \tl_to_str:n, meaning that category codes are ignored.

3 Recovering values from property lists

\prop_get:NnN
\prop_get:(NVN|NoN|cnN|cVN|coN)

Updated: 2011-08-28

\prop_get:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ is set to the special marker \q_no_value. The ⟨*token list variable*⟩ is set within the current TeX group. See also \prop_get:NnNTF.

\prop_pop:NnN
\prop_pop:(NoN|cnN|coN)

Updated: 2011-08-18

\prop_pop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ is set to the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. Both assignments are local. See also \prop_pop:NnNTF.

\prop_gpop:NnN
\prop_gpop:(NoN|cnN|coN)

Updated: 2011-08-18

\prop_gpop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ is set to the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. The ⟨*property list*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local. See also \prop_gpop:NnNTF.

\prop_item:Nn ★
\prop_item:cn ★

New: 2014-07-17

`\prop_item:Nn ⟨property list⟩ {⟨key⟩}`

Expands to the ⟨*value*⟩ corresponding to the ⟨*key*⟩ in the ⟨*property list*⟩. If the ⟨*key*⟩ is missing, this has an empty expansion.

TEXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*value*⟩ does not expand further when appearing in an `x`-type argument expansion.

4 Modifying property lists

\prop_remove:Nn
\prop_remove:(NV|cn|cV)
\prop_gremove:Nn
\prop_gremove:(NV|cn|cV)

New: 2012-05-12

`\prop_remove:Nn ⟨property list⟩ {⟨key⟩}`

Removes the entry listed under ⟨*key*⟩ from the ⟨*property list*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

\prop_if_exist_p:N ★
\prop_if_exist_p:c ★
\prop_if_exist:N*TF* ★
\prop_if_exist:c*TF* ★

New: 2012-03-03

`\prop_if_exist_p:N ⟨property list⟩`
`\prop_if_exist:NTF ⟨property list⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*property list*⟩ is currently defined. This does not check that the ⟨*property list*⟩ really is a property list variable.

\prop_if_empty_p:N ★
\prop_if_empty_p:c ★
\prop_if_empty:N*TF* ★
\prop_if_empty:c*TF* ★

`\prop_if_empty_p:N ⟨property list⟩`
`\prop_if_empty:NTF ⟨property list⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*property list*⟩ is empty (containing no entries).

\prop_if_in_p:Nn ★
\prop_if_in_p:(NV|No|cn|cV|co) ★
\prop_if_in:Nn*TF* ★
\prop_if_in:(NV|No|cn|cV|co)*TF* ★

Updated: 2011-09-15

`\prop_if_in:NnTF ⟨property list⟩ {⟨key⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*key*⟩ is present in the ⟨*property list*⟩, making the comparison using the method described by `\str_if_eq:nnTF`.

TEXhackers note: This function iterates through every key–value pair in the ⟨*property list*⟩ and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnN*TF*
\prop_get:(NVN|NoN|cnN|cVN|coN)*TF*

Updated: 2012-05-19

```
\prop_get:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩
  {⟨true code⟩} {⟨false code⟩}
```

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, stores the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*property list*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*token list variable*⟩ is assigned locally.

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

```
\prop_pop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩}
{⟨false code⟩}
```

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*property list*⟩. Both the ⟨*property list*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

```
\prop_gpop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩}
{⟨false code⟩}
```

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*property list*⟩. The ⟨*property list*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

7 Mapping to property lists

\prop_map_function:NN ☆
\prop_map_function:cN ☆

Updated: 2013-01-08

```
\prop_map_function:NN ⟨property list⟩ ⟨function⟩
```

Applies ⟨*function*⟩ to every ⟨*entry*⟩ stored in the ⟨*property list*⟩. The ⟨*function*⟩ receives two argument for each iteration: the ⟨*key*⟩ and associated ⟨*value*⟩. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon.

\prop_map_inline:Nn
\prop_map_inline:cn

Updated: 2013-01-08

```
\prop_map_inline:Nn ⟨property list⟩ {⟨inline function⟩}
```

Applies ⟨*inline function*⟩ to every ⟨*entry*⟩ stored within the ⟨*property list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*key*⟩ as `#1` and the ⟨*value*⟩ as `#2`. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon.

\prop_map_break: ☆

Updated: 2012-06-29

\prop_map_break:

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {⟨*tokens*⟩}

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

8 Viewing property lists

\prop_show:N
\prop_show:c

Updated: 2015-08-01

\prop_show:N ⟨*property list*⟩

Displays the entries in the ⟨*property list*⟩ in the terminal.

\prop_log:N
\prop_log:c

New: 2014-08-12
Updated: 2015-08-01

\prop_log:N ⟨*property list*⟩

Writes the entries in the ⟨*property list*⟩ in the log file.

9 Scratch property lists

\l_tmpa_prop
\l_tmpb_prop

New: 2012-06-23

Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_prop
\g_tmpb_prop

New: 2012-06-23

Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

11 Internal property list functions

\s__prop

The internal token used at the beginning of property lists. This is also used after each ⟨*key*⟩ (see `\__prop_pair:wn`).

__prop_pair:wn

```
\__prop_pair:wn ⟨key⟩ \s__prop {⟨item⟩}
```

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

\l__prop_internal_tl

Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

__prop_split:NnTF

Updated: 2013-01-08

```
\__prop_split:NnTF ⟨property list⟩ {⟨key⟩} {⟨true code⟩} {⟨false code⟩}
```

Splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three token lists: the ⟨*extract*⟩ of ⟨*property list*⟩ before the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*⟩ of the ⟨*property list*⟩ after the ⟨*value*⟩. Both ⟨*extracts*⟩ retain the internal structure of a property list, and the concatenation of the two ⟨*extracts*⟩ is a property list. If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then the ⟨*true code*⟩ is left in the input stream, with #1, #2, and #3 replaced by the first ⟨*extract*⟩, the ⟨*value*⟩, and the second extract. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the ⟨*false code*⟩ is left in the input stream, with no trailing material. Both ⟨*true code*⟩ and ⟨*false code*⟩ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the ⟨*true code*⟩ for the three extracts from the property list. The ⟨*key*⟩ comparison takes place as described for `\str_if_eq:nn`.

Part XVII
The l3msg package
Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The l3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by l3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `\␣` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the LaTeX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

`\msg_new:nnnn`
`\msg_new:nnn`

Updated: 2011-08-16

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used. An error is raised if the ⟨*message*⟩ already exists.

`\msg_set:nnnn`
`\msg_set:nnn`
`\msg_gset:nnnn`
`\msg_gset:nnn`

`\msg_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used.

\msg_if_exist_p:nn ★
\msg_if_exist:nnTF ★

New: 2012-03-03

```
\msg_if_exist_p:nn {⟨module⟩} {⟨message⟩}
\msg_if_exist:nnTF {⟨module⟩} {⟨message⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

2 Contextual information for messages

\msg_line_context: ☆

```
\msg_line_context:
```

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

\msg_line_number: ★

```
\msg_line_number:
```

Prints the current line number when a message is given.

\msg_fatal_text:n ★

```
\msg_fatal_text:n {⟨module⟩}
```

Produces the standard text

```
Fatal ⟨module⟩ error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

\msg_critical_text:n ★

```
\msg_critical_text:n {⟨module⟩}
```

Produces the standard text

```
Critical ⟨module⟩ error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

\msg_error_text:n ★

```
\msg_error_text:n {⟨module⟩}
```

Produces the standard text

```
⟨module⟩ error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

\msg_warning_text:n ★

```
\msg_warning_text:n {⟨module⟩}
```

Produces the standard text

```
⟨module⟩ warning
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_info_text:n` ⋆

`\msg_info_text:n {⟨module⟩}`

Produces the standard text:

> `⟨module⟩ info`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_see_documentation_text:n` ⋆

`\msg_see_documentation_text:n {⟨module⟩}`

Produces the standard text

> `See the ⟨module⟩ documentation for further information.`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `x`-type variants should be used to expand material.

`\msg_fatal:nnnnnn`
`\msg_fatal:nnxxxx`
`\msg_fatal:nnnnn`
`\msg_fatal:nnxxx`
`\msg_fatal:nnnn`
`\msg_fatal:nnxx`
`\msg_fatal:nnn`
`\msg_fatal:nnx`
`\msg_fatal:nn`

Updated: 2012-08-11

`\msg_fatal:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TeX run halts.

`\msg_critical:nnnnnn`
`\msg_critical:nnxxxx`
`\msg_critical:nnnnn`
`\msg_critical:nnxxx`
`\msg_critical:nnnn`
`\msg_critical:nnxx`
`\msg_critical:nnn`
`\msg_critical:nnx`
`\msg_critical:nn`

Updated: 2012-08-11

`\msg_critical:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a critical error, TeX stops reading the current input file. This may halt the TeX run (if the current file is the main file) or may abort reading a sub-file.

TeXhackers note: The TeX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```
\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn
```

Updated: 2012-08-11

`\msg_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

```
\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn
```

Updated: 2012-08-11

`\msg_warning:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text is added to the log file and the terminal, but the TEX run is not interrupted.

```
\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn
```

Updated: 2012-08-11

`\msg_info:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text is added to the log file.

```
\msg_log:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn
```

Updated: 2012-08-11

`\msg_log:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text is added to the log file: the output is briefer than `\msg_info:nnnnnn`.

`\msg_none:nnnnnn`
`\msg_none:nnxxxx`
`\msg_none:nnnnn`
`\msg_none:nnxxx`
`\msg_none:nnnn`
`\msg_none:nnxx`
`\msg_none:nnn`
`\msg_none:nnx`
`\msg_none:nn`

Updated: 2012-08-11

`\msg_none:nnnnnn` {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

4 Redirecting messages

Each message has a "name", which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \to B$, $B \to C$ and $C \to A$ in this order, then the $A \to B$ redirection is cancelled.

`\msg_redirect_class:nn`

Updated: 2012-04-27

`\msg_redirect_class:nn` {⟨*class one*⟩} {⟨*class two*⟩}

Changes the behaviour of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩.

\msg_redirect_module:nnn

Updated: 2012-04-27

\msg_redirect_module:nnn {⟨*module*⟩} {⟨*class one*⟩} {⟨*class two*⟩}

Redirects message of ⟨*class one*⟩ for ⟨*module*⟩ to act as though they were from ⟨*class two*⟩. Messages of ⟨*class one*⟩ from sources other than ⟨*module*⟩ are not affected by this redirection. This function can be used to make some messages "silent" by default. For example, all of the `warning` messages of ⟨*module*⟩ could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

\msg_redirect_name:nnn

Updated: 2012-04-27

\msg_redirect_name:nnn {⟨*module*⟩} {⟨*message*⟩} {⟨*class*⟩}

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages. No further redirection is performed. This function can be used to make a selected message "silent" without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

\msg_interrupt:nnn

New: 2012-06-28

\msg_interrupt:nnn {⟨*first line*⟩} {⟨*text*⟩} {⟨*extra text*⟩}

Interrupts the TeX run, issuing a formatted message comprising ⟨*first line*⟩ and ⟨*text*⟩ laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!...............................................
```

where the ⟨*text*⟩ is wrapped to fit within the current line length. The user may then request more information, at which stage the ⟨*extra text*⟩ is shown in the terminal in the format

```
|'''''''''''''''''''''''''''''''''''''''''''''''
|  <extra text>
|...............................................
```

where the ⟨*extra text*⟩ is wrapped within the current line length. Wrapping of both ⟨*text*⟩ and ⟨*more text*⟩ takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_log:n`

New: 2012-06-28

`\msg_log:n {⟨text⟩}`

Writes to the log file with the ⟨*text*⟩ laid out in the format

```
................................................
. <text>
................................................
```

where the ⟨*text*⟩ is wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_term:n`

New: 2012-06-28

`\msg_term:n {⟨text⟩}`

Writes to the terminal and log file with the ⟨*text*⟩ laid out in the format

```
************************************************
* <text>
************************************************
```

where the ⟨*text*⟩ is wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 Kernel-specific functions

Messages from LaTeX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

`\__msg_kernel_new:nnnn`
`\__msg_kernel_new:nnn`

Updated: 2011-08-16

`\__msg_kernel_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied and expanded at the time the message is used. An error is raised if the ⟨*message*⟩ already exists.

`\__msg_kernel_set:nnnn`
`\__msg_kernel_set:nnn`

`\__msg_kernel_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied and expanded at the time the message is used.

```
\__msg_kernel_fatal:nnnnnn
\__msg_kernel_fatal:nnxxxx
\__msg_kernel_fatal:nnnnn
\__msg_kernel_fatal:nnxxx
\__msg_kernel_fatal:nnnn
\__msg_kernel_fatal:nnxx
\__msg_kernel_fatal:nnn
\__msg_kernel_fatal:nnx
\__msg_kernel_fatal:nn
```

Updated: 2012-08-11

`\__msg_kernel_fatal:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TEX run halts. Cannot be redirected.

```
\__msg_kernel_error:nnnnnn
\__msg_kernel_error:nnxxxx
\__msg_kernel_error:nnnnn
\__msg_kernel_error:nnxxx
\__msg_kernel_error:nnnn
\__msg_kernel_error:nnxx
\__msg_kernel_error:nnn
\__msg_kernel_error:nnx
\__msg_kernel_error:nn
```

Updated: 2012-08-11

`\__msg_kernel_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.

```
\__msg_kernel_warning:nnnnnn
\__msg_kernel_warning:nnxxxx
\__msg_kernel_warning:nnnnn
\__msg_kernel_warning:nnxxx
\__msg_kernel_warning:nnnn
\__msg_kernel_warning:nnxx
\__msg_kernel_warning:nnn
\__msg_kernel_warning:nnx
\__msg_kernel_warning:nn
```

Updated: 2012-08-11

`\__msg_kernel_warning:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text is added to the log file, but the TEX run is not interrupted.

```
\__msg_kernel_info:nnnnnn
\__msg_kernel_info:nnxxxx
\__msg_kernel_info:nnnnn
\__msg_kernel_info:nnxxx
\__msg_kernel_info:nnnn
\__msg_kernel_info:nnxx
\__msg_kernel_info:nnn
\__msg_kernel_info:nnx
\__msg_kernel_info:nn
```

Updated: 2012-08-11

`\__msg_kernel_info:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text is added to the log file.

7 Expandable errors

In a few places, the LaTeX3 kernel needs to produce errors in an expansion only context. This must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

```
\__msg_kernel_expandable_error:nnnnnn ⋆
\__msg_kernel_expandable_error:nnnnn  ⋆
\__msg_kernel_expandable_error:nnnn   ⋆
\__msg_kernel_expandable_error:nnn    ⋆
\__msg_kernel_expandable_error:nn     ⋆
```

New: 2011-11-23

```
\__msg_kernel_expandable_error:nnnnnn {⟨module⟩} {⟨message⟩}
{⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}
```

Issues an error, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.

```
\__msg_expandable_error:n ⋆
```

New: 2011-08-11
Updated: 2011-08-13

```
\__msg_expandable_error:n {⟨error message⟩}
```

Issues an "Undefined error" message from TeX itself, and prints the ⟨*error message*⟩. The ⟨*error message*⟩ must be short: it is cropped at the end of one line.

TeXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to TeX's prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

8 Internal l3msg functions

The following functions are used in several kernel modules.

```
\__msg_log_next:
```

New: 2015-08-05

```
\__msg_log_next: ⟨show-command⟩
```

Causes the next ⟨*show-command*⟩ to send its output to the log file instead of the terminal. This allows for instance `\cs_log:N` to be defined as `\__msg_log_next: \cs_show:N`. The effect of this command lasts until the next use of `\__msg_show_wrap:Nn` or `\__msg_show_wrap:n` or `\__msg_show_variable:NNNnn`, in other words until the next time the ε-TeX primitive `\showtokens` would have been used for showing to the terminal or until the next `variable-not-defined` error.

```
\__msg_show_pre:nnnnnn
\__msg_show_pre:(nnxxxx|nnnnnV)
```

New: 2015-08-05

```
\__msg_show_pre:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩}
{⟨arg three⟩} {⟨arg four⟩}
```

Prints the ⟨*message*⟩ from ⟨*module*⟩ in the terminal (or log file if `\__msg_log_next:` was issued) without formatting. Used in messages which print complex variable contents completely.

`\__msg_show_variable:NNNnn`

New: 2015-08-04

`\__msg_show_variable:NNNnn` ⟨*variable*⟩ ⟨*if-exist*⟩ ⟨*if-empty*⟩ {⟨*msg*⟩} {⟨*formatted content*⟩}

If the ⟨*variable*⟩ does not exist according to ⟨*if-exist*⟩ (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if ⟨*msg*⟩ is not empty, display the message `LaTeX/kernel/show-`⟨*msg*⟩ with `\token_to_str:N` ⟨*variable*⟩ as a first argument, and a second argument that is `?` or empty depending on the result of ⟨*if-empty*⟩ (typically `\tl_if_empty:NTF`) on the ⟨*variable*⟩. Then display the ⟨*formatted content*⟩ by giving it as an argument to `\__msg_show_wrap:n`.

`\__msg_show_wrap:Nn`

New: 2015-08-03
Updated: 2015-08-07

`\__msg_show_wrap:Nn` ⟨*function*⟩ {⟨*expression*⟩}

Shows or logs the ⟨*expression*⟩ (turned into a string), an equal sign, and the result of applying the ⟨*function*⟩ to the {⟨*expression*⟩}. For instance, if the ⟨*function*⟩ is `\int_eval:n` and the ⟨*expression*⟩ is `1+2` then this logs `> 1+2=3.` The case where the ⟨*function*⟩ is `\tl_to_str:n` is special: then the string representation of the ⟨*expression*⟩ is only logged once.

`\__msg_show_wrap:n`

New: 2015-08-03

`\__msg_show_wrap:n` {⟨*formatted text*⟩}

Shows or logs the ⟨*formatted text*⟩. After expansion, unless it is empty, the ⟨*formatted text*⟩ must contain `>`, and the part of ⟨*formatted text*⟩ before the first `>` is removed. Failure to do so causes low-level TeX errors.

`\__msg_show_item:n`
`\__msg_show_item:nn`
`\__msg_show_item_unbraced:nn`

Updated: 2012-09-09

`\__msg_show_item:n` ⟨*item*⟩
`\__msg_show_item:nn` ⟨*item-key*⟩ ⟨*item-value*⟩

Auxiliary functions used within the last argument of `\__msg_show_variable:NNNnn` or `\__msg_show_wrap:n` to format variable items correctly for display. The `\__msg_show_item:n` version is used for simple lists, the `\__msg_show_item:nn` and `\__msg_show_item_unbraced:nn` versions for key–value like data structures.

`\c__msg_coding_error_text_tl`

The text

```
This is a coding error.
```

used by kernel functions when erroneous programming input is encountered.

Part XVIII

The **l3file** package File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files TEX attempts to locate them both the operating system path and entries in the TEX file database (most TEX systems use such a database). Thus the "current path" for TEX is somewhat broader than that for other programs.

For functions which expect a ⟨*file name*⟩ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* be expanded, allowing the direct use of these in file names. File names are quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1 File operation functions

`\g_file_curr_dir_str`
`\g_file_curr_name_str`
`\g_file_curr_ext_str`

New: 2017-06-21

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the TEX search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The ⟨*name*⟩ and ⟨*ext*⟩ parts together make up the file name, thus the ⟨*name*⟩ part may be thought of as the "job name" for the current file. Note that TEX does not provide information on the ⟨*ext*⟩ part for the main (top level) file and that this file always has an empty ⟨*dir*⟩ component. Also, the ⟨*name*⟩ here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

`\l_file_search_path_seq`

New: 2017-06-18

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

TEXhackers note: When working as a package in LATEX 2ε, expl3 will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist:nTF`

Updated: 2012-02-10

```
\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}
```

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\l_file_search_path_seq`.

`\file_get_full_name:nN`
`\file_get_full_name:VN`

Updated: 2017-06-26

`\file_get_full_name:nN {⟨file name⟩} ⟨str var⟩`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found sets the ⟨*str var*⟩ the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given ⟨*file name*⟩ has no extension but the file found has that extension. If the file is not found then the ⟨*str var*⟩ is empty.

`\file_parse_full_name:nNNN`

New: 2017-06-23
Updated: 2017-06-26

`\file_parse_full_name:nNNN {⟨full name⟩} ⟨dir⟩ ⟨name⟩ ⟨ext⟩`

Parses the ⟨*full name*⟩ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The ⟨*dir*⟩: everything up to the last `/` (path separator) in the ⟨*file path*⟩. As with system `PATH` variables and related functions, the ⟨*dir*⟩ does *not* include the trailing `/` unless it points to the root directory. If there is no path (only a file name), ⟨*dir*⟩ is empty.
- The ⟨*name*⟩: everything after the last `/` up to the last `.`, where both of those characters are optional. The ⟨*name*⟩ may contain multiple `.` characters. It is empty if ⟨*full name*⟩ consists only of a directory name.
- The ⟨*ext*⟩: everything after the last `.` (including the dot). The ⟨*ext*⟩ is empty if there is no `.` after the last `/`.

This function does not expand the ⟨*full name*⟩ before turning it to a string. It assume that the ⟨*full name*⟩ either contains no quote (`"`) characters or is surrounded by a pair of quotes.

`\file_input:n`

Updated: 2017-06-26

`\file_input:n {⟨file name⟩}`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

`\file_show_list:`
`\file_log_list:`

```
\file_show_list:
\file_log_list:
```

These functions list all files loaded by LaTeX 2ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

1.1 Input–output stream management

As TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in LaTeX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

New: 2011-09-26
Updated: 2011-12-27

`\ior_new:N ⟨stream⟩`
`\iow_new:N ⟨stream⟩`

Globally reserves the name of the ⟨*stream*⟩, either for reading or for writing as appropriate. The ⟨*stream*⟩ is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a ⟨*stream*⟩ which has not been opened is an error, and the ⟨*stream*⟩ will behave as the corresponding `\c_term_...`.

\ior_open:Nn
\ior_open:cn

Updated: 2012-02-10

`\ior_open:Nn ⟨stream⟩ {⟨file name⟩}`

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\ior_close:N` instruction is given or the TEX run ends. If the file is not found, an error is raised.

\ior_open:Nn*TF*
\ior_open:cn*TF*

New: 2013-01-12

`\ior_open:NnTF ⟨stream⟩ {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\ior_close:N` instruction is given or the TEX run ends. The ⟨*true code*⟩ is then inserted into the input stream. If the file is not found, no error is raised and the ⟨*false code*⟩ is inserted into the input stream.

\iow_open:Nn
\iow_open:cn

Updated: 2012-02-09

`\iow_open:Nn ⟨stream⟩ {⟨file name⟩}`

Opens ⟨*file name*⟩ for writing using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\iow_close:N` instruction is given or the TEX run ends. Opening a file for writing clears any existing content in the file (*i.e.* writing is *not* additive).

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

`\ior_close:N ⟨stream⟩`
`\iow_close:N ⟨stream⟩`

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

New: 2017-06-27

```
\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:
```

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

1.2 Reading from files

`\ior_get:NN`

New: 2012-06-24

`\ior_get:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

Function that reads one or more lines (until an equal number of left and right braces are found) from the input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. If the ⟨*stream*⟩ is not open, input is requested from the terminal. The material read from the ⟨*stream*⟩ is tokenized by TeX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character `%` have the line ending converted to a space, so for example input

```
a b  c
```

results in a token list `a␣b␣c␣`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens.

TeXhackers note: This protected macro is a wrapper around the TeX primitive `\read`. Regardless of settings, TeX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

`\ior_str_get:NN`

New: 2016-12-04

`\ior_str_get:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

Function that reads one line from the input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. If the ⟨*stream*⟩ is not open, input is requested from the terminal. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the ⟨*token list variable*⟩ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b  c
```

results in a token list `a b  c` with the letters `a`, `b`, and `c` having category code 12.

TeXhackers note: This protected macro is a wrapper around the ε-TeX primitive `\readline`. Regardless of settings, TeX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

\ior_map_inline:Nn

New: 2012-02-11

`\ior_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

Applies the ⟨*inline function*⟩ to each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file. TEX ignores any trailing new-line marker from the file it reads. The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`.

\ior_str_map_inline:Nn

New: 2012-02-11

`\ior_str_map_inline:Nn` {⟨*stream*⟩} {⟨*inline function*⟩}

Applies the ⟨*inline function*⟩ to every ⟨*line*⟩ in the ⟨*stream*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads.

\ior_map_break:

New: 2012-06-29

`\ior_map_break:`

Used to terminate a `\ior_map_...` function before all lines from the ⟨*stream*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before further items are taken from the input stream. This depends on the design of the mapping function.

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n {⟨*tokens*⟩}

Used to terminate a \ior_map_... function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \ior_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N ★
\ior_if_eof:N*TF* ★

Updated: 2012-02-10

\ior_if_eof_p:N ⟨*stream*⟩
\ior_if_eof:NTF ⟨*stream*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test also returns a true value if the ⟨*stream*⟩ is not open.

2 Writing to files

\iow_now:Nn
\iow_now:(Nx|cn|cx)

Updated: 2012-06-05

\iow_now:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ immediately (*i.e.* the write operation is called on expansion of \iow_now:Nn).

\iow_log:n
\iow_log:x

\iow_log:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_term:n
\iow_term:x

\iow_term:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

\iow_shipout:Nn
\iow_shipout:(Nx|cn|cx)

\iow_shipout:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The x-type variants expand the ⟨*tokens*⟩ at the point where the function is used but *not* when the resulting tokens are written to the ⟨*stream*⟩ (*cf.* \iow_shipout_x:Nn).

TEXhackers note: When using expl3 with a format other than LaTeX, new line characters inserted using \iow_newline: or using the line-wrapping code \iow_wrap:nnnN are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

\iow_shipout_x:Nn
\iow_shipout_x:(Nx|cn|cx)
Updated: 2012-09-08

\iow_shipout_x:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The ⟨*tokens*⟩ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TEXhackers note: This is a wrapper around the TEX primitive \write. When using expl3 with a format other than LaTeX, new line characters inserted using \iow_newline: or using the line-wrapping code \iow_wrap:nnnN are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

\iow_char:N ⋆

\iow_char:N \⟨*char*⟩

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of \iow_now:Nn).

\iow_newline: ⋆

\iow_newline:

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of \iow_now:Nn).

TEXhackers note: When using expl3 with a format other than LaTeX, the character inserted by \iow_newline: is not recognized by TEX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects \iow_shipout:Nn, \iow_shipout_x:Nn and direct uses of primitive operations.

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28
Updated: 2017-07-17

`\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨set up⟩} ⟨function⟩`

This function wraps the ⟨*text*⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨*run-on text*⟩ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the ⟨*run-on text*⟩ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The ⟨*text*⟩ and ⟨*run-on text*⟩ are exhaustively expanded by the function, with the following substitutions:

- `\\` may be used to force a new line,
- `\␣` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_indent:n` may be used to indent a part of the ⟨*text*⟩ (not the ⟨*run-on text*⟩).

Additional functions may be added to the wrapping by using the ⟨*set up*⟩, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the ⟨*text*⟩ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the ⟨*function*⟩) consists of characters of category "other" (category code 12), with the exception of spaces which have category "space" (category code 10). This means that the output does *not* expand further when written to a file.

TEXhackers note: Internally, `\iow_wrap:nnnN` carries out an `x`-type expansion on the ⟨*text*⟩ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨*text*⟩.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n {⟨text⟩}`

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents ⟨*text*⟩ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the ⟨*text*⟩. In case the indented ⟨*text*⟩ should appear on separate lines from the surrounding text, use `\\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the TEX system in use: the standard value is 78, which is typically correct for unmodified TEXlive and MiKTEX systems.

2.2 Constant input–output streams

\c_term_ior

Constant input stream for reading from the terminal. Reading from this stream using \ior_get:NN or similar results in a prompt from TeX of the form

```
<tl>=
```

\c_log_iow
\c_term_iow

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

2.3 Primitive conditionals

\if_eof:w ★

```
\if_eof:w ⟨stream⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests if the ⟨*stream*⟩ returns "end of file", which is true for non-existent files. The \else: branch is optional.

> **TeXhackers note:** This is the TeX primitive \ifeof.

2.4 Internal file functions and variables

\g__file_internal_ior

Used to test for the existence of files when opening.

\l__file_base_name_str
\l__file_full_name_str

Used to store and transfer the file name (including extension) and (partial) file path whilst reading files. (The file base is the base name plus any preceding directory name.)

__file_missing:n

New: 2017-06-25

```
\__file_missing:n {⟨name⟩}
```

Expands the ⟨*name*⟩ as per __file_name_sanitize:nN then produces an error message indicating that that file was not found.

__file_name_sanitize:nN

New: 2017-06-19

```
\__file_name_sanitize:nN {⟨name⟩} ⟨str var⟩
```

Exhaustively-expands the ⟨*name*⟩ with the exception of any category ⟨*active*⟩ (catcode 13) tokens, which are not expanded. The list of ⟨*active*⟩ tokens is taken from \l_char_active_seq. The ⟨*str var*⟩ is then set to the ⟨*sanitized name*⟩.

__file_name_quote:nN

New: 2017-06-19
Updated: 2017-06-25

```
\__file_name_quote:nN {⟨name⟩} ⟨str var⟩
```

Expands the ⟨*name*⟩ (without special-casing active tokens), then sets the ⟨*str var*⟩ to the ⟨*name*⟩ quoted using " at each end if required by the presence of spaces in the ⟨*name*⟩. Any existing " tokens is removed and if their number is odd an error is raised.

2.5 Internal input–output functions

`\__ior_open:Nn`
`\__ior_open:No`

New: 2012-01-23

`\__ior_open:Nn ⟨stream⟩ {⟨file name⟩}`

This function has identical syntax to the public version. However, is does not take precautions against active characters in the ⟨*file name*⟩, and it does not attempt to add a ⟨*path*⟩ to the ⟨*file name*⟩: it is therefore intended to be used by higher-level functions which have already fully expanded the ⟨*file name*⟩ and which need to perform multiple open or close operations. See for example the implementation of `\file_get_full_name:nN`,

`\__iow_with:Nnn`

New: 2014-08-23

`\__iow_with:Nnn ⟨integer⟩ {⟨value⟩} {⟨code⟩}`

If the ⟨*integer*⟩ is equal to the ⟨*value*⟩ then this function simply runs the ⟨*code*⟩. Otherwise it saves the current value of the ⟨*integer*⟩, sets it to the ⟨*value*⟩, runs the ⟨*code*⟩, and restores the ⟨*integer*⟩ to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is -1 when displaying a message.

Part XIX
The **l3skip** package Dimensions and skips

LaTeX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

`\dim_new:N`
`\dim_new:c`

`\dim_new:N` ⟨*dimension*⟩

Creates a new ⟨*dimension*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*dimension*⟩ is initially equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`
New: 2012-03-05

`\dim_const:Nn` ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ is set globally to the ⟨*dimension expression*⟩.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N` ⟨*dimension*⟩

Sets ⟨*dimension*⟩ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`
New: 2012-01-07

`\dim_zero_new:N` ⟨*dimension*⟩

Ensures that the ⟨*dimension*⟩ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the ⟨*dimension*⟩ set to zero.

`\dim_if_exist_p:N` ★
`\dim_if_exist_p:c` ★
`\dim_if_exist:NTF` ★
`\dim_if_exist:cTF` ★
New: 2012-03-03

`\dim_if_exist_p:N` ⟨*dimension*⟩
`\dim_if_exist:NTF` ⟨*dimension*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*dimension*⟩ is currently defined. This does not check that the ⟨*dimension*⟩ really is a dimension variable.

2 Setting `dim` variables

\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn

Updated: 2011-10-22

`\dim_add:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Adds the result of the ⟨*dimension expression*⟩ to the current content of the ⟨*dimension*⟩.

\dim_set:Nn
\dim_set:cn
\dim_gset:Nn
\dim_gset:cn

Updated: 2011-10-22

`\dim_set:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Sets ⟨*dimension*⟩ to the value of ⟨*dimension expression*⟩, which must evaluate to a length with units.

\dim_set_eq:NN
\dim_set_eq:(cN|Nc|cc)
\dim_gset_eq:NN
\dim_gset_eq:(cN|Nc|cc)

`\dim_set_eq:NN` ⟨*dimension*$_1$⟩ ⟨*dimension*$_2$⟩

Sets the content of ⟨*dimension*$_1$⟩ equal to that of ⟨*dimension*$_2$⟩.

\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

Updated: 2011-10-22

`\dim_sub:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Subtracts the result of the ⟨*dimension expression*⟩ from the current content of the ⟨*dimension*⟩.

3 Utilities for dimension calculations

\dim_abs:n ⋆

Updated: 2012-09-26

`\dim_abs:n {⟨dimexpr⟩}`

Converts the ⟨*dimexpr*⟩ to its absolute value, leaving the result in the input stream as a ⟨*dimension denotation*⟩.

\dim_max:nn ⋆
\dim_min:nn ⋆

New: 2012-09-09
Updated: 2012-09-26

`\dim_max:nn` {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}
`\dim_min:nn` {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Evaluates the two ⟨*dimension expressions*⟩ and leaves either the maximum or minimum value in the input stream as appropriate, as a ⟨*dimension denotation*⟩.

\dim_ratio:nn ☆

Updated: 2011-10-22

\dim_ratio:nn {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Parses the two ⟨*dimension expressions*⟩ and converts the ratio of the two to a form suitable for use inside a ⟨*dimension expression*⟩. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
  { 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ration expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays `327680/655360` on the terminal.

4 Dimension expression conditionals

\dim_compare_p:nNn ★
\dim_compare:nNn*TF* ★

\dim_compare_p:nNn {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩}
\dim_compare:nNnTF
  {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩}
  {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates each of the ⟨*dimension expressions*⟩ as described for `\dim_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

\dim_compare_p:n ⋆
\dim_compare:n*TF* ⋆

Updated: 2013-01-13

```
\dim_compare_p:n
  {
    ⟨dimexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨dimexpr_N⟩ ⟨relation_N⟩
    ⟨dimexpr_N+1⟩
  }
\dim_compare:nTF
  {
    ⟨dimexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨dimexpr_N⟩ ⟨relation_N⟩
    ⟨dimexpr_N+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*dimension expressions*⟩ as described for `\dim_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*dimexpr*$_1$⟩ and ⟨*dimexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*dimexpr*$_2$⟩ and ⟨*dimexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*dimexpr*$_N$⟩ and ⟨*dimexpr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*dimension expression*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*dimension expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	`=` or `==`
Greater than or equal to	`>=`
Greater than	`>`
Less than or equal to	`<=`
Less than	`<`
Not equal	`!=`

\dim_case:nn ★
\dim_case:nnTF ★

New: 2013-07-24

```
\dim_case:nnTF {⟨test dimension expression⟩}
  {
    {⟨dimexpr case_1⟩} {⟨code case_1⟩}
    {⟨dimexpr case_2⟩} {⟨code case_2⟩}
    ...
    {⟨dimexpr case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  {
    { 5 pt }        { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }     { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream.

5 Dimension expression loops

\dim_do_until:nNnn ☆

```
\dim_do_until:nNnn {⟨dimexpr_1⟩} ⟨relation⟩ {⟨dimexpr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

\dim_do_while:nNnn ☆

```
\dim_do_while:nNnn {⟨dimexpr_1⟩} ⟨relation⟩ {⟨dimexpr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

\dim_until_do:nNnn ☆

```
\dim_until_do:nNnn {⟨dimexpr_1⟩} ⟨relation⟩ {⟨dimexpr_2⟩} {⟨code⟩}
```

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nNnn` ☆

`\dim_while_do:nNnn {⟨dimexpr`$_1$`⟩} ⟨relation⟩ {⟨dimexpr`$_2$`⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

`\dim_do_until:nn` ☆

Updated: 2013-01-13

`\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\dim_do_while:nn` ☆

Updated: 2013-01-13

`\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\dim_until_do:nn` ☆

Updated: 2013-01-13

`\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nn` ☆

Updated: 2013-01-13

`\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

6 Using `dim` expressions and variables

`\dim_eval:n` ★

Updated: 2011-10-22

`\dim_eval:n {⟨dimension expression⟩}`

Evaluates the ⟨*dimension expression*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*dimension denotation*⟩ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TEX-style assignment as it is *not* an ⟨*internal dimension*⟩.

`\dim_use:N` ★
`\dim_use:c` ★

`\dim_use:N ⟨dimension⟩`

Recovers the content of a ⟨*dimension*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\dim_eval:n`).

TEXhackers note: `\dim_use:N` is the TEX primitive `\the`: this is one of several LATEX3 names for this primitive.

`\dim_to_decimal:n` ⋆

New: 2014-07-15

`\dim_to_decimal:n {⟨dimexpr⟩}`

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves `1.00374` in the input stream, *i.e.* the magnitude of one "big point" when converted to (TEX) points.

`\dim_to_decimal_in_bp:n` ⋆

New: 2014-07-15

`\dim_to_decimal_in_bp:n {⟨dimexpr⟩}`

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves `0.99628` in the input stream, *i.e.* the magnitude of one (TEX) point when converted to big points.

`\dim_to_decimal_in_sp:n` ⋆

New: 2015-05-18

`\dim_to_decimal_in_sp:n {⟨dimexpr⟩}`

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result is necessarily an integer.

`\dim_to_decimal_in_unit:nn` ⋆

New: 2014-07-15

`\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}`

Evaluates the ⟨*dimension expressions*⟩, and leaves the value of ⟨$dimexpr_1$⟩, expressed in a unit given by ⟨$dimexpr_2$⟩, in the input stream. The result is a decimal number, rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves `0.35277` in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε-TEX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

`\dim_to_fp:n` ⋆

New: 2012-05-08

`\dim_to_fp:n {⟨dimexpr⟩}`

Expands to an internal floating point number equal to the value of the ⟨*dimexpr*⟩ in `pt`. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

7 Viewing `dim` variables

`\dim_show:N`
`\dim_show:c`

`\dim_show:N ⟨dimension⟩`

Displays the value of the ⟨*dimension*⟩ on the terminal.

`\dim_show:n`

New: 2011-11-22
Updated: 2015-08-07

`\dim_show:n {⟨dimension expression⟩}`

Displays the result of evaluating the ⟨*dimension expression*⟩ on the terminal.

`\dim_log:N`
`\dim_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\dim_log:N ⟨dimension⟩`

Writes the value of the ⟨*dimension*⟩ in the log file.

`\dim_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\dim_log:n {⟨dimension expression⟩}`

Writes the result of evaluating the ⟨*dimension expression*⟩ in the log file.

8 Constant dimensions

`\c_max_dim`

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

`\c_zero_dim`

A zero length as a dimension. This can also be used as a component of a skip.

9 Scratch dimensions

`\l_tmpa_dim`
`\l_tmpb_dim`

Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_dim`
`\g_tmpb_dim`

Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

10 Creating and initialising skip variables

\skip_new:N
\skip_new:c

`\skip_new:N ⟨skip⟩`

Creates a new ⟨*skip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*skip*⟩ is initially equal to 0 pt.

\skip_const:Nn
\skip_const:cn
New: 2012-03-05

`\skip_const:Nn ⟨skip⟩ {⟨skip expression⟩}`

Creates a new constant ⟨*skip*⟩ or raises an error if the name is already taken. The value of the ⟨*skip*⟩ is set globally to the ⟨*skip expression*⟩.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

`\skip_zero:N ⟨skip⟩`

Sets ⟨*skip*⟩ to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c
New: 2012-01-07

`\skip_zero_new:N ⟨skip⟩`

Ensures that the ⟨*skip*⟩ exists globally by applying `\skip_new:N` if necessary, then applies `\skip_(g)zero:N` to leave the ⟨*skip*⟩ set to zero.

\skip_if_exist_p:N ★
\skip_if_exist_p:c ★
\skip_if_exist:N*TF* ★
\skip_if_exist:c*TF* ★
New: 2012-03-03

```
\skip_if_exist_p:N ⟨skip⟩
\skip_if_exist:NTF ⟨skip⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*skip*⟩ is currently defined. This does not check that the ⟨*skip*⟩ really is a skip variable.

11 Setting skip variables

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
Updated: 2011-10-22

`\skip_add:Nn ⟨skip⟩ {⟨skip expression⟩}`

Adds the result of the ⟨*skip expression*⟩ to the current content of the ⟨*skip*⟩.

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn
Updated: 2011-10-22

`\skip_set:Nn ⟨skip⟩ {⟨skip expression⟩}`

Sets ⟨*skip*⟩ to the value of ⟨*skip expression*⟩, which must evaluate to a length with units and may include a rubber component (for example `1 cm plus 0.5 cm`.

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

`\skip_set_eq:NN ⟨skip_1⟩ ⟨skip_2⟩`

Sets the content of ⟨*skip*$_1$⟩ equal to that of ⟨*skip*$_2$⟩.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Updated: 2011-10-22

`\skip_sub:Nn ⟨skip⟩ {⟨skip expression⟩}`

Subtracts the result of the ⟨*skip expression*⟩ from the current content of the ⟨*skip*⟩.

12 Skip expression conditionals

\skip_if_eq_p:nn ⋆
\skip_if_eq:nnTF ⋆

```
\skip_if_eq_p:nn {⟨skipexpr1⟩} {⟨skipexpr2⟩}
\dim_compare:nTF
  {⟨skipexpr1⟩} {⟨skipexpr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*skip expressions*⟩ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

\skip_if_finite_p:n ⋆
\skip_if_finite:nTF ⋆

New: 2012-03-05

```
\skip_if_finite_p:n {⟨skipexpr⟩}
\skip_if_finite:nTF {⟨skipexpr⟩} {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*skip expression*⟩ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

\skip_eval:n ⋆

Updated: 2011-10-22

`\skip_eval:n {⟨skip expression⟩}`

Evaluates the ⟨*skip expression*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\skip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*glue denotation*⟩ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal glue*⟩.

\skip_use:N ⋆
\skip_use:c ⋆

`\skip_use:N ⟨skip⟩`

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\skip_eval:n`).

TeXhackers note: `\skip_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

14 Viewing skip variables

\skip_show:N
\skip_show:c

Updated: 2015-08-03

`\skip_show:N ⟨skip⟩`

Displays the value of the ⟨*skip*⟩ on the terminal.

`\skip_show:n`

New: 2011-11-22
Updated: 2015-08-07

`\skip_show:n {⟨skip expression⟩}`

Displays the result of evaluating the ⟨*skip expression*⟩ on the terminal.

`\skip_log:N`
`\skip_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\skip_log:N ⟨skip⟩`

Writes the value of the ⟨*skip*⟩ in the log file.

`\skip_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\skip_log:n {⟨skip expression⟩}`

Writes the result of evaluating the ⟨*skip expression*⟩ in the log file.

15 Constant skips

`\c_max_skip`

Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.

`\c_zero_skip`

Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

16 Scratch skips

`\l_tmpa_skip`
`\l_tmpb_skip`

Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_skip`
`\g_tmpb_skip`

Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

17 Inserting skips into the output

`\skip_horizontal:N`
`\skip_horizontal:c`
`\skip_horizontal:n`

Updated: 2011-10-22

`\skip_horizontal:N ⟨skip⟩`
`\skip_horizontal:n {⟨skipexpr⟩}`

Inserts a horizontal ⟨*skip*⟩ into the current list.

TeXhackers note: `\skip_horizontal:N` is the TeX primitive `\hskip` renamed.

\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Updated: 2011-10-22

\skip_vertical:N ⟨*skip*⟩
\skip_vertical:n {⟨*skipexpr*⟩}

Inserts a vertical ⟨*skip*⟩ into the current list.

> **TEXhackers note:** \skip_vertical:N is the TEX primitive \vskip renamed.

18 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

\muskip_new:N ⟨*muskip*⟩

Creates a new ⟨*muskip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*muskip*⟩ is initially equal to 0 mu.

\muskip_const:Nn
\muskip_const:cn

New: 2012-03-05

\muskip_const:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Creates a new constant ⟨*muskip*⟩ or raises an error if the name is already taken. The value of the ⟨*muskip*⟩ is set globally to the ⟨*muskip expression*⟩.

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

\skip_zero:N ⟨*muskip*⟩

Sets ⟨*muskip*⟩ to 0 mu.

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

New: 2012-01-07

\muskip_zero_new:N ⟨*muskip*⟩

Ensures that the ⟨*muskip*⟩ exists globally by applying \muskip_new:N if necessary, then applies \muskip_(g)zero:N to leave the ⟨*muskip*⟩ set to zero.

\muskip_if_exist_p:N ⋆
\muskip_if_exist_p:c ⋆
\muskip_if_exist:N*TF* ⋆
\muskip_if_exist:c*TF* ⋆

New: 2012-03-03

\muskip_if_exist_p:N ⟨*muskip*⟩
\muskip_if_exist:NTF ⟨*muskip*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*muskip*⟩ is currently defined. This does not check that the ⟨*muskip*⟩ really is a muskip variable.

19 Setting muskip variables

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn

Updated: 2011-10-22

\muskip_add:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Adds the result of the ⟨*muskip expression*⟩ to the current content of the ⟨*muskip*⟩.

\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn

Updated: 2011-10-22

\muskip_set:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Sets ⟨*muskip*⟩ to the value of ⟨*muskip expression*⟩, which must evaluate to a math length with units and may include a rubber component (for example `1 mu plus 0.5 mu`.

\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)

\muskip_set_eq:NN ⟨*muskip*$_1$⟩ ⟨*muskip*$_2$⟩

Sets the content of ⟨*muskip*$_1$⟩ equal to that of ⟨*muskip*$_2$⟩.

\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Updated: 2011-10-22

\muskip_sub:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Subtracts the result of the ⟨*muskip expression*⟩ from the current content of the ⟨*skip*⟩.

20 Using muskip expressions and variables

\muskip_eval:n ★

Updated: 2011-10-22

\muskip_eval:n {⟨*muskip expression*⟩}

Evaluates the ⟨*muskip expression*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\muskip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*muglue denotation*⟩ after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TEX-style assignment as it is *not* an ⟨*internal muglue*⟩.

\muskip_use:N ★
\muskip_use:c ★

\muskip_use:N ⟨*muskip*⟩

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\muskip_eval:n`).

TEXhackers note: `\muskip_use:N` is the TEX primitive `\the`: this is one of several LATEX3 names for this primitive.

21 Viewing muskip variables

\muskip_show:N
\muskip_show:c

Updated: 2015-08-03

\muskip_show:N ⟨*muskip*⟩

Displays the value of the ⟨*muskip*⟩ on the terminal.

\muskip_show:n

New: 2011-11-22
Updated: 2015-08-07

\muskip_show:n {⟨*muskip expression*⟩}

Displays the result of evaluating the ⟨*muskip expression*⟩ on the terminal.

\muskip_log:N
\muskip_log:c

New: 2014-08-22
Updated: 2015-08-03

`\muskip_log:N ⟨muskip⟩`

Writes the value of the ⟨*muskip*⟩ in the log file.

\muskip_log:n

New: 2014-08-22
Updated: 2015-08-07

`\muskip_log:n {⟨muskip expression⟩}`

Writes the result of evaluating the ⟨*muskip expression*⟩ in the log file.

22 Constant muskips

\c_max_muskip

The maximum value that can be stored as a muskip, with no stretch nor shrink component.

\c_zero_muskip

A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

\l_tmpa_muskip
\l_tmpb_muskip

Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_muskip
\g_tmpb_muskip

Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

24 Primitive conditional

\if_dim:w

```
\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen2⟩
  ⟨true code⟩
\else:
  ⟨false⟩
\fi:
```

Compare two dimensions. The ⟨*relation*⟩ is one of `<`, `=` or `>` with category code 12.

> **TeXhackers note:** This is the TeX primitive `\ifdim`.

25 Internal functions

`\__dim_eval:w` ⋆
`\__dim_eval_end:` ⋆

`\__dim_eval:w` ⟨*dimexpr*⟩ `\__dim_eval_end:`

Evaluates ⟨*dimension expression*⟩ as described for `\dim_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when `\__dim_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\__dim_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

> **TEXhackers note:** This is the ε-TEX primitive `\dimexpr`.

Part XX
The **l3keys** package Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
  {
    key-one .code:n   = code including parameter #1,
    key-two .tl_set:N = \l_mymodule_store_tl
  }
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
  {
    key-one = value one,
    key-two = value two
  }
```

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 }  }
\DeclareDocumentCommand \MyModuleMacro { o m }
  {
    \group_begin:
      \keys_set:nn { mymodule } { #1 }
      % Main code for \MyModuleMacro
    \group_end:
  }
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`; spaces are *ignored* in key names. As discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  {
    \l_mymodule_tmp_tl .code:n = code
  }
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2015-11-07

`\keys_define:nn {⟨module⟩} {⟨keyval list⟩}`

Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. The ⟨*module*⟩ name should be a text value, but there are no restrictions on the nature of the text. In practice the ⟨*module*⟩ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The ⟨*keyval list*⟩ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some~code~using~#1,
    keyname .value_required:n = true
  }
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary ⟨*key*⟩, which when used may be supplied with a ⟨*value*⟩. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define "actions", such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
  {
    keyname .code:n           = Some~code~using~#1,
    keyname .value_required:n = true
  }
\keys_define:nn { mymodule }
```

```
{
  keyname .value_required:n = true,
  keyname .code:n           = Some~code~using~#1
}
```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current TeX scope.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

Updated: 2013-07-08

⟨*key*⟩ `.bool_set:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to ⟨*value*⟩ (which must be either `true` or `false`). If the variable does not exist, it will be created globally at the point that the key is set up.

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

New: 2011-08-28
Updated: 2013-07-08

⟨*key*⟩ `.bool_set_inverse:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to the logical inverse of ⟨*value*⟩ (which must be either `true` or `false`). If the ⟨*boolean*⟩ does not exist, it will be created globally at the point that the key is set up.

`.choice:`

⟨*key*⟩ `.choice:`

Sets ⟨*key*⟩ to act as a choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 3.

`.choices:nn`
`.choices:(Vn|on|xn)`

New: 2011-08-21
Updated: 2013-07-10

⟨*key*⟩ `.choices:nn = {⟨choices⟩} {⟨code⟩}`

Sets ⟨*key*⟩ to act as a choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 1). Choices are discussed in detail in section 3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

New: 2011-09-11

⟨*key*⟩ `.clist_set:N = ⟨comma list variable⟩`

Defines ⟨*key*⟩ to set ⟨*comma list variable*⟩ to ⟨*value*⟩. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

`.code:n`

Updated: 2013-07-10

⟨*key*⟩ `.code:n = {⟨code⟩}`

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is used. The ⟨*code*⟩ can include one parameter (`#1`), which will be the ⟨*value*⟩ given for the ⟨*key*⟩. The `x`-type variant expands ⟨*code*⟩ at the point where the ⟨*key*⟩ is created.

.default:n
.default:(V|o|x)

Updated: 2013-07-09

⟨*key*⟩ .default:n = {⟨*default*⟩}

Creates a ⟨*default*⟩ value for ⟨*key*⟩, which is used if no value is given. This will be used if only the key name is given, but not if a blank ⟨*value*⟩ is given:

```
\keys_define:nn { mymodule }
  {
    key .code:n    = Hello~#1,
    key .default:n = World
  }
\keys_set:nn { mymodule }
  {
    key = Fred, % Prints 'Hello Fred'
    key,        % Prints 'Hello World'
    key = ,     % Prints 'Hello '
  }
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

⟨*key*⟩ .dim_set:N = ⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

⟨*key*⟩ .fp_set:N = ⟨*floating point*⟩

Defines ⟨*key*⟩ to set ⟨*floating point*⟩ to ⟨*value*⟩ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up.

.groups:n

New: 2013-07-14

⟨*key*⟩ .groups:n = {⟨*groups*⟩}

Defines ⟨*key*⟩ as belonging to the ⟨*groups*⟩ declared. Groups provide a "secondary axis" for selectively setting keys, and are described in Section 6.

.inherit:n

New: 2016-11-22

⟨*key*⟩ .inherit:n = {⟨*parents*⟩}

Specifies that the ⟨*key*⟩ path should inherit the keys listed as ⟨*parents*⟩. For example, after setting

```
\keys_define:n { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:n { } { bar .inherit:n = foo }
```

setting

```
\keys_set:n { bar } { test = a }
```

will be equivalent to

```
\keys_set:n { foo } { test = a }
```

.initial:n
.initial:(V|o|x)

Updated: 2013-07-09

⟨key⟩ .initial:n = {⟨value⟩}

Initialises the ⟨key⟩ with the ⟨value⟩, equivalent to

\keys_set:nn {⟨module⟩} { ⟨key⟩ = ⟨value⟩ }

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

⟨key⟩ .int_set:N = ⟨integer⟩

Defines ⟨key⟩ to set ⟨integer⟩ to ⟨value⟩ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up.

.meta:n

Updated: 2013-07-10

⟨key⟩ .meta:n = {⟨keyval list⟩}

Makes ⟨key⟩ a meta-key, which will set ⟨keyval list⟩ in one go. If ⟨key⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨keys⟩ for processing (as #1).

.meta:nn

New: 2013-07-10

⟨key⟩ .meta:nn = {⟨path⟩} {⟨keyval list⟩}

Makes ⟨key⟩ a meta-key, which will set ⟨keyval list⟩ in one go using the ⟨path⟩ in place of the current one. If ⟨key⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨keys⟩ for processing (as #1).

.multichoice:

New: 2011-08-21

⟨key⟩ .multichoice:

Sets ⟨key⟩ to act as a multiple choice key. Each valid choice for ⟨key⟩ must then be created, as discussed in section 3.

.multichoices:nn
.multichoices:(Vn|on|xn)

New: 2011-08-21
Updated: 2013-07-10

⟨key⟩ .multichoices:nn {⟨choices⟩} {⟨code⟩}

Sets ⟨key⟩ to act as a multiple choice key, and defines a series ⟨choices⟩ which are implemented using the ⟨code⟩. Inside ⟨code⟩, \l_keys_choice_tl will be the name of the choice made, and \l_keys_choice_int will be the position of the choice in the list of ⟨choices⟩ (indexed from 1). Choices are discussed in detail in section 3.

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

⟨key⟩ .skip_set:N = ⟨skip⟩

Defines ⟨key⟩ to set ⟨skip⟩ to ⟨value⟩ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set:N
.tl_set:c
.tl_gset:N
.tl_gset:c

⟨key⟩ .tl_set:N = ⟨token list variable⟩

Defines ⟨key⟩ to set ⟨token list variable⟩ to ⟨value⟩. If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set_x:N
.tl_set_x:c
.tl_gset_x:N
.tl_gset_x:c

⟨key⟩ .tl_set_x:N = ⟨token list variable⟩

Defines ⟨key⟩ to set ⟨token list variable⟩ to ⟨value⟩, which will be subjected to an x-type expansion (*i.e.* using \tl_set:Nx). If the variable does not exist, it is created globally at the point that the key is set up.

.undefine:

New: 2015-07-14

⟨*key*⟩ `.undefine:`

Removes the definition of the ⟨*key*⟩ within the current scope.

.value_forbidden:n

New: 2015-07-14

⟨*key*⟩ `.value_forbidden:n = true|false`

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued. Setting the property `false` cancels the restriction.

.value_required:n

New: 2015-07-14

⟨*key*⟩ `.value_required:n = true|false`

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued. Setting the property `false` cancels the restriction.

2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `module/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

3 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. "Multiple" choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
    key / unknown  .code:n =
      \msg_error:nnxxx { mymodule } { unknown-choice }
        { key }                             % Name of choice key
        { choice-a , choice-b ,  choice-c }  % Valid choices
        { \exp_not:n {#1} }                 % Invalid choice given
    %
    %
  }
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```
\keys_define:nn { mymodule }
  {
    key .multichoices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

and

```
\keys_define:nn { mymodule }
  {
    key .multichoice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

are valid.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
  {
    key = { a , b , c } % 'key' defined as a multiple choice
  }
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
  {
    key = a ,
    key = b ,
    key = c ,
  }
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

\keys_set:nn
\keys_set:(nV|nv|no)

Updated: 2015-11-07

\keys_set:nn {⟨*module*⟩} {⟨*keyval list*⟩}

Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl

Updated: 2015-07-14

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a "full" description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset  / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_tl`, and will have been processed by `\tl_to_str:n`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_tl`, and will have been processed by `\tl_to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_tl'~to~'#1'.
  }
```

\keys_set_known:nnN
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)

New: 2011-08-23
Updated: 2017-05-27

\keys_set_known:nnN {⟨*module*⟩} {⟨*keyval list*⟩} ⟨*tl*⟩

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The \keys_set_known:nnN function parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. Any keys which are unknown are not processed further by the parser. The key–value pairs for each *unknown* key name are stored in the ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The \keys_set_known:nn version skips this stage.

Use of \keys_set_known:nnN can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

the use of \keys_set:nn attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-one   .groups:n = { first }           ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-two   .groups:n = { first }           ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-three .groups:n = { second }          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

assigns key-one and key-two to group first, key-three to group second, while key-four is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

```
\keys_set_filter:nnnN
\keys_set_filter:(nnVN|nnvN|nnoN)
\keys_set_filter:nnn
\keys_set_filter:(nnV|nnv|nno)
```

New: 2013-07-14
Updated: 2017-05-27

```
\keys_set_filter:nnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} ⟨tl⟩
```

Activates key filtering in an "opt-out" sense: keys assigned to any of the ⟨*groups*⟩ specified are ignored. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

```
\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)
```

New: 2013-07-14
Updated: 2017-05-27

```
\keys_set_groups:nnn {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩}
```

Activates key filtering in an "opt-in" sense: only keys assigned to one or more of the ⟨*groups*⟩ specified are set. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

7 Utility functions for keys

```
\keys_if_exist_p:nn ★
\keys_if_exist:nnTF ★
```

Updated: 2015-11-07

```
\keys_if_exist_p:nn {⟨module⟩} {⟨key⟩}
\keys_if_exist:nnTF {⟨module⟩} {⟨key⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*key*⟩ exists for ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩.

```
\keys_if_choice_exist_p:nnn ★
\keys_if_choice_exist:nnnTF ★
```

New: 2011-08-21
Updated: 2015-11-07

```
\keys_if_choice_exist_p:nnn {⟨module⟩} {⟨key⟩} {⟨choice⟩}
\keys_if_choice_exist:nnnTF {⟨module⟩} {⟨key⟩} {⟨choice⟩} {⟨true code⟩}
{⟨false code⟩}
```

Tests if the ⟨*choice*⟩ is defined for the ⟨*key*⟩ within the ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩/⟨*choice*⟩. The test is `false` if the ⟨*key*⟩ itself is not defined.

```
\keys_show:nn
```

Updated: 2015-08-09

```
\keys_show:nn {⟨module⟩} {⟨key⟩}
```

Displays in the terminal the information associated to the ⟨*key*⟩ for a ⟨*module*⟩, including the function which is used to actually implement it.

```
\keys_log:nn
```

New: 2014-08-22
Updated: 2015-08-09

```
\keys_log:nn {⟨module⟩} {⟨key⟩}
```

Writes in the log file the information associated to the ⟨*key*⟩ for a ⟨*module*⟩. See also `\keys_show:nn` which displays the result in the terminal.

8 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a ⟨*key–value list*⟩ into ⟨*keys*⟩ and associated ⟨*values*⟩. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double `#` tokens or expand any input. Active tokens `=` and `,` appearing at the outer level of braces are converted to category "other" (12) so that the parser does not "miss" any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
 key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn`

Updated: 2011-09-08

`\keyval_parse:NNn` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*key-value list*⟩}

Parses the ⟨*key–value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function*$_1$⟩ should take one argument, while ⟨*function*$_2$⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key–value list*⟩, ⟨*function*$_1$⟩ is used to process keys given with no value and ⟨*function*$_2$⟩ is used to process keys given with a value. The order of the ⟨*keys*⟩ in the ⟨*key–value list*⟩ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n  { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing.

Part XXI

The l3fp package: floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x/y, square root $\sqrt{x}$, and parentheses.
- Comparison operators: $x < y$, $x <= y$, $x >? y$, $x \,!= y$ *etc.*
- Boolean logic: sign $\operatorname{sign} x$, negation $!\,x$, conjunction $x \,\&\&\, y$, disjunction $x \,||\, y$, ternary operator $x \,?\, y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\operatorname{sind} x$, $\operatorname{cosd} x$, $\operatorname{tand} x$, $\operatorname{cotd} x$, $\operatorname{secd} x$, $\operatorname{cscd} x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\operatorname{asin} x$, $\operatorname{acos} x$, $\operatorname{atan} x$, $\operatorname{acot} x$, $\operatorname{asec} x$, $\operatorname{acsc} x$ giving a result in radians, and $\operatorname{asind} x$, $\operatorname{acosd} x$, $\operatorname{atand} x$, $\operatorname{acotd} x$, $\operatorname{asecd} x$, $\operatorname{acscd} x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\operatorname{sech} x$, csch, and $\operatorname{asinh} x$, $\operatorname{acosh} x$, $\operatorname{atanh} x$, $\operatorname{acoth} x$, $\operatorname{asech} x$, $\operatorname{acsch} x$.

- Extrema: $\max(x, y, \ldots)$, $\min(x, y, \ldots)$, $\operatorname{abs}(x)$.
- Rounding functions ($n = 0$ by default, $t =$ `NaN` by default): $\operatorname{trunc}(x, n)$ rounds towards zero, $\operatorname{floor}(x, n)$ rounds towards $-\infty$, $\operatorname{ceil}(x, n)$ rounds towards $+\infty$, $\operatorname{round}(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$. And *(not yet)* modulo, and “quantize”.
- Random numbers: $rand()$, $randint(m, n)$ in pdfTeX and LuaTeX engines.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\⟨type⟩_use:N`) of integer, dimension, and skip variables to floating points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {sin 3.5 /2 + 2e-3} $.
```

But in all fairness, this module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

1 Creating and initialising floating point variables

`\fp_new:N`
`\fp_new:c`
Updated: 2012-05-08

`\fp_new:N ⟨fp var⟩`

Creates a new ⟨*fp var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*fp var*⟩ is initially +0.

`\fp_const:Nn`
`\fp_const:cn`
Updated: 2012-05-08

`\fp_const:Nn ⟨fp var⟩ {⟨floating point expression⟩}`

Creates a new constant ⟨*fp var*⟩ or raises an error if the name is already taken. The ⟨*fp var*⟩ is set globally equal to the result of evaluating the ⟨*floating point expression*⟩.

`\fp_zero:N`
`\fp_zero:c`
`\fp_gzero:N`
`\fp_gzero:c`
Updated: 2012-05-08

`\fp_zero:N ⟨fp var⟩`

Sets the ⟨*fp var*⟩ to +0.

`\fp_zero_new:N`
`\fp_zero_new:c`
`\fp_gzero_new:N`
`\fp_gzero_new:c`
Updated: 2012-05-08

`\fp_zero_new:N ⟨fp var⟩`

Ensures that the ⟨*fp var*⟩ exists globally by applying `\fp_new:N` if necessary, then applies `\fp_(g)zero:N` to leave the ⟨*fp var*⟩ set to +0.

2 Setting floating point variables

`\fp_set:Nn`
`\fp_set:cn`
`\fp_gset:Nn`
`\fp_gset:cn`
Updated: 2012-05-08

`\fp_set:Nn ⟨fp var⟩ {⟨floating point expression⟩}`

Sets ⟨*fp var*⟩ equal to the result of computing the ⟨*floating point expression*⟩.

\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)

Updated: 2012-05-08

\fp_set_eq:NN ⟨*fp var*$_1$⟩ ⟨*fp var*$_2$⟩

Sets the floating point variable ⟨*fp var*$_1$⟩ equal to the current value of ⟨*fp var*$_2$⟩.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

Updated: 2012-05-08

\fp_add:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Adds the result of computing the ⟨*floating point expression*⟩ to the ⟨*fp var*⟩.

\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn

Updated: 2012-05-08

\fp_sub:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩.

3 Using floating point numbers

\fp_eval:n ★

New: 2012-05-08
Updated: 2012-07-08

\fp_eval:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values ±∞ and NaN trigger an "invalid operation" exception. This function is identical to \fp_to_decimal:n.

\fp_to_decimal:N ★
\fp_to_decimal:c ★
\fp_to_decimal:n ★

New: 2012-05-08
Updated: 2012-07-08

\fp_to_decimal:N ⟨*fp var*⟩
\fp_to_decimal:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values ±∞ and NaN trigger an "invalid operation" exception.

\fp_to_dim:N ★
\fp_to_dim:c ★
\fp_to_dim:n ★

Updated: 2016-03-22

\fp_to_dim:N ⟨*fp var*⟩
\fp_to_dim:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_to_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14}+2^{-17},2^{14}-2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. The values ±∞ and NaN trigger an "invalid operation" exception.

\fp_to_int:N ⋆
\fp_to_int:c ⋆
\fp_to_int:n ⋆

Updated: 2012-07-08

```
\fp_to_int:N ⟨fp var⟩
\fp_to_int:n {⟨floating point expression⟩}
```

Evaluates the ⟨*floating point expression*⟩, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31}+1, 2^{31}-1]$ of valid TEX integers, leading to overflow errors if used in an integer expression. The values ±∞ and `NaN` trigger an "invalid operation" exception.

\fp_to_scientific:N ⋆
\fp_to_scientific:c ⋆
\fp_to_scientific:n ⋆

New: 2012-05-08
Updated: 2016-03-22

```
\fp_to_scientific:N ⟨fp var⟩
\fp_to_scientific:n {⟨floating point expression⟩}
```

Evaluates the ⟨*floating point expression*⟩ and expresses the result in scientific notation:

⟨*optional* `-`⟩⟨*digit*⟩`.`⟨*15 digits*⟩`e`⟨*optional sign*⟩⟨*exponent*⟩

The leading ⟨*digit*⟩ is non-zero except in the case of ±0. The values ±∞ and `NaN` trigger an "invalid operation" exception. Normal category codes apply: thus the `e` is category code 11 (a letter).

\fp_to_tl:N ⋆
\fp_to_tl:c ⋆
\fp_to_tl:n ⋆

Updated: 2016-03-22

```
\fp_to_tl:N ⟨fp var⟩
\fp_to_tl:n {⟨floating point expression⟩}
```

Evaluates the ⟨*floating point expression*⟩ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from `\fp_to_scientific:n`). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`. Negative numbers start with `-`. The special values ±0, ±∞ and `NaN` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, are made up of letters.

\fp_use:N ⋆
\fp_use:c ⋆

Updated: 2012-07-08

```
\fp_use:N ⟨fp var⟩
```

Inserts the value of the ⟨*fp var*⟩ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values ±∞ and `NaN` trigger an "invalid operation" exception. This function is identical to `\fp_to_decimal:N`.

4 Floating point conditionals

\fp_if_exist_p:N ⋆
\fp_if_exist_p:c ⋆
\fp_if_exist:N*TF* ⋆
\fp_if_exist:c*TF* ⋆

Updated: 2012-05-08

```
\fp_if_exist_p:N ⟨fp var⟩
\fp_if_exist:NTF ⟨fp var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*fp var*⟩ is currently defined. This does not check that the ⟨*fp var*⟩ really is a floating point variable.

\fp_compare_p:nNn ★
\fp_compare:nNnTF ★

Updated: 2012-05-08

```
\fp_compare_p:nNn {⟨fpexpr_1⟩} ⟨relation⟩ {⟨fpexpr_2⟩}
\fp_compare:nNnTF {⟨fpexpr_1⟩} ⟨relation⟩ {⟨fpexpr_2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the ⟨*fpexpr*$_1$⟩ and the ⟨*fpexpr*$_2$⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is `NaN`, and this relation is denoted by the symbol `?`. Note that a `NaN` is distinct from any value, even another `NaN`, hence $x = x$ is not true for a `NaN`. To test if a value is `NaN`, compare it to an arbitrary number with the "not ordered" relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
  { } % <value> is nan
  { } % <value> is not nan
```

\fp_compare_p:n ★
\fp_compare:nTF ★

Updated: 2012-12-14

```
\fp_compare_p:n
  {
    ⟨fpexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨fpexpr_N⟩ ⟨relation_N⟩
    ⟨fpexpr_N+1⟩
  }
\fp_compare:nTF
  {
    ⟨fpexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨fpexpr_N⟩ ⟨relation_N⟩
    ⟨fpexpr_N+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*floating point expressions*⟩ as described for `\fp_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*intexpr*$_1$⟩ and ⟨*intexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*intexpr*$_2$⟩ and ⟨*intexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*intexpr*$_N$⟩ and ⟨*intexpr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*floating point expression*⟩ is evaluated only once. Contrarily to `\int_compare:nTF`, all ⟨*floating point expressions*⟩ are computed, even if one comparison is `false`. Two floating point numbers x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is `NaN`, and this relation is denoted by the symbol `?`. Each ⟨*relation*⟩ can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the ⟨*relation*⟩), with the restriction that the ⟨*relation*⟩ may not start with `?`, as this symbol has a different meaning (in combination with `:`) within floatin point expressions. The comparison x ⟨*relation*⟩ y is then `true` if the ⟨*relation*⟩ does not start with `!` and the actual relation (`<`, `=`, `>`, or `?`) between x and y appears within the ⟨*relation*⟩, or on the contrary if the ⟨*relation*⟩ starts with `!` and the relation between x and y does not appear within the ⟨*relation*⟩. Common choices of ⟨*relation*⟩ include `>=` (greater or equal), `!=` (not equal), `!?` or `<=>` (comparable).

5 Floating point expression loops

\fp_do_until:nNnn ☆

New: 2012-08-16

\fp_do_until:nNnn {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF. If the test is false then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is true.

\fp_do_while:nNnn ☆

New: 2012-08-16

\fp_do_while:nNnn {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF. If the test is true then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is false.

\fp_until_do:nNnn ☆

New: 2012-08-16

\fp_until_do:nNnn {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is true.

\fp_while_do:nNnn ☆

New: 2012-08-16

\fp_while_do:nNnn {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is false.

\fp_do_until:nn ☆

New: 2012-08-16

\fp_do_until:nn { ⟨*fpexpr*$_1$⟩ ⟨*relation*⟩ ⟨*fpexpr*$_2$⟩ } {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF. If the test is false then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is true.

\fp_do_while:nn ☆

New: 2012-08-16

\fp_do_while:nn { ⟨*fpexpr*$_1$⟩ ⟨*relation*⟩ ⟨*fpexpr*$_2$⟩ } {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF. If the test is true then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is false.

\fp_until_do:nn ☆

New: 2012-08-16

\fp_until_do:nn { ⟨*fpexpr*$_1$⟩ ⟨*relation*⟩ ⟨*fpexpr*$_2$⟩ } {⟨*code*⟩}

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is true.

`\fp_while_do:nn` ☆

New: 2012-08-16

`\fp_while_do:nn { ⟨fpexpr₁⟩ ⟨relation⟩ ⟨fpexpr₂⟩ } {⟨code⟩}`

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

`\fp_step_function:nnnN` ☆
`\fp_step_function:nnnc` ☆

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0]   [I saw 1.1]   [I saw 1.2]   [I saw 1.3]   [I saw 1.4]   [I saw 1.5]
```

TEXhackers note: Due to rounding, it may happen that adding the ⟨*step*⟩ to the ⟨*value*⟩ does not change the ⟨*value*⟩; such cases give an error, as they would otherwise lead to an infinite loop.

`\fp_step_inline:nnnn`

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`).

`\fp_step_variable:nnnNn`

New: 2017-04-12

```
\fp_step_variable:nnnNn
  {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}
```

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

6 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

New: 2012-05-08

Zero, with either sign.

`\c_one_fp`

New: 2012-05-08

One as an `fp`: useful for comparisons in some places.

`\c_inf_fp`
`\c_minus_inf_fp`

New: 2012-05-08

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

`\c_e_fp`

Updated: 2012-05-08

The value of the base of the natural logarithm, $\mathrm{e} = \exp(1)$.

`\c_pi_fp`

Updated: 2013-11-17

The value of π. This can be input directly in a floating point expression as `pi`.

`\c_one_degree_fp`

New: 2012-05-08
Updated: 2013-11-17

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

`\l_tmpa_fp`
`\l_tmpb_fp`

Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_fp`
`\g_tmpb_fp`

Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

"Exceptions" may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0.
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `NaN`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `NaN` value (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in LaTeX3.

To each exception we associate a "flag": `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from l3flag

By default, the "invalid operation" exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

```
\fp_trap:nn
```

New: 2012-07-19
Updated: 2017-02-13

```
\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}
```

All occurrences of the ⟨*exception*⟩ (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as ⟨*trap type*⟩, which can be

- `none`: the ⟨*exception*⟩ will be entirely ignored, and leave no trace;
- `flag`: the ⟨*exception*⟩ will turn the corresponding flag on when it occurs;
- `error`: additionally, the ⟨*exception*⟩ will halt the TeX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

```
flag␣fp_overflow
flag␣fp_underflow
flag␣fp_invalid_operation
flag␣fp_division_by_zero
```

Flags denoting the occurrence of various floating-point exceptions.

8 Viewing floating points

```
\fp_show:N
\fp_show:c
\fp_show:n
```

New: 2012-05-08
Updated: 2015-08-07

```
\fp_show:N ⟨fp var⟩
\fp_show:n {⟨floating point expression⟩}
```

Evaluates the ⟨*floating point expression*⟩ and displays the result in the terminal.

```
\fp_log:N
\fp_log:c
\fp_log:n
```

New: 2014-08-22
Updated: 2015-08-07

```
\fp_log:N ⟨fp var⟩
\fp_log:n {⟨floating point expression⟩}
```

Evaluates the ⟨*floating point expression*⟩ and writes the result in the log file.

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0, zero, with a given sign;
- $\pm\infty$, infinity, with a given sign;
- `NaN`, is "not a number", and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- ⟨*sign*⟩: a possibly empty string of `+` and `-` characters;
- ⟨*significand*⟩: a non-empty string of digits together with zero or one dot;
- ⟨*exponent*⟩ optionally: the character `e`, followed by a possibly empty string of `+` and `-` tokens, and a non-empty string of digits.

The sign of the resulting number is `+` if ⟨*sign*⟩ contains an even number of `-`, and `-` otherwise, hence, an empty ⟨*sign*⟩ denotes a non-negative input. The stored significand is obtained from ⟨*significand*⟩ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input ⟨*significand*⟩ has at most 16 digits. The stored ⟨*exponent*⟩ is obtained by combining the input ⟨*exponent*⟩ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting ⟨*exponent*⟩ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if ⟨*significand*⟩ contains no non-zero digit (*i.e.*, consists only in characters `0`, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to $+0$, but that is not guaranteed to remain true.

The ⟨*significand*⟩ must be non-empty, so `e1` and `e-1` are not valid floating point numbers. Note that the latter could be mistaken with the difference of "`e`" and 1. To avoid confusions, the base of natural logarithms cannot be input as `e` and should be input as `exp(1)` or `\c_e_fp`.

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any ⟨*sign*⟩, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `NaN`.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).

- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc.*
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\texttt{sin2pi} = \sin(2\pi) = 0,$$
$$\texttt{2\^{}2max}(3,4) = 2^{2\max(3,4)} = 256.$$

Functions are called on the value of their argument, contrarily to TEX macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ±0, and `true` otherwise, including when it is `NaN`.

`?:` `\fp_eval:n {` ⟨*operand*$_1$⟩ `?` ⟨*operand*$_2$⟩ `:` ⟨*operand*$_3$⟩ `}`

The ternary operator `?:` results in ⟨*operand*$_2$⟩ if ⟨*operand*$_1$⟩ is true, and ⟨*operand*$_3$⟩ if it is false (equal to ±0). All three ⟨*operands*⟩ are evaluated in all cases. The operator is right associative, hence

```
\fp_eval:n
  {
    1 + 3 > 4 ? 1 :
    2 + 4 > 5 ? 2 :
    3 + 5 > 6 ? 3 : 4
  }
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2+4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

`||` `\fp_eval:n {` ⟨*operand*$_1$⟩   ⟨*operand*$_2$⟩ `}`

If ⟨*operand*$_1$⟩ is true (non-zero), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

&&

```
\fp_eval:n { ⟨operand1⟩ && ⟨operand2⟩ }
```

If ⟨*operand*$_1$⟩ is false (equal to ±0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

<
=
>
?

Updated: 2013-12-14

```
\fp_eval:n
  {
    ⟨operand1⟩ ⟨relation1⟩
    ...
    ⟨operandN⟩ ⟨relationN⟩
    ⟨operandN+1⟩
  }
```

Each ⟨*relation*⟩ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to +1 if all comparisons ⟨*operand*$_i$⟩ ⟨*relation*$_j$⟩ ⟨*operand*$_{i+1}$⟩ are true, and +0 otherwise. All ⟨*operands*⟩ are evaluated in all cases. See `\fp_compare:nTF` for details.

+
-

```
\fp_eval:n { ⟨operand1⟩ + ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ - ⟨operand2⟩ }
```

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate.

*
/

```
\fp_eval:n { ⟨operand1⟩ * ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ / ⟨operand2⟩ }
```

Computes the product or the ratio of its two ⟨*operands*⟩. The "invalid operation" exception occurs for ∞/∞, $0/0$, or $0 * \infty$. "Division by zero" occurs when dividing a finite non-zero number by ±0. "Underflow" and "overflow" occur when appropriate.

+
-
!

```
\fp_eval:n { + ⟨operand⟩ }
\fp_eval:n { - ⟨operand⟩ }
\fp_eval:n { ! ⟨operand⟩ }
```

The unary `+` does nothing, the unary `-` changes the sign of the ⟨*operand*⟩, and `!` ⟨*operand*⟩ evaluates to 1 if ⟨*operand*⟩ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

**
^

```
\fp_eval:n { ⟨operand1⟩ ** ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ ^ ⟨operand2⟩ }
```

Raises ⟨*operand*$_1$⟩ to the power ⟨*operand*$_2$⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. If ⟨*operand*$_1$⟩ is negative or -0 then: the result's sign is + if the ⟨*operand*$_2$⟩ is infinite and $(-1)^p$ if the ⟨*operand*$_2$⟩ is $p/5^q$ with p, q integers; the result is +0 if `abs(`⟨*operand*$_1$⟩`)^`⟨*operand*$_2$⟩ evaluates to zero; in other cases the "invalid operation" exception occurs because the sign cannot be determined. "Division by zero" occurs when raising ±0 to a finite strictly negative power. "Underflow" and "overflow" occur when appropriate.

abs

```
\fp_eval:n { abs( ⟨fpexpr⟩ ) }
```

Computes the absolute value of the ⟨*fpexpr*⟩. This function does not raise any exception beyond those raised when computing its operand ⟨*fpexpr*⟩. See also `\fp_abs:n`.

exp

```
\fp_eval:n { exp( ⟨fpexpr⟩ ) }
```

Computes the exponential of the ⟨*fpexpr*⟩. "Underflow" and "overflow" occur when appropriate.

ln

```
\fp_eval:n { ln( ⟨fpexpr⟩ ) }
```

Computes the natural logarithm of the ⟨*fpexpr*⟩. Negative numbers have no (real) logarithm, hence the "invalid operation" is raised in that case, including for $\ln(-0)$. "Division by zero" occurs when evaluating $\ln(+0) = -\infty$. "Underflow" and "overflow" occur when appropriate.

max
min

```
\fp_eval:n { max( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ... ) }
\fp_eval:n { min( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ... ) }
```

Evaluates each ⟨*fpexpr*⟩ and computes the largest (smallest) of those. If any of the ⟨*fpexpr*⟩ is a `NaN`, the result is `NaN`. Those operations do not raise exceptions.

round
trunc
ceil
floor

New: 2013-12-14
Updated: 2015-08-08

```
\fp_eval:n { round ( ⟨fpexpr⟩ ) }
\fp_eval:n { round ( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
\fp_eval:n { round ( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ⟨fpexpr3⟩ ) }
```

Only `round` accepts a third argument. Evaluates ⟨*fpexpr*$_1$⟩ $= x$ and ⟨*fpexpr*$_2$⟩ $= n$ and ⟨*fpexpr*$_3$⟩ $= t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n}; if $n = +\infty$, this always yields x; if $n = -\infty$, this yields one of ± 0, $\pm\infty$, or `NaN`; if n is neither $\pm\infty$ nor an integer, then an "invalid operation" exception is raised. When ⟨*fpexpr*$_2$⟩ is omitted, $n = 0$, *i.e.*, ⟨*fpexpr*$_1$⟩ is rounded to an integer. The rounding direction depends on the function.

- `round` yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is `nan` or not given the even multiple is chosen ("ties to even"), if $t = \pm 0$ the multiple closest to 0 is chosen ("ties to zero"), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen ("ties towards positive/negative infinity").
- `floor` yields the largest multiple of 10^{-n} smaller or equal to x ("round towards negative infinity");
- `ceil` yields the smallest multiple of 10^{-n} greater or equal to x ("round towards positive infinity");
- `trunc` yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less that that of x ("round towards zero").

"Overflow" occurs if x is finite and the result is infinite (this can only happen if ⟨*fpexpr*$_2$⟩ < -9984).

sign

```
\fp_eval:n { sign( ⟨fpexpr⟩ ) }
```

Evaluates the ⟨*fpexpr*⟩ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0, and `NaN` for `NaN`. This operation does not raise exceptions.

```
sin     \fp_eval:n { sin( ⟨fpexpr⟩ ) }
cos     \fp_eval:n { cos( ⟨fpexpr⟩ ) }
tan     \fp_eval:n { tan( ⟨fpexpr⟩ ) }
cot     \fp_eval:n { cot( ⟨fpexpr⟩ ) }
csc     \fp_eval:n { csc( ⟨fpexpr⟩ ) }
sec     \fp_eval:n { sec( ⟨fpexpr⟩ ) }
```

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fpexpr*⟩ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate.

```
sind    \fp_eval:n { sind( ⟨fpexpr⟩ ) }
cosd    \fp_eval:n { cosd( ⟨fpexpr⟩ ) }
tand    \fp_eval:n { tand( ⟨fpexpr⟩ ) }
cotd    \fp_eval:n { cotd( ⟨fpexpr⟩ ) }
cscd    \fp_eval:n { cscd( ⟨fpexpr⟩ ) }
secd    \fp_eval:n { secd( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fpexpr*⟩ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate.

```
asin    \fp_eval:n { asin( ⟨fpexpr⟩ ) }
acos    \fp_eval:n { acos( ⟨fpexpr⟩ ) }
acsc    \fp_eval:n { acsc( ⟨fpexpr⟩ ) }
asec    \fp_eval:n { asec( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fpexpr*⟩ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate.

```
asind   \fp_eval:n { asind( ⟨fpexpr⟩ ) }
acosd   \fp_eval:n { acosd( ⟨fpexpr⟩ ) }
acscd   \fp_eval:n { acscd( ⟨fpexpr⟩ ) }
asecd   \fp_eval:n { asecd( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fpexpr*⟩ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate.

atan
acot

New: 2013-11-02

`\fp_eval:n { atan(` $\langle fpexpr \rangle$ `) }`
`\fp_eval:n { atan(` $\langle fpexpr_1 \rangle$ `,` $\langle fpexpr_2 \rangle$ `) }`
`\fp_eval:n { acot(` $\langle fpexpr \rangle$ `) }`
`\fp_eval:n { acot(` $\langle fpexpr_1 \rangle$ `,` $\langle fpexpr_2 \rangle$ `) }`

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. Only the "underflow" exception can occur.

atand
acotd

New: 2013-11-02

`\fp_eval:n { atand(` $\langle fpexpr \rangle$ `) }`
`\fp_eval:n { atand(` $\langle fpexpr_1 \rangle$ `,` $\langle fpexpr_2 \rangle$ `) }`
`\fp_eval:n { acotd(` $\langle fpexpr \rangle$ `) }`
`\fp_eval:n { acotd(` $\langle fpexpr_1 \rangle$ `,` $\langle fpexpr_2 \rangle$ `) }`

Those functions yield an angle in degrees: `atand` and `acotd` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle/\langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. Only the "underflow" exception can occur.

sqrt

New: 2013-12-14

`\fp_eval:n { sqrt(` $\langle fpexpr \rangle$ `) }`

Computes the square root of the $\langle fpexpr \rangle$. The "invalid operation" is raised when the $\langle fpexpr \rangle$ is negative; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\mathtt{NaN}} = \mathtt{NaN}$.

rand

New: 2016-12-05

```
\fp_eval:n { rand() }
```

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. Available in pdfTEX and LuaTEX engines only.

TEXhackers note: This is based on pseudo-random numbers provided by the engine's primitive \pdfuniformdeviate in pdfTEX and \uniformdeviate in LuaTEX. The underlying code in pdfTEX and LuaTEX is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than \uniformdeviate to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

The random seed can be queried using \pdfrandomseed and set using \pdfsetrandomseed (in LuaTEX \randomseed and \setrandomseed). While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

randint

New: 2016-12-05

```
\fp_eval:n { randint( ⟨fpexpr⟩ ) }
\fp_eval:n { randint( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
```

Produces a pseudo-random integer between 1 and ⟨*fpexpr*⟩ or between ⟨*fpexpr*$_1$⟩ and ⟨*fpexpr*$_2$⟩ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See rand for important comments on how these pseudo-random numbers are generated.

inf
nan

The special values $+\infty$, $-\infty$, and NaN are represented as inf, -inf and nan (see \c_inf_fp, \c_minus_inf_fp and \c_nan_fp).

pi

The value of π (see \c_pi_fp).

deg

The value of 1° in radians (see \c_one_degree_fp).

`em`
`ex`
`in`
`pt`
`pc`
`cm`
`mm`
`dd`
`cc`
`nd`
`nc`
`bp`
`sp`

Those units of measurement are equal to their values in `pt`, namely

$$
\begin{aligned}
1\texttt{in} &= 72.27\texttt{pt}\\
1\texttt{pt} &= 1\texttt{pt}\\
1\texttt{pc} &= 12\texttt{pt}\\
1\texttt{cm} &= \frac{1}{2.54}\texttt{in} = 28.45275590551181\texttt{pt}\\
1\texttt{mm} &= \frac{1}{25.4}\texttt{in} = 2.845275590551181\texttt{pt}\\
1\texttt{dd} &= 0.376065\texttt{mm} = 1.07000856496063\texttt{pt}\\
1\texttt{cc} &= 12\texttt{dd} = 12.84010277952756\texttt{pt}\\
1\texttt{nd} &= 0.375\texttt{mm} = 1.066978346456693\texttt{pt}\\
1\texttt{nc} &= 12\texttt{nd} = 12.80374015748031\texttt{pt}\\
1\texttt{bp} &= \frac{1}{72}\texttt{in} = 1.00375\texttt{pt}\\
1\texttt{sp} &= 2^{-16}\texttt{pt} = 1.52587890625e-5\texttt{pt}.
\end{aligned}
$$

The values of the (font-dependent) units `em` and `ex` are gathered from TeX when the surrounding floating point expression is evaluated.

`true`
`false`

Other names for 1 and +0.

`\fp_abs:n` ⋆

New: 2012-05-14
Updated: 2012-07-08

`\fp_abs:n {⟨floating point expression⟩}`

Evaluates the ⟨*floating point expression*⟩ as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `abs()` can be used.

`\fp_max:nn` ⋆
`\fp_min:nn` ⋆

New: 2012-09-26

`\fp_max:nn {⟨fp expression 1⟩} {⟨fp expression 2⟩}`

Evaluates the ⟨*floating point expressions*⟩ as described for `\fp_eval:n` and leaves the resulting larger (`max`) or smaller (`min`) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `max()` and `min()` can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among `0123456789_+`, or if it receives a TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I'll try to time-order the items.

- Decide what exponent range to consider.
- Support signalling `nan`.

- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?

- `\fp_format:nn {`⟨*fpexpr*⟩`} {`⟨*format*⟩`}`, but what should ⟨*format*⟩ be? More general pretty printing?

- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?

- Add $\log(x, b)$ for logarithm of x in base b.

- `hypot` (Euclidean length). Cartesian-to-polar transform.

- Hyperbolic functions `cosh`, `sinh`, `tanh`.

- Inverse hyperbolics.

- Base conversion, input such as `0xAB.CDEF`.

- Factorial (not with `!`), gamma function.

- Improve coefficients of the `sin` and `tan` series.

- Treat upper and lower case letters identically in identifiers, and ignore underscores.

- Add an `array(1,2,3)` and `i=complex(0,1)`.

- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?

- Provide `\fp_if_nan:nTF`, and an `isnan` function?

- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs.

- Check that functions are monotonic when they should.

- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.

- Logarithms of numbers very close to 1 are inaccurate.

- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0, not $+0$.

- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.

- `0e9999999999` gives a TeX "number too large" error.

- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that l3trial/l3fp-types introduces tools for adding new types.

- In subsection 9.1, write a grammar.

- It would be nice if the parse auxiliaries for each operation were set up in the corresponding module, rather than centralizing in l3fp-parse.
- Some functions should get an _o ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the ternary set of functions is ugly.
- There are many ~ missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (l3fp-basics).
- Understand and document __fp_basics_pack_weird_low:NNNNw and __fp_basics_pack_weird_high:NNNNNNNNw better. Move the other basics_pack auxiliaries to l3fp-aux under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TEX fp packages, the international standards,...
- Also take into account the "inexact" exception?
- Support multi-character prefix operators (*e.g.*, @/ or whatever)?

Part XXII
The l3sort package
Sorting functions

1 Controlling sorting

LaTeX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
  {
    \int_compare:nNnTF { #1 } > { #2 }
      { \sort_return_swapped: }
      { \sort_return_same: }
  }
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a ⟨*comparison code*⟩ consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a ⟨*comparison code*⟩ consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

TeXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaTeX), depending on what other packages are loaded.

Internally, the code from l3sort stores items in `\toks` registers allocated locally. Thus, the ⟨*comparison code*⟩ should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

Part XXIII

The l3tl-build package: building token lists

1 l3tl-build documentation

This module provides no user function: it is meant for kernel use only.

There are two main ways of building token lists from individual tokens. Either in one go within an `x`-expanding assignment, or by repeatedly using `\tl_put_right:Nn`. The first method takes a linear time, but only allows expandable operations. The second method takes a time quadratic in the length of the token list, but allows expandable and non-expandable operations.

The goal of this module is to provide functions to build a token list piece by piece in linear time, while allowing non-expandable operations. This is achieved by abusing `\toks`: adding some tokens to the token list is done by storing them in a free token register (time $O(1)$ for each such operation). Those token registers are only put together at the end, within an `x`-expanding assignment, which takes a linear time.[5] Of course, all this must be done in a group: we can't go and clobber the values of legitimate `\toks` used by LaTeX 2ε.

Since none of the current applications need the ability to insert material on the left of the token list, I have not implemented that. This could be done for instance by using odd-numbered `\toks` for the left part, and even-numbered `\toks` for the right part.

1.1 Internal functions

```
\__tl_build:Nw
\__tl_gbuild:Nw
\__tl_build_x:Nw
\__tl_gbuild_x:Nw
```

```
\__tl_build:Nw ⟨tl var⟩ ...
\__tl_build_one:n {⟨tokens₁⟩} ...
\__tl_build_one:n {⟨tokens₂⟩} ...
...
\__tl_build_end:
```

Defines the ⟨*tl var*⟩ to contain the contents of ⟨*tokens1*⟩ followed by ⟨*tokens2*⟩, *etc.* This is built in such a way to be more efficient than repeatedly using `\tl_put_right:Nn`. The code in "`...`" does not need to be expandable. The commands `\__tl_build:Nw` and `\__tl_build_end:` start and end a group. The assignment to the ⟨*tl var*⟩ occurs just after the end of that group, using `\tl_set:Nn`, `\tl_gset:Nn`, `\tl_set:Nx`, or `\tl_gset:Nx`.

```
\__tl_build_one:n
\__tl_build_one:(o|x)
```

```
\__tl_build_one:n {⟨tokens⟩}
```

This function may only be used within the scope of a `\__tl_build:Nw` function. It adds the ⟨*tokens*⟩ on the right of the current token list.

```
\__tl_build_end:
```

Ends the scope started by `\__tl_build:Nw`, and performs the relevant assignment.

[5] If we run out of token registers, then the currently filled-up `\toks` are put together in a temporary token list, and cleared, and we ultimately use `\tl_put_right:Nx` to put those chunks together. Hence the true asymptotic is quadratic, with a very small constant.

Part XXIV

The l3tl-analysis package: analysing token lists

1 l3tl-analysis documentation

This module mostly provides internal functions for use in the l3regex module. However, it provides as a side-effect a user debugging function, very similar to the \ShowTokens macro from the ted package.

```
\tl_show_analysis:N
\tl_show_analysis:n
```

New: 2017-05-26

\tl_show_analysis:n {⟨*token list*⟩}

Displays to the terminal the detailed decomposition of the ⟨*token list*⟩ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

Part XXV

The l3regex package: regular expressions in TEX

1 Regular expressions

The l3regex package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that TEX manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text "`This cat.`", where the first occurrence of "`at`" was replaced by "`is`". A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any "word" character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_const:Nn`. For example,

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\c_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses "capture" the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

1.1 Syntax of regular expressions

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word "Cat" capitalized in this way, but also matches the beginning of the word "Cattle": use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among "a", "b", "c"; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).

- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).

- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.

- `\_[^\_]*\_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `\_.*?\_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.

- `[+-]?\d+` matches an explicit integer with at most one sign.

- `[+-\␣]*\d+\␣*` matches an explicit integer with any number of + and − signs, with spaces allowed except within the mantissa, and sourrounded by spaces.

- `[+-\␣]*(\d+|\d*\.\d+)\␣*` matches an explict integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.

- `[+-\␣]*(\d+|\d*\.\d+)\␣*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\␣*` matches an explicit dimension with any unit that TEX knows, where `(?i)` means to treat lowercase and uppercase letters identically.

- `[+-\␣]*((?i)nan|inf|(\d+|\d*\.\d+)(\␣*e[+-\␣]*\d+)?)\␣*` matches an explicit floating point number or the special values `nan` and `inf` (with signs).

- `[+-\␣]*(\d+|\cC.)\␣*` matches an explicit integer or control sequence (without checking whether it is an integer variable).

- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[+-\(]*\d+\)*([+-*/][+-\(]*\d+\)*)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A`–`Z`, `a`–`z`, `0`–`9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, … have special meanings;

- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`);

- spaces should always be escaped (even in character classes);

- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into TeX under normal category codes. For instance, `\\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{`⟨*regex*⟩`}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

Character types.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^^I\^^J\^^L\^^M]`.

`\v` Any vertical space character, equivalent to `[\^^J\^^K\^^L\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alpha-numerics and underscore, equivalent to `[A-Za-z0-9\_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

`[^...]` Negative character class. Matches any token other than the specified characters.

`x-y` Within a character class, this denotes a range (can be used with escaped characters).

`[:`⟨*name*⟩`:]` Within a character class (one more set of brackets), this denotes the POSIX character class ⟨*name*⟩, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

`[:^`⟨*name*⟩`:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

Quantifiers (repetition).

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

`+?` 1 or more, lazy.

`{`n`}` Exactly n.

`{`n`,}` n or more, greedy.

`{`n`,}?` n or more, lazy.

`{`n, m`}` At least n, no more than m, greedy.

`{`n, m`}?` At least n, no more than m, lazy.

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^`or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\l_tmpa_int` holding the value 1.

Alternation and capturing groups.

`A|B|C` Either one of `A`, `B`, or `C`.

`(...)` Capturing group.

`(?:...)` Non-capturing group.

`(?|...)` Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- `C` for control sequences;
- `B` for begin-group tokens;
- `E` for end-group tokens;
- `M` for math shift;
- `T` for alignment tab tokens;
- `P` for macro parameter tokens;
- `U` for superscript tokens (up);
- `D` for subscript tokens (down);
- `S` for spaces;
- `L` for letters;
- `O` for others; and
- `A` for active characters.

The `\c` escape sequence is used as follows.

`\c{`⟨*regex*⟩`}` A control sequence whose csname matches the ⟨*regex*⟩, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category `X` (any of `CBEMTPUDSLOA`. For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from `A` to `F` with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO\*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters.

Namely, `\u{`⟨*tl var name*⟩`}` matches the exact contents of the token list ⟨*tl var*⟩. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying `A`–`Z` with `a`–`z`; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[Y-\\]` is equivalent to `[YZ\[\\yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[^6-9]` are equivalent.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the "best" match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

1.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2`, ..., `\9` and `\g{`⟨*number*⟩`}`;
- `\␣` inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\x{hhh}` correspond to single characters as in regular expressions;
- `\c{`⟨*cs name*⟩`}` inserts a control sequence;
- `\c`⟨*category*⟩⟨*character*⟩ (see below);
- `\u{`⟨*tl var name*⟩`}` inserts the contents of the ⟨*tl var*⟩ (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for TeX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(ell--el)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n-th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through `\␣` have category code 10, while spaces inserted through `\x20` or `\x{20}` have category code 12. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters "..." with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{`⟨*text*⟩`}` Produces the control sequence with csname ⟨*text*⟩. The ⟨*text*⟩ may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{`⟨*tl var name*⟩`}` allows to insert the contents of the token list with name ⟨*tl var name*⟩ directly into the replacement, giving an easier control of category codes. When nested in `\c{`...`}` and `\u{`...`}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{l_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

1.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the l3regex module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N`

New: 2017-05-26

`\regex_new:N ⟨regex var⟩`

Creates a new ⟨*regex var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*regex var*⟩ is initially such that it never matches.

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_const:Nn`

New: 2017-05-26

`\regex_set:Nn ⟨regex var⟩ {⟨regex⟩}`

Stores a compiled version of the ⟨*regular expression*⟩ in the ⟨*regex var*⟩. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which never change.

`\regex_show:n`
`\regex_show:N`

New: 2017-05-26

`\regex_show:n {⟨regex⟩}`

Shows how l3regex interprets the ⟨*regex*⟩. For instance, `\regex_show:n {\A X|Y}` shows

```
+-branch
  anchor at start (\A)
  char code 88
+-branch
  char code 89
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

1.4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a "standard" regular expression, while the later require a compiled expression as generated by `\regex_(g)set:Nn`.

`\regex_match:nnTF`
`\regex_match:NnTF`

New: 2017-05-26

`\regex_match:nnTF {⟨regex⟩} {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*regular expression*⟩ matches any part of the ⟨*token list*⟩. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves `TRUE` then `FALSE` in the input stream.

\regex_count:nnN
\regex_count:NnN

New: 2017-05-26

\regex_count:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*int var*⟩

Sets ⟨*int var*⟩ within the current TeX group level equal to the number of times ⟨*regular expression*⟩ appears in ⟨*token list*⟩. The search starts by finding the left-most longest match, respecting greedy and ungreedy operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbb } \l_foo_int
```

results in \l_foo_int taking the value 5.

1.5 Submatch extraction

\regex_extract_once:nnNTF
\regex_extract_once:NnNTF

New: 2017-05-26

\regex_extract_once:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_once:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Finds the first match of the ⟨*regular expression*⟩ in the ⟨*token list*⟩. If it exists, the match is stored as the first item of the ⟨*seq var*⟩, and further items are the contents of capturing groups, in the order of their opening parenthesis. The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
  { true } { false }
```

Then the regular expression (anchored at the start with \A and at the end with \Z) must match the whole token list. The first capturing group, (La)?, matches La, and the second capturing group, (!*), matches !!!. Thus, \l_foo_seq contains as a result the items {LaTeX!!!}, {La}, and {!!!}, and the true branch is left in the input stream. Note that the n-th item of \l_foo_seq, as obtained using \seq_item:Nn, correspond to the submatch numbered $(n-1)$ in functions such as \regex_replace_once:nnN.

\regex_extract_all:nnNTF
\regex_extract_all:NnNTF

New: 2017-05-26

\regex_extract_all:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_all:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Finds all matches of the ⟨*regular expression*⟩ in the ⟨*token list*⟩, and stores all the submatch information in a single sequence (concatenating the results of multiple \regex_extract_once:nnN calls). The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
  { true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items {Hello} and {world}, and the true branch is left in the input stream.

\regex_split:nnN*TF*
\regex_split:NnN*TF*

New: 2017-05-26

\regex_split:nnN {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_split:nnNTF {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Splits the ⟨*token list*⟩ into a sequence of parts, delimited by matches of the ⟨*regular expression*⟩. If the ⟨*regular expression*⟩ has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to ⟨*seq var*⟩ is local. If no match is found the resulting ⟨*seq var*⟩ has the ⟨*token list*⟩ as its sole item. If the ⟨*regular expression*⟩ matches the empty token list, then the ⟨*token list*⟩ is split into single tokens. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
  { true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

1.6 Replacement

\regex_replace_once:nnN*TF*
\regex_replace_once:NnN*TF*

New: 2017-05-26

\regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_once:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for the ⟨*regular expression*⟩ in the ⟨*token list*⟩ and replaces the first match with the ⟨*replacement*⟩. The result is assigned locally to ⟨*tl var*⟩. In the ⟨*replacement*⟩, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.*

\regex_replace_all:nnN*TF*
\regex_replace_all:NnN*TF*

New: 2017-05-26

\regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_all:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Replaces all occurrences of the `\regular expression` in the ⟨*token list*⟩ by the ⟨*replacement*⟩, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to ⟨*tl var*⟩.

1.7 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?

  Additional error-checking to come.

- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.

- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `\__regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build .␣. once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `\__regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `current_state` and `current_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.
- Use an array rather than `\l__regex_balance_tl` to build `\__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `\__regex_action_free:n`.
- Optimize the use of `\__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better "show" for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.

- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTEX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any "extended" Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.
- Provide a syntax such as `\ur{l_my_regex}` to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
- Allowing `\u{l_my_tl}` in more places, for instance as the number of repetitions in a quantifier.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break:` and then of playing well with `\tl_map_break:` called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: TEX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to TEX's own `\^^x`.

- Comments: TeX already has its own system for comments.
- `\Q...\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: XeTeX and LuaTeX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Part XXVI
The l3box package
Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

1 Creating and initialising boxes

`\box_new:N`
`\box_new:c`

`\box_new:N ⟨box⟩`

Creates a new ⟨*box*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*box*⟩ is initially void.

`\box_clear:N`
`\box_clear:c`
`\box_gclear:N`
`\box_gclear:c`

`\box_clear:N ⟨box⟩`

Clears the content of the ⟨*box*⟩ by setting the box equal to `\c_empty_box`.

`\box_clear_new:N`
`\box_clear_new:c`
`\box_gclear_new:N`
`\box_gclear_new:c`

`\box_clear_new:N ⟨box⟩`

Ensures that the ⟨*box*⟩ exists globally by applying `\box_new:N` if necessary, then applies `\box_(g)clear:N` to leave the ⟨*box*⟩ empty.

`\box_set_eq:NN`
`\box_set_eq:(cN|Nc|cc)`
`\box_gset_eq:NN`
`\box_gset_eq:(cN|Nc|cc)`

`\box_set_eq:NN ⟨box1⟩ ⟨box2⟩`

Sets the content of ⟨*box*$_1$⟩ equal to that of ⟨*box*$_2$⟩.

`\box_set_eq_clear:NN`
`\box_set_eq_clear:(cN|Nc|cc)`

`\box_set_eq_clear:NN ⟨box1⟩ ⟨box2⟩`

Sets the content of ⟨*box*$_1$⟩ within the current TeX group equal to that of ⟨*box*$_2$⟩, then clears ⟨*box*$_2$⟩ globally.

`\box_gset_eq_clear:NN`
`\box_gset_eq_clear:(cN|Nc|cc)`

`\box_gset_eq_clear:NN ⟨box1⟩ ⟨box2⟩`

Sets the content of ⟨*box*$_1$⟩ equal to that of ⟨*box*$_2$⟩, then clears ⟨*box*$_2$⟩. These assignments are global.

`\box_if_exist_p:N` ★
`\box_if_exist_p:c` ★
`\box_if_exist:NTF` ★
`\box_if_exist:cTF` ★

New: 2012-03-03

`\box_if_exist_p:N ⟨box⟩`
`\box_if_exist:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*box*⟩ is currently defined. This does not check that the ⟨*box*⟩ really is a box.

2 Using boxes

\box_use:N
\box_use:c

\box_use:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting.

> **TEXhackers note:** This is the TEX primitive \copy.

\box_use_drop:N
\box_use_drop:c

\box_use_drop:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting. The ⟨*box*⟩ is then cleared at the group level the box was set at, *i.e.* the current content is "dropped" entirely. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
  \group_begin:
    \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of \box_show:N will show an entirely cleared (void) box, and the second will show the letter A in the box.

This function is useful as boxes can contain an open-ended amount of material. As such, they can have a significant memory impact on TEX. At the same time, it is often the case that once a box has been inserted, it is no longer needed at all. Using \box_use_drop:N in these circumstances therefore offers improved memory use and performance. It should therefore be preferred over \box_use:N where it is clear that the content is no longer needed in the variable.

> **TEXhackers note:** This is the TEX primitive \box.

\box_move_right:nn
\box_move_left:nn

\box_move_right:nn {⟨*dimexpr*⟩} {⟨*box function*⟩}

This function operates in vertical mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced horizontally by the given ⟨*dimexpr*⟩ from the reference point for typesetting, to the right or left as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a "raw" box specification such as \vbox:n { xyz }.

\box_move_up:nn
\box_move_down:nn

\box_move_up:nn {⟨*dimexpr*⟩} {⟨*box function*⟩}

This function operates in horizontal mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced vertically by the given ⟨*dimexpr*⟩ from the reference point for typesetting, up or down as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a "raw" box specification such as \vbox:n { xyz }.

3 Measuring and setting box dimensions

\box_dp:N
\box_dp:c

\box_dp:N ⟨*box*⟩

Calculates the depth (below the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

> **TeXhackers note:** This is the TeX primitive \dp.

\box_ht:N
\box_ht:c

\box_ht:N ⟨*box*⟩

Calculates the height (above the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

> **TeXhackers note:** This is the TeX primitive \ht.

\box_wd:N
\box_wd:c

\box_wd:N ⟨*box*⟩

Calculates the width of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

> **TeXhackers note:** This is the TeX primitive \wd.

\box_set_dp:Nn
\box_set_dp:cn
Updated: 2011-10-22

\box_set_dp:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the depth (below the baseline) of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

\box_set_ht:Nn
\box_set_ht:cn
Updated: 2011-10-22

\box_set_ht:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the height (above the baseline) of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

\box_set_wd:Nn
\box_set_wd:cn
Updated: 2011-10-22

\box_set_wd:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the width of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

4 Box conditionals

\box_if_empty_p:N ⋆
\box_if_empty_p:c ⋆
\box_if_empty:N*TF* ⋆
\box_if_empty:c*TF* ⋆

\box_if_empty_p:N ⟨*box*⟩
\box_if_empty:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is a empty (equal to \c_empty_box).

\box_if_horizontal_p:N ⋆
\box_if_horizontal_p:c ⋆
\box_if_horizontal:N*TF* ⋆
\box_if_horizontal:c*TF* ⋆

\box_if_horizontal_p:N ⟨*box*⟩
\box_if_horizontal:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is a horizontal box.

\box_if_vertical_p:N ⋆
\box_if_vertical_p:c ⋆
\box_if_vertical:NTF ⋆
\box_if_vertical:cTF ⋆

`\box_if_vertical_p:N ⟨box⟩`
`\box_if_vertical:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if ⟨*box*⟩ is a vertical box.

5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

`\box_set_to_last:N ⟨box⟩`

Sets the ⟨*box*⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨*box*⟩ is always void as it is not possible to recover the last added item.

6 Constant boxes

\c_empty_box

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

TEXhackers note: At the TEX level this is a void box.

7 Scratch boxes

\l_tmpa_box
\l_tmpb_box

Updated: 2012-11-04

Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_box
\g_tmpb_box

Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Viewing box contents

\box_show:N
\box_show:c

Updated: 2012-05-11

`\box_show:N ⟨box⟩`

Shows full details of the content of the ⟨*box*⟩ in the terminal.

\box_show:Nnn
\box_show:cnn

New: 2012-05-11

`\box_show:Nnn ⟨box⟩ ⟨intexpr1⟩ ⟨intexpr2⟩`

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ group levels.

\box_log:N
\box_log:c

New: 2012-05-11

`\box_log:N` ⟨*box*⟩

Writes full details of the content of the ⟨*box*⟩ to the log.

\box_log:Nnn
\box_log:cnn

New: 2012-05-11

`\box_log:Nnn` ⟨*box*⟩ ⟨*intexpr*$_1$⟩ ⟨*intexpr*$_2$⟩

Writes the contents of ⟨*box*⟩ to the log, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ group levels.

9 Boxes and color

All LaTeX3 boxes are "color safe": a color set inside the box stops applying after the end of the box has occurred.

10 Horizontal mode boxes

\hbox:n

Updated: 2017-04-05

`\hbox:n {`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ into a horizontal box of natural width and then includes this box in the current list for typesetting.

\hbox_to_wd:nn

Updated: 2017-04-05

`\hbox_to_wd:nn {`⟨*dimexpr*⟩`} {`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ into a horizontal box of width ⟨*dimexpr*⟩ and then includes this box in the current list for typesetting.

\hbox_to_zero:n

Updated: 2017-04-05

`\hbox_to_zero:n {`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width and then includes this box in the current list for typesetting.

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn

Updated: 2017-04-05

`\hbox_set:Nn` ⟨*box*⟩ `{`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩.

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

Updated: 2017-04-05

`\hbox_set_to_wd:Nnn` ⟨*box*⟩ `{`⟨*dimexpr*⟩`} {`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩.

\hbox_overlap_right:n

Updated: 2017-04-05

`\hbox_overlap_right:n {`⟨*contents*⟩`}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the right of the insertion point.

`\hbox_overlap_left:n`

Updated: 2017-04-05

`\hbox_overlap_left:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the left of the insertion point.

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_set_end:`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_gset_end:`

Updated: 2017-04-05

`\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:`

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

`\hbox_set_to_wd:Nnw`
`\hbox_set_to_wd:cnw`
`\hbox_gset_to_wd:Nnw`
`\hbox_gset_to_wd:cnw`

New: 2017-06-08

`\hbox_set_to_wd:Nnw ⟨box⟩ {⟨dimexpr⟩} ⟨contents⟩ \hbox_set_end:`

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩. In contrast to `\hbox_set_to_wd:Nnn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument

`\hbox_unpack:N`
`\hbox_unpack:c`

`\hbox_unpack:N ⟨box⟩`

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

TEXhackers note: This is the TEX primitive `\unhcopy`.

`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

`\hbox_unpack_clear:N ⟨box⟩`

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The ⟨*box*⟩ is then cleared globally.

TEXhackers note: This is the TEX primitive `\unhbox`.

11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

`\vbox:n`

Updated: 2017-04-05

`\vbox:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting.

`\vbox_top:n`

Updated: 2017-04-05

`\vbox_top:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

\vbox_to_ht:nn

Updated: 2017-04-05

`\vbox_to_ht:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of height ⟨*dimexpr*⟩ and then includes this box in the current list for typesetting.

\vbox_to_zero:n

Updated: 2017-04-05

`\vbox_to_zero:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of zero height and then includes this box in the current list for typesetting.

\vbox_set:Nn
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

Updated: 2017-04-05

`\vbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩.

\vbox_set_top:Nn
\vbox_set_top:cn
\vbox_gset_top:Nn
\vbox_gset_top:cn

Updated: 2017-04-05

`\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. The baseline of the box is equal to that of the *first* item added to the box.

\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn

Updated: 2017-04-05

`\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩.

\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:

Updated: 2017-04-05

`\vbox_set:Nw ⟨box⟩ ⟨contents⟩ \vbox_set_end:`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw

New: 2017-06-08

`\vbox_set_to_wd:Nnw ⟨box⟩ {⟨dimexpr⟩} ⟨contents⟩ \vbox_set_end:`

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩. In contrast to `\vbox_set_to_ht:Nnn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument

\vbox_set_split_to_ht:NNn

Updated: 2011-10-22

`\vbox_set_split_to_ht:NNn ⟨box₁⟩ ⟨box₂⟩ {⟨dimexpr⟩}`

Sets ⟨*box*$_1$⟩ to contain material to the height given by the ⟨*dimexpr*⟩ by removing content from the top of ⟨*box*$_2$⟩ (which must be a vertical box).

TeXhackers note: This is the TeX primitive `\vsplit`.

`\vbox_unpack:N`
`\vbox_unpack:c`

`\vbox_unpack:N ⟨box⟩`

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

> **TEXhackers note:** This is the TEX primitive `\unvcopy`.

`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

`\vbox_unpack:N ⟨box⟩`

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The ⟨*box*⟩ is then cleared globally.

> **TEXhackers note:** This is the TEX primitive `\unvbox`.

12 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TEX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

`\box_autosize_to_wd_and_ht:Nnn`
`\box_autosize_to_wd_and_ht:Nnn`

New: 2017-04-04

`\box_autosize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}`

Resizes the ⟨*box*⟩ to fit within the given ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically); both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the height only: it does not include any depth. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. The final size of the ⟨*box*⟩ is the smaller of {⟨*x-size*⟩} and {⟨*y-size*⟩}, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

`\box_autosize_to_wd_and_ht_plus_dp:Nnn`
`\box_autosize_to_wd_and_ht_plus_dp:Nnn`

New: 2017-04-04

`\box_autosize_to_wd_and_ht_plus_dp:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}`

Resizes the ⟨*box*⟩ to fit within the given ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically); both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the total vertical size (height plus depth). The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. The final size of the ⟨*box*⟩ is the smaller of {⟨*x-size*⟩} and {⟨*y-size*⟩}, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

\box_resize_to_ht:Nn
\box_resize_to_ht:cn

`\box_resize_to_ht:Nn ⟨box⟩ {⟨y-size⟩}`

Resizes the ⟨*box*⟩ to ⟨*y-size*⟩ (vertically), scaling the horizontal size by the same amount; ⟨*y-size*⟩ is a dimension expression. The ⟨*y-size*⟩ is the height only: it does not include any depth. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative ⟨*y-size*⟩ causes the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn

`\box_resize_to_ht_plus_dp:Nn ⟨box⟩ {⟨y-size⟩}`

Resizes the ⟨*box*⟩ to ⟨*y-size*⟩ (vertically), scaling the horizontal size by the same amount; ⟨*y-size*⟩ is a dimension expression. The ⟨*y-size*⟩ is the total vertical size (height plus depth). The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative ⟨*y-size*⟩ causes the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

\box_resize_to_wd:Nn
\box_resize_to_wd:cn

`\box_resize_to_wd:Nn ⟨box⟩ {⟨x-size⟩}`

Resizes the ⟨*box*⟩ to ⟨*x-size*⟩ (horizontally), scaling the vertical size by the same amount; ⟨*x-size*⟩ is a dimension expression. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative ⟨*x-size*⟩ causes the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*x-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn

New: 2014-07-03

`\box_resize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}`

Resizes the ⟨*box*⟩ to ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically): both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the height only and does not include any depth. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa.* The resizing applies within the current TEX group level.

\box_resize_to_wd_and_ht_plus_dp:Nnn
\box_resize_to_wd_and_ht_plus_dp:cnn

New: 2017-04-06

```
\box_resize_to_wd_and_ht_plus_dp:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}
```

Resizes the ⟨*box*⟩ to ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically): both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the total vertical size (height plus depth). The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TEX group level.

\box_rotate:Nn
\box_rotate:cn

```
\box_rotate:Nn ⟨box⟩ {⟨angle⟩}
```

Rotates the ⟨*box*⟩ by ⟨*angle*⟩ (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the rotation is applied. The rotation applies within the current TEX group level.

\box_scale:Nnn
\box_scale:cnn

```
\box_scale:Nnn ⟨box⟩ {⟨x-scale⟩} {⟨y-scale⟩}
```

Scales the ⟨*box*⟩ by factors ⟨*x-scale*⟩ and ⟨*y-scale*⟩ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the scaling is applied. Negative scalings cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-scale*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TEX group level.

13 Primitive box conditionals

\if_hbox:N ⋆

```
\if_hbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a horizontal box.

> **TEXhackers note:** This is the TEX primitive `\ifhbox`.

\if_vbox:N ⋆

```
\if_vbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a vertical box.

> **TEXhackers note:** This is the TEX primitive `\ifvbox`.

`\if_box_empty:N` ⋆

```
\if_box_empty:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is an empty (void) box.

> **TeXhackers note:** This is the TeX primitive `\ifvoid`.

Part XXVII
The l3coffins package
Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

1 Creating and initialising coffins

\coffin_new:N
\coffin_new:c

New: 2011-08-17

```
\coffin_new:N ⟨coffin⟩
```

Creates a new ⟨*coffin*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*coffin*⟩ is initially empty.

\coffin_clear:N
\coffin_clear:c

New: 2011-08-17

```
\coffin_clear:N ⟨coffin⟩
```

Clears the content of the ⟨*coffin*⟩ within the current TEX group level.

\coffin_set_eq:NN
\coffin_set_eq:(Nc|cN|cc)

New: 2011-08-17

```
\coffin_set_eq:NN ⟨coffin1⟩ ⟨coffin2⟩
```

Sets both the content and poles of ⟨*coffin*$_1$⟩ equal to those of ⟨*coffin*$_2$⟩ within the current TEX group level.

\coffin_if_exist_p:N ★
\coffin_if_exist_p:c ★
\coffin_if_exist:NTF ★
\coffin_if_exist:cTF ★

New: 2012-06-20

```
\coffin_if_exist_p:N ⟨box⟩
\coffin_if_exist:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*coffin*⟩ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current TEX group level.

\hcoffin_set:Nn
\hcoffin_set:cn

New: 2011-08-17
Updated: 2011-09-03

```
\hcoffin_set:Nn ⟨coffin⟩ {⟨material⟩}
```

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:

New: 2011-09-10

```
\hcoffin_set:Nw ⟨coffin⟩ ⟨material⟩ \hcoffin_set_end:
```

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn
\vcoffin_set:cnn

New: 2011-08-17
Updated: 2012-05-22

```
\vcoffin_set:Nnn ⟨coffin⟩ {⟨width⟩} {⟨material⟩}
```

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:

New: 2011-09-10
Updated: 2012-05-22

```
\vcoffin_set:Nnw ⟨coffin⟩ {⟨width⟩} ⟨material⟩ \vcoffin_set_end:
```

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn

New: 2012-07-20

```
\coffin_set_horizontal_pole:Nnn ⟨coffin⟩
  {⟨pole⟩} {⟨offset⟩}
```

Sets the ⟨*pole*⟩ to run horizontally through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the bottom edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn

New: 2012-07-20

```
\coffin_set_vertical_pole:Nnn ⟨coffin⟩ {⟨pole⟩} {⟨offset⟩}
```

Sets the ⟨*pole*⟩ to run vertically through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the left-hand edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

3 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)

```
\coffin_attach:NnnNnnnn
  ⟨coffin1⟩ {⟨coffin1-pole1⟩} {⟨coffin1-pole2⟩}
  ⟨coffin2⟩ {⟨coffin2-pole1⟩} {⟨coffin2-pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

This function attaches ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ is not altered, *i.e.* ⟨*coffin*$_2$⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

```
\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)
```

```
\coffin_join:NnnNnnnn
  ⟨coffin1⟩ {⟨coffin1-pole1⟩} {⟨coffin1-pole2⟩}
  ⟨coffin2⟩ {⟨coffin2-pole1⟩} {⟨coffin2-pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

This function joins ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn ⟨coffin⟩ {⟨pole1⟩} {⟨pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

Typesetting is carried out by first calculating ⟨*handle*⟩, the point of intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the ⟨*handle*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

4 Measuring coffins

```
\coffin_dp:N
\coffin_dp:c
```

```
\coffin_dp:N ⟨coffin⟩
```

Calculates the depth (below the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

```
\coffin_ht:N
\coffin_ht:c
```

```
\coffin_ht:N ⟨coffin⟩
```

Calculates the height (above the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

```
\coffin_wd:N
\coffin_wd:c
```

```
\coffin_wd:N ⟨coffin⟩
```

Calculates the width of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

5 Coffin diagnostics

```
\coffin_display_handles:Nn
\coffin_display_handles:cn
```

Updated: 2011-09-02

```
\coffin_display_handles:Nn ⟨coffin⟩ {⟨color⟩}
```

This function first calculates the intersections between all of the ⟨*poles*⟩ of the ⟨*coffin*⟩ to give a set of ⟨*handles*⟩. It then prints the ⟨*coffin*⟩ at the current location in the source, with the position of the ⟨*handles*⟩ marked on the coffin. The ⟨*handles*⟩ are labelled as part of this process: the locations of the ⟨*handles*⟩ and the labels are both printed in the ⟨*color*⟩ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn

Updated: 2011-09-02

`\coffin_mark_handle:Nnnn` ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ are labelled as part of this process: the location of the ⟨*handle*⟩ and the label are both printed in the ⟨*color*⟩ specified.

\coffin_show_structure:N
\coffin_show_structure:c

Updated: 2015-08-01

`\coffin_show_structure:N` ⟨*coffin*⟩

This function shows the structural information about the ⟨*coffin*⟩ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

\coffin_log_structure:N
\coffin_log_structure:c

New: 2014-08-22
Updated: 2015-08-01

`\coffin_log_structure:N` ⟨*coffin*⟩

This function writes the structural information about the ⟨*coffin*⟩ in the log file. See also `\coffin_show_structure:N` which displays the result in the terminal.

5.1 Constants and variables

\c_empty_coffin

A permanently empty coffin.

\l_tmpa_coffin
\l_tmpb_coffin

New: 2012-06-19

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part XXVIII

The l3color package Color support

This module provides support for color in LaTeX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

`\color_group_begin:`
`\color_group_end:`

New: 2011-09-03

```
\color_group_begin:
...
\color_group_end:
```

Creates a color group: one used to "trap" color settings.

`\color_ensure_current:`

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin:` . . . `\color_group_end:` group.

1.1 Internal functions

`\l__color_current_tl`

New: 2017-06-15

The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values in the range $[0, 1]$: these determine the color. The model and applicable data format must be one of the following:

- `gray` ⟨*gray*⟩ Grayscale color with the ⟨*gray*⟩ value running from 0 (fully black) to 1 (fully white)
- `cmyk` ⟨*cyan*⟩ ⟨*magenta*⟩ ⟨*yellow*⟩ ⟨*black*⟩
- `rgb` ⟨*red*⟩ ⟨*green*⟩ ⟨*blue*⟩

Notice that the value are separated by spaces.

TeXhackers note: This format is the native one for `dvips` color specials: other drivers are expected to convert to their own format when writing color data to output.

Part XXIX
The l3sys package
System/runtime functions

1 The name of the job

```
\c_sys_jobname_str
New: 2015-09-19
```

Constant that gets the "job name" assigned when TeX starts.

TeXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by TeX and should not be overwritten by the package.

2 Date and time

```
\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int
New: 2015-09-22
```

The date and time at which the current job was started: these are all reported as integers.

TeXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the "real" values.

3 Engine

```
\sys_if_engine_luatex_p: ★
\sys_if_engine_luatex:TF ★
\sys_if_engine_pdftex_p: ★
\sys_if_engine_pdftex:TF ★
\sys_if_engine_ptex_p:   ★
\sys_if_engine_ptex:TF   ★
\sys_if_engine_uptex_p:  ★
\sys_if_engine_uptex:TF  ★
\sys_if_engine_xetex_p:  ★
\sys_if_engine_xetex:TF  ★
New: 2015-09-07
```

```
\sys_if_engine_pdftex:TF {⟨true code⟩} {⟨false code⟩}
```

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the `(u)ptex` tests are for ε-pTeX and ε-upTeX as expl3 requires the ε-TeX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for ε-pTeX but false for ε-upTeX.

```
\c_sys_engine_str
New: 2015-09-19
```

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

4 Output format

`\sys_if_output_dvi_p:` ★
`\sys_if_output_dvi:TF` ★
`\sys_if_output_pdf_p:` ★
`\sys_if_output_pdf:TF` ★

New: 2015-09-19

`\sys_if_output_dvi:TF {⟨true code⟩} {⟨false code⟩}`

Conditionals which give the current output mode the TEX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

`\c_sys_output_str`

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

Part XXX
The l3deprecation package Deprecation errors

1 l3deprecation documentation

A few commands have had to be deprecated over the years. This module defines deprecated and deleted commands to produce an error.

Part XXXI
The **l3candidates** package Experimental additions to **l3kernel**

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

> **As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.**

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the `LaTeX-L` mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the `LaTeX-L` mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to l3basics

\debug_on:n
\debug_off:n

New: 2017-07-16
Updated: 2017-08-02

```
\debug_on:n { ⟨comma-separated list⟩ }
\debug_off:n { ⟨comma-separated list⟩ }
```

Turn on and off within a group various debugging code, some of which is also available as expl3 load-time options. The items that can be used in the ⟨*list*⟩ are

- `check-declarations` that checks all expl3 variables used were previously declared;
- `check-expressions` that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- `deprecation` that makes soon-to-be-deprecated commands produce errors;
- `log-functions` that logs function definitions;

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing. These functions can only be used in LaTeX 2ε package mode loaded with `enable-debug` or another option implying it.

\mode_leave_vertical:

New: 2017-07-04

```
\mode_leave_vertical:
```

Ensures that TeX is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

TeXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the LaTeX 2ε `\leavevmode` approach, no box is used by the method implemented here.

3 Additions to l3box

3.1 Viewing part of a box

\box_clip:N
\box_clip:c

```
\box_clip:N ⟨box⟩
```

Clips the ⟨*box*⟩ in the output so that only material inside the bounding box is displayed in the output. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the clipping is applied. The clipping applies within the current TeX group level.

These functions require the LaTeX3 native drivers: they do not work with the LaTeX 2ε graphics drivers!

TeXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

\box_trim:Nnnnn
\box_trim:cnnnn

`\box_trim:Nnnnn ⟨box⟩ {⟨left⟩} {⟨bottom⟩} {⟨right⟩} {⟨top⟩}`

Adjusts the bounding box of the ⟨*box*⟩ ⟨*left*⟩ is removed from the left-hand edge of the bounding box, ⟨*right*⟩ from the right-hand edge and so fourth. All adjustments are ⟨*dimension expressions*⟩. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the trim operation is applied. The adjustment applies within the current TeX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

\box_viewport:Nnnnn
\box_viewport:cnnnn

`\box_viewport:Nnnnn ⟨box⟩ {⟨llx⟩} {⟨lly⟩} {⟨urx⟩} {⟨ury⟩}`

Adjusts the bounding box of the ⟨*box*⟩ such that it has lower-left co-ordinates (⟨*llx*⟩, ⟨*lly*⟩) and upper-right co-ordinates (⟨*urx*⟩, ⟨*ury*⟩). All four co-ordinate positions are ⟨*dimension expressions*⟩. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the viewport operation is applied. The adjustment applies within the current TeX group level.

4 Additions to l3clist

\clist_rand_item:N ★
\clist_rand_item:c ★
\clist_rand_item:n ★

New: 2016-12-06

`\clist_rand_item:N ⟨clist var⟩`
`\clist_rand_item:n {⟨comma list⟩}`

Selects a pseudo-random item of the ⟨*comma list*⟩. If the ⟨*comma list*⟩ has no item, the result is empty. This is only available in pdfTeX and LuaTeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

5 Additions to l3coffins

\coffin_resize:Nnn
\coffin_resize:cnn

`\coffin_resize:Nnn ⟨coffin⟩ {⟨width⟩} {⟨total-height⟩}`

Resized the ⟨*coffin*⟩ to ⟨*width*⟩ and ⟨*total-height*⟩, both of which should be given as dimension expressions.

\coffin_rotate:Nn
\coffin_rotate:cn

`\coffin_rotate:Nn ⟨coffin⟩ {⟨angle⟩}`

Rotates the ⟨*coffin*⟩ by the given ⟨*angle*⟩ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

\coffin_scale:Nnn
\coffin_scale:cnn

`\coffin_scale:Nnn ⟨coffin⟩ {⟨x-scale⟩} {⟨y-scale⟩}`

Scales the ⟨*coffin*⟩ by a factors ⟨*x-scale*⟩ and ⟨*y-scale*⟩ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

6 Additions to l3file

\file_get_mdfive_hash:nN

New: 2017-07-11

`\file_get_mdfive_hash:nN {⟨file name⟩} ⟨str var⟩`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`. If found, sets the ⟨*str var*⟩ to the MD5 sum generated from the content of the file. Where the file is not found, the ⟨*str var*⟩ will be empty.

\file_get_size:nN

New: 2017-07-09

`\file_get_size:nN {⟨file name⟩} ⟨str var⟩`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`. If found, sets the ⟨*str var*⟩ to the size of the file in bytes. Where the file is not found, the ⟨*str var*⟩ will be empty.

TEXhackers note: The XETEX engine provides no way to implement this function.

\file_get_timestamp:nN

New: 2017-07-09

`\file_get_timestamp:nN {⟨file name⟩} ⟨str var⟩`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`. If found, sets the ⟨*str var*⟩ to the modification timestamp of the file in the form `D:`⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*minute*⟩⟨*second*⟩⟨*offset*⟩, where the latter may be `Z` (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩`'`⟨*minutes*⟩`'`. Where the file is not found, the ⟨*str var*⟩ will be empty.

TEXhackers note: The XETEX engine provides no way to implement this function.

\file_if_exist_input:n
\file_if_exist_input:nF

New: 2014-07-02

`\file_if_exist_input:n {⟨file name⟩}`
`\file_if_exist_input:nF {⟨file name⟩} {⟨false code⟩}`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`. If found then reads in the file as additional LATEX source as described for `\file_input:n`, otherwise inserts the ⟨*false code*⟩. Note that these functions do not raise an error if the file is not found, in contrast to `\file_input:n`.

\file_input_stop:

New: 2017-07-07

`\file_input_stop:`

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

TEXhackers note: This function must be used on a line on its own: TEX reads files line-by-line and so any additional tokens in the "current" line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn't help as it is the line where it is used that counts!

7 Additions to l3int

\int_rand:nn ⋆

New: 2016-12-06

`\int_rand:nn {⟨intexpr_1⟩} {⟨intexpr_2⟩}`

Evaluates the two ⟨*integer expressions*⟩ and produces a pseudo-random number between the two (with bounds included). This is only available in pdfTeX and LuaTeX.

8 Additions to l3msg

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, the message text and arguments are not expanded, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn ⋆
\msg_expandable_error:nnffff ⋆
\msg_expandable_error:nnnnn ⋆
\msg_expandable_error:nnfff ⋆
\msg_expandable_error:nnnn ⋆
\msg_expandable_error:nnff ⋆
\msg_expandable_error:nnn ⋆
\msg_expandable_error:nnf ⋆
\msg_expandable_error:nn ⋆

New: 2015-08-06

`\msg_expandable_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues an “Undefined error” message from TeX itself using the undefined control sequence `\::error` then prints “! ⟨*module*⟩: ”⟨*error message*⟩, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

9 Additions to l3prop

\prop_count:N ⋆
\prop_count:c ⋆

`\prop_count:N ⟨property list⟩`

Leaves the number of key–value pairs in the ⟨*property list*⟩ in the input stream as an ⟨*integer denotation*⟩.

\prop_map_tokens:Nn ☆
\prop_map_tokens:cn ☆

\prop_map_tokens:Nn ⟨*property list*⟩ {⟨*code*⟩}

Analogue of \prop_map_function:NN which maps several tokens instead of a single function. The ⟨*code*⟩ receives each key–value pair in the ⟨*property list*⟩ as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to mykey: for each pair in \l_my_prop the function \str_if_eq:nnT receives mykey, the ⟨*key*⟩ and the ⟨*value*⟩ as its three arguments. For that specific task, \prop_item:Nn is faster.

\prop_rand_key_value:N ★
\prop_rand_key_value:c ★
New: 2016-12-06

\prop_rand_key_value:N ⟨*prop var*⟩

Selects a pseudo-random key–value pair in the ⟨*property list*⟩ and returns {⟨*key*⟩}{⟨*value*⟩}. If the ⟨*property list*⟩ is empty the result is empty. This is only available in pdfTEX and LuaTEX.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*value*⟩ does not expand further when appearing in an x-type argument expansion.

10 Additions to l3seq

\seq_mapthread_function:NNN ☆
\seq_mapthread_function:(NcN|cNN|ccN) ☆

\seq_mapthread_function:NNN ⟨*seq*$_1$⟩ ⟨*seq*$_2$⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every pair of items ⟨*seq*$_1$*-item*⟩–⟨*seq*$_2$*-item*⟩ from the two sequences, returning items from both sequences from left to right. The ⟨*function*⟩ receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

\seq_set_filter:NNn
\seq_gset_filter:NNn

\seq_set_filter:NNn ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ {⟨*inline boolexpr*⟩}

Evaluates the ⟨*inline boolexpr*⟩ for every ⟨*item*⟩ stored within the ⟨*sequence*$_2$⟩. The ⟨*inline boolexpr*⟩ receives the ⟨*item*⟩ as #1. The sequence of all ⟨*items*⟩ for which the ⟨*inline boolexpr*⟩ evaluated to true is assigned to ⟨*sequence*$_1$⟩.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level TEX errors.

\seq_set_map:NNn
\seq_gset_map:NNn
New: 2011-12-22

\seq_set_map:NNn ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence*$_2$⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as #1. The sequence resulting from x-expanding ⟨*inline function*⟩ applied to each ⟨*item*⟩ is assigned to ⟨*sequence*$_1$⟩. As such, the code in ⟨*inline function*⟩ should be expandable.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level TEX errors.

\seq_rand_item:N ⋆
\seq_rand_item:c ⋆

New: 2016-12-06

`\seq_rand_item:N ⟨seq var⟩`

Selects a pseudo-random item of the ⟨*sequence*⟩. If the ⟨*sequence*⟩ is empty the result is empty. This is only available in pdfTEX and LuaTEX.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an x-type argument expansion.

11 Additions to l3skip

\skip_split_finite_else_action:nnNN

`\skip_split_finite_else_action:nnNN {⟨skipexpr⟩} {⟨action⟩}`
`⟨dimen₁⟩ ⟨dimen₂⟩`

Checks if the ⟨*skipexpr*⟩ contains finite glue. If it does then it assigns ⟨*dimen*$_1$⟩ the stretch component and ⟨*dimen*$_2$⟩ the shrink component. If it contains infinite glue set ⟨*dimen*$_1$⟩ and ⟨*dimen*$_2$⟩ to 0 pt and place `#2` into the input stream: this is usually an error or warning message of some sort.

12 Additions to l3sys

\sys_if_rand_exist_p: ⋆
\sys_if_rand_exist:*TF* ⋆

New: 2017-05-27

`\sys_if_rand_exist_p:`
`\sys_if_rand_exist:TF {⟨true code⟩} {⟨false code⟩}`

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTEX and LuaTEX.

\sys_rand_seed: ⋆

New: 2017-05-27

`\sys_rand_seed:`

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

\sys_gset_rand_seed:n

New: 2017-05-27

`\sys_gset_rand_seed:n {⟨intexpr⟩}`

Sets the seed for the engine's pseudo-random number generator to the ⟨*integer expression*⟩. The assignment is global. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. Currently only the absolute value of the seed is used. In engines without random number support this produces an error.

\c_sys_shell_escape_int

New: 2017-05-27

This variable exposes the internal triple of the shell escape status. The possible values are

0 Shell escape is disabled

1 Unrestricted shell escape is enabled

2 Restricted shell escape is enabled

`\sys_if_shell_p:` ★
`\sys_if_shell:TF` ★

New: 2017-05-27

```
\sys_if_shell_p:
\sys_if_shell:TF {⟨true code⟩} {⟨false code⟩}
```

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

`\sys_if_shell_unrestricted_p:` ★
`\sys_if_shell_unrestricted:TF` ★

New: 2017-05-27

```
\sys_if_shell_unrestricted_p:
\sys_if_shell_unrestricted:TF {⟨true code⟩} {⟨false code⟩}
```

Performs a check for whether *unrestricted* shell escape is enabled.

`\sys_if_shell_restricted_p:` ★
`\sys_if_shell_restricted:TF` ★

New: 2017-05-27

```
\sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF {⟨true code⟩} {⟨false code⟩}
```

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

`\sys_shell_now:n`
`\sys_shell_now:x`

New: 2017-05-27

```
\sys_shell_now:n {⟨tokens⟩}
```

Execute ⟨*tokens*⟩ through shell escape immediately.

`\sys_shell_shipout:n`
`\sys_shell_shipout:x`

New: 2017-05-27

```
\sys_shell_shipout:n {⟨tokens⟩}
```

Execute ⟨*tokens*⟩ through shell escape at shipout.

13 Additions to l3tl

`\tl_if_single_token_p:n` ★
`\tl_if_single_token:nTF` ★

```
\tl_if_single_token_p:n {⟨token list⟩}
\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single "normal" token. Token groups (`{...}`) are not single tokens.

`\tl_reverse_tokens:n` ★

```
\tl_reverse_tokens:n {⟨tokens⟩}
```

This function, which works directly on TeX tokens, reverses the order of the ⟨*tokens*⟩: the first becomes the last and the last becomes first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{)(b}~a` in the input stream. This function requires two steps of expansion.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list does not expand further when appearing in an `x`-type argument expansion.

`\tl_count_tokens:n` ★

```
\tl_count_tokens:n {⟨tokens⟩}
```

Counts the number of TEX tokens in the ⟨*tokens*⟩ and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6. This function requires three expansions, giving an ⟨*integer denotation*⟩.

`\tl_lower_case:n` ★
`\tl_upper_case:n` ★
`\tl_mixed_case:n` ★
`\tl_lower_case:nn` ★
`\tl_upper_case:nn` ★
`\tl_mixed_case:nn` ★

New: 2014-06-30
Updated: 2016-01-12

```
\tl_upper_case:n  {⟨tokens⟩}
\tl_upper_case:nn {⟨language⟩} {⟨tokens⟩}
```

These functions are intended to be applied to input which may be regarded broadly as "text". They traverse the ⟨*tokens*⟩ and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the ⟨*tokens*⟩ are normalized and become `{` and `}`, respectively.

Importantly, notice that these functions are intended for working with user text for typesetting. For case changing programmatic data see the l3str module and discussion there of `\str_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions is expanded *unless* it falls within one of the special handling classes described below. This expansion approach means that in general the result of case changing matches the "natural" outcome expected from a "functional" approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

produces

```
HELLO WORLD
```

The expansion approach taken means that in package mode any LATEX 2$_\varepsilon$ "robust" commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the etoolbox package.

`\l_tl_case_change_math_tl`

Case changing does not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_tl_case_change_math_tl`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \( \)
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be "hidden" inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of "text" the case changing functions are intended to apply to this should not be an issue.

`\l_tl_case_change_exclude_tl`

Case changing can be prevented by using any command on the list `\l_tl_case_change_exclude_tl`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_tl_case_change_exclude_tl { \NoChangeCase }
```

the input

```
\tl_upper_case:n
  { Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with LaTeX 2ε the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

\l_tl_case_change_accents_tl

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\tl_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of \".

The standard contents of this variable is \", \', \., \^, \`, \~, \c, \H, \k, \r, \t, \u and \v.

"Mixed" case conversion may be regarded informally as converting the first character of the ⟨*tokens*⟩ to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example ij in Dutch which becomes IJ. As such, \tl_mixed_case:n(n) implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the ⟨*tokens*⟩ are ignored when finding the first "letter" for conversion.

```
\tl_mixed_case:n { hello~WORLD }   % => "Hello world"
\tl_mixed_case:n { ~hello~WORLD }  % => " Hello world"
\tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first "letter" for this process, any content in math mode or covered by \l_tl_case_change_exclude_tl is ignored.

(Note that the Unicode Consortium describe this as "title case", but that in English title case applies on a word-by-word basis. The "mixed" case implemented here is a lower level concept needed for both "title" and "sentence" casing of text.)

\l_tl_mixed_case_ignore_tl

The list of characters to ignore when searching for the first "letter" in mixed-casing is determined by \l_tl_mixed_change_ignore_tl. This has the standard setting

```
( [ { ` -
```

where comparisons are made on a character basis.

As is generally true for expl3, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with XeTeX or LuaTeX a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the Unicode Consortium in UnicodeData.txt and SpecialCasing.txt. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1 font encoding. Thus for example Ãď can be case-changed using pdfTeX. For pTeX only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection expands input but treats any unexpandable control sequences as "failures" to match a context.

Language-sensitive conversions are enabled using the ⟨*language*⟩ argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lower casing I-dot and introduced when upper casing i-dotless.
- German (`de-alt`). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (`lt`). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdfTEX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

```
\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
New: 2014-06-25
```

`\tl_set_from_file:Nnn ⟨tl⟩ {⟨setup⟩} {⟨filename⟩}`

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument.

```
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
New: 2014-06-25
```

`\tl_set_from_file_x:Nnn ⟨tl⟩ {⟨setup⟩} {⟨filename⟩}`

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the ⟨*setup*⟩ argument.

```
\tl_rand_item:N ⋆
\tl_rand_item:c ⋆
\tl_rand_item:n ⋆
New: 2016-12-06
```

```
\tl_rand_item:N ⟨tl var⟩
\tl_rand_item:n {⟨token list⟩}
```

Selects a pseudo-random item of the ⟨*token list*⟩. If the ⟨*token list*⟩ is blank, the result is empty. This is only available in pdfTEX and LuaTEX.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

\tl_range:Nnn ⋆ \tl_range:nnn ⋆

New: 2017-02-17
Updated: 2017-07-15

\tl_range:Nnn ⟨*tl var*⟩ {⟨*start index*⟩} {⟨*end index*⟩}
\tl_range:nnn {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

Leaves in the input stream the items from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Positive ⟨*indices*⟩ are counted from the start of the ⟨*token list*⟩, 1 being the first item, and negative ⟨*indices*⟩ are counted from the end of the token list, −1 being the last item. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is 0, the result is empty. For instance,

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `bcd␣{e{}}`, `cd␣{e{}}f`, `{e{}}f` and an empty line to the terminal. The ⟨*start index*⟩ must always be smaller than or equal to the ⟨*end index*⟩: if this is not the case then no output is generated. Thus

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { 5 } { 2 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}f } { -1 } { -4 } }
```

both yield empty token lists. For improved performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

`\tl_range_braced:Nnn` ★
`\tl_range_braced:cnn` ★
`\tl_range_braced:nnn` ★
`\tl_range_unbraced:Nnn` ★
`\tl_range_unbraced:cnn` ★
`\tl_range_unbraced:nnn` ★

New: 2017-07-15

```
\tl_range_braced:Nnn ⟨tl var⟩ {⟨start index⟩} {⟨end index⟩}
\tl_range_braced:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\tl_range_unbraced:Nnn ⟨tl var⟩ {⟨start index⟩} {⟨end index⟩}
\tl_range_unbraced:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
```

Leaves in the input stream the items from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive, using the same indexing as `\tl_range:nnn`. Spaces are ignored. Regardless of whether items appear with or without braces in the ⟨*token list*⟩, the `\tl_range_braced:nnn` function wraps each item in braces, while `\tl_range_unbraced:nnn` does not (overall it removes an outer set of braces). For instance,

```
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `{b}{c}{d}{e{}}`, `{c}{d}{e{}}{f}`, `{e{}}{f}`, and an empty line to the terminal, while

```
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `bcde{}`, `cde{}f`, `e{}f`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `x`-type argument expansion.

14 Additions to l3token

`\c_catcode_active_space_tl`

New: 2017-08-07

Token list containing one character with category code 13, ("active"), and character code 32 (space).

`\peek_N_type:TF`

Updated: 2012-12-20

```
\peek_N_type:TF {⟨true code⟩} {⟨false code⟩}
```

Tests if the next ⟨*token*⟩ in the input stream can be safely grabbed as an `N`-type argument. The test is ⟨*false*⟩ if the next ⟨*token*⟩ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in LaTeX3) and ⟨*true*⟩ in all other cases. Note that a ⟨*true*⟩ result ensures that the next ⟨*token*⟩ is a valid `N`-type argument. However, if the next ⟨*token*⟩ is for instance `\c_space_token`, the test takes the ⟨*false*⟩ branch, even though the next ⟨*token*⟩ is in fact a valid `N`-type argument. The ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

Part XXXII
The **l3luatex** package LuaTeX-specific functions

1 Breaking out to Lua

The LuaTEX engine provides access to the Lua programming language, and with it access to the "internals" of TEX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTEX or XeTEX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTEX engine are given in the LuaTEX manual.

1.1 TEX code interfaces

`\lua_now_x:n` ★
`\lua_now:n` ★

New: 2015-06-29

`\lua_now:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TEX, which includes converting line ends to spaces in the usual TEX manner and which respects currently-applicable TEX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the ⟨*Lua input*⟩ immediately, and in an expandable manner.

In the case of the `\lua_now_x:n` version the input is fully expanded by TEX in an `x`-type manner *but* the function remains fully expandable.

TEXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua`: when LuaTEX is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_x:n`
`\lua_shipout:n`

New: 2015-06-30

`\lua_shipout:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TEX, which includes converting line ends to spaces in the usual TEX manner and which respects currently-applicable TEX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ during the page-building routine: no TEX expansion of the ⟨*Lua input*⟩ will occur at this stage.

In the case of the `\lua_shipout_x:n` version the input is fully expanded by TEX in an `x`-type manner during the shipout operation.

TEXhackers note: At a TEX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

`\lua_escape_x:n` ★
`\lua_escape:n` ★

New: 2015-06-29

`\lua_escape:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

In the case of the `\lua_escape_x:n` version the input is fully expanded by TEX in an `x`-type manner *but* the function remains fully expandable.

TEXhackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when LuaTEX is in use two expansions are required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for TEX, there are a small number of Lua functions provided here. Currently these are intended for internal use only.

`l3kernel.charcat`

`\l3kernel.charcat(⟨charcode⟩, ⟨catcode⟩)`

Constructs a character of ⟨*charcode*⟩ and ⟨*catcode*⟩ and returns the result to TEX.

`l3kernel.filemdfivesum`

`\l3kernel.filemdfivesum(⟨file⟩)`

Returns the of the MD5 sum of the file contents read as bytes. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

`l3kernel.filemoddate`

`\l3kernel.filemoddate(⟨file⟩)`

Returns the of the date/time of last modification of the ⟨*file*⟩ in the format `D:`⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*m* where the latter may be `Z` (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩`'`⟨*minutes*⟩`'`. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

`l3kernel.filesize`

`\l3kernel.filesize(⟨file⟩)`

Returns the size of the ⟨*file*⟩ in bytes. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

`l3kernel.strcmp`

`\l3kernel.strcmp(⟨str one⟩, ⟨str two⟩)`

Compares the two strings and returns `0` to TEX if the two are identical.

Part XXXIII
The l3drivers package Drivers

TEX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, LATEX3 is aware of the following drivers:

- `pdfmode`: The "driver" for direct PDF output by *both* pdfTEX and LuaTEX (no separate driver is used in this case: the engine deals with PDF creation itself).
- `dvips`: The `dvips` program, which works in conjugation with pdfTEX or LuaTEX in DVI mode.
- `dvipdfmx`: The `dvipdfmx` program, which works in conjugation with pdfTEX or LuaTEX in DVI mode.
- `dvisvgm`: The `dvisvgm` program, which works in conjugation with pdfTEX or LuaTEX when run in DVI mode as well as with (u)pTEX and XƎTEX.
- `xdvipdfmx`: The driver used by XƎTEX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level "up", and they must be used in the correct contexts.

1 Box clipping

`\__driver_box_use_clip:N`

New: 2011-11-11

`\__driver_box_use_clip:N ⟨box⟩`

Inserts the content of the ⟨*box*⟩ at the current insertion point such that any material outside of the bounding box is not displayed by the driver. The material in the ⟨*box*⟩ is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

`\__driver_box_use_rotate:Nn`

New: 2016-05-12

`\__driver_box_use_rotate:Nn ⟨box⟩ {⟨angle⟩}`

Inserts the content of the ⟨*box*⟩ at the current insertion point rotated by the ⟨*angle*⟩ (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the TEX reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibility of the code using this function to adjust the apparent size of the box to be correct at the TEX side.

This function should only be used within a surrounding horizontal box construct.

```
\__driver_box_use_scale:Nnn    \__driver_box_use_scale:Nnn ⟨box⟩ {⟨x-scale⟩} {⟨y-scale⟩}
           New: 2016-05-12
```

Inserts the content of the ⟨*box*⟩ at the current insertion point scale by the ⟨*x-scale*⟩ and ⟨*y-scale*⟩. The material is inserted with no apparent height or width. It is the responsibility of the code using this function to adjust the apparent size of the box to be correct at the TeX side.

This function should only be used within a surrounding horizontal box construct.

3 Color support

```
\__driver_color_ensure_current:    \__driver_color_ensure_current:
                New: 2011-09-03
            Updated: 2012-05-18
```

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any "color safe" box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of `pgf` (most have the same interface as the same functions in the latter's `\pgfsys@...` namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by `pgf` and Ti*k*z.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

```
\__driver_draw_begin:    \__driver_draw_begin:
\__driver_draw_end:      ⟨content⟩
                         \__driver_draw_end:
```

Defines a drawing environment. This is a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the ⟨*content*⟩ should be zero from the TeX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

```
\__driver_draw_scope_begin:    \__driver_draw_scope_begin:
\__driver_draw_scope_end:      ⟨content⟩
                               \__driver_draw_scope_end:
```

Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form TeX groups and may not be aligned with them.

4.1 Path construction

`\__driver_draw_moveto:nn`

`\__driver_draw_move:nn` {⟨x⟩} {⟨y⟩}

Moves the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix applies.

`\__driver_draw_lineto:nn`

`\__driver_draw_lineto:nn` {⟨x⟩} {⟨y⟩}

Adds a path from the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_curveto:nnnnnn`

`\__driver_draw_curveto:nnnnnn` {⟨x_1⟩} {⟨y_1⟩}
{⟨x_2⟩} {⟨y_2⟩} {⟨x_3⟩} {⟨y_3⟩}

Adds a Bezier curve path from the current drawing reference point to (⟨x_3⟩, ⟨y_3⟩), using (⟨x_1⟩, ⟨y_1⟩) and (⟨x_2⟩, ⟨y_2⟩) as control points; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_rectangle:nnnn`

`\__driver_draw_rectangle:nnnn` {⟨x⟩} {⟨y⟩} {⟨*width*⟩} {⟨*height*⟩}

Adds rectangular path from (⟨x_1⟩, ⟨y_1⟩) of ⟨*height*⟩ and ⟨*width*⟩; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_closepath:`

`\__driver_draw_closepath:`

Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

`\__driver_draw_stroke:`
`\__driver_draw_closestroke:`

⟨*path construction*⟩
`\__driver_draw_stroke:`

Draws a line along the current path, which is also closed by `\__driver_draw_closestroke:`. The nature of the line drawn is influenced by settings for

- Line thickness
- Stroke color (or the current color if no specific stroke color is set)
- Line capping (how non-closed line ends should look)
- Join style (how a bend in the path should be rendered)
- Dash pattern

The path may also be used for clipping.

```
\__driver_draw_fill:
\__driver_draw_fillstroke:
```

⟨*path construction*⟩
`\__driver_draw_fill:`

Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version also strokes the path as described for `\__driver_draw_stroke:`. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

```
\__driver_draw_nonzero_rule:
\__driver_draw_evenodd_rule:
```

`\__driver_draw_nonzero_rule:`

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

```
\__driver_draw_clip:
```

⟨*path construction*⟩
`\__driver_draw_clip:`

Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

```
\__driver_draw_discardpath:
```

⟨*path construction*⟩
`\__driver_draw_discardpath:`

Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

4.3 Stroke options

```
\__driver_draw_linewidth:n
```

`\__driver_draw_linewidth:n {`⟨*dimexpr*⟩`}`

Sets the width to be used for stroking to ⟨*dimexpr*⟩.

```
\__driver_draw_dash:nn
```

`\__driver_draw_dash:nn {`⟨*dash pattern*⟩`} {`⟨*phase*⟩`}`

Sets the pattern of dashing to be used when stroking a line. The ⟨*dash pattern*⟩ should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example `3pt, 4pt` means that 3 pt on, 4 pt off, 3 pt on, and so on. A more complex pattern will also repeat: `3pt, 4pt, 1pt, 2pt` results in 3 pt on, 4 pt off, 1 pt on, 2 pt off, 3 pt on, and so on. An odd number of entries means that the last is repeated, for example `3pt` is equal to `3pt, 3pt`. An empty pattern yields a solid line.

The ⟨*phase*⟩ specifies an offset at the start of the cycle. For example, with a pattern `3pt` a phase of `1pt` means that the output is 2 pt on, 3 pt off, 3 pt on, 3 pt on, *etc.*

```
\__driver_draw_cap_butt:
\__driver_draw_cap_rectangle:
\__driver_draw_cap_round:
```

`\__driver_draw_cap_butt:`

Sets the style of terminal stroke position to one of butt, rectangle or round.

```
\__driver_draw_join_bevel:
\__driver_draw_join_miter:
\__driver_draw_join_round:
```

`\__driver_draw_cap_butt:`

Sets the style of stroke joins to one of bevel, miter or round.

```
\__driver_draw_miterlimit:n
```

`\__driver_draw_miterlimit:n {⟨dimexpr⟩}`

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

```
\__driver_draw_color_cmyk:nnnn
\__driver_draw_color_cmyk_fill:nnnn
\__driver_draw_color_cmyk_stroke:nnnn
```

`\__driver_draw_color_cmyk:nnnn {⟨cyan⟩} {⟨magneta⟩} {⟨yellow⟩} {⟨black⟩}`

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_gray:n
\__driver_draw_color_gray_fill:n
\__driver_draw_color_gray_stroke:n
```

`\__driver_draw_color_gray:n {⟨gray⟩}`

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_rgb:nnn
\__driver_draw_color_rgb_fill:nnn
\__driver_draw_color_rgb_stroke:nnn
```

`\__driver_draw_color_gray:n {⟨red⟩} {⟨green⟩} {⟨blue⟩}`

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting TEX material

`\__driver_draw_hbox:Nnnnnnn`

```
\__driver_draw_hbox:Nnnnnnn ⟨box⟩
{⟨a⟩} {⟨b⟩} {⟨c⟩} {⟨d⟩} {⟨x⟩} {⟨y⟩}
```

Inserts the ⟨*box*⟩ as an hbox with the box reference point placed at (x, y). The transformation matrix [*abcd*] is applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that TEX material should not be inserted directly into a drawing as it would not be in the correct location. Also note that as for other drawing elements the box here has no size from a TEX perspective.

4.6 Coordinate system transformations

`\__driver_draw_transformcm:nnnnnn`

```
\__driver_draw_transformcm:nnnnnn {⟨a⟩} {⟨b⟩} {⟨c⟩} {⟨d⟩}
{⟨x⟩} {⟨y⟩}
```

Applies the transformation matrix [*abcd*] and offset vector (x, y) to the current graphic state. This affects any subsequent items in the same scope but not those already given.

Part XXXIV
Implementation

1 l3bootstrap implementation

```
⟨*initex | package⟩
⟨@@=kernel⟩
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniTEX system so that everything else will actually work. TEX does not start with some pretty basic character codes set up.

```
⟨*initex⟩
\catcode `\{ = 1 %
\catcode `\} = 2 %
\catcode `\# = 6 %
\catcode `\^ = 7 %
⟨/initex⟩
```

Tab characters should not show up in the code, but to be on the safe side.

```
⟨*initex⟩
\catcode `\^^I = 10 %
⟨/initex⟩
```

For LuaTEX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

```
⟨*initex⟩
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
\else
  \directlua{tex.enableprimitives("", tex.extraprimitives())}%
\fi
⟨/initex⟩
```

Depending on the versions available, the LaTeX format may not have the raw `\Umath` primitive names available. We fix that globally: it should cause no issues. Older LuaTeX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start `\U` this should be reasonably safe.

```
⟨*package⟩
\begingroup
  \expandafter\ifx\csname directlua\endcsname\relax
  \else
    \directlua{%
      local i
      local t = { }
      for _,i in pairs(tex.extraprimitives("luatex")) do
        if string.match(i,"^U") then
          if not string.match(i,"^Uchar$") then %$
            table.insert(t,i)
          end
        end
      end
      tex.enableprimitives("", t)
    }%
  \fi
\endgroup
⟨/package⟩
```

1.2 The `\pdfstrcmp` primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfTeX name is "safe".

```
\begingroup\expandafter\expandafter\expandafter\endgroup
  \expandafter\ifx\csname pdfstrcmp\endcsname\relax
  \let\pdfstrcmp\strcmp
\fi
```

1.3 Loading support Lua code

When LuaTeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in l3luatex and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
\else
  \ifnum\luatexversion<70 %
  \else
```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the LaTeX 2ε-based generic code. In format mode the table used here can be hard-coded into the Lua.

```
⟨*package⟩
    \begingroup\expandafter\expandafter\expandafter\endgroup
    \expandafter\ifx\csname newcatcodetable\endcsname\relax
```

```
      \input{ltluatex}%
    \fi
    \newcatcodetable\ucharcat@table
    \directlua{
      l3kernel = l3kernel or { }
      local charcat_table = \number\ucharcat@table\space
      l3kernel.charcat_table = charcat_table
    }%
⟨/package⟩
    \directlua{require("expl3")}%
```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```
    \ifnum 0%
      \directlua{
        if status.ini_version then
          tex.write("1")
        end
      }>0 %
      \everyjob\expandafter{%
        \the\expandafter\everyjob
        \csname\detokenize{lua_now_x:n}\endcsname{require("expl3")}%
      }%
    \fi
  \fi
\fi
```

1.4 Engine requirements

The code currently requires ε-TEX and functionality equivalent to \pdfstrcmp, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

```
\begingroup
  \def\next{\endgroup}%
  \def\ShortText{Required primitives not found}%
  \def\LongText%
    {%
      LaTeX3 requires the e-TeX primitives and additional functionality as
      described in the README file.
      \LineBreak
      These are available in the engines\LineBreak
      - pdfTeX v1.40\LineBreak
      - XeTeX v0.9994\LineBreak
      - LuaTeX v0.70\LineBreak
      - e-(u)pTeX mid-2012\LineBreak
      or later.\LineBreak
      \LineBreak
    }%
  \ifnum0%
    \expandafter\ifx\csname pdfstrcmp\endcsname\relax
    \else
      \expandafter\ifx\csname pdftexversion\endcsname\relax
        1%
      \else
        \ifnum\pdftexversion<140 \else 1\fi
      \fi
    \fi
    \expandafter\ifx\csname directlua\endcsname\relax
    \else
      \ifnum\luatexversion<70 \else 1\fi
    \fi
    =0 %
      \newlinechar`\^^J %
⟨*initex⟩
      \def\LineBreak{^^J}%
      \edef\next
        {%
          \errhelp
            {%
              \LongText
              For pdfTeX and XeTeX the '-etex' command-line switch is also
              needed.\LineBreak
              \LineBreak
              Format building will abort!\LineBreak
            }%
          \errmessage{\ShortText}%
          \endgroup
          \noexpand\end
        }%
⟨/initex⟩
⟨*package⟩
      \def\LineBreak{\noexpand\MessageBreak}%
      \expandafter\ifx\csname PackageError\endcsname\relax
        \def\LineBreak{^^J}%
        \def\PackageError#1#2#3%
          {%
            \errhelp{#3}%
            \errmessage{#1 Error: #2}%
          }%
      \fi
      \edef\next
        {%
          \noexpand\PackageError{expl3}{\ShortText}
            {\LongText Loading of expl3 will abort!}%
          \endgroup
          \noexpand\endinput
        }%
⟨/package⟩
  \fi
\next
```

1.5 Extending allocators

In format mode, allocating registers is handled by l3alloc. However, in package mode it's much safer to rely on more general code. For example, the ability to extend TeX's allocation routine to allow for ε-TeX has been around since 1997 in the etex package.

Loading this support is delayed until here as we are now sure that the ε-TeX extensions and \pdfstrcmp or equivalent are available. Thus there is no danger of an "uncontrolled" error if the engine requirements are not met.

For LaTeX 2ε we need to make sure that the extended pool is being used: expl3 uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the etex package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that \reserveinserts is \outer and so is accessed here by csname. In earlier versions, loading etex was done directly and so \reserveinserts appeared in the code: this then required a \relax after \RequirePackage to prevent an error with "unsafe" definitions as seen for example with capoptions. The optional loading here is done using a group and \ifx test as we are not quite in the position to have a single name for \pdfstrcmp just yet.

```
⟨*package⟩
\begingroup
  \def\@tempa{LaTeX2e}%
  \def\next{}%
  \ifx\fmtname\@tempa
    \expandafter\ifx\csname extrafloats\endcsname\relax
      \def\next
        {%
          \RequirePackage{etex}%
          \csname reserveinserts\endcsname{32}%
        }%
    \fi
  \fi
\expandafter\endgroup
\next
⟨/package⟩
```

1.6 Character data

TeX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which \lccode values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for LuaTeX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini)TeX category codes and primitive availability and must therefore loaded *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For XeTeX and LuaTeX, which are natively Unicode engines, simply load the Unicode data.

```
⟨*initex⟩
\ifdefined\Umathcode
  \input load-unicode-data %
  \input load-unicode-math-classes %
\else
```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```
\begingroup
```

Lower case chars: map to themselves when lower casing and down by "20 when upper casing. (The characters a–z are set up correctly by IniTEX.)

```
  \def\temp{%
    \ifnum\count0>\count2 %
    \else
      \global\lccode\count0 = \count0 %
      \global\uccode\count0 = \numexpr\count0 - "20\relax
      \advance\count0 by 1 %
      \expandafter\temp
    \fi
  }
  \count0 = "A0 %
  \count2 = "BC %
  \temp
  \count0 = "E0 %
  \count2 = "FF %
  \temp
```

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an \sfcode of 999. (The characters A–Z are set up correctly by IniTEX.)

```
  \def\temp{%
    \ifnum\count0>\count2 %
    \else
      \global\lccode\count0 = \numexpr\count0 + "20\relax
      \global\uccode\count0 = \count0 %
      \global\sfcode\count0 = 999 %
      \advance\count0 by 1 %
      \expandafter\temp
    \fi
  }
  \count0 = "80 %
  \count2 = "9C %
  \temp
  \count0 = "C0 %
  \count2 = "DF %
  \temp
```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```
  \global\lccode`\^^Y = `\^^Y %
  \global\uccode`\^^Y = `\I %
  \global\lccode`\^^Z = `\^^Z %
  \global\uccode`\^^Y = `\J %
  \global\lccode"9D = `\i %
  \global\uccode"9D = "9D %
  \global\lccode"9E = "9E %
  \global\uccode"9E = "D0 %
```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```
    \global\lccode23 = 23 %
  \endgroup
\fi
```

In all cases it makes sense to set up `-` to map to itself: this allows hyphenation of the rest of a word following it (suggested by Lars Helström).

```
\global\lccode`\- = `\- %
⟨/initex⟩
```

1.7 The LaTeX3 code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` becomes a "do nothing" command until `\ExplSyntaxOn` is used. For format mode, there is no need to save category codes so that step is skipped.

```
\protected\def\ExplSyntaxOff{}%
⟨*package⟩
\protected\edef\ExplSyntaxOff
  {%
    \protected\def\ExplSyntaxOff{}%
    \catcode   9 = \the\catcode   9\relax
    \catcode  32 = \the\catcode  32\relax
    \catcode  34 = \the\catcode  34\relax
    \catcode  38 = \the\catcode  38\relax
    \catcode  58 = \the\catcode  58\relax
    \catcode  94 = \the\catcode  94\relax
    \catcode  95 = \the\catcode  95\relax
    \catcode 124 = \the\catcode 124\relax
    \catcode 126 = \the\catcode 126\relax
    \endlinechar = \the\endlinechar\relax
    \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
  }%
⟨/package⟩
```

(*End definition for* `\ExplSyntaxOff`. *This function is documented on page 6.*)

The code environment is now set up.

```
\catcode 9   = 9\relax
\catcode 32  = 9\relax
\catcode 34  = 12\relax
\catcode 38 =  4\relax
\catcode 58  = 11\relax
\catcode 94  = 7\relax
\catcode 95  = 11\relax
\catcode 124 = 12\relax
\catcode 126 = 10\relax
\endlinechar = 32\relax
```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```
\chardef\l__kernel_expl_bool = 1\relax
```

(*End definition for* `\l__kernel_expl_bool`.)

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```
\protected \def \ExplSyntaxOn
  {
    \bool_if:NF \l__kernel_expl_bool
      {
        \cs_set_protected:Npx \ExplSyntaxOff
          {
            \char_set_catcode:nn { 9 }   { \char_value_catcode:n { 9 } }
            \char_set_catcode:nn { 32 }  { \char_value_catcode:n { 32 } }
            \char_set_catcode:nn { 34 }  { \char_value_catcode:n { 34 } }
            \char_set_catcode:nn { 38 }  { \char_value_catcode:n { 38 } }
            \char_set_catcode:nn { 58 }  { \char_value_catcode:n { 58 } }
            \char_set_catcode:nn { 94 }  { \char_value_catcode:n { 94 } }
            \char_set_catcode:nn { 95 }  { \char_value_catcode:n { 95 } }
            \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
            \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
            \tex_endlinechar:D =
              \tex_the:D \tex_endlinechar:D \scan_stop:
            \bool_set_false:N \l__kernel_expl_bool
            \cs_set_protected:Npn \ExplSyntaxOff { }
          }
      }
    \char_set_catcode_ignore:n           { 9 }   % tab
    \char_set_catcode_ignore:n           { 32 }  % space
    \char_set_catcode_other:n            { 34 }  % double quote
    \char_set_catcode_alignment:n        { 38 }  % ampersand
    \char_set_catcode_letter:n           { 58 }  % colon
    \char_set_catcode_math_superscript:n { 94 }  % circumflex
    \char_set_catcode_letter:n           { 95 }  % underscore
    \char_set_catcode_other:n            { 124 } % pipe
    \char_set_catcode_space:n            { 126 } % tilde
    \tex_endlinechar:D = 32 \scan_stop:
    \bool_set_true:N \l__kernel_expl_bool
  }
```

(*End definition for \ExplSyntaxOn. This function is documented on page 6.*)

```
⟨/initex | package⟩
```

2 l3names implementation

```
⟨*initex | package⟩
```

The prefix here is kernel. A few places need @@ to be left as is; this is obtained as @@@@.

```
⟨@@=kernel⟩
```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

\tex_undefined:D This function does not exist at all, but is the name used by the plain TeX format for an undefined function. So it should be marked here as "taken".

(*End definition for* `\tex_undefined:D`.)

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
\let \tex_global:D \global
\let \tex_let:D    \let
```

Everything is inside a (rather long) group, which keeps `\__kernel_primitive:NN` trapped.

```
\begingroup
```

`\__kernel_primitive:NN`

A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
  \long \def \__kernel_primitive:NN #1#2
    {
      \tex_global:D \tex_let:D #2 #1
⟨*initex⟩
      \tex_global:D \tex_let:D #1 \tex_undefined:D
⟨/initex⟩
    }
```

(*End definition for* `\__kernel_primitive:NN`.)

To allow extracting "just the names", a bit of DocStrip fiddling.

```
⟨/initex | package⟩
⟨*initex | names | package⟩
```

In the current incarnation of this package, all TEX primitives are given a new name of the form `\tex_`*oldname*`:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
  \__kernel_primitive:NN \                          \tex_space:D
  \__kernel_primitive:NN \/                         \tex_italiccorrection:D
  \__kernel_primitive:NN \-                         \tex_hyphen:D
```

Now all the other primitives.

```
  \__kernel_primitive:NN \above                       \tex_above:D
  \__kernel_primitive:NN \abovedisplayshortskip       \tex_abovedisplayshortskip:D
  \__kernel_primitive:NN \abovedisplayskip            \tex_abovedisplayskip:D
  \__kernel_primitive:NN \abovewithdelims             \tex_abovewithdelims:D
  \__kernel_primitive:NN \accent                      \tex_accent:D
  \__kernel_primitive:NN \adjdemerits                 \tex_adjdemerits:D
  \__kernel_primitive:NN \advance                     \tex_advance:D
  \__kernel_primitive:NN \afterassignment             \tex_afterassignment:D
  \__kernel_primitive:NN \aftergroup                  \tex_aftergroup:D
  \__kernel_primitive:NN \atop                        \tex_atop:D
  \__kernel_primitive:NN \atopwithdelims              \tex_atopwithdelims:D
  \__kernel_primitive:NN \badness                     \tex_badness:D
  \__kernel_primitive:NN \baselineskip                \tex_baselineskip:D
  \__kernel_primitive:NN \batchmode                   \tex_batchmode:D
  \__kernel_primitive:NN \begingroup                  \tex_begingroup:D
  \__kernel_primitive:NN \belowdisplayshortskip       \tex_belowdisplayshortskip:D
  \__kernel_primitive:NN \belowdisplayskip            \tex_belowdisplayskip:D
  \__kernel_primitive:NN \binoppenalty                \tex_binoppenalty:D
  \__kernel_primitive:NN \botmark                     \tex_botmark:D
```

```
\__kernel_primitive:NN \box                         \tex_box:D
\__kernel_primitive:NN \boxmaxdepth                 \tex_boxmaxdepth:D
\__kernel_primitive:NN \brokenpenalty               \tex_brokenpenalty:D
\__kernel_primitive:NN \catcode                     \tex_catcode:D
\__kernel_primitive:NN \char                        \tex_char:D
\__kernel_primitive:NN \chardef                     \tex_chardef:D
\__kernel_primitive:NN \cleaders                    \tex_cleaders:D
\__kernel_primitive:NN \closein                     \tex_closein:D
\__kernel_primitive:NN \closeout                    \tex_closeout:D
\__kernel_primitive:NN \clubpenalty                 \tex_clubpenalty:D
\__kernel_primitive:NN \copy                        \tex_copy:D
\__kernel_primitive:NN \count                       \tex_count:D
\__kernel_primitive:NN \countdef                    \tex_countdef:D
\__kernel_primitive:NN \cr                          \tex_cr:D
\__kernel_primitive:NN \crcr                        \tex_crcr:D
\__kernel_primitive:NN \csname                      \tex_csname:D
\__kernel_primitive:NN \day                         \tex_day:D
\__kernel_primitive:NN \deadcycles                  \tex_deadcycles:D
\__kernel_primitive:NN \def                         \tex_def:D
\__kernel_primitive:NN \defaulthyphenchar           \tex_defaulthyphenchar:D
\__kernel_primitive:NN \defaultskewchar             \tex_defaultskewchar:D
\__kernel_primitive:NN \delcode                     \tex_delcode:D
\__kernel_primitive:NN \delimiter                   \tex_delimiter:D
\__kernel_primitive:NN \delimiterfactor             \tex_delimiterfactor:D
\__kernel_primitive:NN \delimitershortfall          \tex_delimitershortfall:D
\__kernel_primitive:NN \dimen                       \tex_dimen:D
\__kernel_primitive:NN \dimendef                    \tex_dimendef:D
\__kernel_primitive:NN \discretionary               \tex_discretionary:D
\__kernel_primitive:NN \displayindent               \tex_displayindent:D
\__kernel_primitive:NN \displaylimits               \tex_displaylimits:D
\__kernel_primitive:NN \displaystyle                \tex_displaystyle:D
\__kernel_primitive:NN \displaywidowpenalty         \tex_displaywidowpenalty:D
\__kernel_primitive:NN \displaywidth                \tex_displaywidth:D
\__kernel_primitive:NN \divide                      \tex_divide:D
\__kernel_primitive:NN \doublehyphendemerits        \tex_doublehyphendemerits:D
\__kernel_primitive:NN \dp                          \tex_dp:D
\__kernel_primitive:NN \dump                        \tex_dump:D
\__kernel_primitive:NN \edef                        \tex_edef:D
\__kernel_primitive:NN \else                        \tex_else:D
\__kernel_primitive:NN \emergencystretch            \tex_emergencystretch:D
\__kernel_primitive:NN \end                         \tex_end:D
\__kernel_primitive:NN \endcsname                   \tex_endcsname:D
\__kernel_primitive:NN \endgroup                    \tex_endgroup:D
\__kernel_primitive:NN \endinput                    \tex_endinput:D
\__kernel_primitive:NN \endlinechar                 \tex_endlinechar:D
\__kernel_primitive:NN \eqno                        \tex_eqno:D
\__kernel_primitive:NN \errhelp                     \tex_errhelp:D
\__kernel_primitive:NN \errmessage                  \tex_errmessage:D
\__kernel_primitive:NN \errorcontextlines           \tex_errorcontextlines:D
\__kernel_primitive:NN \errorstopmode               \tex_errorstopmode:D
\__kernel_primitive:NN \escapechar                  \tex_escapechar:D
\__kernel_primitive:NN \everycr                     \tex_everycr:D
\__kernel_primitive:NN \everydisplay                \tex_everydisplay:D
\__kernel_primitive:NN \everyhbox                   \tex_everyhbox:D
```

```
\__kernel_primitive:NN \everyjob                    \tex_everyjob:D
\__kernel_primitive:NN \everymath                   \tex_everymath:D
\__kernel_primitive:NN \everypar                    \tex_everypar:D
\__kernel_primitive:NN \everyvbox                   \tex_everyvbox:D
\__kernel_primitive:NN \exhyphenpenalty             \tex_exhyphenpenalty:D
\__kernel_primitive:NN \expandafter                 \tex_expandafter:D
\__kernel_primitive:NN \fam                         \tex_fam:D
\__kernel_primitive:NN \fi                          \tex_fi:D
\__kernel_primitive:NN \finalhyphendemerits         \tex_finalhyphendemerits:D
\__kernel_primitive:NN \firstmark                   \tex_firstmark:D
\__kernel_primitive:NN \floatingpenalty             \tex_floatingpenalty:D
\__kernel_primitive:NN \font                        \tex_font:D
\__kernel_primitive:NN \fontdimen                   \tex_fontdimen:D
\__kernel_primitive:NN \fontname                    \tex_fontname:D
\__kernel_primitive:NN \futurelet                   \tex_futurelet:D
\__kernel_primitive:NN \gdef                        \tex_gdef:D
\__kernel_primitive:NN \global                      \tex_global:D
\__kernel_primitive:NN \globaldefs                  \tex_globaldefs:D
\__kernel_primitive:NN \halign                      \tex_halign:D
\__kernel_primitive:NN \hangafter                   \tex_hangafter:D
\__kernel_primitive:NN \hangindent                  \tex_hangindent:D
\__kernel_primitive:NN \hbadness                    \tex_hbadness:D
\__kernel_primitive:NN \hbox                        \tex_hbox:D
\__kernel_primitive:NN \hfil                        \tex_hfil:D
\__kernel_primitive:NN \hfill                       \tex_hfill:D
\__kernel_primitive:NN \hfilneg                     \tex_hfilneg:D
\__kernel_primitive:NN \hfuzz                       \tex_hfuzz:D
\__kernel_primitive:NN \hoffset                     \tex_hoffset:D
\__kernel_primitive:NN \holdinginserts              \tex_holdinginserts:D
\__kernel_primitive:NN \hrule                       \tex_hrule:D
\__kernel_primitive:NN \hsize                       \tex_hsize:D
\__kernel_primitive:NN \hskip                       \tex_hskip:D
\__kernel_primitive:NN \hss                         \tex_hss:D
\__kernel_primitive:NN \ht                          \tex_ht:D
\__kernel_primitive:NN \hyphenation                 \tex_hyphenation:D
\__kernel_primitive:NN \hyphenchar                  \tex_hyphenchar:D
\__kernel_primitive:NN \hyphenpenalty               \tex_hyphenpenalty:D
\__kernel_primitive:NN \if                          \tex_if:D
\__kernel_primitive:NN \ifcase                      \tex_ifcase:D
\__kernel_primitive:NN \ifcat                       \tex_ifcat:D
\__kernel_primitive:NN \ifdim                       \tex_ifdim:D
\__kernel_primitive:NN \ifeof                       \tex_ifeof:D
\__kernel_primitive:NN \iffalse                     \tex_iffalse:D
\__kernel_primitive:NN \ifhbox                      \tex_ifhbox:D
\__kernel_primitive:NN \ifhmode                     \tex_ifhmode:D
\__kernel_primitive:NN \ifinner                     \tex_ifinner:D
\__kernel_primitive:NN \ifmmode                     \tex_ifmmode:D
\__kernel_primitive:NN \ifnum                       \tex_ifnum:D
\__kernel_primitive:NN \ifodd                       \tex_ifodd:D
\__kernel_primitive:NN \iftrue                      \tex_iftrue:D
\__kernel_primitive:NN \ifvbox                      \tex_ifvbox:D
\__kernel_primitive:NN \ifvmode                     \tex_ifvmode:D
\__kernel_primitive:NN \ifvoid                      \tex_ifvoid:D
\__kernel_primitive:NN \ifx                         \tex_ifx:D
```

```
\__kernel_primitive:NN \ignorespaces                \tex_ignorespaces:D
\__kernel_primitive:NN \immediate                   \tex_immediate:D
\__kernel_primitive:NN \indent                      \tex_indent:D
\__kernel_primitive:NN \input                       \tex_input:D
\__kernel_primitive:NN \inputlineno                 \tex_inputlineno:D
\__kernel_primitive:NN \insert                      \tex_insert:D
\__kernel_primitive:NN \insertpenalties             \tex_insertpenalties:D
\__kernel_primitive:NN \interlinepenalty            \tex_interlinepenalty:D
\__kernel_primitive:NN \jobname                     \tex_jobname:D
\__kernel_primitive:NN \kern                        \tex_kern:D
\__kernel_primitive:NN \language                    \tex_language:D
\__kernel_primitive:NN \lastbox                     \tex_lastbox:D
\__kernel_primitive:NN \lastkern                    \tex_lastkern:D
\__kernel_primitive:NN \lastpenalty                 \tex_lastpenalty:D
\__kernel_primitive:NN \lastskip                    \tex_lastskip:D
\__kernel_primitive:NN \lccode                      \tex_lccode:D
\__kernel_primitive:NN \leaders                     \tex_leaders:D
\__kernel_primitive:NN \left                        \tex_left:D
\__kernel_primitive:NN \lefthyphenmin               \tex_lefthyphenmin:D
\__kernel_primitive:NN \leftskip                    \tex_leftskip:D
\__kernel_primitive:NN \leqno                       \tex_leqno:D
\__kernel_primitive:NN \let                         \tex_let:D
\__kernel_primitive:NN \limits                      \tex_limits:D
\__kernel_primitive:NN \linepenalty                 \tex_linepenalty:D
\__kernel_primitive:NN \lineskip                    \tex_lineskip:D
\__kernel_primitive:NN \lineskiplimit               \tex_lineskiplimit:D
\__kernel_primitive:NN \long                        \tex_long:D
\__kernel_primitive:NN \looseness                   \tex_looseness:D
\__kernel_primitive:NN \lower                       \tex_lower:D
\__kernel_primitive:NN \lowercase                   \tex_lowercase:D
\__kernel_primitive:NN \mag                         \tex_mag:D
\__kernel_primitive:NN \mark                        \tex_mark:D
\__kernel_primitive:NN \mathaccent                  \tex_mathaccent:D
\__kernel_primitive:NN \mathbin                     \tex_mathbin:D
\__kernel_primitive:NN \mathchar                    \tex_mathchar:D
\__kernel_primitive:NN \mathchardef                 \tex_mathchardef:D
\__kernel_primitive:NN \mathchoice                  \tex_mathchoice:D
\__kernel_primitive:NN \mathclose                   \tex_mathclose:D
\__kernel_primitive:NN \mathcode                    \tex_mathcode:D
\__kernel_primitive:NN \mathinner                   \tex_mathinner:D
\__kernel_primitive:NN \mathop                      \tex_mathop:D
\__kernel_primitive:NN \mathopen                    \tex_mathopen:D
\__kernel_primitive:NN \mathord                     \tex_mathord:D
\__kernel_primitive:NN \mathpunct                   \tex_mathpunct:D
\__kernel_primitive:NN \mathrel                     \tex_mathrel:D
\__kernel_primitive:NN \mathsurround                \tex_mathsurround:D
\__kernel_primitive:NN \maxdeadcycles               \tex_maxdeadcycles:D
\__kernel_primitive:NN \maxdepth                    \tex_maxdepth:D
\__kernel_primitive:NN \meaning                     \tex_meaning:D
\__kernel_primitive:NN \medmuskip                   \tex_medmuskip:D
\__kernel_primitive:NN \message                     \tex_message:D
\__kernel_primitive:NN \mkern                       \tex_mkern:D
\__kernel_primitive:NN \month                       \tex_month:D
\__kernel_primitive:NN \moveleft                    \tex_moveleft:D
```

```
\__kernel_primitive:NN \moveright                   \tex_moveright:D
\__kernel_primitive:NN \mskip                       \tex_mskip:D
\__kernel_primitive:NN \multiply                    \tex_multiply:D
\__kernel_primitive:NN \muskip                      \tex_muskip:D
\__kernel_primitive:NN \muskipdef                   \tex_muskipdef:D
\__kernel_primitive:NN \newlinechar                 \tex_newlinechar:D
\__kernel_primitive:NN \noalign                     \tex_noalign:D
\__kernel_primitive:NN \noboundary                  \tex_noboundary:D
\__kernel_primitive:NN \noexpand                    \tex_noexpand:D
\__kernel_primitive:NN \noindent                    \tex_noindent:D
\__kernel_primitive:NN \nolimits                    \tex_nolimits:D
\__kernel_primitive:NN \nonscript                   \tex_nonscript:D
\__kernel_primitive:NN \nonstopmode                 \tex_nonstopmode:D
\__kernel_primitive:NN \nulldelimiterspace          \tex_nulldelimiterspace:D
\__kernel_primitive:NN \nullfont                    \tex_nullfont:D
\__kernel_primitive:NN \number                      \tex_number:D
\__kernel_primitive:NN \omit                        \tex_omit:D
\__kernel_primitive:NN \openin                      \tex_openin:D
\__kernel_primitive:NN \openout                     \tex_openout:D
\__kernel_primitive:NN \or                          \tex_or:D
\__kernel_primitive:NN \outer                       \tex_outer:D
\__kernel_primitive:NN \output                      \tex_output:D
\__kernel_primitive:NN \outputpenalty               \tex_outputpenalty:D
\__kernel_primitive:NN \over                        \tex_over:D
\__kernel_primitive:NN \overfullrule                \tex_overfullrule:D
\__kernel_primitive:NN \overline                    \tex_overline:D
\__kernel_primitive:NN \overwithdelims              \tex_overwithdelims:D
\__kernel_primitive:NN \pagedepth                   \tex_pagedepth:D
\__kernel_primitive:NN \pagefilllstretch            \tex_pagefilllstretch:D
\__kernel_primitive:NN \pagefillstretch             \tex_pagefillstretch:D
\__kernel_primitive:NN \pagefilstretch              \tex_pagefilstretch:D
\__kernel_primitive:NN \pagegoal                    \tex_pagegoal:D
\__kernel_primitive:NN \pageshrink                  \tex_pageshrink:D
\__kernel_primitive:NN \pagestretch                 \tex_pagestretch:D
\__kernel_primitive:NN \pagetotal                   \tex_pagetotal:D
\__kernel_primitive:NN \par                         \tex_par:D
\__kernel_primitive:NN \parfillskip                 \tex_parfillskip:D
\__kernel_primitive:NN \parindent                   \tex_parindent:D
\__kernel_primitive:NN \parshape                    \tex_parshape:D
\__kernel_primitive:NN \parskip                     \tex_parskip:D
\__kernel_primitive:NN \patterns                    \tex_patterns:D
\__kernel_primitive:NN \pausing                     \tex_pausing:D
\__kernel_primitive:NN \penalty                     \tex_penalty:D
\__kernel_primitive:NN \postdisplaypenalty          \tex_postdisplaypenalty:D
\__kernel_primitive:NN \predisplaypenalty           \tex_predisplaypenalty:D
\__kernel_primitive:NN \predisplaysize              \tex_predisplaysize:D
\__kernel_primitive:NN \pretolerance                \tex_pretolerance:D
\__kernel_primitive:NN \prevdepth                   \tex_prevdepth:D
\__kernel_primitive:NN \prevgraf                    \tex_prevgraf:D
\__kernel_primitive:NN \radical                     \tex_radical:D
\__kernel_primitive:NN \raise                       \tex_raise:D
\__kernel_primitive:NN \read                        \tex_read:D
\__kernel_primitive:NN \relax                       \tex_relax:D
\__kernel_primitive:NN \relpenalty                  \tex_relpenalty:D
```

```
\__kernel_primitive:NN \right                       \tex_right:D
\__kernel_primitive:NN \righthyphenmin              \tex_righthyphenmin:D
\__kernel_primitive:NN \rightskip                   \tex_rightskip:D
\__kernel_primitive:NN \romannumeral                \tex_romannumeral:D
\__kernel_primitive:NN \scriptfont                  \tex_scriptfont:D
\__kernel_primitive:NN \scriptscriptfont            \tex_scriptscriptfont:D
\__kernel_primitive:NN \scriptscriptstyle           \tex_scriptscriptstyle:D
\__kernel_primitive:NN \scriptspace                 \tex_scriptspace:D
\__kernel_primitive:NN \scriptstyle                 \tex_scriptstyle:D
\__kernel_primitive:NN \scrollmode                  \tex_scrollmode:D
\__kernel_primitive:NN \setbox                      \tex_setbox:D
\__kernel_primitive:NN \setlanguage                 \tex_setlanguage:D
\__kernel_primitive:NN \sfcode                      \tex_sfcode:D
\__kernel_primitive:NN \shipout                     \tex_shipout:D
\__kernel_primitive:NN \show                        \tex_show:D
\__kernel_primitive:NN \showbox                     \tex_showbox:D
\__kernel_primitive:NN \showboxbreadth              \tex_showboxbreadth:D
\__kernel_primitive:NN \showboxdepth                \tex_showboxdepth:D
\__kernel_primitive:NN \showlists                   \tex_showlists:D
\__kernel_primitive:NN \showthe                     \tex_showthe:D
\__kernel_primitive:NN \skewchar                    \tex_skewchar:D
\__kernel_primitive:NN \skip                        \tex_skip:D
\__kernel_primitive:NN \skipdef                     \tex_skipdef:D
\__kernel_primitive:NN \spacefactor                 \tex_spacefactor:D
\__kernel_primitive:NN \spaceskip                   \tex_spaceskip:D
\__kernel_primitive:NN \span                        \tex_span:D
\__kernel_primitive:NN \special                     \tex_special:D
\__kernel_primitive:NN \splitbotmark                \tex_splitbotmark:D
\__kernel_primitive:NN \splitfirstmark              \tex_splitfirstmark:D
\__kernel_primitive:NN \splitmaxdepth               \tex_splitmaxdepth:D
\__kernel_primitive:NN \splittopskip                \tex_splittopskip:D
\__kernel_primitive:NN \string                      \tex_string:D
\__kernel_primitive:NN \tabskip                     \tex_tabskip:D
\__kernel_primitive:NN \textfont                    \tex_textfont:D
\__kernel_primitive:NN \textstyle                   \tex_textstyle:D
\__kernel_primitive:NN \the                         \tex_the:D
\__kernel_primitive:NN \thickmuskip                 \tex_thickmuskip:D
\__kernel_primitive:NN \thinmuskip                  \tex_thinmuskip:D
\__kernel_primitive:NN \time                        \tex_time:D
\__kernel_primitive:NN \toks                        \tex_toks:D
\__kernel_primitive:NN \toksdef                     \tex_toksdef:D
\__kernel_primitive:NN \tolerance                   \tex_tolerance:D
\__kernel_primitive:NN \topmark                     \tex_topmark:D
\__kernel_primitive:NN \topskip                     \tex_topskip:D
\__kernel_primitive:NN \tracingcommands             \tex_tracingcommands:D
\__kernel_primitive:NN \tracinglostchars            \tex_tracinglostchars:D
\__kernel_primitive:NN \tracingmacros               \tex_tracingmacros:D
\__kernel_primitive:NN \tracingonline               \tex_tracingonline:D
\__kernel_primitive:NN \tracingoutput               \tex_tracingoutput:D
\__kernel_primitive:NN \tracingpages                \tex_tracingpages:D
\__kernel_primitive:NN \tracingparagraphs           \tex_tracingparagraphs:D
\__kernel_primitive:NN \tracingrestores             \tex_tracingrestores:D
\__kernel_primitive:NN \tracingstats                \tex_tracingstats:D
\__kernel_primitive:NN \uccode                      \tex_uccode:D
```

```
\__kernel_primitive:NN \uchyph                      \tex_uchyph:D
\__kernel_primitive:NN \underline                   \tex_underline:D
\__kernel_primitive:NN \unhbox                      \tex_unhbox:D
\__kernel_primitive:NN \unhcopy                     \tex_unhcopy:D
\__kernel_primitive:NN \unkern                      \tex_unkern:D
\__kernel_primitive:NN \unpenalty                   \tex_unpenalty:D
\__kernel_primitive:NN \unskip                      \tex_unskip:D
\__kernel_primitive:NN \unvbox                      \tex_unvbox:D
\__kernel_primitive:NN \unvcopy                     \tex_unvcopy:D
\__kernel_primitive:NN \uppercase                   \tex_uppercase:D
\__kernel_primitive:NN \vadjust                     \tex_vadjust:D
\__kernel_primitive:NN \valign                      \tex_valign:D
\__kernel_primitive:NN \vbadness                    \tex_vbadness:D
\__kernel_primitive:NN \vbox                        \tex_vbox:D
\__kernel_primitive:NN \vcenter                     \tex_vcenter:D
\__kernel_primitive:NN \vfil                        \tex_vfil:D
\__kernel_primitive:NN \vfill                       \tex_vfill:D
\__kernel_primitive:NN \vfilneg                     \tex_vfilneg:D
\__kernel_primitive:NN \vfuzz                       \tex_vfuzz:D
\__kernel_primitive:NN \voffset                     \tex_voffset:D
\__kernel_primitive:NN \vrule                       \tex_vrule:D
\__kernel_primitive:NN \vsize                       \tex_vsize:D
\__kernel_primitive:NN \vskip                       \tex_vskip:D
\__kernel_primitive:NN \vsplit                      \tex_vsplit:D
\__kernel_primitive:NN \vss                         \tex_vss:D
\__kernel_primitive:NN \vtop                        \tex_vtop:D
\__kernel_primitive:NN \wd                          \tex_wd:D
\__kernel_primitive:NN \widowpenalty                \tex_widowpenalty:D
\__kernel_primitive:NN \write                       \tex_write:D
\__kernel_primitive:NN \xdef                        \tex_xdef:D
\__kernel_primitive:NN \xleaders                    \tex_xleaders:D
\__kernel_primitive:NN \xspaceskip                  \tex_xspaceskip:D
\__kernel_primitive:NN \year                        \tex_year:D
```

Since LaTeX3 requires at least the ε-TeX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

```
\__kernel_primitive:NN \beginL                      \etex_beginL:D
\__kernel_primitive:NN \beginR                      \etex_beginR:D
\__kernel_primitive:NN \botmarks                    \etex_botmarks:D
\__kernel_primitive:NN \clubpenalties               \etex_clubpenalties:D
\__kernel_primitive:NN \currentgrouplevel           \etex_currentgrouplevel:D
\__kernel_primitive:NN \currentgrouptype            \etex_currentgrouptype:D
\__kernel_primitive:NN \currentifbranch             \etex_currentifbranch:D
\__kernel_primitive:NN \currentiflevel              \etex_currentiflevel:D
\__kernel_primitive:NN \currentiftype               \etex_currentiftype:D
\__kernel_primitive:NN \detokenize                  \etex_detokenize:D
\__kernel_primitive:NN \dimexpr                     \etex_dimexpr:D
\__kernel_primitive:NN \displaywidowpenalties       \etex_displaywidowpenalties:D
\__kernel_primitive:NN \endL                        \etex_endL:D
\__kernel_primitive:NN \endR                        \etex_endR:D
\__kernel_primitive:NN \eTeXrevision                \etex_eTeXrevision:D
\__kernel_primitive:NN \eTeXversion                 \etex_eTeXversion:D
\__kernel_primitive:NN \everyeof                    \etex_everyeof:D
\__kernel_primitive:NN \firstmarks                  \etex_firstmarks:D
```

```
\__kernel_primitive:NN \fontchardp                  \etex_fontchardp:D
\__kernel_primitive:NN \fontcharht                  \etex_fontcharht:D
\__kernel_primitive:NN \fontcharic                  \etex_fontcharic:D
\__kernel_primitive:NN \fontcharwd                  \etex_fontcharwd:D
\__kernel_primitive:NN \glueexpr                    \etex_glueexpr:D
\__kernel_primitive:NN \glueshrink                  \etex_glueshrink:D
\__kernel_primitive:NN \glueshrinkorder             \etex_glueshrinkorder:D
\__kernel_primitive:NN \gluestretch                 \etex_gluestretch:D
\__kernel_primitive:NN \gluestretchorder            \etex_gluestretchorder:D
\__kernel_primitive:NN \gluetomu                    \etex_gluetomu:D
\__kernel_primitive:NN \ifcsname                    \etex_ifcsname:D
\__kernel_primitive:NN \ifdefined                   \etex_ifdefined:D
\__kernel_primitive:NN \iffontchar                  \etex_iffontchar:D
\__kernel_primitive:NN \interactionmode             \etex_interactionmode:D
\__kernel_primitive:NN \interlinepenalties          \etex_interlinepenalties:D
\__kernel_primitive:NN \lastlinefit                 \etex_lastlinefit:D
\__kernel_primitive:NN \lastnodetype                \etex_lastnodetype:D
\__kernel_primitive:NN \marks                       \etex_marks:D
\__kernel_primitive:NN \middle                      \etex_middle:D
\__kernel_primitive:NN \muexpr                      \etex_muexpr:D
\__kernel_primitive:NN \mutoglue                    \etex_mutoglue:D
\__kernel_primitive:NN \numexpr                     \etex_numexpr:D
\__kernel_primitive:NN \pagediscards                \etex_pagediscards:D
\__kernel_primitive:NN \parshapedimen               \etex_parshapedimen:D
\__kernel_primitive:NN \parshapeindent              \etex_parshapeindent:D
\__kernel_primitive:NN \parshapelength              \etex_parshapelength:D
\__kernel_primitive:NN \predisplaydirection         \etex_predisplaydirection:D
\__kernel_primitive:NN \protected                   \etex_protected:D
\__kernel_primitive:NN \readline                    \etex_readline:D
\__kernel_primitive:NN \savinghyphcodes             \etex_savinghyphcodes:D
\__kernel_primitive:NN \savingvdiscards             \etex_savingvdiscards:D
\__kernel_primitive:NN \scantokens                  \etex_scantokens:D
\__kernel_primitive:NN \showgroups                  \etex_showgroups:D
\__kernel_primitive:NN \showifs                     \etex_showifs:D
\__kernel_primitive:NN \showtokens                  \etex_showtokens:D
\__kernel_primitive:NN \splitbotmarks               \etex_splitbotmarks:D
\__kernel_primitive:NN \splitdiscards               \etex_splitdiscards:D
\__kernel_primitive:NN \splitfirstmarks             \etex_splitfirstmarks:D
\__kernel_primitive:NN \TeXXeTstate                 \etex_TeXXeTstate:D
\__kernel_primitive:NN \topmarks                    \etex_topmarks:D
\__kernel_primitive:NN \tracingassigns              \etex_tracingassigns:D
\__kernel_primitive:NN \tracinggroups               \etex_tracinggroups:D
\__kernel_primitive:NN \tracingifs                  \etex_tracingifs:D
\__kernel_primitive:NN \tracingnesting              \etex_tracingnesting:D
\__kernel_primitive:NN \tracingscantokens           \etex_tracingscantokens:D
\__kernel_primitive:NN \unexpanded                  \etex_unexpanded:D
\__kernel_primitive:NN \unless                      \etex_unless:D
\__kernel_primitive:NN \widowpenalties              \etex_widowpenalties:D
```

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective, based on those also available in LuaTEX or used in expl3. In the case of the pdfTEX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTEX primitives that start `\pdf...` but are not related to PDF output. These ones related to

PDF output or only work in PDF mode.

```
  \__kernel_primitive:NN \pdfannot                    \pdftex_pdfannot:D
  \__kernel_primitive:NN \pdfcatalog                  \pdftex_pdfcatalog:D
  \__kernel_primitive:NN \pdfcompresslevel            \pdftex_pdfcompresslevel:D
  \__kernel_primitive:NN \pdfcolorstack               \pdftex_pdfcolorstack:D
  \__kernel_primitive:NN \pdfcolorstackinit           \pdftex_pdfcolorstackinit:D
  \__kernel_primitive:NN \pdfcreationdate             \pdftex_pdfcreationdate:D
  \__kernel_primitive:NN \pdfdecimaldigits            \pdftex_pdfdecimaldigits:D
  \__kernel_primitive:NN \pdfdest                     \pdftex_pdfdest:D
  \__kernel_primitive:NN \pdfdestmargin               \pdftex_pdfdestmargin:D
  \__kernel_primitive:NN \pdfendlink                  \pdftex_pdfendlink:D
  \__kernel_primitive:NN \pdfendthread                \pdftex_pdfendthread:D
  \__kernel_primitive:NN \pdffontattr                 \pdftex_pdffontattr:D
  \__kernel_primitive:NN \pdffontname                 \pdftex_pdffontname:D
  \__kernel_primitive:NN \pdffontobjnum               \pdftex_pdffontobjnum:D
  \__kernel_primitive:NN \pdfgamma                    \pdftex_pdfgamma:D
  \__kernel_primitive:NN \pdfimageapplygamma          \pdftex_pdfimageapplygamma:D
  \__kernel_primitive:NN \pdfimagegamma               \pdftex_pdfimagegamma:D
  \__kernel_primitive:NN \pdfgentounicode             \pdftex_pdfgentounicode:D
  \__kernel_primitive:NN \pdfglyphtounicode           \pdftex_pdfglyphtounicode:D
  \__kernel_primitive:NN \pdfhorigin                  \pdftex_pdfhorigin:D
  \__kernel_primitive:NN \pdfimagehicolor             \pdftex_pdfimagehicolor:D
  \__kernel_primitive:NN \pdfimageresolution          \pdftex_pdfimageresolution:D
  \__kernel_primitive:NN \pdfincludechars             \pdftex_pdfincludechars:D
  \__kernel_primitive:NN \pdfinclusioncopyfonts       \pdftex_pdfinclusioncopyfonts:D
  \__kernel_primitive:NN \pdfinclusionerrorlevel      \pdftex_pdfinclusionerrorlevel:D
  \__kernel_primitive:NN \pdfinfo                     \pdftex_pdfinfo:D
  \__kernel_primitive:NN \pdflastannot                \pdftex_pdflastannot:D
  \__kernel_primitive:NN \pdflastlink                 \pdftex_pdflastlink:D
  \__kernel_primitive:NN \pdflastobj                  \pdftex_pdflastobj:D
  \__kernel_primitive:NN \pdflastxform                \pdftex_pdflastxform:D
  \__kernel_primitive:NN \pdflastximage               \pdftex_pdflastximage:D
  \__kernel_primitive:NN \pdflastximagecolordepth     \pdftex_pdflastximagecolordepth:D
  \__kernel_primitive:NN \pdflastximagepages          \pdftex_pdflastximagepages:D
  \__kernel_primitive:NN \pdflinkmargin               \pdftex_pdflinkmargin:D
  \__kernel_primitive:NN \pdfliteral                  \pdftex_pdfliteral:D
  \__kernel_primitive:NN \pdfminorversion             \pdftex_pdfminorversion:D
  \__kernel_primitive:NN \pdfnames                    \pdftex_pdfnames:D
  \__kernel_primitive:NN \pdfobj                      \pdftex_pdfobj:D
  \__kernel_primitive:NN \pdfobjcompresslevel         \pdftex_pdfobjcompresslevel:D
  \__kernel_primitive:NN \pdfoutline                  \pdftex_pdfoutline:D
  \__kernel_primitive:NN \pdfoutput                   \pdftex_pdfoutput:D
  \__kernel_primitive:NN \pdfpageattr                 \pdftex_pdfpageattr:D
  \__kernel_primitive:NN \pdfpagebox                  \pdftex_pdfpagebox:D
  \__kernel_primitive:NN \pdfpageref                  \pdftex_pdfpageref:D
  \__kernel_primitive:NN \pdfpageresources            \pdftex_pdfpageresources:D
  \__kernel_primitive:NN \pdfpagesattr                \pdftex_pdfpagesattr:D
  \__kernel_primitive:NN \pdfrefobj                   \pdftex_pdfrefobj:D
  \__kernel_primitive:NN \pdfrefxform                 \pdftex_pdfrefxform:D
  \__kernel_primitive:NN \pdfrefximage                \pdftex_pdfrefximage:D
  \__kernel_primitive:NN \pdfrestore                  \pdftex_pdfrestore:D
  \__kernel_primitive:NN \pdfretval                   \pdftex_pdfretval:D
  \__kernel_primitive:NN \pdfsave                     \pdftex_pdfsave:D
  \__kernel_primitive:NN \pdfsetmatrix                \pdftex_pdfsetmatrix:D
```

```
\__kernel_primitive:NN \pdfstartlink                \pdftex_pdfstartlink:D
\__kernel_primitive:NN \pdfstartthread              \pdftex_pdfstartthread:D
\__kernel_primitive:NN \pdfsuppressptexinfo         \pdftex_pdfsuppressptexinfo:D
\__kernel_primitive:NN \pdfthread                   \pdftex_pdfthread:D
\__kernel_primitive:NN \pdfthreadmargin             \pdftex_pdfthreadmargin:D
\__kernel_primitive:NN \pdftrailer                  \pdftex_pdftrailer:D
\__kernel_primitive:NN \pdfuniqueresname            \pdftex_pdfuniqueresname:D
\__kernel_primitive:NN \pdfvorigin                  \pdftex_pdfvorigin:D
\__kernel_primitive:NN \pdfxform                    \pdftex_pdfxform:D
\__kernel_primitive:NN \pdfxformattr                \pdftex_pdfxformattr:D
\__kernel_primitive:NN \pdfxformname                \pdftex_pdfxformname:D
\__kernel_primitive:NN \pdfxformresources           \pdftex_pdfxformresources:D
\__kernel_primitive:NN \pdfximage                   \pdftex_pdfximage:D
\__kernel_primitive:NN \pdfximagebbox               \pdftex_pdfximagebbox:D
```

While these are not.

```
\__kernel_primitive:NN \ifpdfabsdim                 \pdftex_ifabsdim:D
\__kernel_primitive:NN \ifpdfabsnum                 \pdftex_ifabsnum:D
\__kernel_primitive:NN \ifpdfprimitive              \pdftex_ifprimitive:D
\__kernel_primitive:NN \pdfadjustspacing            \pdftex_adjustspacing:D
\__kernel_primitive:NN \pdfcopyfont                 \pdftex_copyfont:D
\__kernel_primitive:NN \pdfdraftmode                \pdftex_draftmode:D
\__kernel_primitive:NN \pdfeachlinedepth            \pdftex_eachlinedepth:D
\__kernel_primitive:NN \pdfeachlineheight           \pdftex_eachlineheight:D
\__kernel_primitive:NN \pdffilemoddate              \pdftex_filemoddate:D
\__kernel_primitive:NN \pdffilesize                 \pdftex_filesize:D
\__kernel_primitive:NN \pdffirstlineheight          \pdftex_firstlineheight:D
\__kernel_primitive:NN \pdffontexpand               \pdftex_fontexpand:D
\__kernel_primitive:NN \pdffontsize                 \pdftex_fontsize:D
\__kernel_primitive:NN \pdfignoreddimen             \pdftex_ignoreddimen:D
\__kernel_primitive:NN \pdfinsertht                 \pdftex_insertht:D
\__kernel_primitive:NN \pdflastlinedepth            \pdftex_lastlinedepth:D
\__kernel_primitive:NN \pdflastxpos                 \pdftex_lastxpos:D
\__kernel_primitive:NN \pdflastypos                 \pdftex_lastypos:D
\__kernel_primitive:NN \pdfmapfile                  \pdftex_mapfile:D
\__kernel_primitive:NN \pdfmapline                  \pdftex_mapline:D
\__kernel_primitive:NN \pdfmdfivesum                \pdftex_mdfivesum:D
\__kernel_primitive:NN \pdfnoligatures              \pdftex_noligatures:D
\__kernel_primitive:NN \pdfnormaldeviate            \pdftex_normaldeviate:D
\__kernel_primitive:NN \pdfpageheight               \pdftex_pageheight:D
\__kernel_primitive:NN \pdfpagewidth                \pdftex_pagewidth:D
\__kernel_primitive:NN \pdfpkmode                   \pdftex_pkmode:D
\__kernel_primitive:NN \pdfpkresolution             \pdftex_pkresolution:D
\__kernel_primitive:NN \pdfprimitive                \pdftex_primitive:D
\__kernel_primitive:NN \pdfprotrudechars            \pdftex_protrudechars:D
\__kernel_primitive:NN \pdfpxdimen                  \pdftex_pxdimen:D
\__kernel_primitive:NN \pdfrandomseed               \pdftex_randomseed:D
\__kernel_primitive:NN \pdfsavepos                  \pdftex_savepos:D
\__kernel_primitive:NN \pdfstrcmp                   \pdftex_strcmp:D
\__kernel_primitive:NN \pdfsetrandomseed            \pdftex_setrandomseed:D
\__kernel_primitive:NN \pdfshellescape              \pdftex_shellescape:D
\__kernel_primitive:NN \pdftracingfonts             \pdftex_tracingfonts:D
\__kernel_primitive:NN \pdfuniformdeviate           \pdftex_uniformdeviate:D
```

The version primitives are not related to PDF mode but are related to pdfTEX so retain the full prefix.

```
  \__kernel_primitive:NN \pdftexbanner                \pdftex_pdftexbanner:D
  \__kernel_primitive:NN \pdftexrevision              \pdftex_pdftexrevision:D
  \__kernel_primitive:NN \pdftexversion               \pdftex_pdftexversion:D
```

These ones appear in pdfTEX but don't have `pdf` in the name at all. (`\synctex` is odd as it's really not from pdfTEX but from SyncTeX!)

```
  \__kernel_primitive:NN \efcode                      \pdftex_efcode:D
  \__kernel_primitive:NN \ifincsname                  \pdftex_ifincsname:D
  \__kernel_primitive:NN \leftmarginkern              \pdftex_leftmarginkern:D
  \__kernel_primitive:NN \letterspacefont             \pdftex_letterspacefont:D
  \__kernel_primitive:NN \lpcode                      \pdftex_lpcode:D
  \__kernel_primitive:NN \quitvmode                   \pdftex_quitvmode:D
  \__kernel_primitive:NN \rightmarginkern             \pdftex_rightmarginkern:D
  \__kernel_primitive:NN \rpcode                      \pdftex_rpcode:D
  \__kernel_primitive:NN \synctex                     \pdftex_synctex:D
  \__kernel_primitive:NN \tagcode                     \pdftex_tagcode:D
```

Post pdfTEX primitive availability gets more complex. Both XƎTEX and LuaTEX have varying names for some primitives from pdfTEX. Particularly for LuaTEX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```
⟨/initex | names | package⟩
⟨*initex | package⟩
  \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
  \tex_long:D \tex_def:D \use_none:n #1 { }
  \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
    {
      \etex_ifdefined:D #1
        \tex_expandafter:D \use_ii:nn
      \tex_fi:D
        \use_none:n { \tex_global:D \tex_let:D #2 #1 }
⟨*initex⟩
      \tex_global:D \tex_let:D #1 \tex_undefined:D
⟨/initex⟩
    }
⟨/initex | package⟩
⟨*initex | names | package⟩
```

XƎTEX-specific primitives. Note that XƎTEX's `\strcmp` is handled earlier and is "rolled up" into `\pdfstrcmp`. With the exception of the version primitives these don't carry `XeTeX` through into the "base" name. A few cross-compatibility names which lack the `pdf` of the original are handled later.

```
  \__kernel_primitive:NN \suppressfontnotfounderror   \xetex_suppressfontnotfounderror:D
  \__kernel_primitive:NN \XeTeXcharclass              \xetex_charclass:D
  \__kernel_primitive:NN \XeTeXcharglyph              \xetex_charglyph:D
  \__kernel_primitive:NN \XeTeXcountfeatures          \xetex_countfeatures:D
  \__kernel_primitive:NN \XeTeXcountglyphs            \xetex_countglyphs:D
  \__kernel_primitive:NN \XeTeXcountselectors         \xetex_countselectors:D
  \__kernel_primitive:NN \XeTeXcountvariations        \xetex_countvariations:D
  \__kernel_primitive:NN \XeTeXdefaultencoding        \xetex_defaultencoding:D
  \__kernel_primitive:NN \XeTeXdashbreakstate         \xetex_dashbreakstate:D
  \__kernel_primitive:NN \XeTeXfeaturecode            \xetex_featurecode:D
  \__kernel_primitive:NN \XeTeXfeaturename            \xetex_featurename:D
```

```
\__kernel_primitive:NN \XeTeXfindfeaturebyname      \xetex_findfeaturebyname:D
\__kernel_primitive:NN \XeTeXfindselectorbyname     \xetex_findselectorbyname:D
\__kernel_primitive:NN \XeTeXfindvariationbyname    \xetex_findvariationbyname:D
\__kernel_primitive:NN \XeTeXfirstfontchar          \xetex_firstfontchar:D
\__kernel_primitive:NN \XeTeXfonttype               \xetex_fonttype:D
\__kernel_primitive:NN \XeTeXgenerateactualtext     \xetex_generateactualtext:D
\__kernel_primitive:NN \XeTeXglyph                  \xetex_glyph:D
\__kernel_primitive:NN \XeTeXglyphbounds            \xetex_glyphbounds:D
\__kernel_primitive:NN \XeTeXglyphindex             \xetex_glyphindex:D
\__kernel_primitive:NN \XeTeXglyphname              \xetex_glyphname:D
\__kernel_primitive:NN \XeTeXinputencoding          \xetex_inputencoding:D
\__kernel_primitive:NN \XeTeXinputnormalization     \xetex_inputnormalization:D
\__kernel_primitive:NN \XeTeXinterchartokenstate    \xetex_interchartokenstate:D
\__kernel_primitive:NN \XeTeXinterchartoks          \xetex_interchartoks:D
\__kernel_primitive:NN \XeTeXisdefaultselector      \xetex_isdefaultselector:D
\__kernel_primitive:NN \XeTeXisexclusivefeature     \xetex_isexclusivefeature:D
\__kernel_primitive:NN \XeTeXlastfontchar           \xetex_lastfontchar:D
\__kernel_primitive:NN \XeTeXlinebreakskip          \xetex_linebreakskip:D
\__kernel_primitive:NN \XeTeXlinebreaklocale        \xetex_linebreaklocale:D
\__kernel_primitive:NN \XeTeXlinebreakpenalty       \xetex_linebreakpenalty:D
\__kernel_primitive:NN \XeTeXOTcountfeatures        \xetex_OTcountfeatures:D
\__kernel_primitive:NN \XeTeXOTcountlanguages       \xetex_OTcountlanguages:D
\__kernel_primitive:NN \XeTeXOTcountscripts         \xetex_OTcountscripts:D
\__kernel_primitive:NN \XeTeXOTfeaturetag           \xetex_OTfeaturetag:D
\__kernel_primitive:NN \XeTeXOTlanguagetag          \xetex_OTlanguagetag:D
\__kernel_primitive:NN \XeTeXOTscripttag            \xetex_OTscripttag:D
\__kernel_primitive:NN \XeTeXpdffile                \xetex_pdffile:D
\__kernel_primitive:NN \XeTeXpdfpagecount           \xetex_pdfpagecount:D
\__kernel_primitive:NN \XeTeXpicfile                \xetex_picfile:D
\__kernel_primitive:NN \XeTeXselectorname           \xetex_selectorname:D
\__kernel_primitive:NN \XeTeXtracingfonts           \xetex_tracingfonts:D
\__kernel_primitive:NN \XeTeXupwardsmode            \xetex_upwardsmode:D
\__kernel_primitive:NN \XeTeXuseglyphmetrics        \xetex_useglyphmetrics:D
\__kernel_primitive:NN \XeTeXvariation              \xetex_variation:D
\__kernel_primitive:NN \XeTeXvariationdefault       \xetex_variationdefault:D
\__kernel_primitive:NN \XeTeXvariationmax           \xetex_variationmax:D
\__kernel_primitive:NN \XeTeXvariationmin           \xetex_variationmin:D
\__kernel_primitive:NN \XeTeXvariationname          \xetex_variationname:D
```

The version primitives retain `XeTeX`.

```
\__kernel_primitive:NN \XeTeXrevision               \xetex_XeTeXrevision:D
\__kernel_primitive:NN \XeTeXversion                \xetex_XeTeXversion:D
```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```
\__kernel_primitive:NN \mdfivesum                   \pdftex_mdfivesum:D
\__kernel_primitive:NN \ifprimitive                 \pdftex_ifprimitive:D
\__kernel_primitive:NN \primitive                   \pdftex_primitive:D
\__kernel_primitive:NN \shellescape                 \pdftex_shellescape:D
```

Primitives from LuaTeX, some of which have been ported back to XeTeX. Notice that `\expanded` was intended for pdfTeX 1.50 but as that was not released we call this a LuaTeX primitive.

```
\__kernel_primitive:NN \alignmark                   \luatex_alignmark:D
\__kernel_primitive:NN \aligntab                    \luatex_aligntab:D
\__kernel_primitive:NN \attribute                   \luatex_attribute:D
```

```
  \__kernel_primitive:NN \attributedef                \luatex_attributedef:D
  \__kernel_primitive:NN \automatichyphenpenalty      \luatex_automatichyphenpenalty:D
  \__kernel_primitive:NN \begincsname                 \luatex_begincsname:D
  \__kernel_primitive:NN \catcodetable                \luatex_catcodetable:D
  \__kernel_primitive:NN \clearmarks                  \luatex_clearmarks:D
  \__kernel_primitive:NN \crampeddisplaystyle         \luatex_crampeddisplaystyle:D
  \__kernel_primitive:NN \crampedscriptscriptstyle    \luatex_crampedscriptscriptstyle:D
  \__kernel_primitive:NN \crampedscriptstyle          \luatex_crampedscriptstyle:D
  \__kernel_primitive:NN \crampedtextstyle            \luatex_crampedtextstyle:D
  \__kernel_primitive:NN \directlua                   \luatex_directlua:D
  \__kernel_primitive:NN \dviextension                \luatex_dviextension:D
  \__kernel_primitive:NN \dvifeedback                 \luatex_dvifeedback:D
  \__kernel_primitive:NN \dvivariable                 \luatex_dvivariable:D
  \__kernel_primitive:NN \etoksapp                    \luatex_etoksapp:D
  \__kernel_primitive:NN \etokspre                    \luatex_etokspre:D
  \__kernel_primitive:NN \explicithyphenpenalty       \luatex_explicithyphenpenalty:D
  \__kernel_primitive:NN \expanded                    \luatex_expanded:D
  \__kernel_primitive:NN \firstvalidlanguage          \luatex_firstvalidlanguage:D
  \__kernel_primitive:NN \fontid                      \luatex_fontid:D
  \__kernel_primitive:NN \formatname                  \luatex_formatname:D
  \__kernel_primitive:NN \hjcode                      \luatex_hjcode:D
  \__kernel_primitive:NN \hpack                       \luatex_hpack:D
  \__kernel_primitive:NN \hyphenationbounds           \luatex_hyphenationbounds:D
  \__kernel_primitive:NN \hyphenationmin              \luatex_hyphenationmin:D
  \__kernel_primitive:NN \hyphenpenaltymode           \luatex_hyphenpenaltymode:D
  \__kernel_primitive:NN \gleaders                    \luatex_gleaders:D
  \__kernel_primitive:NN \initcatcodetable            \luatex_initcatcodetable:D
  \__kernel_primitive:NN \lastnamedcs                 \luatex_lastnamedcs:D
  \__kernel_primitive:NN \latelua                     \luatex_latelua:D
  \__kernel_primitive:NN \letcharcode                 \luatex_letcharcode:D
  \__kernel_primitive:NN \luaescapestring             \luatex_luaescapestring:D
  \__kernel_primitive:NN \luafunction                 \luatex_luafunction:D
  \__kernel_primitive:NN \luatexbanner                \luatex_luatexbanner:D
  \__kernel_primitive:NN \luatexdatestamp             \luatex_luatexdatestamp:D
  \__kernel_primitive:NN \luatexrevision              \luatex_luatexrevision:D
  \__kernel_primitive:NN \luatexversion               \luatex_luatexversion:D
  \__kernel_primitive:NN \mathdisplayskipmode         \luatex_mathdisplayskipmode:D
  \__kernel_primitive:NN \matheqnogapstep             \luatex_matheqnogapstep:D
  \__kernel_primitive:NN \mathnolimitsmode            \luatex_mathnolimitsmode:D
  \__kernel_primitive:NN \mathoption                  \luatex_mathoption:D
  \__kernel_primitive:NN \mathrulesfam                \luatex_mathrulesfam:D
  \__kernel_primitive:NN \mathscriptsmode             \luatex_mathscriptsmode:D
  \__kernel_primitive:NN \mathstyle                   \luatex_mathstyle:D
  \__kernel_primitive:NN \mathsurroundmode            \luatex_mathsurroundmode:D
  \__kernel_primitive:NN \mathsurroundskip            \luatex_mathsurroundskip:D
  \__kernel_primitive:NN \nohrule                     \luatex_nohrule:D
  \__kernel_primitive:NN \nokerns                     \luatex_nokerns:D
  \__kernel_primitive:NN \noligs                      \luatex_noligs:D
  \__kernel_primitive:NN \nospaces                    \luatex_nospaces:D
  \__kernel_primitive:NN \novrule                     \luatex_novrule:D
  \__kernel_primitive:NN \outputbox                   \luatex_outputbox:D
  \__kernel_primitive:NN \pagebottomoffset            \luatex_pagebottomoffset:D
  \__kernel_primitive:NN \pageleftoffset              \luatex_pageleftoffset:D
  \__kernel_primitive:NN \pagerightoffset             \luatex_pagerightoffset:D
```

```
\__kernel_primitive:NN \pagetopoffset               \luatex_pagetopoffset:D
\__kernel_primitive:NN \pdfextension                \luatex_pdfextension:D
\__kernel_primitive:NN \pdffeedback                 \luatex_pdffeedback:D
\__kernel_primitive:NN \pdfvariable                 \luatex_pdfvariable:D
\__kernel_primitive:NN \postexhyphenchar            \luatex_postexhyphenchar:D
\__kernel_primitive:NN \posthyphenchar              \luatex_posthyphenchar:D
\__kernel_primitive:NN \predisplaygapfactor         \luatex_predisplaygapfactor:D
\__kernel_primitive:NN \preexhyphenchar             \luatex_preexhyphenchar:D
\__kernel_primitive:NN \prehyphenchar               \luatex_prehyphenchar:D
\__kernel_primitive:NN \savecatcodetable            \luatex_savecatcodetable:D
\__kernel_primitive:NN \scantextokens               \luatex_scantextokens:D
\__kernel_primitive:NN \setfontid                   \luatex_setfontid:D
\__kernel_primitive:NN \shapemode                   \luatex_shapemode:D
\__kernel_primitive:NN \suppressifcsnameerror       \luatex_suppressifcsnameerror:D
\__kernel_primitive:NN \suppresslongerror           \luatex_suppresslongerror:D
\__kernel_primitive:NN \suppressmathparerror        \luatex_suppressmathparerror:D
\__kernel_primitive:NN \suppressoutererror          \luatex_suppressoutererror:D
\__kernel_primitive:NN \toksapp                     \luatex_toksapp:D
\__kernel_primitive:NN \tokspre                     \luatex_tokspre:D
\__kernel_primitive:NN \tpack                       \luatex_tpack:D
\__kernel_primitive:NN \vpack                       \luatex_vpack:D
```

Slightly more awkward are the directional primitives in LuaTEX. These come from Omega/Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTEX primitives for prefix purposes. One here is "new" but fits into the general set.

```
\__kernel_primitive:NN \bodydir                     \luatex_bodydir:D
\__kernel_primitive:NN \boxdir                      \luatex_boxdir:D
\__kernel_primitive:NN \leftghost                   \luatex_leftghost:D
\__kernel_primitive:NN \linedir                     \luatex_linedir:D
\__kernel_primitive:NN \localbrokenpenalty          \luatex_localbrokenpenalty:D
\__kernel_primitive:NN \localinterlinepenalty       \luatex_localinterlinepenalty:D
\__kernel_primitive:NN \localleftbox                \luatex_localleftbox:D
\__kernel_primitive:NN \localrightbox               \luatex_localrightbox:D
\__kernel_primitive:NN \mathdir                     \luatex_mathdir:D
\__kernel_primitive:NN \pagedir                     \luatex_pagedir:D
\__kernel_primitive:NN \pardir                      \luatex_pardir:D
\__kernel_primitive:NN \rightghost                  \luatex_rightghost:D
\__kernel_primitive:NN \textdir                     \luatex_textdir:D
```

Primitives from pdfTEX that LuaTEX renames.

```
\__kernel_primitive:NN \adjustspacing               \pdftex_adjustspacing:D
\__kernel_primitive:NN \copyfont                    \pdftex_copyfont:D
\__kernel_primitive:NN \draftmode                   \pdftex_draftmode:D
\__kernel_primitive:NN \expandglyphsinfont          \pdftex_fontexpand:D
\__kernel_primitive:NN \ifabsdim                    \pdftex_ifabsdim:D
\__kernel_primitive:NN \ifabsnum                    \pdftex_ifabsnum:D
\__kernel_primitive:NN \ignoreligaturesinfont       \pdftex_ignoreligaturesinfont:D
\__kernel_primitive:NN \insertht                    \pdftex_insertht:D
\__kernel_primitive:NN \lastsavedboxresourceindex   \pdftex_pdflastxform:D
\__kernel_primitive:NN \lastsavedimageresourceindex \pdftex_pdflastximage:D
\__kernel_primitive:NN \lastsavedimageresourcepages \pdftex_pdflastximagepages:D
\__kernel_primitive:NN \lastxpos                    \pdftex_lastxpos:D
\__kernel_primitive:NN \lastypos                    \pdftex_lastypos:D
\__kernel_primitive:NN \normaldeviate               \pdftex_normaldeviate:D
```

```
\__kernel_primitive:NN \outputmode                   \pdftex_pdfoutput:D
\__kernel_primitive:NN \pageheight                   \pdftex_pageheight:D
\__kernel_primitive:NN \pagewidth                    \pdftex_pagewith:D
\__kernel_primitive:NN \protrudechars                \pdftex_protrudechars:D
\__kernel_primitive:NN \pxdimen                      \pdftex_pxdimen:D
\__kernel_primitive:NN \randomseed                   \pdftex_randomseed:D
\__kernel_primitive:NN \useboxresource               \pdftex_pdfrefxform:D
\__kernel_primitive:NN \useimageresource             \pdftex_pdfrefximage:D
\__kernel_primitive:NN \savepos                      \pdftex_savepos:D
\__kernel_primitive:NN \saveboxresource              \pdftex_pdfxform:D
\__kernel_primitive:NN \saveimageresource            \pdftex_pdfximage:D
\__kernel_primitive:NN \setrandomseed                \pdftex_setrandomseed:D
\__kernel_primitive:NN \tracingfonts                 \pdftex_tracingfonts:D
\__kernel_primitive:NN \uniformdeviate               \pdftex_uniformdeviate:D
```

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as `\XeTeX...` which were then renamed with LuaTeX having a lot more. These names now all start `\U...` and mainly `\Umath....` To keep things somewhat clear we therefore prefix all of these as `\utex...` (introduced by a Unicode TeX engine) and drop `\U(math)` from the names. Where there is a related TeX90 primitive or where it really seems required we keep the `math` part of the name.

```
\__kernel_primitive:NN \Uchar                        \utex_char:D
\__kernel_primitive:NN \Ucharcat                     \utex_charcat:D
\__kernel_primitive:NN \Udelcode                     \utex_delcode:D
\__kernel_primitive:NN \Udelcodenum                  \utex_delcodenum:D
\__kernel_primitive:NN \Udelimiter                   \utex_delimiter:D
\__kernel_primitive:NN \Udelimiterover               \utex_delimiterover:D
\__kernel_primitive:NN \Udelimiterunder              \utex_delimiterunder:D
\__kernel_primitive:NN \Uhextensible                 \utex_hextensible:D
\__kernel_primitive:NN \Umathaccent                  \utex_mathaccent:D
\__kernel_primitive:NN \Umathaxis                    \utex_mathaxis:D
\__kernel_primitive:NN \Umathbinbinspacing           \utex_binbinspacing:D
\__kernel_primitive:NN \Umathbinclosespacing         \utex_binclosespacing:D
\__kernel_primitive:NN \Umathbininnerspacing         \utex_bininnerspacing:D
\__kernel_primitive:NN \Umathbinopenspacing          \utex_binopenspacing:D
\__kernel_primitive:NN \Umathbinopspacing            \utex_binopspacing:D
\__kernel_primitive:NN \Umathbinordspacing           \utex_binordspacing:D
\__kernel_primitive:NN \Umathbinpunctspacing         \utex_binpunctspacing:D
\__kernel_primitive:NN \Umathbinrelspacing           \utex_binrelspacing:D
\__kernel_primitive:NN \Umathchar                    \utex_mathchar:D
\__kernel_primitive:NN \Umathcharclass               \utex_mathcharclass:D
\__kernel_primitive:NN \Umathchardef                 \utex_mathchardef:D
\__kernel_primitive:NN \Umathcharfam                 \utex_mathcharfam:D
\__kernel_primitive:NN \Umathcharnum                 \utex_mathcharnum:D
\__kernel_primitive:NN \Umathcharnumdef              \utex_mathcharnumdef:D
\__kernel_primitive:NN \Umathcharslot                \utex_mathcharslot:D
\__kernel_primitive:NN \Umathclosebinspacing         \utex_closebinspacing:D
\__kernel_primitive:NN \Umathcloseclosespacing       \utex_closeclosespacing:D
\__kernel_primitive:NN \Umathcloseinnerspacing       \utex_closeinnerspacing:D
\__kernel_primitive:NN \Umathcloseopenspacing        \utex_closeopenspacing:D
\__kernel_primitive:NN \Umathcloseopspacing          \utex_closeopspacing:D
\__kernel_primitive:NN \Umathcloseordspacing         \utex_closeordspacing:D
\__kernel_primitive:NN \Umathclosepunctspacing       \utex_closepunctspacing:D
\__kernel_primitive:NN \Umathcloserelspacing         \utex_closerelspacing:D
```

```
\__kernel_primitive:NN \Umathcode                   \utex_mathcode:D
\__kernel_primitive:NN \Umathcodenum                \utex_mathcodenum:D
\__kernel_primitive:NN \Umathconnectoroverlapmin    \utex_connectoroverlapmin:D
\__kernel_primitive:NN \Umathfractiondelsize        \utex_fractiondelsize:D
\__kernel_primitive:NN \Umathfractiondenomdown      \utex_fractiondenomdown:D
\__kernel_primitive:NN \Umathfractiondenomvgap      \utex_fractiondenomvgap:D
\__kernel_primitive:NN \Umathfractionnumup          \utex_fractionnumup:D
\__kernel_primitive:NN \Umathfractionnumvgap        \utex_fractionnumvgap:D
\__kernel_primitive:NN \Umathfractionrule           \utex_fractionrule:D
\__kernel_primitive:NN \Umathinnerbinspacing        \utex_innerbinspacing:D
\__kernel_primitive:NN \Umathinnerclosespacing      \utex_innerclosespacing:D
\__kernel_primitive:NN \Umathinnerinnerspacing      \utex_innerinnerspacing:D
\__kernel_primitive:NN \Umathinneropenspacing       \utex_inneropenspacing:D
\__kernel_primitive:NN \Umathinneropspacing         \utex_inneropspacing:D
\__kernel_primitive:NN \Umathinnerordspacing        \utex_innerordspacing:D
\__kernel_primitive:NN \Umathinnerpunctspacing      \utex_innerpunctspacing:D
\__kernel_primitive:NN \Umathinnerrelspacing        \utex_innerrelspacing:D
\__kernel_primitive:NN \Umathlimitabovebgap         \utex_limitabovebgap:D
\__kernel_primitive:NN \Umathlimitabovekern         \utex_limitabovekern:D
\__kernel_primitive:NN \Umathlimitabovevgap         \utex_limitabovevgap:D
\__kernel_primitive:NN \Umathlimitbelowbgap         \utex_limitbelowbgap:D
\__kernel_primitive:NN \Umathlimitbelowkern         \utex_limitbelowkern:D
\__kernel_primitive:NN \Umathlimitbelowvgap         \utex_limitbelowvgap:D
\__kernel_primitive:NN \Umathnolimitsubfactor       \utex_nolimitsubfactor:D
\__kernel_primitive:NN \Umathnolimitsupfactor       \utex_nolimitsupfactor:D
\__kernel_primitive:NN \Umathopbinspacing           \utex_opbinspacing:D
\__kernel_primitive:NN \Umathopclosespacing         \utex_opclosespacing:D
\__kernel_primitive:NN \Umathopenbinspacing         \utex_openbinspacing:D
\__kernel_primitive:NN \Umathopenclosespacing       \utex_openclosespacing:D
\__kernel_primitive:NN \Umathopeninnerspacing       \utex_openinnerspacing:D
\__kernel_primitive:NN \Umathopenopenspacing        \utex_openopenspacing:D
\__kernel_primitive:NN \Umathopenopspacing          \utex_openopspacing:D
\__kernel_primitive:NN \Umathopenordspacing         \utex_openordspacing:D
\__kernel_primitive:NN \Umathopenpunctspacing       \utex_openpunctspacing:D
\__kernel_primitive:NN \Umathopenrelspacing         \utex_openrelspacing:D
\__kernel_primitive:NN \Umathoperatorsize           \utex_operatorsize:D
\__kernel_primitive:NN \Umathopinnerspacing         \utex_opinnerspacing:D
\__kernel_primitive:NN \Umathopopenspacing          \utex_opopenspacing:D
\__kernel_primitive:NN \Umathopopspacing            \utex_opopspacing:D
\__kernel_primitive:NN \Umathopordspacing           \utex_opordspacing:D
\__kernel_primitive:NN \Umathoppunctspacing         \utex_oppunctspacing:D
\__kernel_primitive:NN \Umathoprelspacing           \utex_oprelspacing:D
\__kernel_primitive:NN \Umathordbinspacing          \utex_ordbinspacing:D
\__kernel_primitive:NN \Umathordclosespacing        \utex_ordclosespacing:D
\__kernel_primitive:NN \Umathordinnerspacing        \utex_ordinnerspacing:D
\__kernel_primitive:NN \Umathordopenspacing         \utex_ordopenspacing:D
\__kernel_primitive:NN \Umathordopspacing           \utex_ordopspacing:D
\__kernel_primitive:NN \Umathordordspacing          \utex_ordordspacing:D
\__kernel_primitive:NN \Umathordpunctspacing        \utex_ordpunctspacing:D
\__kernel_primitive:NN \Umathordrelspacing          \utex_ordrelspacing:D
\__kernel_primitive:NN \Umathoverbarkern            \utex_overbarkern:D
\__kernel_primitive:NN \Umathoverbarrule            \utex_overbarrule:D
\__kernel_primitive:NN \Umathoverbarvgap            \utex_overbarvgap:D
\__kernel_primitive:NN \Umathoverdelimiterbgap      \utex_overdelimiterbgap:D
```

```
\__kernel_primitive:NN \Umathoverdelimitervgap      \utex_overdelimitervgap:D
\__kernel_primitive:NN \Umathpunctbinspacing        \utex_punctbinspacing:D
\__kernel_primitive:NN \Umathpunctclosespacing      \utex_punctclosespacing:D
\__kernel_primitive:NN \Umathpunctinnerspacing      \utex_punctinnerspacing:D
\__kernel_primitive:NN \Umathpunctopenspacing       \utex_punctopenspacing:D
\__kernel_primitive:NN \Umathpunctopspacing         \utex_punctopspacing:D
\__kernel_primitive:NN \Umathpunctordspacing        \utex_punctordspacing:D
\__kernel_primitive:NN \Umathpunctpunctspacing      \utex_punctpunctspacing:D
\__kernel_primitive:NN \Umathpunctrelspacing        \utex_punctrelspacing:D
\__kernel_primitive:NN \Umathquad                   \utex_quad:D
\__kernel_primitive:NN \Umathradicaldegreeafter     \utex_radicaldegreeafter:D
\__kernel_primitive:NN \Umathradicaldegreebefore    \utex_radicaldegreebefore:D
\__kernel_primitive:NN \Umathradicaldegreeraise     \utex_radicaldegreeraise:D
\__kernel_primitive:NN \Umathradicalkern            \utex_radicalkern:D
\__kernel_primitive:NN \Umathradicalrule            \utex_radicalrule:D
\__kernel_primitive:NN \Umathradicalvgap            \utex_radicalvgap:D
\__kernel_primitive:NN \Umathrelbinspacing          \utex_relbinspacing:D
\__kernel_primitive:NN \Umathrelclosespacing        \utex_relclosespacing:D
\__kernel_primitive:NN \Umathrelinnerspacing        \utex_relinnerspacing:D
\__kernel_primitive:NN \Umathrelopenspacing         \utex_relopenspacing:D
\__kernel_primitive:NN \Umathrelopspacing           \utex_relopspacing:D
\__kernel_primitive:NN \Umathrelordspacing          \utex_relordspacing:D
\__kernel_primitive:NN \Umathrelpunctspacing        \utex_relpunctspacing:D
\__kernel_primitive:NN \Umathrelrelspacing          \utex_relrelspacing:D
\__kernel_primitive:NN \Umathskewedfractionhgap     \utex_skewedfractionhgap:D
\__kernel_primitive:NN \Umathskewedfractionvgap     \utex_skewedfractionvgap:D
\__kernel_primitive:NN \Umathspaceafterscript       \utex_spaceafterscript:D
\__kernel_primitive:NN \Umathstackdenomdown         \utex_stackdenomdown:D
\__kernel_primitive:NN \Umathstacknumup             \utex_stacknumup:D
\__kernel_primitive:NN \Umathstackvgap              \utex_stackvgap:D
\__kernel_primitive:NN \Umathsubshiftdown           \utex_subshiftdown:D
\__kernel_primitive:NN \Umathsubshiftdrop           \utex_subshiftdrop:D
\__kernel_primitive:NN \Umathsubsupshiftdown        \utex_subsupshiftdown:D
\__kernel_primitive:NN \Umathsubsupvgap             \utex_subsupvgap:D
\__kernel_primitive:NN \Umathsubtopmax              \utex_subtopmax:D
\__kernel_primitive:NN \Umathsupbottommin           \utex_supbottommin:D
\__kernel_primitive:NN \Umathsupshiftdrop           \utex_supshiftdrop:D
\__kernel_primitive:NN \Umathsupshiftup             \utex_supshiftup:D
\__kernel_primitive:NN \Umathsupsubbottommax        \utex_supsubbottommax:D
\__kernel_primitive:NN \Umathunderbarkern           \utex_underbarkern:D
\__kernel_primitive:NN \Umathunderbarrule           \utex_underbarrule:D
\__kernel_primitive:NN \Umathunderbarvgap           \utex_underbarvgap:D
\__kernel_primitive:NN \Umathunderdelimiterbgap     \utex_underdelimiterbgap:D
\__kernel_primitive:NN \Umathunderdelimitervgap     \utex_underdelimitervgap:D
\__kernel_primitive:NN \Uoverdelimiter              \utex_overdelimiter:D
\__kernel_primitive:NN \Uradical                    \utex_radical:D
\__kernel_primitive:NN \Uroot                       \utex_root:D
\__kernel_primitive:NN \Uskewed                     \utex_skewed:D
\__kernel_primitive:NN \Uskewedwithdelims           \utex_skewedwithdelims:D
\__kernel_primitive:NN \Ustack                      \utex_stack:D
\__kernel_primitive:NN \Ustartdisplaymath           \utex_startdisplaymath:D
\__kernel_primitive:NN \Ustartmath                  \utex_startmath:D
\__kernel_primitive:NN \Ustopdisplaymath            \utex_stopdisplaymath:D
\__kernel_primitive:NN \Ustopmath                   \utex_stopmath:D
```

```
\__kernel_primitive:NN \Usubscript                  \utex_subscript:D
\__kernel_primitive:NN \Usuperscript                \utex_superscript:D
\__kernel_primitive:NN \Uunderdelimiter             \utex_underdelimiter:D
\__kernel_primitive:NN \Uvextensible                \utex_vextensible:D
```

Primitives from pTeX.

```
\__kernel_primitive:NN \autospacing                 \ptex_autospacing:D
\__kernel_primitive:NN \autoxspacing                \ptex_autoxspacing:D
\__kernel_primitive:NN \dtou                        \ptex_dtou:D
\__kernel_primitive:NN \euc                         \ptex_euc:D
\__kernel_primitive:NN \ifdbox                      \ptex_ifdbox:D
\__kernel_primitive:NN \ifddir                      \ptex_ifddir:D
\__kernel_primitive:NN \ifmdir                      \ptex_ifmdir:D
\__kernel_primitive:NN \iftbox                      \ptex_iftbox:D
\__kernel_primitive:NN \iftdir                      \ptex_iftdir:D
\__kernel_primitive:NN \ifybox                      \ptex_ifybox:D
\__kernel_primitive:NN \ifydir                      \ptex_ifydir:D
\__kernel_primitive:NN \inhibitglue                 \ptex_inhibitglue:D
\__kernel_primitive:NN \inhibitxspcode              \ptex_inhibitxspcode:D
\__kernel_primitive:NN \jcharwidowpenalty           \ptex_jcharwidowpenalty:D
\__kernel_primitive:NN \jfam                        \ptex_jfam:D
\__kernel_primitive:NN \jfont                       \ptex_jfont:D
\__kernel_primitive:NN \jis                         \ptex_jis:D
\__kernel_primitive:NN \kanjiskip                   \ptex_kanjiskip:D
\__kernel_primitive:NN \kansuji                     \ptex_kansuji:D
\__kernel_primitive:NN \kansujichar                 \ptex_kansujichar:D
\__kernel_primitive:NN \kcatcode                    \ptex_kcatcode:D
\__kernel_primitive:NN \kuten                       \ptex_kuten:D
\__kernel_primitive:NN \noautospacing               \ptex_noautospacing:D
\__kernel_primitive:NN \noautoxspacing              \ptex_noautoxspacing:D
\__kernel_primitive:NN \postbreakpenalty            \ptex_postbreakpenalty:D
\__kernel_primitive:NN \prebreakpenalty             \ptex_prebreakpenalty:D
\__kernel_primitive:NN \showmode                    \ptex_showmode:D
\__kernel_primitive:NN \sjis                        \ptex_sjis:D
\__kernel_primitive:NN \tate                        \ptex_tate:D
\__kernel_primitive:NN \tbaselineshift              \ptex_tbaselineshift:D
\__kernel_primitive:NN \tfont                       \ptex_tfont:D
\__kernel_primitive:NN \xkanjiskip                  \ptex_xkanjiskip:D
\__kernel_primitive:NN \xspcode                     \ptex_xspcode:D
\__kernel_primitive:NN \ybaselineshift              \ptex_ybaselineshift:D
\__kernel_primitive:NN \yoko                        \ptex_yoko:D
```

Primitives from upTeX.

```
\__kernel_primitive:NN \disablecjktoken             \uptex_disablecjktoken:D
\__kernel_primitive:NN \enablecjktoken              \uptex_enablecjktoken:D
\__kernel_primitive:NN \forcecjktoken               \uptex_forcecjktoken:D
\__kernel_primitive:NN \kchar                       \uptex_kchar:D
\__kernel_primitive:NN \kchardef                    \uptex_kchardef:D
\__kernel_primitive:NN \kuten                       \uptex_kuten:D
\__kernel_primitive:NN \ucs                         \uptex_ucs:D
```

End of the “just the names” part of the source.

```
⟨/initex | names | package⟩
⟨*initex | package⟩
```

The job is done: close the group (using the primitive renamed!).

```
\tex_endgroup:D
```

LaTeX 2ε moves a few primitives, so these are sorted out. A convenient test for LaTeX 2ε is the `\@@end` saved primitive.

```
⟨*package⟩
\etex_ifdefined:D \@@end
  \tex_let:D \tex_end:D                  \@@end
  \tex_let:D \tex_everydisplay:D         \frozen@everydisplay
  \tex_let:D \tex_everymath:D            \frozen@everymath
  \tex_let:D \tex_hyphen:D               \@@hyph
  \tex_let:D \tex_input:D                \@@input
  \tex_let:D \tex_italiccorrection:D     \@@italiccorr
  \tex_let:D \tex_underline:D            \@@underline
```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the LaTeX 2ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@@@tracingfonts` available: this might be useful and almost all LaTeX 2ε users will have expl3 loaded by fontspec. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```
  \tex_let:D \pdftex_tracingfonts:D \tex_undefined:D
  \etex_ifdefined:D \pdftracingfonts
    \tex_let:D \pdftex_tracingfonts:D \pdftracingfonts
  \tex_else:D
    \etex_ifdefined:D \luatex_directlua:D
      \luatex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
      \tex_let:D \pdftex_tracingfonts:D \luatextracingfonts
    \tex_fi:D
  \tex_fi:D
\tex_fi:D
```

That is also true for the LuaTeX primitives under LaTeX 2ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```
\etex_ifdefined:D \luatexsuppressfontnotfounderror
  \tex_let:D \luatex_alignmark:D                 \luatexalignmark
  \tex_let:D \luatex_aligntab:D                  \luatexaligntab
  \tex_let:D \luatex_attribute:D                 \luatexattribute
  \tex_let:D \luatex_attributedef:D              \luatexattributedef
  \tex_let:D \luatex_catcodetable:D              \luatexcatcodetable
  \tex_let:D \luatex_clearmarks:D                \luatexclearmarks
  \tex_let:D \luatex_crampeddisplaystyle:D       \luatexcrampeddisplaystyle
  \tex_let:D \luatex_crampedscriptscriptstyle:D  \luatexcrampedscriptscriptstyle
  \tex_let:D \luatex_crampedscriptstyle:D        \luatexcrampedscriptstyle
  \tex_let:D \luatex_crampedtextstyle:D          \luatexcrampedtextstyle
  \tex_let:D \luatex_fontid:D                    \luatexfontid
  \tex_let:D \luatex_formatname:D                \luatexformatname
  \tex_let:D \luatex_gleaders:D                  \luatexgleaders
  \tex_let:D \luatex_initcatcodetable:D          \luatexinitcatcodetable
  \tex_let:D \luatex_latelua:D                   \luatexlatelua
  \tex_let:D \luatex_luaescapestring:D           \luatexluaescapestring
  \tex_let:D \luatex_luafunction:D               \luatexluafunction
  \tex_let:D \luatex_mathstyle:D                 \luatexmathstyle
  \tex_let:D \luatex_nokerns:D                  \luatexnokerns
  \tex_let:D \luatex_noligs:D                   \luatexnoligs
  \tex_let:D \luatex_outputbox:D                \luatexoutputbox
  \tex_let:D \luatex_pageleftoffset:D           \luatexpageleftoffset
  \tex_let:D \luatex_pagetopoffset:D            \luatexpagetopoffset
  \tex_let:D \luatex_postexhyphenchar:D         \luatexpostexhyphenchar
  \tex_let:D \luatex_posthyphenchar:D           \luatexposthyphenchar
  \tex_let:D \luatex_preexhyphenchar:D          \luatexpreexhyphenchar
  \tex_let:D \luatex_prehyphenchar:D            \luatexprehyphenchar
  \tex_let:D \luatex_savecatcodetable:D         \luatexsavecatcodetable
  \tex_let:D \luatex_scantextokens:D            \luatexscantextokens
  \tex_let:D \luatex_suppressifcsnameerror:D    \luatexsuppressifcsnameerror
  \tex_let:D \luatex_suppresslongerror:D        \luatexsuppresslongerror
  \tex_let:D \luatex_suppressmathparerror:D     \luatexsuppressmathparerror
  \tex_let:D \luatex_suppressoutererror:D       \luatexsuppressoutererror
  \tex_let:D \utex_char:D                       \luatexUchar
  \tex_let:D \xetex_suppressfontnotfounderror:D \luatexsuppressfontnotfounderror
```

Which also covers those slightly odd ones.

```
  \tex_let:D \luatex_bodydir:D               \luatexbodydir
  \tex_let:D \luatex_boxdir:D                \luatexboxdir
  \tex_let:D \luatex_leftghost:D             \luatexleftghost
  \tex_let:D \luatex_localbrokenpenalty:D    \luatexlocalbrokenpenalty
  \tex_let:D \luatex_localinterlinepenalty:D \luatexlocalinterlinepenalty
  \tex_let:D \luatex_localleftbox:D          \luatexlocalleftbox
  \tex_let:D \luatex_localrightbox:D         \luatexlocalrightbox
  \tex_let:D \luatex_mathdir:D               \luatexmathdir
  \tex_let:D \luatex_pagebottomoffset:D      \luatexpagebottomoffset
  \tex_let:D \luatex_pagedir:D               \luatexpagedir
  \tex_let:D \pdftex_pageheight:D            \luatexpageheight
  \tex_let:D \luatex_pagerightoffset:D       \luatexpagerightoffset
  \tex_let:D \pdftex_pagewidth:D             \luatexpagewidth
  \tex_let:D \luatex_pardir:D                \luatexpardir
  \tex_let:D \luatex_rightghost:D            \luatexrightghost
  \tex_let:D \luatex_textdir:D               \luatextextdir
\tex_fi:D
```

Only pdfTeX and LuaTeX define `\pdfmapfile` and `\pdfmapline`: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```
\tex_ifnum:D 0
  \etex_ifdefined:D \pdftex_pdftexversion:D 1 \tex_fi:D
  \etex_ifdefined:D \luatex_luatexversion:D 1 \tex_fi:D
    = 0 %
  \tex_let:D \pdftex_mapfile:D \tex_undefined:D
  \tex_let:D \pdftex_mapline:D \tex_undefined:D
\tex_fi:D
⟨/package⟩
```

Older XeTeX versions use `\XeTeX` as the prefix for the Unicode math primitives it knows. That is tided up here (we support XeTeX versions from 0.9994 but this change was in 0.9999).

```
⟨*initex | package⟩
\etex_ifdefined:D \XeTeXdelcode
  \tex_let:D \utex_delcode:D        \XeTeXdelcode
  \tex_let:D \utex_delcodenum:D     \XeTeXdelcodenum
  \tex_let:D \utex_delimiter:D      \XeTeXdelimiter
  \tex_let:D \utex_mathaccent:D     \XeTeXmathaccent
  \tex_let:D \utex_mathchar:D       \XeTeXmathchar
  \tex_let:D \utex_mathchardef:D    \XeTeXmathchardef
  \tex_let:D \utex_mathcharnum:D    \XeTeXmathcharnum
  \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
  \tex_let:D \utex_mathcode:D       \XeTeXmathcode
  \tex_let:D \utex_mathcodenum:D    \XeTeXmathcodenum
\tex_fi:D
```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that `\pdftex_pdftexversion:D` is a marker for pdfTeX alone: useful in engine-dependent code later.

```
\etex_ifdefined:D \luatex_luatexversion:D
  \tex_let:D \pdftex_pdftexbanner:D   \tex_undefined:D
  \tex_let:D \pdftex_pdftexrevision:D \tex_undefined:D
  \tex_let:D \pdftex_pdftexversion:D  \tex_undefined:D
\tex_fi:D
⟨/initex | package⟩
```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```
⟨*package⟩
\etex_ifdefined:D \normalend
  \tex_let:D \tex_end:D         \normalend
  \tex_let:D \tex_everyjob:D    \normaleveryjob
  \tex_let:D \tex_input:D       \normalinput
  \tex_let:D \tex_language:D    \normallanguage
  \tex_let:D \tex_mathop:D      \normalmathop
  \tex_let:D \tex_month:D       \normalmonth
  \tex_let:D \tex_outer:D       \normalouter
  \tex_let:D \tex_over:D        \normalover
  \tex_let:D \tex_vcenter:D     \normalvcenter
  \tex_let:D \etex_unexpanded:D \normalunexpanded
  \tex_let:D \luatex_expanded:D \normalexpanded
\tex_fi:D
\etex_ifdefined:D \normalitaliccorrection
  \tex_let:D \tex_hoffset:D          \normalhoffset
  \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
  \tex_let:D \tex_voffset:D          \normalvoffset
  \tex_let:D \etex_showtokens:D      \normalshowtokens
  \tex_let:D \luatex_bodydir:D       \spac_directions_normal_body_dir
  \tex_let:D \luatex_pagedir:D       \spac_directions_normal_page_dir
\tex_fi:D
\etex_ifdefined:D \normalleft
  \tex_let:D \tex_left:D   \normalleft
  \tex_let:D \tex_middle:D \normalmiddle
  \tex_let:D \tex_right:D  \normalright
\tex_fi:D
⟨/package⟩

⟨/initex | package⟩
```

3 l3basics implementation

```
⟨*initex | package⟩
```

3.1 Renaming some TEX primitives (again)

Having given all the TEX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.[6]

\if_true:
\if_false:
\or:
\else:
\fi:
\reverse_if:N
\if:w
\if_charcode:w
\if_catcode:w
\if_meaning:w

Then some conditionals.

```
\tex_let:D \if_true:           \tex_iftrue:D
\tex_let:D \if_false:          \tex_iffalse:D
\tex_let:D \or:                \tex_or:D
\tex_let:D \else:              \tex_else:D
\tex_let:D \fi:                \tex_fi:D
\tex_let:D \reverse_if:N       \etex_unless:D
\tex_let:D \if:w               \tex_if:D
\tex_let:D \if_charcode:w      \tex_if:D
\tex_let:D \if_catcode:w       \tex_ifcat:D
\tex_let:D \if_meaning:w       \tex_ifx:D
```

(*End definition for* \if_true: *and others. These functions are documented on page 21.*)

\if_mode_math:
\if_mode_horizontal:
\if_mode_vertical:
\if_mode_inner:

TEX lets us detect some if its modes.

```
\tex_let:D \if_mode_math:       \tex_ifmmode:D
\tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\tex_let:D \if_mode_vertical:   \tex_ifvmode:D
\tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(*End definition for* \if_mode_math: *and others. These functions are documented on page 21.*)

\if_cs_exist:N
\if_cs_exist:w
\cs:w
\cs_end:

Building csnames and testing if control sequences exist.

```
\tex_let:D \if_cs_exist:N      \etex_ifdefined:D
\tex_let:D \if_cs_exist:w      \etex_ifcsname:D
\tex_let:D \cs:w               \tex_csname:D
\tex_let:D \cs_end:            \tex_endcsname:D
```

(*End definition for* \if_cs_exist:N *and others. These functions are documented on page 21.*)

\exp_after:wN
\exp_not:N
\exp_not:n

The five \exp_ functions are used in the l3expan module where they are described.

```
\tex_let:D \exp_after:wN       \tex_expandafter:D
\tex_let:D \exp_not:N          \tex_noexpand:D
\tex_let:D \exp_not:n          \etex_unexpanded:D
\tex_let:D \exp:w              \tex_romannumeral:D
\tex_chardef:D \exp_end:  = 0 ~
```

(*End definition for* \exp_after:wN*,* \exp_not:N*, and* \exp_not:n*. These functions are documented on page 31.*)

\token_to_meaning:N
\cs_meaning:N

Examining a control sequence or token.

```
\tex_let:D \token_to_meaning:N \tex_meaning:D
\tex_let:D \cs_meaning:N       \tex_meaning:D
```

[6] This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the \tex_...:D name in the cases where no good alternative exists.

(*End definition for* `\token_to_meaning:N` *and* `\cs_meaning:N`*. These functions are documented on page 113.*)

`\tl_to_str:n`
`\token_to_str:N`

Making strings.

```
\tex_let:D \tl_to_str:n        \etex_detokenize:D
\tex_let:D \token_to_str:N     \tex_string:D
```

(*End definition for* `\tl_to_str:n` *and* `\token_to_str:N`*. These functions are documented on page 42.*)

`\scan_stop:`
`\group_begin:`
`\group_end:`

The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the l3prg module.

```
\tex_let:D \scan_stop:         \tex_relax:D
\tex_let:D \group_begin:       \tex_begingroup:D
\tex_let:D \group_end:         \tex_endgroup:D
```

(*End definition for* `\scan_stop:`*,* `\group_begin:`*, and* `\group_end:`*. These functions are documented on page 9.*)

```
⟨@@=int⟩
```

`\if_int_compare:w`
`\__int_to_roman:w`

For integers.

```
\tex_let:D \if_int_compare:w   \tex_ifnum:D
\tex_let:D \__int_to_roman:w     \tex_romannumeral:D
```

(*End definition for* `\if_int_compare:w` *and* `\__int_to_roman:w`*. These functions are documented on page 80.*)

`\group_insert_after:N`

Adding material after the end of a group.

```
\tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(*End definition for* `\group_insert_after:N`*. This function is documented on page 9.*)

`\exp_args:Nc`
`\exp_args:cc`

Discussed in l3expan, but needed much earlier.

```
\tex_long:D \tex_def:D \exp_args:Nc #1#2
  { \exp_after:wN #1 \cs:w #2 \cs_end: }
\tex_long:D \tex_def:D \exp_args:cc #1#2
  { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(*End definition for* `\exp_args:Nc` *and* `\exp_args:cc`*. These functions are documented on page 28.*)

`\token_to_meaning:c`
`\token_to_str:c`
`\cs_meaning:c`

A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:NNc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```
\tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
\tex_long:D \tex_def:D \cs_meaning:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \exp_args:Nc \cs_meaning:N {#1} }
    { \tl_to_str:n {undefined} }
  }
\tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(*End definition for* `\token_to_meaning:c`*,* `\token_to_str:c`*, and* `\cs_meaning:c`*. These functions are documented on page 113.*)

3.2 Defining some constants

\c_zero

We need the constant \c_zero which is used by some functions in the l3alloc module. The rest are defined in the l3int module – at least for the ones that can be defined with \tex_chardef:D or \tex_mathchardef:D. For other constants the l3int module is required but it can't be used until the allocation has been set up properly!

```
\tex_chardef:D \c_zero    = 0 ~
```

(*End definition for* \c_zero*. This variable is documented on page 79.*)

\c_max_register_int

This is here as this particular integer is needed both in package mode and to bootstrap l3alloc, and is documented in l3int.

```
\etex_ifdefined:D \luatex_luatexversion:D
  \tex_chardef:D \c_max_register_int = 65 535 ~
\tex_else:D
  \tex_mathchardef:D \c_max_register_int = 32 767 ~
\tex_fi:D
```

(*End definition for* \c_max_register_int*. This variable is documented on page 79.*)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

\cs_set_nopar:Npn
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx

All assignment functions in LaTeX3 should be naturally protected; after all, the TeX primitives for assignments are and it can be a cause of problems if others aren't.

```
\tex_let:D \cs_set_nopar:Npn            \tex_def:D
\tex_let:D \cs_set_nopar:Npx            \tex_edef:D
\etex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
  { \tex_long:D \tex_def:D }
\etex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
  { \tex_long:D \tex_edef:D }
\etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
  { \etex_protected:D \tex_def:D }
\etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
  { \etex_protected:D \tex_edef:D }
\etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
  { \etex_protected:D \tex_long:D \tex_def:D }
\etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
  { \etex_protected:D \tex_long:D \tex_edef:D }
```

(*End definition for* \cs_set_nopar:Npn *and others. These functions are documented on page 11.*)

\cs_gset_nopar:Npn
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npx

Global versions of the above functions.

```
\tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
\tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D
\cs_set_protected:Npn \cs_gset:Npn
  { \tex_long:D \tex_gdef:D }
\cs_set_protected:Npn \cs_gset:Npx
  { \tex_long:D \tex_xdef:D }
\cs_set_protected:Npn \cs_gset_protected_nopar:Npn
  { \etex_protected:D \tex_gdef:D }
\cs_set_protected:Npn \cs_gset_protected_nopar:Npx
  { \etex_protected:D \tex_xdef:D }
```

```
\cs_set_protected:Npn \cs_gset_protected:Npn
  { \etex_protected:D \tex_long:D \tex_gdef:D }
\cs_set_protected:Npn \cs_gset_protected:Npx
  { \etex_protected:D \tex_long:D \tex_xdef:D }
```

(*End definition for* \cs_gset_nopar:Npn *and others. These functions are documented on page 12.*)

3.4 Selecting tokens

```
⟨@@=exp⟩
```

\l__exp_internal_tl

Scratch token list variable for l3expan, used by \use:x, used in defining conditionals. We don't use tl methods because l3basics is loaded earlier.

```
\cs_set_nopar:Npn \l__exp_internal_tl { }
```

(*End definition for* \l__exp_internal_tl.)

\use:c

This macro grabs its argument and returns a csname from it.

```
\cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(*End definition for* \use:c. *This function is documented on page 16.*)

\use:x

Fully expands its argument and passes it to the input stream. Uses the reserved \l__exp_internal_tl which will be set up in l3expan.

```
\cs_set_protected:Npn \use:x #1
  {
    \cs_set_nopar:Npx \l__exp_internal_tl {#1}
    \l__exp_internal_tl
  }
```

(*End definition for* \use:x. *This function is documented on page 19.*)

\use:n
\use:nn
\use:nnn
\use:nnnn

These macros grab their arguments and returns them back to the input (with outer braces removed).

```
\cs_set:Npn \use:n    #1       {#1}
\cs_set:Npn \use:nn   #1#2     {#1#2}
\cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
\cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(*End definition for* \use:n *and others. These functions are documented on page 17.*)

\use_i:nn
\use_ii:nn

The equivalent to LaTeX 2ε's \@firstoftwo and \@secondoftwo.

```
\cs_set:Npn \use_i:nn  #1#2 {#1}
\cs_set:Npn \use_ii:nn #1#2 {#2}
```

(*End definition for* \use_i:nn *and* \use_ii:nn. *These functions are documented on page 18.*)

\use_i:nnn
\use_ii:nnn
\use_iii:nnn
\use_i_ii:nnn
\use_i:nnnn
\use_ii:nnnn
\use_iii:nnnn
\use_iv:nnnn

We also need something for picking up arguments from a longer list.

```
\cs_set:Npn \use_i:nnn    #1#2#3 {#1}
\cs_set:Npn \use_ii:nnn   #1#2#3 {#2}
\cs_set:Npn \use_iii:nnn  #1#2#3 {#3}
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
\cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
\cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
\cs_set:Npn \use_iv:nnnn  #1#2#3#4 {#4}
```

(*End definition for* `\use_i:nnn` *and others. These functions are documented on page 18.*)

`\use_none_delimit_by_q_nil:w`
`\use_none_delimit_by_q_stop:w`
`\use_none_delimit_by_q_recursion_stop:w`

Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\cs_set:Npn \use_none_delimit_by_q_nil:w  #1 \q_nil  { }
\cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
\cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(*End definition for* `\use_none_delimit_by_q_nil:w`*,* `\use_none_delimit_by_q_stop:w`*, and* `\use_none_delimit_by_q_recursion_stop:w`*. These functions are documented on page 19.*)

`\use_i_delimit_by_q_nil:nw`
`\use_i_delimit_by_q_stop:nw`
`\use_i_delimit_by_q_recursion_stop:nw`

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\cs_set:Npn \use_i_delimit_by_q_nil:nw  #1#2 \q_nil  {#1}
\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
\cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(*End definition for* `\use_i_delimit_by_q_nil:nw`*,* `\use_i_delimit_by_q_stop:nw`*, and* `\use_i_delimit_by_q_recursion_stop:nw`*. These functions are documented on page 19.*)

3.5 Gobbling tokens from input

`\use_none:n`
`\use_none:nn`
`\use_none:nnn`
`\use_none:nnnn`
`\use_none:nnnnn`
`\use_none:nnnnnn`
`\use_none:nnnnnnn`
`\use_none:nnnnnnnn`
`\use_none:nnnnnnnnn`

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```
\cs_set:Npn \use_none:n         #1                 { }
\cs_set:Npn \use_none:nn        #1#2               { }
\cs_set:Npn \use_none:nnn       #1#2#3             { }
\cs_set:Npn \use_none:nnnn      #1#2#3#4           { }
\cs_set:Npn \use_none:nnnnn     #1#2#3#4#5         { }
\cs_set:Npn \use_none:nnnnnn    #1#2#3#4#5#6       { }
\cs_set:Npn \use_none:nnnnnnn   #1#2#3#4#5#6#7     { }
\cs_set:Npn \use_none:nnnnnnnn  #1#2#3#4#5#6#7#8   { }
\cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(*End definition for* `\use_none:n` *and others. These functions are documented on page 18.*)

3.6 Debugging and patching later definitions

```
⟨@@=debug⟩
```

`\__debug:TF`

A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. At present, debugging is disabled in the format and in generic mode, while in LaTeX 2_ε mode it is enabled if one of the options `enable-debug`, `log-functions` or `check-declarations` was given.

```
\cs_set_protected:Npn \__debug:TF #1#2 {#2}
⟨*package⟩
\tex_ifodd:D \l@expl@enable@debug@bool
  \cs_set_protected:Npn \__debug:TF #1#2 {#1}
\fi:
⟨/package⟩
```

(*End definition for* `\__debug:TF`*.*)

`\debug_on:n`
`\debug_off:n`

```
\__debug:TF
  {
    \cs_set_protected:Npn \debug_on:n #1
      {
        \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
          {
            \cs_if_exist_use:cF { __debug_##1_on: }
              { \__msg_kernel_error:nnn { kernel } { debug } {##1} }
          }
      }
    \cs_set_protected:Npn \debug_off:n #1
      {
        \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
          {
            \cs_if_exist_use:cF { __debug_##1_off: }
              { \__msg_kernel_error:nnn { kernel } { debug } {##1} }
          }
      }
  }
  {
    \cs_set_protected:Npn \debug_on:n #1
      {
        \__msg_kernel_error:nnx { kernel } { enable-debug }
          { \tl_to_str:n { \debug_on:n {#1} } }
      }
    \cs_set_protected:Npn \debug_off:n #1
      {
        \__msg_kernel_error:nnx { kernel } { enable-debug }
          { \tl_to_str:n { \debug_off:n {#1} } }
      }
  }
```

(*End definition for* `\debug_on:n` *and* `\debug_off:n`*. These functions are documented on page 232.*)

`\__debug_check-declarations_on:`
`\__debug_check-declarations_off:`
`\__debug_chk_var_exist:N`
`\__debug_chk_cs_exist:N`
`\__debug_chk_cs_exist:c`

When debugging is enabled these two functions set up `\__debug_chk_var_exist:N` and `\__debug_chk_cs_exist:N`, two functions that test (when `check-declarations` is active) that their argument is defined.

```
\__debug:TF
  {
    \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_on: }
      {
        \cs_set_protected:Npn \__debug_chk_var_exist:N ##1
          {
            \cs_if_exist:NF ##1
              {
                \__msg_kernel_error:nnx { kernel } { non-declared-variable }
                  { \token_to_str:N ##1 }
              }
          }
        \cs_set_protected:Npn \__debug_chk_cs_exist:N ##1
          {
```

```
      \cs_if_exist:NF ##1
       {
        \__msg_kernel_error:nnx { kernel } { command-not-defined }
         { \token_to_str:N ##1 }
       }
     }
   }
  \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_off: }
   {
    \cs_set_protected:Npn \__debug_chk_var_exist:N ##1 { }
    \cs_set_protected:Npn \__debug_chk_cs_exist:N ##1 { }
   }
  \cs_set_protected:Npn \__debug_chk_cs_exist:c
   { \exp_args:Nc \__debug_chk_cs_exist:N }
  \tex_ifodd:D \l@expl@check@declarations@bool
   \use:c { __debug_check-declarations_on: }
  \else:
   \use:c { __debug_check-declarations_off: }
  \fi:
 }
 { }
```

(*End definition for* `\__debug_check-declarations_on:` *and others.*)

__debug_check-expressions_on:
__debug_check-expressions_off:
__debug_chk_expr:nNnN
__debug_chk_expr_aux:nNnN

When debugging is enabled these two functions set `\__debug_chk_expr:nNnN` to test or not whether the given expression is valid. The idea is to evaluate the expression within a brace group (to catch trailing `\use_none:nn` or similar), then test that the result is what we expect. This is done by turning it to an integer and hitting that with `\tex_romannumeral:D` after replacing the first character by `-0`. If all goes well, that primitive finds a non-positive integer and gives an empty output. If the original expression evaluation stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation (used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_romannumeral:D` does not absorb `\tex_relax:D` the output will be nonempty. Note that `#3` is empty except for mu expressions for which it is `\etex_mutoglue:D` to avoid an "incompatible glue units" error. Note also that if we had omitted the first `\tex_relax:D` then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer expression.

```
\__debug:TF
 {
  \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_on: }
   {
    \cs_set:Npn \__debug_chk_expr:nNnN ##1##2
     {
      \exp_after:wN \__debug_chk_expr_aux:nNnN
      \exp_after:wN { \tex_the:D ##2 ##1 \tex_relax:D }
      ##2
     }
   }
  \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_off: }
   { \cs_set:Npn \__debug_chk_expr:nNnN ##1##2##3##4 {##1} }
  \use:c { __debug_check-expressions_off: }
  \cs_set:Npn \__debug_chk_expr_aux:nNnN #1#2#3#4
   {
        \tl_if_empty:oF
          {
            \tex_romannumeral:D - 0
            \exp_after:wN \use_none:n
            \__int_value:w #3 #2 #1 \tex_relax:D
          }
          {
            \__msg_kernel_expandable_error:nnnn
              { kernel } { expr } {#4} {#1}
          }
        #1
      }
  }
  { }
```

(*End definition for* `\__debug_check-expressions_on:` *and others.*)

`\__debug_log-functions_on:`
`\__debug_log-functions_off:`
`\__debug_log:x`
`\__debug_suspend_log:`
`\__debug_resume_log:`

These two functions (corresponding to the expl3 option `log-functions`) control whether `\__debug_log:x` writes to the log file or not. Since `\iow_log:x` does not yet have its final definition we do not use `\cs_set_eq:NN` (not defined yet anyway). The `\__debug_suspend_log:` function disables `\__debug_log:x` until the matching `\__debug_resume_log:`. These two commands are used to improve the logging for datatypes with multiple parts, currently only coffins. They should come in pairs, which can be nested (this complicates the code here and is currently unused). The function `\exp_not:o` is defined in l3expan later on but `\__debug_suspend_log:` and `\__debug_resume_log:` are not used before that point. Once everything is defined, turn logging on or off depending on what option was given. When debugging is not enabled, simply produce an error.

```
\__debug:TF
  {
    \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_on: }
      {
        \cs_set_protected:Npn \__debug_log:x { \iow_log:x }
        \cs_set_protected:Npn \__debug_suspend_log:
          {
            \cs_set_protected:Npx \__debug_resume_log:
              {
                \cs_set_protected:Npn \__debug_resume_log:
                  { \exp_not:o { \__debug_resume_log: } }
                \cs_set_protected:Npn \__debug_log:x
                  { \exp_not:o { \__debug_log:x } }
              }
            \cs_set_protected:Npn \__debug_log:x { \use_none:n }
          }
        \cs_set_protected:Npn \__debug_resume_log: { }
      }
    \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_off: }
      {
        \cs_set_protected:Npn \__debug_log:x { \use_none:n }
        \cs_set_protected:Npn \__debug_suspend_log: { }
        \cs_set_protected:Npn \__debug_resume_log: { }
      }
    \tex_ifodd:D \l@expl@log@functions@bool
      \use:c { __debug_log-functions_on: }
    \else:
      \use:c { __debug_log-functions_off: }
    \fi:
  }
  { }
```

(*End definition for* `\__debug_log-functions_on:` *and others.*)

`\__debug_deprecation_on:`
`\__debug_deprecation_off:`
`\g__debug_deprecation_on_tl`
`\g__debug_deprecation_off_tl`

Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, filled up by calls to `\__debug_deprecation:nnNNpn` in each module.

```
\__debug:TF
  {
    \cs_set_protected:Npn \__debug_deprecation_on:
      { \g__debug_deprecation_on_tl }
    \cs_set_protected:Npn \__debug_deprecation_off:
      { \g__debug_deprecation_off_tl }
    \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
    \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
  }
  { }
```

(*End definition for* `\__debug_deprecation_on:` *and others.*)

`\__debug_deprecation:nnNNpn`
`\__debug_deprecation_aux:nnNnn`

Grab a definition (at present, must be `\cs_new_protected:Npn`). Add to `\g__debug_deprecation_on_tl` some code that makes the defined macro `#3` outer (and defines it as an error). Add to `\g__debug_deprecation_off_tl` the definition itself. In both cases we undefine the token with `\tex_let:D` to avoid taking a potentially outer macro as the argument of some expl3 function. Finally define the macro itself to produce a warning then redefine and call itself. The macro initially takes no parameters: together with the `x`-expanding assignment and `\exp_not:n` this gives a convenient way of storing the macro's definition in itself in order to only produce the warning once for each macro. If debugging is disabled, `\__debug_deprecation:nnNNpn` lets the definition happen.

```
\__debug:TF
  {
    \cs_set_protected:Npn \__debug_deprecation:nnNNpn #1#2#3#4#5#
      {
        \if_meaning:w \cs_new_protected:Npn #3
        \else:
          \__msg_kernel_error:nnx { kernel } { debug-unpatchable }
            { \token_to_str:N #3 ~(for~deprecation) }
        \fi:
        \__debug_deprecation_aux:nnNnn {#1} {#2} #4 {#5}
      }
    \cs_set_protected:Npn \__debug_deprecation_aux:nnNnn #1#2#3#4#5
      {
        \tl_gput_right:Nn \g__debug_deprecation_on_tl
          {
            \tex_let:D #3 \scan_stop:
            \__deprecation_error:Nnn #3 {#2} {#1}
          }
        \tl_gput_right:Nn \g__debug_deprecation_off_tl
          {
            \tex_let:D #3 \scan_stop:
            \cs_set_protected:Npn #3 #4 {#5}
          }
        \cs_new_protected:Npx #3
          {
            \exp_not:N \__msg_kernel_warning:nnxxx
              { kernel } { deprecated-command }
              {#1} { \token_to_str:N #3 } { \tl_to_str:n {#2} }
            \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
            \exp_not:N #3
          }
      }
  }
  { \cs_set_protected:Npn \__debug_deprecation:nnNNpn #1#2 { } }
```

(*End definition for* __debug_deprecation:nnNNpn *and* __debug_deprecation_aux:nnNnn.)

__debug_patch:nnNNpn
__debug_patch_conditional:nNNpnn
__debug_patch_aux:nnNNnn
__debug_patch_aux:nNNnnn

When debugging is not enabled, __debug_patch:nnNNpn and __debug_patch_conditional:nNNpnn throw the patch away. Otherwise they can be followed by \cs_new:Npn (or similar), and \prg_new_conditional:Npnn (or similar), respectively. In each case, grab the name of the function to be defined and its parameters then insert tokens before and/or after the definition.

```
\__debug:TF
  {
    \cs_set_protected:Npn \__debug_patch:nnNNpn #1#2#3#4#5#
      { \__debug_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
    \cs_set_protected:Npn \__debug_patch_conditional:nNNpnn #1#2#3#4#
      { \__debug_patch_aux:nNNnnn {#1} #2 #3 {#4} }
    \cs_set_protected:Npn \__debug_patch_aux:nnNNnn #1#2#3#4#5#6
      { #3 #4 #5 { #1 #6 #2 } }
    \cs_set_protected:Npn \__debug_patch_aux:nNNnnn #1#2#3#4#5#6
      { #2 #3 #4 {#5} { #1 #6 } }
  }
  {
    \cs_set_protected:Npn \__debug_patch:nnNNpn #1#2 { }
    \cs_set_protected:Npn \__debug_patch_conditional:nNNpnn #1 { }
  }
```

(*End definition for* __debug_patch:nnNNpn *and others.*)

__debug_patch_args:nNNpn
__debug_patch_conditional_args:nNNpnn
__debug_tmp:w
__debug_patch_args_aux:nNNnn
__debug_patch_args_aux:nNNnnn

See __debug_patch:nnNNpn. The first argument is something like {#1}{(#2)}. Define a temporary macro using the ⟨*parameters*⟩ and ⟨*code*⟩ of the definition that follows, then expand that temporary macro in front of the first argument to obtain new ⟨*code*⟩. Then perform the definition as if that new ⟨*code*⟩ was directly typed in the file. To make it easy to expand in the definition, treat it as a "pre"-code to an empty definition.

```
\__debug:TF
  {
    \cs_set_protected:Npn \__debug_patch_args:nNNpn #1#2#3#4#
      { \__debug_patch_args_aux:nNNnn {#1} #2 #3 {#4} }
    \cs_set_protected:Npn \__debug_patch_conditional_args:nNNpnn #1#2#3#4#
      { \__debug_patch_args_aux:nNNnnn {#1} #2 #3 {#4} }
    \cs_set_protected:Npn \__debug_patch_args_aux:nNNnn #1#2#3#4#5
      {
        \cs_set:Npn \__debug_tmp:w #4 {#5}
        \exp_after:wN \__debug_patch_aux:nnNNnn \exp_after:wN
          { \__debug_tmp:w #1 } { } #2 #3 {#4} { }
      }
    \cs_set_protected:Npn \__debug_patch_args_aux:nNNnnn #1#2#3#4#5#6
      {
        \cs_set:Npn \__debug_tmp:w #4 {#6}
        \exp_after:wN \__debug_patch_aux:nNNnnn \exp_after:wN
          { \__debug_tmp:w #1 } #2 #3 {#4} {#5} { }
      }
  }
  {
    \cs_set_protected:Npn \__debug_patch_args:nNNpn #1 { }
    \cs_set_protected:Npn \__debug_patch_conditional_args:nNNpnn #1 { }
  }
```

(*End definition for* `\__debug_patch_args:nNNpn` *and others.*)

3.7 Conditional processing and definitions

```
⟨@@=prg⟩
```

Underneath any predicate function (`_p`) or other conditional forms (`TF`, etc.) is a built-in logic saying that it after all of the testing and processing must return the ⟨*state*⟩ this leaves TEX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:
```

Usually, a TEX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the TEX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:`
`\prg_return_false:`

The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
\cs_set:Npn \prg_return_true:
  { \exp_after:wN \use_i:nn  \exp:w }
\cs_set:Npn \prg_return_false:
  { \exp_after:wN \use_ii:nn \exp:w}
```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn`/`\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(*End definition for* `\prg_return_true:` *and* `\prg_return_false:`. *These functions are documented on page 92.*)

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
__prg_generate_conditional_parm:nnNpnn

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those Npnn type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed {⟨*name*⟩} {⟨*signature*⟩} ⟨*boolean*⟩ {⟨*set or new*⟩} {⟨*maybe protected*⟩} {⟨*parameters*⟩} {TF,...} {⟨*code*⟩} to the auxiliary function responsible for defining all conditionals.

```
\cs_set_protected:Npn \prg_set_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { set } { } }
\cs_set_protected:Npn \prg_new_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { new } { } }
\cs_set_protected:Npn \prg_set_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
\cs_set_protected:Npn \prg_new_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
\cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
  {
    \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
    {#1} {#2} {#4}
  }
```

(*End definition for* \prg_set_conditional:Npnn *and others. These functions are documented on page 90.*)

\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
__prg_generate_conditional_count:nnNnn
__prg_generate_conditional_count:nnNnnnn

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed {⟨*name*⟩} {⟨*signature*⟩} ⟨*boolean*⟩ {⟨*set or new*⟩} {⟨*maybe protected*⟩} {⟨*parameters*⟩} {TF,...} {⟨*code*⟩} to the auxiliary function responsible for defining all conditionals. If the ⟨*signature*⟩ has more than 9 letters, the definition is aborted since TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```
\cs_set_protected:Npn \prg_set_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { set } { } }
\cs_set_protected:Npn \prg_new_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { new } { } }
\cs_set_protected:Npn \prg_set_protected_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
\cs_set_protected:Npn \prg_new_protected_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
\cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
  {
    \__cs_split_function:NN #3 \__prg_generate_conditional_count:nnNnnnn
    {#1} {#2}
  }
\cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
  {
    \__cs_parm_from_arg_count:nnF
      { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
      { \tl_count:n {#2} }
      {
        \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
          { \token_to_str:c { #1 : #2 } }
          { \tl_count:n {#2} }
        \use_none:nn
      }
  }
```

(*End definition for* \prg_set_conditional:Nnn *and others. These functions are documented on page 90.*)

__prg_generate_conditional:nnNnnnnn
__prg_generate_conditional:nnnnnnw

The workhorse here is going through a list of desired forms, *i.e.*, `p`, `TF`, `T` and `F`. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

```
\cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none:nn
    \fi:
    \use:x
      {
        \exp_not:N \__prg_generate_conditional:nnnnnnw
        \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
        \tl_to_str:n {#7}
        \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```
\cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
  {
    \if_meaning:w \q_recursion_tail #7
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_generate_ #7 _form:wnnnnnn }
        \tl_if_empty:nF {#7}
          {
            \__msg_kernel_error:nnxx
              { kernel } { conditional-form-unknown }
              {#7} { \token_to_str:c { #3 : #4 } }
          }
        \use_none:nnnnnnn
      \q_stop
      {#1} {#2} {#3} {#4} {#5} {#6}
    \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
  }
```

(*End definition for* `\__prg_generate_conditional:nnNnnnnn` *and* `\__prg_generate_conditional:nnnnnnw`*.*)

__prg_generate_p_form:wnnnnnn
__prg_generate_TF_form:wnnnnnn
__prg_generate_T_form:wnnnnnn
__prg_generate_F_form:wnnnnnn

How to generate the various forms. Those functions take the following arguments: 1: `set` or `new`, 2: empty or `_protected`, 3: function name 4: signature, 5: parameter text (or empty), 6: replacement. Remember that the logic-returning functions expect two arguments to be present after `\exp_end:`: notice the construction of the different variants relies on this, and that the `TF` and `F` variants will be slightly faster than the `T` version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty.

```
\cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn
    #1 \q_stop #2#3#4#5#6#7
  {
    \if_meaning:w \scan_stop: #3 \scan_stop:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
          { #7 \exp_end: \c_true_bool \c_false_bool }
      }
      {
        \__msg_kernel_error:nnx { kernel } { protected-predicate }
          { \token_to_str:c { #4 _p: #5 } }
      }
  }
\cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn
    #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
      { #7 \exp_end: \use:n \use_none:n }
  }
\cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn
    #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
      { #7 \exp_end: { } }
  }
\cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn
    #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
      { #7 \exp_end: }
  }
```

(*End definition for* `\__prg_generate_p_form:wnnnnnn` *and others.*)

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
__prg_set_eq_conditional:NNNn

The setting-equal functions. Split both functions and feed `{`⟨*name*$_1$⟩`}` `{`⟨*signature*$_1$⟩`}` ⟨*boolean*$_1$⟩ `{`⟨*name*$_2$⟩`}` `{`⟨*signature*$_2$⟩`}` ⟨*boolean*$_2$⟩ ⟨*copying function*⟩ ⟨*conditions*⟩ `,` `\q_recursion_tail` `,` `\q_recursion_stop` to a first auxiliary.

```
\cs_set_protected:Npn \prg_set_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
\cs_set_protected:Npn \prg_new_eq_conditional:NNn
```

```
  { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
\cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
  {
    \use:x
      {
        \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
          \__cs_split_function:NN #2 \prg_do_nothing:
          \__cs_split_function:NN #3 \prg_do_nothing:
          \exp_not:N #1
          \tl_to_str:n {#4}
          \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
```

(*End definition for* `\prg_set_eq_conditional:NNn`*,* `\prg_new_eq_conditional:NNn`*, and* `\__prg_set_eq_conditional:NNNn`*. These functions are documented on page 91.*)

__prg_set_eq_conditional:nnNnnNNw
__prg_set_eq_conditional_loop:nnnnNw
__prg_set_eq_conditional_p_form:nnn
__prg_set_eq_conditional_TF_form:nnn
__prg_set_eq_conditional_T_form:nnn
__prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument `#6` of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments {⟨*name*$_1$⟩} {⟨*signature*$_1$⟩} {⟨*name*$_2$⟩} {⟨*signature*$_2$⟩} ⟨*copying function*⟩ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
\cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \if_meaning:w \c_false_bool #6
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#4} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
  }
\cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
  {
    \if_meaning:w \q_recursion_tail #6
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
        \tl_if_empty:nF {#6}
          {
            \__msg_kernel_error:nnxx
              { kernel } { conditional-form-unknown }
              {#6} { \token_to_str:c { #1 : #2 } }
          }
        \use_none:nnnnnn
      \q_stop
      #5 {#1} {#2} {#3} {#4}
    \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
  }
\__debug_patch:nnNNpn
  { \__debug_chk_cs_exist:c { #5 _p : #6    } } { }
\cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
  { #2 { #3 _p : #4    }    { #5 _p : #6    } }
\__debug_patch:nnNNpn
  { \__debug_chk_cs_exist:c { #5    : #6 TF } } { }
\cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
  { #2 { #3    : #4 TF }    { #5    : #6 TF } }
\__debug_patch:nnNNpn
  { \__debug_chk_cs_exist:c { #5    : #6 T  } } { }
\cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
  { #2 { #3    : #4 T  }    { #5    : #6 T  } }
\__debug_patch:nnNNpn
  { \__debug_chk_cs_exist:c { #5    : #6  F } } { }
\cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
  { #2 { #3    : #4  F }    { #5    : #6  F } }
```

(*End definition for* __prg_set_eq_conditional:nnNnnNNw *and others.*)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using \if:w but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

\c_true_bool
\c_false_bool

Here are the canonical boolean values.

```
\tex_chardef:D \c_true_bool  = 1 ~
\tex_chardef:D \c_false_bool = 0 ~
```

(*End definition for* \c_true_bool *and* \c_false_bool*. These variables are documented on page 20.*)

3.8 Dissecting a control sequence

```
⟨@@=cs⟩
```

\cs_to_str:N
__cs_to_str:N
__cs_to_str:w

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the \escapechar is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from \token_to_str:N \a. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, \token_to_str:N␣\␣ yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TEX reads `\__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N␣\␣`, and the auxiliary `\__cs_to_str:w` is expanded, feeding `-` as a second character for the test; the test is false, and TEX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `\__cs_to_str:w` comes into play, inserting `-\__int_value:w`, which expands `\c_zero` to the character `0`. The initial `\tex_romannumeral:D` then sees `0`, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
\cs_set:Npn \cs_to_str:N
  {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero` so we make this dependency explicit.

```
    \tex_romannumeral:D
      \if:w \token_to_str:N \ \__cs_to_str:w \fi:
      \exp_after:wN \__cs_to_str:N \token_to_str:N
  }
\cs_set:Npn \__cs_to_str:N #1 { \c_zero }
\cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
  { - \__int_value:w \fi: \exp_after:wN \c_zero }
```

If speed is a concern we could use `\csstring` in LuaTEX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(*End definition for* `\cs_to_str:N`*,* `\__cs_to_str:N`*, and* `\__cs_to_str:w`*. These functions are documented on page 17.*)

`\__cs_split_function:NN`
`\__cs_split_function_auxi:w`
`\__cs_split_function_auxii:w`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean ⟨*true*⟩ or ⟨*false*⟩ is returned with ⟨*true*⟩ for when there is a colon in the function and ⟨*false*⟩ if there is not. Lastly, the second argument of `\__cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `\@@_split_function:NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\q_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\q_stop`. Otherwise, the `#1` contains the function name and `\q_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. In both cases, `#5` is the ⟨*processor*⟩. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```
\cs_set:Npx \__cs_split_function:NN #1
```

```
  {
    \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
    \exp_not:N \exp_after:wN \exp_not:N \__cs_split_function_auxi:w
      \exp_not:N \cs_to_str:N #1 \exp_not:N \q_mark \c_true_bool
      \token_to_str:N : \exp_not:N \q_mark \c_false_bool
      \exp_not:N \q_stop
  }
\use:x
  {
    \cs_set:Npn \exp_not:N \__cs_split_function_auxi:w
      ##1 \token_to_str:N : ##2 \exp_not:N \q_mark ##3##4 \exp_not:N \q_stop ##5
  }
  { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
\cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
  { #1 {#2} }
```

(*End definition for* `\__cs_split_function:NN`, `\__cs_split_function_auxi:w`, *and* `\__cs_split_function_auxii:w`.)

`\__cs_get_function_name:N`
`\__cs_get_function_signature:N`

Simple wrappers.

```
\cs_set:Npn \__cs_get_function_name:N #1
  { \__cs_split_function:NN #1 \use_i:nnn }
\cs_set:Npn \__cs_get_function_signature:N #1
  { \__cs_split_function:NN #1 \use_ii:nnn }
```

(*End definition for* `\__cs_get_function_name:N` *and* `\__cs_get_function_signature:N`.)

3.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N`
`\cs_if_exist_p:c`
`\cs_if_exist:NTF`
`\cs_if_exist:cTF`

Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as TEX will only ever skip input in case the token tested against is `\scan_stop:`.

```
\prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```
\prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    {
      \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    }
    \prg_return_false:
  }
```

(*End definition for* `\cs_if_exist:NTF`*. This function is documented on page 20.*)

\cs_if_free_p:N
\cs_if_free_p:c
\cs_if_free:N*TF*
\cs_if_free:c*TF*

The logical reversal of the above.

```
\prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_true:
    \else:
      \if_cs_exist:N #1
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    \fi:
  }
\prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
      { \prg_return_true: }
  }
```

(*End definition for* `\cs_if_free:NTF`*. This function is documented on page 20.*)

\cs_if_exist_use:N
\cs_if_exist_use:c
\cs_if_exist_use:N*TF*
\cs_if_exist_use:c*TF*

The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In LuaTeX we could use the `\lastnamedcs` primitive.

```
\cs_set:Npn \cs_if_exist_use:NTF #1#2
  { \cs_if_exist:NTF #1 { #1 #2 } }
\cs_set:Npn \cs_if_exist_use:NF #1
  { \cs_if_exist:NTF #1 { #1 } }
\cs_set:Npn \cs_if_exist_use:NT #1 #2
  { \cs_if_exist:NTF #1 { #1 #2 } { } }
\cs_set:Npn \cs_if_exist_use:N #1
  { \cs_if_exist:NTF #1 { #1 } { } }
\cs_set:Npn \cs_if_exist_use:cTF #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
\cs_set:Npn \cs_if_exist_use:cF #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } }
\cs_set:Npn \cs_if_exist_use:cT #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
\cs_set:Npn \cs_if_exist_use:c #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(*End definition for* \cs_if_exist_use:NTF*. This function is documented on page 16.*)

3.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called \..._new. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

__msg_kernel_error:nnxx
__msg_kernel_error:nnx
__msg_kernel_error:nn

If an internal error occurs before LaTeX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the \newlinechar is needed, to turn ^^J into a proper line break in plain TeX.

```
\cs_set_protected:Npn \__msg_kernel_error:nnxx #1#2#3#4
  {
    \tex_newlinechar:D = `\^^J \tex_relax:D
    \tex_errmessage:D
      {
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
        Argh,~internal~LaTeX3~error! ^^J ^^J
        Module ~ #1 , ~ message~name~"#2": ^^J
        Arguments~'#3'~and~'#4' ^^J ^^J
        This~is~one~for~The~LaTeX3~Project:~bailing~out
      }
    \tex_end:D
  }
\cs_set_protected:Npn \__msg_kernel_error:nnx #1#2#3
  { \__msg_kernel_error:nnxx {#1} {#2} {#3} { } }
\cs_set_protected:Npn \__msg_kernel_error:nn #1#2
  { \__msg_kernel_error:nnxx {#1} {#2} { } { } }
```

(*End definition for* __msg_kernel_error:nnxx *,* __msg_kernel_error:nnx *, and* __msg_kernel_error:nn*.*)

\msg_line_context:

Another one from l3msg which will be altered later.

```
\cs_set:Npn \msg_line_context:
  { on~line~ \tex_the:D \tex_inputlineno:D }
```

(*End definition for* `\msg_line_context:`. *This function is documented on page 130.*)

\iow_log:x
\iow_term:x

We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by l3io.

```
\cs_set_protected:Npn \iow_log:x
  { \tex_immediate:D \tex_write:D -1 }
\cs_set_protected:Npn \iow_term:x
  { \tex_immediate:D \tex_write:D 16 }
```

(*End definition for* `\iow_log:x` *and* `\iow_term:x`. *These functions are documented on page 144.*)

__chk_if_free_cs:N
__chk_if_free_cs:c

This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` *etc.* to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if ⟨*csname*⟩ is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```
\__debug_patch:nnNNpn { }
  { \__debug_log:x { Defining~\token_to_str:N #1~ \msg_line_context: } }
\cs_set_protected:Npn \__chk_if_free_cs:N #1
  {
    \cs_if_free:NF #1
      {
        \__msg_kernel_error:nnxx { kernel } { command-already-defined }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      }
  }
\cs_set_protected:Npn \__chk_if_free_cs:c
  { \exp_args:Nc \__chk_if_free_cs:N }
```

(*End definition for* `\__chk_if_free_cs:N`.)

3.11 Defining new functions

```
⟨@@=cs⟩
```

\cs_new_nopar:Npn
\cs_new_nopar:Npx
\cs_new:Npn
\cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
\cs_new_protected:Npn
\cs_new_protected:Npx
__cs_tmp:w

Function which check that the control sequence is free before defining it.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_set_protected:Npn #1 ##1
       {
         \__chk_if_free_cs:N ##1
         #2 ##1
      }
  }
\__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
\__cs_tmp:w \cs_new_nopar:Npx           \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
\__cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
\__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
```

```
\__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
\__cs_tmp:w \cs_new_protected:Npx       \cs_gset_protected:Npx
```

(*End definition for* `\cs_new_nopar:Npn` *and others. These functions are documented on page 11.*)

```
\cs_set_nopar:cpn
\cs_set_nopar:cpx
\cs_gset_nopar:cpn
\cs_gset_nopar:cpx
\cs_new_nopar:cpn
\cs_new_nopar:cpx
```

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for csname argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn`⟨*string*⟩⟨*rep-text*⟩ turns ⟨*string*⟩ into a csname and then assigns ⟨*rep-text*⟩ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```
\cs_set:Npn \__cs_tmp:w #1#2
  { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
\__cs_tmp:w \cs_set_nopar:cpn  \cs_set_nopar:Npn
\__cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
\__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
\__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
\__cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
```

(*End definition for* `\cs_set_nopar:cpn` *and others. These functions are documented on page 11.*)

```
\cs_set:cpn
\cs_set:cpx
\cs_gset:cpn
\cs_gset:cpx
\cs_new:cpn
\cs_new:cpx
```

Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set:cpn  \cs_set:Npn
\__cs_tmp:w \cs_set:cpx  \cs_set:Npx
\__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
\__cs_tmp:w \cs_new:cpn  \cs_new:Npn
\__cs_tmp:w \cs_new:cpx  \cs_new:Npx
```

(*End definition for* `\cs_set:cpn` *and others. These functions are documented on page 11.*)

```
\cs_set_protected_nopar:cpn
\cs_set_protected_nopar:cpx
\cs_gset_protected_nopar:cpn
\cs_gset_protected_nopar:cpx
\cs_new_protected_nopar:cpn
\cs_new_protected_nopar:cpx
```

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set_protected_nopar:cpn  \cs_set_protected_nopar:Npn
\__cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
\__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected_nopar:cpn  \cs_new_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
```

(*End definition for* `\cs_set_protected_nopar:cpn` *and others. These functions are documented on page 12.*)

```
\cs_set_protected:cpn
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpx
\cs_new_protected:cpn
\cs_new_protected:cpx
```

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set_protected:cpn  \cs_set_protected:Npn
\__cs_tmp:w \cs_set_protected:cpx  \cs_set_protected:Npx
\__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
\__cs_tmp:w \cs_new_protected:cpn  \cs_new_protected:Npn
\__cs_tmp:w \cs_new_protected:cpx  \cs_new_protected:Npx
```

(*End definition for* `\cs_set_protected:cpn` *and others. These functions are documented on page 11.*)

3.12 Copying definitions

\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

These macros allow us to copy the definition of a control sequence to another control sequence.

The `=` sign allows us to define funny char tokens like `=` itself or ␣ with this function. For the definition of `\c_space_char{~}` to work we need the `~` after the `=`.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an "already defined" error rather than "runaway argument".

```
\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
\cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc  \cs_set_eq:NN }
\cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D  \cs_set_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_new_eq:NN #1
  {
    \__chk_if_free_cs:N #1
    \tex_global:D \cs_set_eq:NN #1
  }
\cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
\cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
\cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(*End definition for* `\cs_set_eq:NN`*,* `\cs_gset_eq:NN`*, and* `\cs_new_eq:NN`*. These functions are documented on page 15.*)

3.13 Undefining functions

\cs_undefine:N
\cs_undefine:c

The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting TEX conditionals in case `#1` is unbalanced in this matter.

```
\cs_new_protected:Npn \cs_undefine:N #1
  { \cs_gset_eq:NN #1 \tex_undefined:D }
\cs_new_protected:Npn \cs_undefine:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
    { \cs_gset_eq:cN {#1} \tex_undefined:D }
  }
```

(*End definition for* `\cs_undefine:N`*. This function is documented on page 15.*)

3.14 Generating parameter text from argument count

```
⟨@@=cs⟩
```

__cs_parm_from_arg_count:nnF
__cs_parm_from_arg_count_test:nnF

LaTeX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#`n`}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```
\cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
  {
    \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
      {
        \exp_after:wN \exp_not:n
        \if_case:w \__int_eval:w (#2) \__int_eval_end:
             { }
        \or: { ##1 }
        \or: { ##1##2 }
        \or: { ##1##2##3 }
        \or: { ##1##2##3##4 }
        \or: { ##1##2##3##4##5 }
        \or: { ##1##2##3##4##5##6 }
        \or: { ##1##2##3##4##5##6##7 }
        \or: { ##1##2##3##4##5##6##7##8 }
        \or: { ##1##2##3##4##5##6##7##8##9 }
        \else: { \c_false_bool }
        \fi:
      }
      {#1}
  }
\cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
  {
    \if_meaning:w \c_false_bool #1
      \exp_after:wN \use_ii:nn
    \else:
      \exp_after:wN \use_i:nn
    \fi:
    { #2 {#1} }
  }
```

(*End definition for* `\__cs_parm_from_arg_count:nnF` *and* `\__cs_parm_from_arg_count_test:nnF`.)

3.15 Defining functions from a given number of arguments

```
⟨@@=cs⟩
```

__cs_count_signature:N
__cs_count_signature:c
__cs_count_signature:nnN

Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise -1 arguments to signal an error. We need a variant form right away.

```
\cs_new:Npn \__cs_count_signature:N #1
  { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
\cs_new:Npn \__cs_count_signature:nnN #1#2#3
  {
    \if_meaning:w \c_true_bool #3
      \tl_count:n {#2}
    \else:
      -1
    \fi:
  }
\cs_new:Npn \__cs_count_signature:c
  { \exp_args:Nc \__cs_count_signature:N }
```

(*End definition for* __cs_count_signature:N *and* __cs_count_signature:nnN.)

\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count:cNnn
\cs_generate_from_arg_count:Ncnn

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TEX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
  {
    \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
      {
        \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
          { \token_to_str:N #1 } { \int_eval:n {#3} }
        \use_none:n
      }
      {#4}
  }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
\cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
  { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
\cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
  { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(*End definition for* \cs_generate_from_arg_count:NNnn. *This function is documented on page 14.*)

3.16 Using the signature to define functions

```
⟨@@=cs⟩
```

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, \cs_set:Nn \foo_bar:nn {#1,#2}.

We want to define \cs_set:Nn as

```
\cs_set_protected:Npn \cs_set:Nn #1#2
  {
    \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
      { \__cs_count_signature:N #1 } {#2}
  }
```

In short, to define \cs_set:Nn we need just use \cs_set:Npn, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
\cs_set:Npn \__cs_tmp:w #1#2#3
  {
    \cs_new_protected:cpx { cs_ #1 : #2 }
      {
        \exp_not:N \__cs_generate_from_signature:NNn
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
      }
  }
\cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
  {
    \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
    #1 #2
  }
\cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
  {
    \bool_if:NTF #3
      {
        \str_if_eq_x:nnF { }
          { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
          {
            \__msg_kernel_error:nnx { kernel } { non-base-function }
              { \token_to_str:N #5 }
          }
        \cs_generate_from_arg_count:NNnn
          #5 #4 { \tl_count:n {#2} } {#6}
      }
      {
        \__msg_kernel_error:nnx { kernel } { missing-colon }
          { \token_to_str:N #5 }
      }
  }
\cs_new:Npn \__cs_generate_from_signature:n #1
  {
    \if:w n #1 \else: \if:w N #1 \else:
    \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
  }
```

Then we define the 24 variants beginning with N.

```
\__cs_tmp:w { set }                   { Nn } { Npn }
\__cs_tmp:w { set }                   { Nx } { Npx }
\__cs_tmp:w { set_nopar }             { Nn } { Npn }
\__cs_tmp:w { set_nopar }             { Nx } { Npx }
\__cs_tmp:w { set_protected }         { Nn } { Npn }
\__cs_tmp:w { set_protected }         { Nx } { Npx }
\__cs_tmp:w { set_protected_nopar }   { Nn } { Npn }
\__cs_tmp:w { set_protected_nopar }   { Nx } { Npx }
\__cs_tmp:w { gset }                  { Nn } { Npn }
\__cs_tmp:w { gset }                  { Nx } { Npx }
\__cs_tmp:w { gset_nopar }            { Nn } { Npn }
\__cs_tmp:w { gset_nopar }            { Nx } { Npx }
\__cs_tmp:w { gset_protected }        { Nn } { Npn }
```

```
\__cs_tmp:w { gset_protected }       { Nx } { Npx }
\__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
\__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
\__cs_tmp:w { new }                  { Nn } { Npn }
\__cs_tmp:w { new }                  { Nx } { Npx }
\__cs_tmp:w { new_nopar }            { Nn } { Npn }
\__cs_tmp:w { new_nopar }            { Nx } { Npx }
\__cs_tmp:w { new_protected }        { Nn } { Npn }
\__cs_tmp:w { new_protected }        { Nx } { Npx }
\__cs_tmp:w { new_protected_nopar }  { Nn } { Npn }
\__cs_tmp:w { new_protected_nopar }  { Nx } { Npx }
```

(*End definition for* `\cs_set:Nn` *and others. These functions are documented on page 13.*)

\cs_set:cn
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

The 24 `c` variants simply use `\exp_args:Nc`.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_new_protected:cpx { cs_ #1 : c #2 }
      {
        \exp_not:N \exp_args:Nc
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
      }
  }
\__cs_tmp:w { set }                  { n }
\__cs_tmp:w { set }                  { x }
\__cs_tmp:w { set_nopar }            { n }
\__cs_tmp:w { set_nopar }            { x }
\__cs_tmp:w { set_protected }        { n }
\__cs_tmp:w { set_protected }        { x }
\__cs_tmp:w { set_protected_nopar }  { n }
\__cs_tmp:w { set_protected_nopar }  { x }
\__cs_tmp:w { gset }                 { n }
\__cs_tmp:w { gset }                 { x }
\__cs_tmp:w { gset_nopar }           { n }
\__cs_tmp:w { gset_nopar }           { x }
\__cs_tmp:w { gset_protected }       { n }
\__cs_tmp:w { gset_protected }       { x }
\__cs_tmp:w { gset_protected_nopar } { n }
\__cs_tmp:w { gset_protected_nopar } { x }
\__cs_tmp:w { new }                  { n }
\__cs_tmp:w { new }                  { x }
\__cs_tmp:w { new_nopar }            { n }
\__cs_tmp:w { new_nopar }            { x }
\__cs_tmp:w { new_protected }        { n }
\__cs_tmp:w { new_protected }        { x }
\__cs_tmp:w { new_protected_nopar }  { n }
\__cs_tmp:w { new_protected_nopar }  { x }
```

(*End definition for* `\cs_set:cn` *and others. These functions are documented on page 13.*)

3.17 Checking control sequence equality

\cs_if_eq_p:NN
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NN*TF*
\cs_if_eq:cN*TF*
\cs_if_eq:Nc*TF*
\cs_if_eq:cc*TF*

Check if two control sequences are identical.

```
\prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
```

```
  {
    \if_meaning:w #1#2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc  \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc  \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:cNT  { \exp_args:Nc  \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:cNF  { \exp_args:Nc  \cs_if_eq:NNF }
\cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:NcT  { \exp_args:NNc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:NcF  { \exp_args:NNc \cs_if_eq:NNF }
\cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:ccT  { \exp_args:Ncc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:ccF  { \exp_args:Ncc \cs_if_eq:NNF }
```

(*End definition for* `\cs_if_eq:NNTF`*. This function is documented on page 20.*)

3.18 Diagnostic functions

```
⟨@@=kernel⟩
```

__kernel_register_show:N
__kernel_register_show:c
__kernel_register_show_aux:n

Simply using the \showthe primitive does not allow for line-wrapping, so instead use __msg_show_variable:NNNnn (defined in l3msg). This checks that the variable exists (using \cs_if_exist:NTF), then displays the third argument, namely >~⟨*variable*⟩=⟨*value*⟩. We expand the value before-hand as otherwise some integers (such as \currentgrouplevel or \currentgrouptype) altered by the line-wrapping code would show wrong values.

```
\cs_new_protected:Npn \__kernel_register_show:N #1
  { \exp_args:No \__kernel_register_show_aux:nN { \tex_the:D #1 } #1 }
\cs_new_protected:Npn \__kernel_register_show_aux:nN #1#2
  {
    \__msg_show_variable:NNNnn #2 \cs_if_exist:NTF ? { }
      { > ~ \token_to_str:N #2 = #1 }
  }
\cs_new_protected:Npn \__kernel_register_show:c
  { \exp_args:Nc \__kernel_register_show:N }
```

(*End definition for* `\__kernel_register_show:N` *and* `\__kernel_register_show_aux:n`*.*)

__kernel_register_log:N
__kernel_register_log:c

Redirect the output of __kernel_register_show:N to the log.

```
\cs_new_protected:Npn \__kernel_register_log:N
  { \__msg_log_next: \__kernel_register_show:N }
\cs_new_protected:Npn \__kernel_register_log:c
  { \exp_args:Nc \__kernel_register_log:N }
```

(*End definition for* `\__kernel_register_log:N`*.*)

\cs_show:N
\cs_show:c

Some control sequences have a very long name or meaning. Thus, simply using TEX's primitive \show could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to \iow_wrap:nnnN for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the \escapechar in a group; this also localizes the assignment performed by x-expansion.

The \cs_show:c command also converts its argument to a control sequence within a group to avoid showing \relax for undefined control sequences.

```
\cs_new_protected:Npn \cs_show:N #1
  {
    \group_begin:
      \int_set:Nn \tex_escapechar:D { `\\ }
      \exp_args:NNx
    \group_end:
    \__msg_show_wrap:n { > ~ \token_to_str:N #1 = \cs_meaning:N #1 }
  }
\cs_new_protected:Npn \cs_show:c
  { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
```

(*End definition for \cs_show:N. This function is documented on page 16.*)

\cs_log:N
\cs_log:c

Use \cs_show:N or \cs_show:c after calling __msg_log_next: to redirect their output to the log file only. Note that \cs_log:c is not just a variant of \cs_log:N as the csname should be turned to a control sequence within a group (see \cs_show:c).

```
\cs_new_protected:Npn \cs_log:N { \__msg_log_next: \cs_show:N }
\cs_new_protected:Npn \cs_log:c { \__msg_log_next: \cs_show:c }
```

(*End definition for \cs_log:N. This function is documented on page 16.*)

3.19 Doing nothing functions

\prg_do_nothing:

This does not fit anywhere else!

```
\cs_new_nopar:Npn \prg_do_nothing: { }
```

(*End definition for \prg_do_nothing:. This function is documented on page 9.*)

3.20 Breaking out of mapping functions

```
⟨@@=prg⟩
```

__prg_break_point:Nn
__prg_map_break:Nn

In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of __prg_break_point:Nn. The breaking functions are then defined to jump to that point and perform the argument of __prg_break_point:Nn, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```
\cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
\cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
  {
    #5
    \if_meaning:w #1 #4
      \exp_after:wN \use_iii:nnn
    \fi:
    \__prg_map_break:Nn #1 {#2}
  }
```

(*End definition for __prg_break_point:Nn and __prg_map_break:Nn.*)

__prg_break_point:
__prg_break:
__prg_break:n

Very simple analogues of `\__prg_break_point:Nn` and `\__prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```
\cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
\cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
\cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}
```

(*End definition for* `\__prg_break_point:` *,* `\__prg_break:` *, and* `\__prg_break:n`*.*)

```
⟨/initex | package⟩
```

4 l3expan implementation

```
⟨*initex | package⟩
⟨@@=exp⟩
```

\exp_after:wN
\exp_not:N
\exp_not:n

These are defined in l3basics.

(*End definition for* `\exp_after:wN` *,* `\exp_not:N` *, and* `\exp_not:n`*. These functions are documented on page 31.*)

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the LaTeX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 4.3. In section 4.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

\l__exp_internal_tl

This scratch token list variable is defined in l3basics, as it is needed "early". This is just a reminder that is the case!

(*End definition for* `\l__exp_internal_tl`*.*)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` as this turned out to be desirable since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::`⟨*Z*⟩ always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

__exp_arg_next:nnn
__exp_arg_next:Nnn

`#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

```
\cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
\cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(*End definition for* `\__exp_arg_next:nnn` *and* `\__exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

```
\cs_new:Npn \::: #1 {#1}
```

(*End definition for* `\:::`.)

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
\cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End definition for* `\::n`.)

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
\cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for* `\::N`.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It should not be wrapped in braces in the result.

```
\cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for* `\::p`.)

`\::c` This function is used to skip an argument that is turned into a control sequence without expansion.

```
\cs_new:Npn \::c #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for* `\::c`.)

`\::o` This function is used to expand an argument once.

```
\cs_new:Npn \::o #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for* `\::o`.)

`\::f` `\exp_stop_f:` This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker. In the example shown earlier the scanning was stopped once TEX had fully expanded `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` into `\cs_set_eq:NN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NN`. Since the expansion of `\exp:w \exp_end_continue_f:w` is ⟨*null*⟩, we wind up with a fully expanded list, only TEX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
\cs_new:Npn \::f #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
      {#1} {#2}
  }
\use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End definition for* `\::f` *and* `\exp_stop_f:`.)

`\::x` This function is used to expand an argument fully.

```
\cs_new_protected:Npn \::x #1 \::: #2#3
  {
    \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
    \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
  }
```

(*End definition for* `\::x`.)

`\::v`
`\::V` These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim` and `muskip`. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```
\cs_new:Npn \::V #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
      {#1} {#2}
}
\cs_new:Npn \::v # 1\::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
      {#1} {#2}
  }
```

(*End definition for* `\::v` *and* `\::V`.)

`\__exp_eval_register:N`
`\__exp_eval_register:c`
`\__exp_eval_error_msg:w` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TEX register such as `\count`. For the TEX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:`.

```
\cs_new:Npn \__exp_eval_register:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let TEX do it for us but that would result in the rather obscure

```
! You can't use `\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
      \if_meaning:w \scan_stop: #1
        \__exp_eval_error_msg:w
      \fi:
```

The next bit requires some explanation. The function must be initiated by \exp:w and we want to terminate this expansion chain by inserting the \exp_end: token. However, we have to expand the register #1 before we do that. If it is a TeX register, we need to execute the sequence \exp_after:wN \exp_end: \tex_the:D #1 and if it is a macro we need to execute \exp_after:wN \exp_end: #1. We therefore issue the longer of the two sequences and if the register is a macro, we remove the \tex_the:D.

```
    \else:
      \exp_after:wN \use_i_ii:nnn
    \fi:
    \exp_after:wN \exp_end: \tex_the:D #1
  }
\cs_new:Npn \__exp_eval_register:c #1
  { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                            Erroneous variable used!
l.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
\cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
  {
      \fi:
    \fi:
    \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
    \exp_end:
  }
```

(*End definition for* __exp_eval_register:N *and* __exp_eval_error_msg:w.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with \tex_global:D for example.

\exp_args:No
\exp_args:NNo
\exp_args:NNNo

Those lovely runs of expansion!

```
\cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
\cs_new:Npn \exp_args:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
\cs_new:Npn \exp_args:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(*End definition for* \exp_args:No *,* \exp_args:NNo *, and* \exp_args:NNNo*. These functions are documented on page 28.*)

\exp_args:Nc
\exp_args:cc

In l3basics.

(*End definition for* \exp_args:Nc *and* \exp_args:cc*. These functions are documented on page 28.*)

\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc

Here are the functions that turn their argument into csnames but are expandable.

```
\cs_new:Npn \exp_args:NNc #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
\cs_new:Npn \exp_args:Ncc #1#2#3
```

```
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
\cs_new:Npn \exp_args:Nccc #1#2#3#4
  {
    \exp_after:wN #1
      \cs:w #2 \exp_after:wN \cs_end:
      \cs:w #3 \exp_after:wN \cs_end:
      \cs:w #4 \cs_end:
  }
```

(*End definition for* `\exp_args:NNc`*,* `\exp_args:Ncc`*, and* `\exp_args:Nccc`*. These functions are documented on page 29.*)

\exp_args:Nf
\exp_args:NV
\exp_args:Nv

```
\cs_new:Npn \exp_args:Nf #1#2
  { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
\cs_new:Npn \exp_args:Nv #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \exp:w \__exp_eval_register:c {#2} }
  }
\cs_new:Npn \exp_args:NV #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \exp:w \__exp_eval_register:N #2 }
  }
```

(*End definition for* `\exp_args:Nf`*,* `\exp_args:NV`*, and* `\exp_args:Nv`*. These functions are documented on page 28.*)

\exp_args:NNV
\exp_args:NNv
\exp_args:NNf
\exp_args:NVV
\exp_args:Ncf
\exp_args:Nco

Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```
\cs_new:Npn \exp_args:NNf #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
  }
\cs_new:Npn \exp_args:NNv #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
  }
\cs_new:Npn \exp_args:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
  }
\cs_new:Npn \exp_args:Nco #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN {#3}
  }
\cs_new:Npn \exp_args:Ncf #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
  }
\cs_new:Npn \exp_args:NVV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN { \exp:w \exp_after:wN
      \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
    \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
  }
```

(*End definition for* `\exp_args:NNV` *and others. These functions are documented on page 29.*)

`\exp_args:Ncco`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:NNNV`

A few more that we can hand-tune.

```
\cs_new:Npn \exp_args:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
  }
\cs_new:Npn \exp_args:NcNc #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \cs:w #4 \cs_end:
  }
\cs_new:Npn \exp_args:NcNo #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \exp_after:wN {#4}
  }
\cs_new:Npn \exp_args:Ncco #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \cs:w #3 \exp_after:wN \cs_end:
    \exp_after:wN {#4}
  }
```

(*End definition for* `\exp_args:Ncco` *and others. These functions are documented on page 29.*)

4.3 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions are all not long: they don't actually take any arguments themselves.

\exp_args:Nx

```
\cs_new_protected:Npn \exp_args:Nx { \::x \::: }
```

(*End definition for* \exp_args:Nx*. This function is documented on page 29.*)

\exp_args:Nnc
\exp_args:Nfo
\exp_args:Nff
\exp_args:Nnf
\exp_args:Nno
\exp_args:NnV
\exp_args:Noo
\exp_args:Nof
\exp_args:Noc
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

Here are the actual function definitions, using the helper functions above.

```
\cs_new:Npn \exp_args:Nnc { \::n \::c \::: }
\cs_new:Npn \exp_args:Nfo { \::f \::o \::: }
\cs_new:Npn \exp_args:Nff { \::f \::f \::: }
\cs_new:Npn \exp_args:Nnf { \::n \::f \::: }
\cs_new:Npn \exp_args:Nno { \::n \::o \::: }
\cs_new:Npn \exp_args:NnV { \::n \::V \::: }
\cs_new:Npn \exp_args:Noo { \::o \::o \::: }
\cs_new:Npn \exp_args:Nof { \::o \::f \::: }
\cs_new:Npn \exp_args:Noc { \::o \::c \::: }
\cs_new_protected:Npn \exp_args:NNx { \::N \::x \::: }
\cs_new_protected:Npn \exp_args:Ncx { \::c \::x \::: }
\cs_new_protected:Npn \exp_args:Nnx { \::n \::x \::: }
\cs_new_protected:Npn \exp_args:Nox { \::o \::x \::: }
\cs_new_protected:Npn \exp_args:Nxo { \::x \::o \::: }
\cs_new_protected:Npn \exp_args:Nxx { \::x \::x \::: }
```

(*End definition for* \exp_args:Nnc *and others. These functions are documented on page 29.*)

\exp_args:NNno
\exp_args:NNoo
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nooo
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Noox

```
\cs_new:Npn \exp_args:NNno { \::N \::n \::o \::: }
\cs_new:Npn \exp_args:NNoo { \::N \::o \::o \::: }
\cs_new:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
\cs_new:Npn \exp_args:Nnno { \::n \::n \::o \::: }
\cs_new:Npn \exp_args:Nooo { \::o \::o \::o \::: }
\cs_new_protected:Npn \exp_args:NNNx { \::N \::N \::x \::: }
\cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \::: }
\cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \::: }
\cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
\cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \::: }
\cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \::: }
\cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
\cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(*End definition for* \exp_args:NNno *and others. These functions are documented on page 30.*)

4.4 Last-unbraced versions

__exp_arg_last_unbraced:nn
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
\cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
\cs_new:Npn \::f_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
  }
\cs_new:Npn \::o_unbraced \::: #1#2
  { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
\cs_new:Npn \::V_unbraced \::: #1#2
```

```
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
  }
\cs_new:Npn \::v_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
  }
\cs_new_protected:Npn \::x_unbraced \::: #1#2
  {
    \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
    \l__exp_internal_tl
  }
```

(*End definition for* `\__exp_arg_last_unbraced:nn` *and others.*)

\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:No
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:Nx

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```
\cs_new:Npn \exp_last_unbraced:NV #1#2
  { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
\cs_new:Npn \exp_last_unbraced:Nv #1#2
  { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
\cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
\cs_new:Npn \exp_last_unbraced:Nf #1#2
  { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
\cs_new:Npn \exp_last_unbraced:Nco #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\cs_new:Npn \exp_last_unbraced:NcV #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp:w \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp:w \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 #3 }
\cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp:w \__exp_eval_register:N #4
  }
\cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
```

```
\cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
\cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }
```

(*End definition for* `\exp_last_unbraced:NV` *and others. These functions are documented on page 30.*)

`\exp_last_two_unbraced:Noo`
`\__exp_last_two_unbraced:noN`

If #2 is a single token then this can be implemented as

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
 { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
  { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
\cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
    { \exp_after:wN #3 #2 #1 }
```

(*End definition for* `\exp_last_two_unbraced:Noo` *and* `\__exp_last_two_unbraced:noN`*. These functions are documented on page 30.*)

4.5 Preventing expansion

`\exp_not:o`
`\exp_not:c`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

```
\cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
\cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\cs_new:Npn \exp_not:f #1
  { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
\cs_new:Npn \exp_not:V #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \exp:w \__exp_eval_register:N #1 }
  }
\cs_new:Npn \exp_not:v #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \exp:w \__exp_eval_register:c {#1} }
  }
```

(*End definition for* `\exp_not:o` *and others. These functions are documented on page 31.*)

4.6 Controlled expansion

`\exp:w`
`\exp_end:`
`\exp_end_continue_f:w`
`\exp_end_continue_f:nw`

To trigger a sequence of “arbitrary” many expansions we need a method to invoke TeX’s expansion mechanism in such a way that a) we are able to stop it in a controlled manner and b) that the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence.

```
%\cs_new_eq:NN \exp:w    \tex_romannumeral:D
```

So to stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumerl:D`'s search for a number.

```
%\int_const:Nn \exp_end: { 0 }
```

(Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a "number" that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an `f`-type expansion we provide the alphabetic constant `` `^^@ `` that also represents `0` but this time TEX's syntax for a ⟨*number*⟩ continues searching for an optional space (and it continues expansion doing that) — see TEXbook page 269 for details.

```
\tex_catcode:D `\^^@=13
\cs_new_protected:Npn \exp_end_continue_f:w {`^^@}
```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error.[7]

```
\cs_new:Npn ^^@{\expansionERROR}
```

```
\cs_new:Npn \exp_end_continue_f:nw #1 { `^^@ #1 }
\tex_catcode:D `\^^@=15
```

(*End definition for* `\exp:w` *and others. These functions are documented on page 32.*)

4.7 Defining function variants

```
⟨@@=cs⟩
```

`\cs_generate_variant:Nn`

#1 : Base form of a function; *e.g.*, `\tl_set:Nn`

#2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `\__cs_tmp:w` as either `\cs_new:Npx` or `\cs_new_protected:Npx`, which is then used to define all the variants (except those involving `x`-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```
\__debug_patch:nnNNpn { \__debug_chk_cs_exist:N #1 } { }
\cs_new_protected:Npn \cs_generate_variant:Nn #1#2
  {
    \__cs_generate_variant:N #1
    \exp_after:wN \__cs_split_function:NN
    \exp_after:wN #1
    \exp_after:wN \__cs_generate_variant:nnNN
    \exp_after:wN #1
    \tl_to_str:n {#2} , \scan_stop: , \q_recursion_stop
  }
```

(*End definition for* `\cs_generate_variant:Nn`. *This function is documented on page 26.*)

[7] Need to get a real error message.

__cs_generate_variant:N
__cs_generate_variant:ww
__cs_generate_variant:wwNw

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because all primitive TEX conditionals are expandable.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the fist occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long␣`, `\protected␣`, `\protected\long␣`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\q_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```
\cs_new_protected:Npx \__cs_generate_variant:N #1
  {
    \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
      \exp_not:N \exp_not:N #1 #1
      \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
    \exp_not:N \else:
      \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
        \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
          \exp_not:N \q_mark
        \exp_not:N \q_mark \cs_new_protected:Npx
        \tl_to_str:n { pr }
        \exp_not:N \q_mark \cs_new:Npx
        \exp_not:N \q_stop
    \exp_not:N \fi:
  }
\use:x
  {
    \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:ww
      ##1 \tl_to_str:n { ma } ##2 \exp_not:N \q_mark
  }
  { \__cs_generate_variant:wwNw #1 }
\use:x
  {
    \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:wwNw
      ##1 \tl_to_str:n { pr } ##2 \exp_not:N \q_mark
      ##3 ##4 \exp_not:N \q_stop
  }
  { \cs_set_eq:NN \__cs_tmp:w #3 }
```

(*End definition for __cs_generate_variant:N, __cs_generate_variant:ww, and __cs_generate_variant:wwNw.*)

__cs_generate_variant:nnNN

`#1` : Base name.
`#2` : Base signature.
`#3` : Boolean.
`#4` : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```
\cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__cs_generate_variant:Nnnw #4 {#1}{#2}
  }
```

(*End definition for* `\__cs_generate_variant:nnNN`*.*)

`\__cs_generate_variant:Nnnw`

#1 : Base function.
#2 : Base name.
#3 : Base signature.
#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a `cV` variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, it would be better to define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` should be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` should trigger an error, because we do not have a means to replace `o`-expansion by `x`-expansion.

All this boils down to a few rules. Only `n` and `N`-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to `n` (except for two cases: `N` and `p`-type arguments). A common trailing part is ignored.

We compare the base and variant signatures one character at a time within `x`-expansion. The result is given to `\__cs_generate_variant:wwNN` in the form ⟨*processed variant signature*⟩ `\q_mark` ⟨*errors*⟩ `\q_stop` ⟨*base function*⟩ ⟨*new function*⟩. If all went well, ⟨*errors*⟩ is empty; otherwise, it is a kernel error message, followed by some clean-up code (`\use_none:nnn`).

Note the space after `#3` and after the following brace group. Those are ignored by TeX when fetching the last argument for `\__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `\__cs_generate_variant_loop_end:nwwwNNnn`.

```
\cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
  {
    \if_meaning:w \scan_stop: #4
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:x
      {
```

```
      \exp_not:N \__cs_generate_variant:wwNN
      \__cs_generate_variant_loop:nNwN { }
        #4
        \__cs_generate_variant_loop_end:nwwwNNnn
        \q_mark
        #3 ~
        { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
        { }
        \q_stop
      \exp_not:N #1 {#2} {#4}
    }
  \__cs_generate_variant:Nnnw #1 {#2} {#3}
}
```

(*End definition for* `\__cs_generate_variant:Nnnw`*.*)

__cs_generate_variant_loop:nNwN
__cs_generate_variant_loop_same:w
__cs_generate_variant_loop_end:nwwwNNnn
__cs_generate_variant_loop_long:wNNnn
__cs_generate_variant_loop_invalid:NNwNNnn

#1 : Last few (consecutive) letters common between the base and variant (in fact, `\__cs_generate_variant_same:N` ⟨*letter*⟩ for each letter).
#2 : Next variant letter.
#3 : Remainder of variant form.
#4 : Next base letter.

The first argument is populated by `\__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed with a base letter of `N` or `n`. Otherwise, call `\__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `\__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument was collected, and the next variant letter `#2`, then loop by calling `\__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, `#2` is `\__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the ⟨*base name*⟩ as `#7`, the ⟨*variant signature*⟩ `#8`, the ⟨*next base letter*⟩ `#1` and the part `#3` of the base signature that wasn't read yet; and combines those into the ⟨*new function*⟩ to be defined.

- If the end of the base form is encountered first, `#4` is `~{}\fi:` which ends the conditional (with an empty expansion), followed by `\__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `\__cs_generate_variant:wwNN`.

- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is neither `n` nor `N`. Again, an error is placed as the second argument of `\__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, `#2` is as described in the first point, and `#4` as described in the second point above. The `\__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in `#4` as its first argument: this empty brace group produces the correct signature for the full variant.

```
\cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
  {
    \if:w #2 #4
      \exp_after:wN \__cs_generate_variant_loop_same:w
    \else:
      \if:w N #4 \else:
        \if:w n #4 \else:
          \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
        \fi:
      \fi:
    \fi:
    #1
    \prg_do_nothing:
    #2
    \__cs_generate_variant_loop:nNwN { } #3 \q_mark
  }
\cs_new:Npn \__cs_generate_variant_loop_same:w
    #1 \prg_do_nothing: #2#3#4
  {
    #3 { #1 \__cs_generate_variant_same:N #2 }
  }
\cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
    #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
  {
    \scan_stop: \scan_stop: \fi:
    \exp_not:N \q_mark
    \exp_not:N \q_stop
    \exp_not:N #6
    \exp_not:c { #7 : #8 #1 #3 }
  }
\cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
  {
    \exp_not:n
      {
        \q_mark
        \__msg_kernel_error:nnxx { kernel } { variant-too-long }
          {#5} { \token_to_str:N #3 }
        \use_none:nnn
        \q_stop
        #3
        #3
      }
  }
\cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
    #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
  {
    \fi: \fi: \fi:
    \exp_not:n
      {
        \q_mark
        \__msg_kernel_error:nnxxxx { kernel } { invalid-variant }
          {#7} { \token_to_str:N #5 } {#1} {#2}
        \use_none:nnn
        \q_stop
        #5
        #5
      }
  }
```

(*End definition for* `\__cs_generate_variant_loop:nNwN` *and others.*)

__cs_generate_variant_same:N

When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the `n`-type no-expansion, but the `N` and `p` types require a slightly different behaviour with respect to braces.

```
\cs_new:Npn \__cs_generate_variant_same:N #1
  {
    \if:w N #1
      N
    \else:
      \if:w p #1
        p
      \else:
        n
      \fi:
    \fi:
  }
```

(*End definition for* `\__cs_generate_variant_same:N`.)

__cs_generate_variant:wwNN

If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `\__cs_tmp:w` locally to `\cs_new_protected:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```
\__debug_patch:nnNNpn
  {
    \cs_if_free:NF #4
      {
        \__debug_log:x
          {
            Variant~\token_to_str:N #4~%
            already~defined;~ not~ changing~ it~ \msg_line_context:
          }
      }
  }
  { }
\cs_new_protected:Npn \__cs_generate_variant:wwNN
    #1 \q_mark #2 \q_stop #3#4
  {
    #2
    \cs_if_free:NT #4
      {
        \group_begin:
          \__cs_generate_internal_variant:n {#1}
          \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
        \group_end:
      }
  }
```

(*End definition for* `\__cs_generate_variant:wwNN`*.*)

__cs_generate_internal_variant:n
__cs_generate_internal_variant:wwnw
__cs_generate_internal_variant_loop:n

Test if `\exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in `#1`. If `#1` contains an `x` (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```
\cs_new_protected:Npx \__cs_generate_internal_variant:n #1
  {
    \exp_not:N \__cs_generate_internal_variant:wwnNwnn
      #1 \exp_not:N \q_mark
        { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
        \cs_new_protected:cpx
      \token_to_str:N x \exp_not:N \q_mark
        { }
        \cs_new:cpx
    \exp_not:N \q_stop
      { exp_args:N #1 }
      {
        \exp_not:N \__cs_generate_internal_variant_loop:n #1
          { : \exp_not:N \use_i:nn }
      }
  }
\use:x
  {
    \cs_new_protected:Npn \exp_not:N \__cs_generate_internal_variant:wwnNwnn
        ##1 \token_to_str:N x ##2 \exp_not:N \q_mark
        ##3 ##4 ##5 \exp_not:N \q_stop ##6 ##7
  }
  {
    #3
    \cs_if_free:cT {#6} { #4 {#6} {#7} }
  }
```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `: \use_i:nn`, which leaves `\cs_end:` and removes the looping macro. The colon is in fact also turned into `\:::` so that the required structure for `\exp_args:N...` commands is correctly terminated.

```
\cs_new:Npn \__cs_generate_internal_variant_loop:n #1
  {
    \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
    \__cs_generate_internal_variant_loop:n
  }
```

(*End definition for* `\__cs_generate_internal_variant:n`*,* `\__cs_generate_internal_variant:wwnw`*, and* `\__cs_generate_internal_variant_loop:n`*.*)

```
⟨/initex | package⟩
```

5 l3tl implementation

```
⟨*initex | package⟩
⟨@@=tl⟩
```

A token list variable is a TeX macro that holds tokens. By using the ε-TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including `#`, in this way.

5.1 Functions

`\tl_new:N`
`\tl_new:c`

Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
\cs_new_protected:Npn \tl_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_tl
  }
\cs_generate_variant:Nn \tl_new:N { c }
```

(*End definition for* `\tl_new:N`*. This function is documented on page 35.*)

`\tl_const:Nn`
`\tl_const:Nx`
`\tl_const:cn`
`\tl_const:cx`

Constants are also easy to generate.

```
\cs_new_protected:Npn \tl_const:Nn #1#2
  {
    \__chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
  }
\cs_new_protected:Npn \tl_const:Nx #1#2
  {
    \__chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 {#2}
  }
\cs_generate_variant:Nn \tl_const:Nn { c }
\cs_generate_variant:Nn \tl_const:Nx { c }
```

(*End definition for* `\tl_const:Nn`*. This function is documented on page 35.*)

`\tl_clear:N`
`\tl_clear:c`
`\tl_gclear:N`
`\tl_gclear:c`

Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear:N  #1
  { \tl_set_eq:NN #1 \c_empty_tl }
\cs_new_protected:Npn \tl_gclear:N #1
  { \tl_gset_eq:NN #1 \c_empty_tl }
\cs_generate_variant:Nn \tl_clear:N  { c }
\cs_generate_variant:Nn \tl_gclear:N { c }
```

(*End definition for* `\tl_clear:N` *and* `\tl_gclear:N`*. These functions are documented on page 35.*)

`\tl_clear_new:N`
`\tl_clear_new:c`
`\tl_gclear_new:N`
`\tl_gclear_new:c`

Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear_new:N  #1
  { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
\cs_new_protected:Npn \tl_gclear_new:N #1
  { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
\cs_generate_variant:Nn \tl_clear_new:N  { c }
\cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(*End definition for* `\tl_clear_new:N` *and* `\tl_gclear_new:N`*. These functions are documented on page 36.*)

\tl_set_eq:NN
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc

For setting token list variables equal to each other. When checking is turned on, make sure both variables exist.

```
\tex_ifodd:D \l@expl@enable@debug@bool
  \cs_new_protected:Npn \tl_set_eq:NN #1#2
    {
      \__debug_chk_var_exist:N #1
      \__debug_chk_var_exist:N #2
      \cs_set_eq:NN #1 #2
    }
  \cs_new_protected:Npn \tl_gset_eq:NN #1#2
    {
      \__debug_chk_var_exist:N #1
      \__debug_chk_var_exist:N #2
      \cs_gset_eq:NN #1 #2
    }
\else:
  \cs_new_eq:NN \tl_set_eq:NN  \cs_set_eq:NN
  \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\fi:
\cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
\cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
```

(*End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 36.*)

\tl_concat:NNN
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc

Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to \scan_stop: would lead to problems later on.

```
\__debug_patch:nnNNpn
  {
    \__debug_chk_var_exist:N #1
    \__debug_chk_var_exist:N #2
    \__debug_chk_var_exist:N #3
  }
  { }
\cs_new_protected:Npn \tl_concat:NNN #1#2#3
  { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\__debug_patch:nnNNpn
  {
    \__debug_chk_var_exist:N #1
    \__debug_chk_var_exist:N #2
    \__debug_chk_var_exist:N #3
  }
  { }
\cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
  { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_generate_variant:Nn \tl_concat:NNN  { ccc }
\cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
```

(*End definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 36.*)

\tl_if_exist_p:N
\tl_if_exist_p:c
\tl_if_exist:NTF
\tl_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for `\tl_if_exist:NTF`. This function is documented on page 36.*)

5.2 Constant token lists

\c_empty_tl

Never full. We need to define that constant before using \tl_new:N.

```
\tl_const:Nn \c_empty_tl { }
```

(*End definition for* \c_empty_tl*. This variable is documented on page 47.*)

\c_space_tl

A space as a token list (as opposed to as a character).

```
\tl_const:Nn \c_space_tl { ~ }
```

(*End definition for* \c_space_tl*. This variable is documented on page 47.*)

5.3 Adding to token list variables

\tl_set:Nn
\tl_set:NV
\tl_set:Nv
\tl_set:No
\tl_set:Nf
\tl_set:Nx
\tl_set:cn
\tl_set:cV
\tl_set:cv
\tl_set:co
\tl_set:cf
\tl_set:cx
\tl_gset:Nn
\tl_gset:NV
\tl_gset:Nv
\tl_gset:No
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cn
\tl_gset:cV
\tl_gset:cv
\tl_gset:co
\tl_gset:cf
\tl_gset:cx

By using \exp_not:n token list variables can contain # tokens, which makes the token list registers provided by TEX more or less redundant. The \tl_set:No version is done "by hand" as it is used quite a lot. Each definition is prefixed by a call to __debug_patch:nnNNpn which adds an existence check to the definition.

```
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_set:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_set:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_set:Nx #1#2
  { \cs_set_nopar:Npx #1 {#2} }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gset:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gset:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gset:Nx #1#2
  { \cs_gset_nopar:Npx #1 {#2} }
\cs_generate_variant:Nn \tl_set:Nn  {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_set:Nx  { c }
\cs_generate_variant:Nn \tl_set:Nn  { c, co , cV , cv , cf }
\cs_generate_variant:Nn \tl_gset:Nn {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_gset:Nx { c }
\cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }
```

(*End definition for* \tl_set:Nn *and* \tl_gset:Nn*. These functions are documented on page 36.*)

\tl_put_left:Nn
\tl_put_left:NV
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:co
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:co
\tl_gput_left:cx

Adding to the left is done directly to gain a little performance.

```
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_left:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_left:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_left:No #1#2
```

```
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_left:Nx #1#2
  { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_left:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_left:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_left:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_left:Nx #1#2
  { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
\cs_generate_variant:Nn \tl_put_left:Nn  { c }
\cs_generate_variant:Nn \tl_put_left:NV  { c }
\cs_generate_variant:Nn \tl_put_left:No  { c }
\cs_generate_variant:Nn \tl_put_left:Nx  { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:NV { c }
\cs_generate_variant:Nn \tl_gput_left:No { c }
\cs_generate_variant:Nn \tl_gput_left:Nx { c }
```

(*End definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 36.*)

`\tl_put_right:Nn`
`\tl_put_right:NV`
`\tl_put_right:No`
`\tl_put_right:Nx`
`\tl_put_right:cn`
`\tl_put_right:cV`
`\tl_put_right:co`
`\tl_put_right:cx`
`\tl_gput_right:Nn`
`\tl_gput_right:NV`
`\tl_gput_right:No`
`\tl_gput_right:Nx`
`\tl_gput_right:cn`
`\tl_gput_right:cV`
`\tl_gput_right:co`
`\tl_gput_right:cx`

The same on the right.

```
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_right:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_right:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_right:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_put_right:Nx #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:Nx #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
\cs_generate_variant:Nn \tl_put_right:Nn  { c }
```

```
\cs_generate_variant:Nn \tl_put_right:NV  { c }
\cs_generate_variant:Nn \tl_put_right:No  { c }
\cs_generate_variant:Nn \tl_put_right:Nx  { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\cs_generate_variant:Nn \tl_gput_right:Nx { c }
```

(*End definition for* `\tl_put_right:Nn` *and* `\tl_gput_right:Nn`*. These functions are documented on page 36.*)

5.4 Reassigning token list category codes

`\c__tl_rescan_marker_tl`

The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```
\tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }
```

(*End definition for* `\c__tl_rescan_marker_tl`*.*)

`\tl_set_rescan:Nnn`
`\tl_set_rescan:Nno`
`\tl_set_rescan:Nnx`
`\tl_set_rescan:cnn`
`\tl_set_rescan:cno`
`\tl_set_rescan:cnx`
`\tl_gset_rescan:Nnn`
`\tl_gset_rescan:Nno`
`\tl_gset_rescan:Nnx`
`\tl_gset_rescan:cnn`
`\tl_gset_rescan:cno`
`\tl_gset_rescan:cnx`
`\tl_rescan:nn`
`\__tl_set_rescan:NNnn`
`\__tl_set_rescan_multi:n`
`\__tl_rescan:w`

These functions use a common auxiliary. After some initial setup explained below, and the user setup `#3` (followed by `\scan_stop:` to be safe), the tokens are rescanned by `\__tl_set_rescan:n` and stored into `\l__tl_internal_a_tl`, then passed to `#1#2` outside the group after expansion. The auxiliary `\__tl_set_rescan:n` is defined later: in the simplest case, this auxiliary calls `\__tl_set_rescan_multi:n`, whose code is included here to help understand the approach.

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a TeX error occurs:

```
! File ended while scanning definition of ...
```

The standard solution is to use an x-expanding assignment and set `\everyeof` to `\exp_not:N` to suppress the error at the end of the file. Since the rescanned tokens should not be expanded, they are taken as a delimited argument of an auxiliary which wraps them in `\exp_not:n` (in fact `\exp_not:o`, as there is a `\prg_do_nothing:` to avoid losing braces). The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

The difference between single-line and multiple-line files complicates the story, as explained below.

```
\cs_new_protected:Npn \tl_set_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_set:Nn }
\cs_new_protected:Npn \tl_gset_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_gset:Nn }
\cs_new_protected:Npn \tl_rescan:nn
  { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
\cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
  {
    \tl_if_empty:nTF {#4}
      {
        \group_begin:
          #3
        \group_end:
        #1 #2 { }
```

```
   }
   {
    \group_begin:
     \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
     \int_compare:nNnT \tex_endlinechar:D = { 32 }
      { \int_set:Nn \tex_endlinechar:D { -1 } }
     \tex_newlinechar:D \tex_endlinechar:D
     #3 \scan_stop:
     \exp_args:No \__tl_set_rescan:n { \tl_to_str:n {#4} }
     \exp_args:NNNo
    \group_end:
    #1 #2 \l__tl_internal_a_tl
  }
 }
\cs_new_protected:Npn \__tl_set_rescan_multi:n #1
 {
  \tl_set:Nx \l__tl_internal_a_tl
   {
    \exp_after:wN \__tl_rescan:w
    \exp_after:wN \prg_do_nothing:
    \etex_scantokens:D {#1}
   }
 }
\exp_args:Nno \use:nn
 { \cs_new:Npn \__tl_rescan:w #1 } \c__tl_rescan_marker_tl
 { \exp_not:o {#1} }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  {     Nno , Nnx }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  { c , cno , cnx }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn {     Nno , Nnx }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }
```

(*End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 38.*)

`\__tl_set_rescan:n`
`\__tl_set_rescan:NnTF`
`\__tl_set_rescan_single:nn`
`\__tl_set_rescan_single_aux:nn`

This function calls `\__tl_set_rescan_multiple:n` or `\__tl_set_rescan_single:nn { ' }` depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

The case of multiple lines is a straightforward application of `\scantokens` as described above. The only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become `\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to -1 if it was 32 (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

For a single line, no `\endlinechar` should be added, so it is set to -1, and spaces should not be removed.

Trailing spaces and tabs are a difficult matter, as TeX removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, 11 (letter), 12 (other) and 13 (active) are accepted, as these are suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from ’ (ASCII 39) to ~ (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). Once a valid character is found, run some code very similar to `\__tl_set_rescan_multi:n`, except that `\__tl_rescan:w` must be redefined to also remove the additional character (with the appropriate catcode). Getting the delimiter with the right catcode requires using `\scantokens` inside an `x`-expansion, hence using the previous definition of `\__tl_rescan:w` as well. The odd `\exp_not:N \use:n` ensures that the trailing `\exp_not:N` in `\everyeof` does not prevent the expansion of `\c__tl_rescan_marker_tl`, but rather of a closing brace (this does nothing). If no valid character is found, similar code is ran, and the only difference is that trailing spaces are not preserved (bear in mind that this only happens if no character between 39 and 127 has catcode letter, other or active).

There is also some work to preserve leading spaces: test whether the first character (given by `\str_head:n`, with an extra space to circumvent a limitation of `f`-expansion) has catcode 10 and add what TeX would add in the middle of a line for any sequence of such characters: a single space with catcode 10 and character code 32.

```
\group_begin:
  \tex_catcode:D ‘\^^@ = 12 \scan_stop:
  \cs_new_protected:Npn \__tl_set_rescan:n #1
    {
      \int_compare:nNnTF \tex_newlinechar:D < 0
        { \use_ii:nn }
        {
          \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
          \tex_lowercase:D { \__tl_set_rescan:NnTF ^^@ } {#1}
        }
          { \__tl_set_rescan_multi:n }
          { \__tl_set_rescan_single:nn { ’ } }
      {#1}
    }
  \cs_new_protected:Npn \__tl_set_rescan:NnTF #1#2
    { \tl_if_in:nnTF {#2} {#1} }
  \cs_new_protected:Npn \__tl_set_rescan_single:nn #1
    {
      \int_compare:nNnTF
        { \char_value_catcode:n { ‘#1 } / 3 } = 4
        { \__tl_set_rescan_single_aux:nn {#1} }
        {
          \int_compare:nNnTF { ‘#1 } < { ‘\~ }
            {
              \char_set_lccode:nn { 0 } { ‘#1 + 1 }
              \tex_lowercase:D { \__tl_set_rescan_single:nn { ^^@ } }
            }
            { \__tl_set_rescan_single_aux:nn { } }
        }
    }
  \cs_new_protected:Npn \__tl_set_rescan_single_aux:nn #1#2
    {
      \int_set:Nn \tex_endlinechar:D { -1 }
      \use:x
```

```
        {
          \exp_not:N \use:n
            {
              \exp_not:n { \cs_set:Npn \__tl_rescan:w ##1 }
              \exp_after:wN \__tl_rescan:w
              \exp_after:wN \prg_do_nothing:
              \etex_scantokens:D {#1}
            }
          \c__tl_rescan_marker_tl
        }
        { \exp_not:o {##1} }
      \tl_set:Nx \l__tl_internal_a_tl
        {
          \int_compare:nNnT
            {
              \char_value_catcode:n
                { \exp_last_unbraced:Nf ` \str_head:n {#2} ~ }
            }
            = { 10 } { ~ }
          \exp_after:wN \__tl_rescan:w
          \exp_after:wN \prg_do_nothing:
          \etex_scantokens:D { #2 #1 }
        }
    }
\group_end:
```

(*End definition for* `\__tl_set_rescan:n` *and others.*)

5.5 Modifying token list variables

\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn

All of the `replace` functions call `\__tl_replace:NnNNNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`\__tl_replace_next:w`) or stops (`\__tl_replace_wrap:w`) after the first replacement. Next comes an `x`-type assignment function `\tl_set:Nx` or `\tl_gset:Nx` for local or global replacements. Finally, the three arguments ⟨*tl var*⟩ {⟨*pattern*⟩} {⟨*replacement*⟩} provided by the user. When describing the auxiliary functions below, we denote the contents of the ⟨*tl var*⟩ by ⟨*token list*⟩.

```
\cs_new_protected:Npn \tl_replace_once:Nnn
  { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_set:Nx  }
\cs_new_protected:Npn \tl_greplace_once:Nnn
  { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
\cs_new_protected:Npn \tl_replace_all:Nnn
  { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx  }
\cs_new_protected:Npn \tl_greplace_all:Nnn
  { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
\cs_generate_variant:Nn \tl_replace_once:Nnn  { c }
\cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
\cs_generate_variant:Nn \tl_replace_all:Nnn   { c }
\cs_generate_variant:Nn \tl_greplace_all:Nnn  { c }
```

(*End definition for* `\tl_replace_all:Nnn` *and others. These functions are documented on page 37.*)

__tl_replace:NnNNNnn
__tl_replace_auxi:NnnNNNnn
__tl_replace_auxii:nNNNnn
__tl_replace_next:w
__tl_replace_wrap:w

To implement the actual replacement auxiliary `\__tl_replace_auxii:nNNNnn` we need a ⟨*delimiter*⟩ with the following properties:

- all occurrences of the ⟨*pattern*⟩ `#6` in “⟨*token list*⟩ ⟨*delimiter*⟩” belong to the ⟨*token list*⟩ and have no overlap with the ⟨*delimiter*⟩,

- the first occurrence of the ⟨*delimiter*⟩ in “⟨*token list*⟩ ⟨*delimiter*⟩” is the trailing ⟨*delimiter*⟩.

We first find the building blocks for the ⟨*delimiter*⟩, namely two tokens $\langle A\rangle$ and $\langle B\rangle$ such that $\langle A\rangle$ does not appear in `#6` and `#6` is not $\langle B\rangle$ (this condition is trivial if `#6` has more than one token). Then we consider the delimiters “$\langle A\rangle$” and “$\langle A\rangle$ $\langle A\rangle^n$ $\langle B\rangle$ $\langle A\rangle^n$ $\langle B\rangle$”, for $n \geq 1$, where $\langle A\rangle^n$ denotes n copies of $\langle A\rangle$, and we choose as our ⟨*delimiter*⟩ the first one which is not in the ⟨*token list*⟩.

Every delimiter in the set obeys the first condition: `#6` does not contain $\langle A\rangle$ hence cannot be overlapping with the ⟨*token list*⟩ and the ⟨*delimiter*⟩, and it cannot be within the ⟨*delimiter*⟩ since it would have to be in one of the two $\langle B\rangle$ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the ⟨*delimiter*⟩ we choose does not appear in the ⟨*token list*⟩. Additionally, the set of delimiters is such that a ⟨*token list*⟩ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a ⟨*delimiter*⟩ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “$\langle A\rangle$” in the list of delimiters to try, so that the ⟨*delimiter*⟩ is simply `\q_mark` in the most common situation where neither the ⟨*token list*⟩ nor the ⟨*pattern*⟩ contains `\q_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty ⟨*pattern*⟩ `#6` is an error, and if `#1` is absent from both the ⟨*token list*⟩ `#5` and the ⟨*pattern*⟩ `#6` then we can use it as the ⟨*delimiter*⟩ through `\__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `\__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q_mark {?}`, `\? {??}`, `\?? {???}`, and so on, until `#6` does not contain the control sequence `#1`, which we take as our $\langle A\rangle$. The argument `#2` only serves to collect `?` characters for `#1`. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptyness of `#6`). However, this is rare enough not to matter. Finally, choose $\langle B\rangle$ to be `\q_nil` or `\q_stop` such that it is not equal to `#6`.

The `\__tl_replace_auxi:NnnNNNnn` auxiliary receives `{`$\langle A\rangle$`}` and `{`$\langle A\rangle^n\langle B\rangle$`}` as its arguments, initially with $n = 1$. If “$\langle A\rangle$ $\langle A\rangle^n\langle B\rangle$ $\langle A\rangle^n\langle B\rangle$” is in the ⟨*token list*⟩ then increase n and try again. Once it is not anymore in the ⟨*token list*⟩ we take it as our ⟨*delimiter*⟩ and pass this to the `auxii` auxiliary.

```
\cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
  {
    \tl_if_empty:nTF {#6}
      {
        \__msg_kernel_error:nnx { kernel } { empty-search-pattern }
          { \tl_to_str:n {#7} }
      }
      {
        \tl_if_in:onTF { #5 #6 } {#1}
          {
            \tl_if_in:nnTF {#6} {#1}
              { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
              {
                \quark_if_nil:nTF {#6}
```

```
                  { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
                  { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil  } }
              }
          }
          { \__tl_replace_auxii:nNNNnn {#1} }
          #3#4#5 {#6} {#7}
      }
  }
\cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
  {
    \tl_if_in:NnTF #1 { #2 #3 #3 }
      { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
      { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
  }
```

The auxiliary __tl_replace_auxii:nNNNnn receives the following arguments: {⟨*delimiter*⟩} ⟨*function*⟩ ⟨*assignment*⟩ ⟨*tl var*⟩ {⟨*pattern*⟩} {⟨*replacement*⟩}. All of its work is done between \group_align_safe_begin: and \group_align_safe_end: to avoid issues in alignments. It does the actual replacement within #3 #4 {...}, an x-expanding ⟨*assignment*⟩ #3 to the ⟨*tl var*⟩ #4. The auxiliary __tl_replace_next:w is called, followed by the ⟨*token list*⟩, some tokens including the ⟨*delimiter*⟩ #1, followed by the ⟨*pattern*⟩ #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls __tl_replace_wrap:w to test whether this #5 is found within the ⟨*token list*⟩ or is the trailing one.

If on the one hand it is found within the ⟨*token list*⟩, then ##1 cannot contain the ⟨*delimiter*⟩ #1 that we worked so hard to obtain, thus __tl_replace_wrap:w gets ##1 as its own argument ##1, and protects it against the x-expanding assignment. It also finds \exp_not:n as ##2 and does nothing to it, thus letting through \exp_not:n {⟨*replacement*⟩} into the assignment. Note that __tl_replace_next:w and __tl_replace_wrap:w are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of \exp_not:o and \use_none:nn, rather than simply \exp_not:n. Afterwards, __tl_replace_next:w is called to repeat the replacement, or __tl_replace_wrap:w if we only want a single replacement. In this second case, ##1 is the ⟨*remaining tokens*⟩ in the ⟨*token list*⟩ and ##2 is some ⟨*ending code*⟩ which ends the assignment and removes the trailing tokens #5 using some \if_false: { \fi: } trickery because #5 may contain any delimiter.

If on the other hand the argument ##1 of __tl_replace_next:w is delimited by the trailing ⟨*pattern*⟩ #5, then ##1 is "{ } { } ⟨*token list*⟩ ⟨*delimiter*⟩ {⟨*ending code*⟩}", hence __tl_replace_wrap:w finds "{ } { } ⟨*token list*⟩" as ##1 and the ⟨*ending code*⟩ as ##2. It leaves the ⟨*token list*⟩ into the assignment and unbraces the ⟨*ending code*⟩ which removes what remains (essentially the ⟨*delimiter*⟩ and ⟨*replacement*⟩).

```
\cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
  {
    \group_align_safe_begin:
    \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
      { \exp_not:o { \use_none:nn ##1 } ##2 }
    \cs_set:Npx \__tl_replace_next:w ##1 #5
      {
        \exp_not:N \__tl_replace_wrap:w ##1
        \exp_not:n { #1 }
        \exp_not:n { \exp_not:n {#6} }
        \exp_not:n { #2 { } { } }
      }
    #3 #4
      {
        \exp_after:wN \__tl_replace_next:w
        \exp_after:wN { \exp_after:wN }
        \exp_after:wN { \exp_after:wN }
        #4
        #1
        {
          \if_false: { \fi: }
          \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
        }
        #5
      }
    \group_align_safe_end:
  }
\cs_new_eq:NN \__tl_replace_wrap:w ?
\cs_new_eq:NN \__tl_replace_next:w ?
```

(*End definition for* `\__tl_replace:NnNNNnn` *and others.*)

\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_once:Nn #1#2
  { \tl_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_once:Nn #1#2
  { \tl_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_once:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(*End definition for* `\tl_remove_once:Nn` *and* `\tl_gremove_once:Nn`*. These functions are documented on page 37.*)

\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_all:Nn #1#2
  { \tl_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_all:Nn #1#2
  { \tl_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_all:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

(*End definition for* `\tl_remove_all:Nn` *and* `\tl_gremove_all:Nn`*. These functions are documented on page 37.*)

5.6 Token list conditionals

\tl_if_blank_p:n
\tl_if_blank_p:V
\tl_if_blank_p:o
\tl_if_blank:nTF
\tl_if_blank:VTF
\tl_if_blank:oTF
__tl_if_blank_p:NNw

TeX skips spaces when reading a non-delimited arguments. Thus, a ⟨*token list*⟩ is blank if and only if `\use_none:n` ⟨*token list*⟩ `?` is empty after one expansion. The auxiliary `\__tl_if_empty_return:o` is a fast emptyness test, converting its argument to a string (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.

```
\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
  { \__tl_if_empty_return:o { \use_none:n #1 ? } }
\cs_generate_variant:Nn \tl_if_blank_p:n { V }
\cs_generate_variant:Nn \tl_if_blank:nT  { V }
```

```
\cs_generate_variant:Nn \tl_if_blank:nF  { V }
\cs_generate_variant:Nn \tl_if_blank:nTF { V }
\cs_generate_variant:Nn \tl_if_blank_p:n { o }
\cs_generate_variant:Nn \tl_if_blank:nT  { o }
\cs_generate_variant:Nn \tl_if_blank:nF  { o }
\cs_generate_variant:Nn \tl_if_blank:nTF { o }
```

(*End definition for* \tl_if_blank:nTF *and* __tl_if_blank_p:NNw*. These functions are documented on page 38.*)

\tl_if_empty_p:N
\tl_if_empty_p:c
\tl_if_empty:N*TF*
\tl_if_empty:c*TF*

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```
\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:N { c }
\cs_generate_variant:Nn \tl_if_empty:NT  { c }
\cs_generate_variant:Nn \tl_if_empty:NF  { c }
\cs_generate_variant:Nn \tl_if_empty:NTF { c }
```

(*End definition for* \tl_if_empty:NTF*. This function is documented on page 39.*)

\tl_if_empty_p:n
\tl_if_empty_p:V
\tl_if_empty:n*TF*
\tl_if_empty:V*TF*

Convert the argument to a string: this is empty if and only if the argument is. Then \if_meaning:w \q_nil ... \q_nil is true if and only if the string ... is empty. It could be tempting to use \if_meaning:w \q_nil #1 \q_nil directly. This fails on a token list starting with \q_nil of course but more troubling is the case where argument is a complete conditional such as \if_true: a \else: b \fi: because then \if_true: is used by \if_meaning:w, the test turns out false, the \else: executes the false branch, the \fi: ends it and the \q_nil at the end starts executing…

```
\prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
        \tl_to_str:n {#1} \q_nil
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:n { V }
\cs_generate_variant:Nn \tl_if_empty:nTF { V }
\cs_generate_variant:Nn \tl_if_empty:nT  { V }
\cs_generate_variant:Nn \tl_if_empty:nF  { V }
```

(*End definition for* \tl_if_empty:nTF*. This function is documented on page 39.*)

\tl_if_empty_p:o
\tl_if_empty:o*TF*
__tl_if_empty_return:o

The auxiliary function __tl_if_empty_return:o is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl_if_empty:nTF, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because \etex_detokenize:D expands tokens that follow until reading a catcode 1 (begin-group) token.

```
\cs_new:Npn \__tl_if_empty_return:o #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \etex_detokenize:D \exp_after:wN {#1} \q_nil
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
  { \__tl_if_empty_return:o {#1} }
```

(*End definition for* \tl_if_empty:oTF *and* __tl_if_empty_return:o*. These functions are documented on page 39.*)

\tl_if_eq_p:NN
\tl_if_eq_p:Nc
\tl_if_eq_p:cN
\tl_if_eq_p:cc
\tl_if_eq:NN*TF*
\tl_if_eq:Nc*TF*
\tl_if_eq:cN*TF*
\tl_if_eq:cc*TF*

Returns \c_true_bool if and only if the two token list variables are equal.

```
\prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1 #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNT  { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNF  { Nc , c , cc }
```

(*End definition for* \tl_if_eq:NNTF*. This function is documented on page 39.*)

\tl_if_eq:nn*TF*
\l__tl_internal_a_tl
\l__tl_internal_b_tl

A simple store and compare routine.

```
\prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F ,  TF }
  {
    \group_begin:
      \tl_set:Nn \l__tl_internal_a_tl {#1}
      \tl_set:Nn \l__tl_internal_b_tl {#2}
      \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
        \group_end:
        \prg_return_true:
      \else:
        \group_end:
        \prg_return_false:
      \fi:
  }
\tl_new:N \l__tl_internal_a_tl
\tl_new:N \l__tl_internal_b_tl
```

(*End definition for* \tl_if_eq:nnTF*,* \l__tl_internal_a_tl*, and* \l__tl_internal_b_tl*. These functions are documented on page 39.*)

\tl_if_in:NnTF
\tl_if_in:cnTF

See \tl_if_in:nnTF for further comments. Here we simply expand the token list variable and pass it to \tl_if_in:nnTF.

```
\cs_new_protected:Npn \tl_if_in:NnT  { \exp_args:No \tl_if_in:nnT  }
\cs_new_protected:Npn \tl_if_in:NnF  { \exp_args:No \tl_if_in:nnF  }
\cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
\cs_generate_variant:Nn \tl_if_in:NnT { c }
\cs_generate_variant:Nn \tl_if_in:NnF  { c }
\cs_generate_variant:Nn \tl_if_in:NnTF { c }
```

(*End definition for* \tl_if_in:NnTF*. This function is documented on page 39.*)

\tl_if_in:nnTF
\tl_if_in:VnTF
\tl_if_in:onTF
\tl_if_in:noTF

Once more, the test relies on the emptiness test for robustness. The function __tl_tmp:w removes tokens until the first occurrence of #2. If this does not appear in #1, then the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See \tl_if_empty:nTF for details on the emptiness test.

Treating correctly cases like \tl_if_in:nnTF {a state}{states}, where #1#2 contains #2 before the end, requires special care. To cater for this case, we insert {}{} between the two token lists. This marker may not appear in #2 because of TEX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The \if_false: constructions are a faster way to do \group_align_safe_begin: and \group_align_safe_end:.

```
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T   , F , TF }
  {
    \if_false: { \fi:
    \cs_set:Npn \__tl_tmp:w ##1 #2 { }
    \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
      { \prg_return_false: } { \prg_return_true: }
    \if_false: } \fi:
  }
\cs_generate_variant:Nn \tl_if_in:nnT  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnF  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }
```

(*End definition for* \tl_if_in:nnTF*. This function is documented on page 39.*)

\tl_if_single_p:N
\tl_if_single:NTF

Expand the token list and feed it to \tl_if_single:n.

```
\cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
\cs_new:Npn \tl_if_single:NT  { \exp_args:No \tl_if_single:nT  }
\cs_new:Npn \tl_if_single:NF  { \exp_args:No \tl_if_single:nF  }
\cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
```

(*End definition for* \tl_if_single:NTF*. This function is documented on page 39.*)

\tl_if_single_p:n
\tl_if_single:nTF
__tl_if_single_p:n
__tl_if_single:nTF

This test is similar to \tl_if_empty:nTF. Expanding \use_none:nn #1 ?? once yields an empty result if #1 is blank, a single ? if #1 has a single item, and otherwise yields some tokens ending with ??. Then, \tl_to_str:n makes sure there are no odd category codes. An earlier version would compare the result to a single ? using string comparison, but the Lua call is slow in LuaTEX. Instead, __tl_if_single:nnw picks the second token in front of it. If #1 is empty, this token is the trailing ? and the catcode test yields false. If #1 has a single item, the token is ^ and the catcode test yields true. Otherwise, it is one of the characters resulting from \tl_to_str:n, and the catcode test yields false. Note that \if_catcode:w takes care of the expansions, and that \tl_to_str:n (the \detokenize primitive) actually expands tokens until finding a begin-group token.

```
\prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
  {
    \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
        \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}
```

(*End definition for* `\tl_if_single:nTF` *and* `\__tl_if_single:nTF`*. These functions are documented on page 40.*)

`\tl_case:Nn`
`\tl_case:cn`
`\tl_case:NnTF`
`\tl_case:cnTF`
`\__tl_case:nnTF`
`\__tl_case:Nw`
`\__prg_case_end:nw`
`\__tl_case_end:nw`

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit "end of recursion" marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.

```
\cs_new:Npn \tl_case:Nn #1#2
  {
    \exp:w
    \__tl_case:NnTF #1 {#2} { } { }
  }
\cs_new:Npn \tl_case:NnT #1#2#3
  {
    \exp:w
    \__tl_case:NnTF #1 {#2} {#3} { }
  }
\cs_new:Npn \tl_case:NnF #1#2#3
  {
    \exp:w
    \__tl_case:NnTF #1 {#2} { } {#3}
  }
\cs_new:Npn \tl_case:NnTF #1#2
  {
    \exp:w
    \__tl_case:NnTF #1 {#2}
  }
\cs_new:Npn \__tl_case:NnTF #1#2#3#4
  { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__tl_case:Nw #1#2#3
  {
    \tl_if_eq:NNTF #1 #2
      { \__tl_case_end:nw {#3} }
      { \__tl_case:Nw #1 }
  }
\cs_generate_variant:Nn \tl_case:Nn   { c }
\cs_generate_variant:Nn \tl_case:NnT  { c }
\cs_generate_variant:Nn \tl_case:NnF  { c }
\cs_generate_variant:Nn \tl_case:NnTF { c }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then `#1` is the code to insert, `#2` is the *next* case to check on and `#3` is all of the rest of the cases code. That means that `#4` is the `true` branch code, and `#5` tidies up the spare \q_mark and the false branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \q_mark and so #4 is the false code (the true code is mopped up by #3).

```
\cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
  { \exp_end: #1 #4 }
\cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw
```

(*End definition for* \tl_case:NnTF *and others. These functions are documented on page 40.*)

5.7 Mapping to token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
__tl_map_function:Nn

Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the loop terminated.

```
\cs_new:Npn \tl_map_function:nN #1#2
  {
    \__tl_map_function:Nn #2 #1
      \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { }
  }
\cs_new:Npn \tl_map_function:NN
  { \exp_args:No \tl_map_function:nN }
\cs_new:Npn \__tl_map_function:Nn #1#2
  {
    \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
    #1 {#2} \__tl_map_function:Nn #1
  }
\cs_generate_variant:Nn \tl_map_function:NN { c }
```

(*End definition for* \tl_map_function:nN*,* \tl_map_function:NN*, and* __tl_map_function:Nn*. These functions are documented on page 40.*)

\tl_map_inline:nn
\tl_map_inline:Nn
\tl_map_inline:cn

The inline functions are straight forward by now. We use a little trick with the counter \g__prg_map_int to make them nestable. We can also make use of __tl_map_function:Nn from before.

```
\cs_new_protected:Npn \tl_map_inline:nn #1#2
  {
    \int_gincr:N \g__prg_map_int
    \cs_gset_protected:cpn
      { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
    \exp_args:Nc \__tl_map_function:Nn
      { __prg_map_ \int_use:N \g__prg_map_int :w }
      #1 \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
  }
\cs_new_protected:Npn \tl_map_inline:Nn
  { \exp_args:No \tl_map_inline:nn }
\cs_generate_variant:Nn \tl_map_inline:Nn { c }
```

(*End definition for* \tl_map_inline:nn *and* \tl_map_inline:Nn*. These functions are documented on page 40.*)

\tl_map_variable:nNn
\tl_map_variable:NNn
\tl_map_variable:cNn
__tl_map_variable:Nnn

\tl_map_variable:nNn ⟨*token list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
  {
    \__tl_map_variable:Nnn #2 {#3} #1
      \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { }
  }
\cs_new_protected:Npn \tl_map_variable:NNn
  { \exp_args:No \tl_map_variable:nNn }
\cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
  {
    \tl_set:Nn #1 {#3}
    \__quark_if_recursion_tail_break:NN #1 \tl_map_break:
    \use:n {#2}
    \__tl_map_variable:Nnn #1 {#2}
  }
\cs_generate_variant:Nn \tl_map_variable:NNn { c }
```

(*End definition for* `\tl_map_variable:nNn`*,* `\tl_map_variable:NNn`*, and* `\__tl_map_variable:Nnn`*. These functions are documented on page 41.*)

\tl_map_break:
\tl_map_break:n

The break statements use the general `\__prg_map_break:Nn`.

```
\cs_new:Npn \tl_map_break:
  { \__prg_map_break:Nn \tl_map_break: { } }
\cs_new:Npn \tl_map_break:n
  { \__prg_map_break:Nn \tl_map_break: }
```

(*End definition for* `\tl_map_break:` *and* `\tl_map_break:n`*. These functions are documented on page 41.*)

5.8 Using token lists

\tl_to_str:n
\tl_to_str:V

Another name for a primitive: defined in l3basics.

```
\cs_generate_variant:Nn \tl_to_str:n { V }
```

(*End definition for* `\tl_to_str:n`*. This function is documented on page 42.*)

\tl_to_str:N
\tl_to_str:c

These functions return the replacement text of a token list as a string.

```
\cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
\cs_generate_variant:Nn \tl_to_str:N { c }
```

(*End definition for* `\tl_to_str:N`*. This function is documented on page 42.*)

\tl_use:N
\tl_use:c

Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```
\cs_new:Npn \tl_use:N #1
  {
    \tl_if_exist:NTF #1 {#1}
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \tl_use:N { c }
```

(*End definition for* `\tl_use:N`. *This function is documented on page 42.*)

5.9 Working with the contents of token lists

`\tl_count:n`
`\tl_count:V`
`\tl_count:o`
`\tl_count:N`
`\tl_count:c`
`\__tl_count:n`

Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. `\__tl_count:n` grabs the element and replaces it by `+1`. The `0` ensures that it works on an empty list.

```
\cs_new:Npn \tl_count:n #1
  {
    \int_eval:n
      { 0 \tl_map_function:nN {#1} \__tl_count:n }
  }
\cs_new:Npn \tl_count:N #1
  {
    \int_eval:n
      { 0 \tl_map_function:NN #1 \__tl_count:n }
  }
\cs_new:Npn \__tl_count:n #1 { + 1 }
\cs_generate_variant:Nn \tl_count:n { V , o }
\cs_generate_variant:Nn \tl_count:N { c }
```

(*End definition for* `\tl_count:n`, `\tl_count:N`, *and* `\__tl_count:n`. *These functions are documented on page 42.*)

`\tl_reverse_items:n`
`\__tl_reverse_items:nwNwn`
`\__tl_reverse_items:wn`

Reversal of a token list is done by taking one item at a time and putting it after `\q_stop`.

```
\cs_new:Npn \tl_reverse_items:n #1
  {
    \__tl_reverse_items:nwNwn #1 ?
      \q_mark \__tl_reverse_items:nwNwn
      \q_mark \__tl_reverse_items:wn
      \q_stop { }
  }
\cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
  {
    #3 #2
      \q_mark \__tl_reverse_items:nwNwn
      \q_mark \__tl_reverse_items:wn
      \q_stop { {#1} #5 }
  }
\cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
  { \exp_not:o { \use_none:nn #2 } }
```

(*End definition for* `\tl_reverse_items:n`, `\__tl_reverse_items:nwNwn`, *and* `\__tl_reverse_items:wn`. *These functions are documented on page 43.*)

`\tl_trim_spaces:n`
`\tl_trim_spaces:N`
`\tl_trim_spaces:c`
`\tl_gtrim_spaces:N`
`\tl_gtrim_spaces:c`

Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `\q_mark`, and whose second argument is a ⟨*continuation*⟩, which receives as a braced argument `\use_none:n \q_mark` ⟨*trimmed token list*⟩. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming behaves correctly within an `x`-type expansion.

```
\cs_new:Npn \tl_trim_spaces:n #1
  { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
\cs_new_protected:Npn \tl_trim_spaces:N #1
  { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
```

```
\cs_new_protected:Npn \tl_gtrim_spaces:N #1
  { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
\cs_generate_variant:Nn \tl_trim_spaces:N  { c }
\cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

(*End definition for* `\tl_trim_spaces:n`*,* `\tl_trim_spaces:N`*, and* `\tl_gtrim_spaces:N`*. These functions are documented on page 43.*)

`\__tl_trim_spaces:nn`
`\__tl_trim_spaces_auxi:w`
`\__tl_trim_spaces_auxii:w`
`\__tl_trim_spaces_auxiii:w`
`\__tl_trim_spaces_auxiv:w`

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `\__tl_tmp:w`, which then receives a single space as its argument: `#1` is ␣. Removing leading spaces is done with `\__tl_trim_spaces_auxi:w`, which loops until `\q_mark␣` matches the end of the token list: then `##1` is the token list and `##3` is `\__tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `\__tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when ␣ `\q_nil` matches the one present in the definition of `\tl_trim_spacs:n`. Then `\__tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the ⟨*continuation*⟩.

```
\cs_set:Npn \__tl_tmp:w #1
  {
    \cs_new:Npn \__tl_trim_spaces:nn ##1
      {
        \__tl_trim_spaces_auxi:w
          ##1
          \q_nil
          \q_mark #1 { }
          \q_mark \__tl_trim_spaces_auxii:w
          \__tl_trim_spaces_auxiii:w
          #1 \q_nil
          \__tl_trim_spaces_auxiv:w
        \q_stop
      }
    \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
      {
        ##3
        \__tl_trim_spaces_auxi:w
        \q_mark
        ##2
        \q_mark #1 {##1}
      }
    \cs_new:Npn \__tl_trim_spaces_auxii:w
        \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
      {
        \__tl_trim_spaces_auxiii:w
        ##1
      }
    \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
      {
        ##2
        ##1 \q_nil
        \__tl_trim_spaces_auxiii:w
      }
    \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
      { ##3 { \use_none:n ##1 } }
  }
\__tl_tmp:w { ~ }
```

(*End definition for* `\__tl_trim_spaces:nn` *and others.*)

\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn
\tl_sort:nN

Implemented in l3sort.

(*End definition for* `\tl_sort:Nn`*,* `\tl_gsort:Nn`*, and* `\tl_sort:nN`*. These functions are documented on page 44.*)

5.10 Token by token changes

\q__tl_act_mark
\q__tl_act_stop

The `\tl_act` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list.Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `\__tl_act:NNNnn` functions. The quarks are effectively defined in l3quark.

(*End definition for* `\q__tl_act_mark` *and* `\q__tl_act_stop`*.*)

__tl_act:NNNnn
__tl_act_output:n
__tl_act_reverse_output:n
__tl_act_loop:w
__tl_act_normal:NwnNNN
__tl_act_group:nwnNNN
__tl_act_space:wwnNNN
__tl_act_end:w

To help control the expansion, `\__tl_act:NNNnn` should always be proceeded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in `#5`. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `\__tl_act_result:n`.

```
\cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
  {
    \group_align_safe_begin:
    \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
    {#4} #1 #2 #3
    \__tl_act_result:n { }
  }
```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q__tl_act_mark`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, `#3` and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `\__tl_act_space:wwnNNN` gobble the space.

```
\cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_act_normal:NwnNNN }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__tl_act_group:nwnNNN }
          { \__tl_act_space:wwnNNN }
      }
    #1 \q__tl_act_stop
  }
\cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
  {
    \if_meaning:w \q__tl_act_mark #1
      \exp_after:wN \__tl_act_end:wn
    \fi:
```

```
    #4 {#3} #1
    \__tl_act_loop:w #2 \q__tl_act_stop
    {#3} #4
  }
\cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
  { \group_align_safe_end: \exp_end: #2 }
\cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
  {
    #5 {#3} {#1}
    \__tl_act_loop:w #2 \q__tl_act_stop
    {#3} #4 #5
  }
\exp_last_unbraced:NNo
  \cs_new:Npn \__tl_act_space:wwnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
  {
    #5 {#2}
    \__tl_act_loop:w #1 \q__tl_act_stop
    {#2} #3 #4 #5
  }
```

Typically, the output is done to the right of what was already output, using `\__tl_act_output:n`, but for the `\__tl_act_reverse` functions, it should be done to the left.

```
\cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #3 #1 } }
\cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #1 #3 } }
```

(*End definition for* `\__tl_act:NNNnn` *and others.*)

`\tl_reverse:n`
`\tl_reverse:o`
`\tl_reverse:V`
`\__tl_reverse_normal:nN`
`\__tl_reverse_group_preserve:nn`
`\__tl_reverse_space:n`

The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `\__tl_act:NNNnn`. Spaces and "normal" tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by `\__tl_act:NNNnn` when changing case (to record which direction the change is in), but not when reversing the tokens.

```
\cs_new:Npn \tl_reverse:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \exp:w
        \__tl_act:NNNnn
          \__tl_reverse_normal:nN
          \__tl_reverse_group_preserve:nn
          \__tl_reverse_space:n
          { }
          {#1}
      }
  }
\cs_generate_variant:Nn \tl_reverse:n { o , V }
\cs_new:Npn \__tl_reverse_normal:nN #1#2
  { \__tl_act_reverse_output:n {#2} }
\cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
  { \__tl_act_reverse_output:n { {#2} } }
\cs_new:Npn \__tl_reverse_space:n #1
  { \__tl_act_reverse_output:n { ~ } }
```

(*End definition for* `\tl_reverse:n` *and others. These functions are documented on page 43.*)

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

This reverses the list, leaving `\exp_stop_f:` in front, which stops the `f`-expansion.

```
\cs_new_protected:Npn \tl_reverse:N #1
  { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_new_protected:Npn \tl_greverse:N #1
  { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_generate_variant:Nn \tl_reverse:N  { c }
\cs_generate_variant:Nn \tl_greverse:N { c }
```

(*End definition for* `\tl_reverse:N` *and* `\tl_greverse:N`*. These functions are documented on page 43.*)

5.11 The first token from a token list

\tl_head:N
\tl_head:n
\tl_head:V
\tl_head:v
\tl_head:f
__tl_head_auxi:nw
__tl_head_auxii:n
\tl_head:w
\tl_tail:N
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f

Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping to a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. Using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. Is there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in `http://tex.stackexchange.com/a/70168`.

```
\cs_new:Npn \tl_head:n #1
  {
    \etex_unexpanded:D
      \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
  }
\cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
  {
    \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
      \if_false: } \fi: {#1}
  }
\cs_new:Npn \__tl_head_auxii:n #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {#1}
      { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
  }
\cs_generate_variant:Nn \tl_head:n { V , v , f }
\cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
\cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }
```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop
```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens "banned" in the input, which we do not have with this definition.

```
\cs_new:Npn \tl_tail:n #1
  {
    \etex_unexpanded:D
      \tl_if_blank:nTF {#1}
        { { } }
        { \exp_after:wN { \use_none:n #1 } }
  }
\cs_generate_variant:Nn \tl_tail:n { V , v , f }
\cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(*End definition for `\tl_head:N` and others. These functions are documented on page 44.*)

`\tl_if_head_eq_meaning_p:nN`
`\tl_if_head_eq_meaning:nNTF`
`\tl_if_head_eq_charcode_p:nN`
`\tl_if_head_eq_charcode:nNTF`
`\tl_if_head_eq_charcode_p:fN`
`\tl_if_head_eq_charcode:fNTF`
`\tl_if_head_eq_catcode_p:nN`
`\tl_if_head_eq_catcode:nNTF`

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
    \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
    \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an `N`-type token (the extra `?` sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two tokens: `?` which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was `true` or `false`.

```
\prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
  {
    \if_charcode:w
        \exp_not:N #2
        \tl_if_head_is_N_type:nTF { #1 ? }
          {
            \exp_after:wN \exp_not:N
            \tl_head:w #1 { ? \use_none:nn } \q_stop
          }
          { \str_head:n {#1} }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
```

```
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT  { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF  { f }
```

For \tl_if_head_eq_catcode:nN, again we detect special cases with a \tl_if_head_is_N_type:n. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either \c_group_begin_token or \c_space_token. Again, for an empty argument, a hack is used, removing \prg_return_true: and \else: with \use_none:nn in case the catcode test with the (arbitrarily chosen) ? is true.

```
\prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
  {
    \if_catcode:w
        \exp_not:N #2
        \tl_if_head_is_N_type:nTF { #1 ? }
          {
            \exp_after:wN \exp_not:N
            \tl_head:w #1 { ? \use_none:nn } \q_stop
          }
          {
            \tl_if_head_is_group:nTF {#1}
              { \c_group_begin_token }
              { \c_space_token }
          }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

For \tl_if_head_eq_meaning:nN, again, detect special cases. In the normal case, use \tl_head:w, with no \exp_not:N this time, since \if_meaning:w causes no expansion. With an empty argument, the test is true, and \use_none:nnn removes #2 and the usual \prg_return_true: and \else:. In the special cases, we know that the first token is a character, hence \if_charcode:w and \if_catcode:w together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```
\prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF { #1 ? }
      { \__tl_if_head_eq_meaning_normal:nN }
      { \__tl_if_head_eq_meaning_special:nN }
    {#1} #2
  }
\cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
  {
    \exp_after:wN \if_meaning:w
        \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
  {
```

```
  \if_charcode:w \str_head:n {#1} \exp_not:N #2
   \exp_after:wN \use:n
  \else:
   \prg_return_false:
   \exp_after:wN \use_none:n
  \fi:
  {
   \if_catcode:w \exp_not:N #2
          \tl_if_head_is_group:nTF {#1}
           { \c_group_begin_token }
           { \c_space_token }
    \prg_return_true:
   \else:
    \prg_return_false:
   \fi:
  }
 }
```

(*End definition for* `\tl_if_head_eq_meaning:nNTF` *and others. These functions are documented on page 45.*)

\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
__tl_if_head_is_N_type:w

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving `\__tl_if_head_is_N_type:w` produces ^ (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `*` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the `true` branch of the conditional.

```
\prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
 {
  \if_catcode:w
    \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
    \exp_after:wN \use_none:n
     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
    * *
   \prg_return_true:
  \else:
   \prg_return_false:
  \fi:
 }
\cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
 {
  \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
  \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
 }
```

(*End definition for* `\tl_if_head_is_N_type:nTF` *and* `\__tl_if_head_is_N_type:w`*. These functions are documented on page 46.*)

\tl_if_head_is_group_p:n
\tl_if_head_is_group:nTF

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument.[8]

[8] Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we need all brace "tricks" to happen in one step of expansion, keeping the token list brace balanced at all times.

```
\prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
  {
    \if_catcode:w
        \exp_after:wN \use_none:n
          \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
        * *
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\tl_if_head_is_group:nTF`*. This function is documented on page 46.*)

\tl_if_head_is_space_p:n
\tl_if_head_is_space:nTF
__tl_if_head_is_space:w

The auxiliary's argument is all that is before the first explicit space in `?#1?~`. If that is a single `?` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from TeX in a table, and to allow for removing what remains of the token list after its first space. The `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```
\prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
  {
    \exp:w \if_false: { \fi:
      \__tl_if_head_is_space:w ? #1 ? ~ }
  }
\cs_new:Npn \__tl_if_head_is_space:w #1 ~
  {
    \tl_if_empty:oTF { \use_none:n #1 }
      { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
      { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
```

(*End definition for* `\tl_if_head_is_space:nTF` *and* `\__tl_if_head_is_space:w`*. These functions are documented on page 46.*)

5.12 Using a single item

\tl_item:nn
\tl_item:Nn
\tl_item:cn
__tl_item_aux:nn
__tl_item:nn

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\quark_if_recursion_tail_stop:n` terminates the loop, and returns nothing at all.

```
\cs_new:Npn \tl_item:nn #1#2
  {
    \exp_args:Nf \__tl_item:nn
      { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
    #1
    \q_recursion_tail
    \__prg_break_point:
  }
\cs_new:Npn \__tl_item_aux:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
      {#1}
  }
\cs_new:Npn \__tl_item:nn #1#2
  {
    \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
    \int_compare:nNnTF {#1} = 1
      { \__prg_break:n { \exp_not:n {#2} } }
      { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
  }
\cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
\cs_generate_variant:Nn \tl_item:Nn { c }
```

(*End definition for* `\tl_item:nn` *and others. These functions are documented on page 46.*)

5.13 Viewing token lists

\tl_show:N
\tl_show:c

Showing token list variables is done after checking that the variable is defined (see __kernel_register_show:N).

```
\cs_new_protected:Npn \tl_show:N #1
  {
    \__msg_show_variable:NNNnn #1 \tl_if_exist:NTF ? { }
      { > ~ \token_to_str:N #1 = \tl_to_str:N #1 }
  }
\cs_generate_variant:Nn \tl_show:N { c }
```

(*End definition for* `\tl_show:N`*. This function is documented on page 46.*)

\tl_show:n

The __msg_show_wrap:n internal function performs line-wrapping and shows the result using the \etex_showtokens:D primitive. Since \tl_to_str:n is expanded within the line-wrapping code, the escape character is always a backslash.

```
\cs_new_protected:Npn \tl_show:n #1
  { \__msg_show_wrap:n { > ~ \tl_to_str:n {#1} } }
```

(*End definition for* `\tl_show:n`*. This function is documented on page 47.*)

\tl_log:N
\tl_log:c
\tl_log:n

Redirect output of \tl_show:N and \tl_show:n to the log.

```
\cs_new_protected:Npn \tl_log:N
  { \__msg_log_next: \tl_show:N }
\cs_generate_variant:Nn \tl_log:N { c }
\cs_new_protected:Npn \tl_log:n
  { \__msg_log_next: \tl_show:n }
```

(*End definition for* `\tl_log:N` *and* `\tl_log:n`*. These functions are documented on page 47.*)

5.14 Scratch token lists

\g_tmpa_tl
\g_tmpb_tl

Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
\tl_new:N \g_tmpa_tl
\tl_new:N \g_tmpb_tl
```

(*End definition for* `\g_tmpa_tl` *and* `\g_tmpb_tl`*. These variables are documented on page 47.*)

\l_tmpa_tl
\l_tmpb_tl

These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
\tl_new:N \l_tmpa_tl
\tl_new:N \l_tmpb_tl
```

(*End definition for* `\l_tmpa_tl` *and* `\l_tmpb_tl`*. These variables are documented on page 47.*)

5.15 Deprecated functions

\tl_to_lowercase:n
\tl_to_uppercase:n

For removal after 2017-12-31.

```
\__debug_deprecation:nnNNpn { 2017-12-31 } { \tex_lowercase:D }
\cs_new_protected:Npn \tl_to_lowercase:n #1 { \tex_lowercase:D {#1} }
\__debug_deprecation:nnNNpn { 2017-12-31 } { \tex_uppercase:D }
\cs_new_protected:Npn \tl_to_uppercase:n #1 { \tex_uppercase:D {#1} }
```

(*End definition for* `\tl_to_lowercase:n` *and* `\tl_to_uppercase:n`*.*)

```
⟨/initex | package⟩
```

6 l3str implementation

```
⟨*initex | package⟩
```

```
⟨@@=str⟩
```

6.1 Creating and setting string variables

\str_new:N
\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc

A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```
\group_begin:
  \cs_set_protected:Npn \__str_tmp:n #1
    {
      \tl_if_blank:nF {#1}
        {
          \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
          \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
          \__str_tmp:n
        }
    }
  \__str_tmp:n
    { new }
    { use }
    { clear }
    { gclear }
    { clear_new }
    { gclear_new }
    { }
\group_end:
\cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
\cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \str_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
```

(*End definition for* `\str_new:N` *and others. These functions are documented on page 48.*)

\str_set:Nn
\str_set:Nx
\str_set:cn
\str_set:cx
\str_gset:Nn
\str_gset:Nx
\str_gset:cn
\str_gset:cx
\str_const:Nn
\str_const:Nx
\str_const:cn
\str_const:cx
\str_put_left:Nn
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cx
\str_put_right:Nn
\str_put_right:Nx
\str_put_right:cn
\str_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx

Simply convert the token list inputs to ⟨*strings*⟩.

```
\group_begin:
  \cs_set_protected:Npn \__str_tmp:n #1
    {
      \tl_if_blank:nF {#1}
        {
          \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
            { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
          \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
          \__str_tmp:n
        }
    }
  \__str_tmp:n
    { set }
    { gset }
    { const }
    { put_left }
    { gput_left }
    { put_right }
    { gput_right }
    { }
\group_end:
```

(*End definition for* `\str_set:Nn` *and others. These functions are documented on page 49.*)

6.2 String comparisons

\str_if_empty_p:N
\str_if_empty_p:c
\str_if_empty:N*TF*
\str_if_empty:c*TF*
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:N*TF*
\str_if_exist:c*TF*

More copy-paste!

```
\prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c { p , T , F , TF }
```

(*End definition for* `\str_if_empty:NTF` *and* `\str_if_exist:NTF`*. These functions are documented on page 50.*)

__str_if_eq_x:nn
__str_escape_x:n

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTEX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTEX case. Note that the necessary Lua code is covered in l3boostrap: long-term this may need to go into a separate Lua file, but at present it's somewhere that spaces are not skipped for ease-of-input. The need to detokenize and force expansion of input arises from the case where a `#` token is used in the input, *e.g.* `\__str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise would fail as `\luatex_luaescapestring:D` does not double such tokens.

```
\cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdftex_strcmp:D {#1} {#2} }
\cs_if_exist:NT \luatex_luatexversion:D
   {
     \cs_set:Npn \__str_if_eq_x:nn #1#2
       {
          \luatex_directlua:D
            {
              l3kernel.strcmp
```

```
        (
         " \__str_escape_x:n {#1} " ,
         " \__str_escape_x:n {#2} "
        )
      }
   }
  \cs_new:Npn \__str_escape_x:n #1
   {
    \luatex_luaescapestring:D
     {
      \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
     }
   }
 }
```

(*End definition for* `\__str_if_eq_x:nn` *and* `\__str_escape_x:n`.)

`\__str_if_eq_x_return:nn`

It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF` (see l3str), but is hard-coded for speed.

```
\cs_new:Npn \__str_if_eq_x_return:nn #1 #2
 {
  \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = 0 \exp_stop_f:
   \prg_return_true:
  \else:
   \prg_return_false:
  \fi:
 }
```

(*End definition for* `\__str_if_eq_x_return:nn`.)

`\str_if_eq_p:nn`
`\str_if_eq_p:Vn`
`\str_if_eq_p:on`
`\str_if_eq_p:nV`
`\str_if_eq_p:no`
`\str_if_eq_p:VV`
`\str_if_eq:nnTF`
`\str_if_eq:VnTF`
`\str_if_eq:onTF`
`\str_if_eq:nVTF`
`\str_if_eq:noTF`
`\str_if_eq:VVTF`
`\str_if_eq_x_p:nn`
`\str_if_eq_x:nnTF`

Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

```
\prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
 {
  \if_int_compare:w
   \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
   = 0 \exp_stop_f:
   \prg_return_true: \else: \prg_return_false: \fi:
 }
\cs_generate_variant:Nn \str_if_eq_p:nn {  V ,  o }
\cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnT  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnT  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnF  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnF  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnTF {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
\prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
 {
  \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = 0 \exp_stop_f:
   \prg_return_true: \else: \prg_return_false: \fi:
 }
```

(*End definition for* \str_if_eq:nnTF *and* \str_if_eq_x:nnTF*. These functions are documented on page 50.*)

\str_if_eq_p:NN
\str_if_eq_p:Nc
\str_if_eq_p:cN
\str_if_eq_p:cc
\str_if_eq:NN*TF*
\str_if_eq:Nc*TF*
\str_if_eq:cN*TF*
\str_if_eq:cc*TF*

Note that \str_if_eq:NN is different from \tl_if_eq:NN because it needs to ignore category codes.

```
\prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
  {
    \if_int_compare:w \__str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
      = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
  }
\cs_generate_variant:Nn \str_if_eq:NNT  { c , Nc , cc }
\cs_generate_variant:Nn \str_if_eq:NNF  { c , Nc , cc }
\cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }
\cs_generate_variant:Nn \str_if_eq_p:NN { c , Nc , cc }
```

(*End definition for* \str_if_eq:NNTF*. This function is documented on page 50.*)

\str_case:nn
\str_case:on
\str_case:nV
\str_case:nv
\str_case:nn*TF*
\str_case:on*TF*
\str_case:nV*TF*
\str_case:nv*TF*
\str_case_x:nn
\str_case_x:nn*TF*
__str_case:nnTF
__str_case_x:nnTF
__str_case:nw
__str_case_x:nw
__str_case_end:nw

Much the same as \tl_case:nn(TF) here: just a change in the internal comparison.

```
\cs_new:Npn \str_case:nn #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case:nnT #1#2#3
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case:nnF #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case:nnTF #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2}
  }
\cs_new:Npn \__str_case:nnTF #1#2#3#4
  { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_generate_variant:Nn \str_case:nn   { o , nV , nv }
\cs_generate_variant:Nn \str_case:nnT  { o , nV , nv }
\cs_generate_variant:Nn \str_case:nnF  { o , nV , nv }
\cs_generate_variant:Nn \str_case:nnTF { o , nV , nv }
\cs_new:Npn \__str_case:nw #1#2#3
  {
    \str_if_eq:nnTF {#1} {#2}
      { \__str_case_end:nw {#3} }
      { \__str_case:nw {#1} }
  }
\cs_new:Npn \str_case_x:nn #1#2
  {
    \exp:w
    \__str_case_x:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case_x:nnT #1#2#3
  {
    \exp:w
    \__str_case_x:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case_x:nnF #1#2
  {
    \exp:w
    \__str_case_x:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case_x:nnTF #1#2
  {
    \exp:w
    \__str_case_x:nnTF {#1} {#2}
  }
\cs_new:Npn \__str_case_x:nnTF #1#2#3#4
  { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__str_case_x:nw #1#2#3
  {
    \str_if_eq_x:nnTF {#1} {#2}
      { \__str_case_end:nw {#3} }
      { \__str_case_x:nw {#1} }
  }
\cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\str_case:nnTF` *and others. These functions are documented on page 51.*)

6.3 Accessing specific characters in a string

__str_to_other:n
__str_to_other_loop:w
__str_to_other_end:w

First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\q_mark` and `\q_stop` markers. The end is detected when `\__str_to_other_loop:w` finds one of the trailing `A`, distinguished from any contents of the initial token list by their category. Then `\__str_to_other_end:w` is called, and finds the result between `\q_mark` and the first `A` (well, there is also the need to remove a space).

```
\cs_new:Npn \__str_to_other:n #1
  {
    \exp_after:wN \__str_to_other_loop:w
      \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
  }
\group_begin:
\tex_lccode:D ‘\* = ‘\  %
\tex_lccode:D ‘\A = ‘\A
\tex_lowercase:D
  {
    \group_end:
    \cs_new:Npn \__str_to_other_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
      {
        \if_meaning:w A #8
          \__str_to_other_end:w
        \fi:
        \__str_to_other_loop:w
        #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
      }
    \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
      { \fi: #2 }
  }
```

(*End definition for* `\__str_to_other:n`*,* `\__str_to_other_loop:w`*, and* `\__str_to_other_end:w`*.*)

`\__str_to_other_fast:n`
`\__str_to_other_fast_loop:w`
`\__str_to_other_fast_end:w`

The difference with `\__str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```
\cs_new:Npn \__str_to_other_fast:n #1
  {
    \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
      A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop
  }
\group_begin:
\tex_lccode:D `\* = `\  %
\tex_lccode:D `\A = `\A
\tex_lowercase:D
  {
    \group_end:
    \cs_new:Npn \__str_to_other_fast_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
      {
        \if_meaning:w A #9
          \__str_to_other_fast_end:w
        \fi:
        #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
        \__str_to_other_fast_loop:w *
      }
    \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \q_stop {#1}
  }
```

(*End definition for* `\__str_to_other_fast:n`*,* `\__str_to_other_fast_loop:w`*, and* `\__str_to_other_fast_end:w`*.*)

`\str_item:Nn`
`\str_item:cn`
`\str_item:nn`
`\str_item_ignore_spaces:nn`
`\__str_item:nn`
`\__str_item:w`

The `\str_item:nn` hands its argument with spaces escaped to `\__str_item:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `\__str_item:nn` since everything else is done with undelimited arguments. Evaluate the ⟨*index*⟩ argument `#2` and count characters in the string, passing those two numbers to `\__str_item:w` for further analysis. If the ⟨*index*⟩ is negative, shift it by the ⟨*count*⟩ to know the how many character to discard, and if that is still negative give an empty result. If the ⟨*index*⟩ is larger than the ⟨*count*⟩, give an empty result, and otherwise discard ⟨*index*⟩ $-$ 1 characters before returning the following one. The shift by -1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the ⟨*index*⟩ is zero.

```
\cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
\cs_generate_variant:Nn \str_item:Nn { c }
\cs_new:Npn \str_item:nn #1#2
  {
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \__str_item:nn
```

```
          { \__str_to_other:n {#1} } {#2}
      }
  }
\cs_new:Npn \str_item_ignore_spaces:nn #1
  { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new:Npn \__str_item:nn #1#2
  {
    \exp_after:wN \__str_item:w
    \__int_value:w \__int_eval:w #2 \exp_after:wN ;
    \__int_value:w \__str_count:n {#1} ;
    #1 \q_stop
  }
\cs_new:Npn \__str_item:w #1; #2;
  {
    \int_compare:nNnTF {#1} < 0
      {
        \int_compare:nNnTF {#1} < {-#2}
          { \use_none_delimit_by_q_stop:w }
          {
            \exp_after:wN \use_i_delimit_by_q_stop:nw
            \exp:w \exp_after:wN \__str_skip_exp_end:w
              \__int_value:w \__int_eval:w #1 + #2 ;
          }
      }
      {
        \int_compare:nNnTF {#1} > {#2}
          { \use_none_delimit_by_q_stop:w }
          {
            \exp_after:wN \use_i_delimit_by_q_stop:nw
            \exp:w \__str_skip_exp_end:w #1 ; { }
          }
      }
  }
```

(*End definition for* `\str_item:Nn` *and others. These functions are documented on page 53.*)

__str_skip_exp_end:w
__str_skip_loop:wNNNNNNNN
__str_skip_end:w
__str_skip_end:NNNNNNNN

Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `\__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `\__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `\__str_skip_exp_end:w` is called).

```
\cs_new:Npn \__str_skip_exp_end:w #1;
  {
    \if_int_compare:w #1 > 8 \exp_stop_f:
      \exp_after:wN \__str_skip_loop:wNNNNNNNN
    \else:
      \exp_after:wN \__str_skip_end:w
      \__int_value:w \__int_eval:w
    \fi:
```

```
    #1 ;
  }
\cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  { \exp_after:wN \__str_skip_exp_end:w \__int_value:w \__int_eval:w #1 - 8 ; }
\cs_new:Npn \__str_skip_end:w #1 ;
  {
    \exp_after:wN \__str_skip_end:NNNNNNNN
    \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
  }
\cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }
```

(*End definition for* `\__str_skip_exp_end:w` *and others.*)

`\str_range:Nnn`
`\str_range:nnn`
`\str_range_ignore_spaces:nnn`
`\__str_range:nnn`
`\__str_range:w`
`\__str_range:nnw`

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the ⟨*start index*⟩, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```
\cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
\cs_generate_variant:Nn \str_range:Nnn { c }
\cs_new:Npn \str_range:nnn #1#2#3
  {
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \__str_range:nnn
          { \__str_to_other:n {#1} } {#2} {#3}
      }
  }
\cs_new:Npn \str_range_ignore_spaces:nnn #1
  { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
\__debug_patch_args:nNNpn { {#1} { (#2) } { (#3) } }
\cs_new:Npn \__str_range:nnn #1#2#3
  {
    \exp_after:wN \__str_range:w
    \__int_value:w \__str_count:n {#1} \exp_after:wN ;
    \__int_value:w \__int_eval:w #2 - 1 \exp_after:wN ;
    \__int_value:w \__int_eval:w #3 ;
    #1 \q_stop
  }
\cs_new:Npn \__str_range:w #1; #2; #3;
  {
    \exp_args:Nf \__str_range:nnw
      { \__str_range_normalize:nn {#2} {#1} }
      { \__str_range_normalize:nn {#3} {#1} }
  }
\cs_new:Npn \__str_range:nnw #1#2
  {
    \exp_after:wN \__str_collect_delimit_by_q_stop:w
    \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
    \exp:w \__str_skip_exp_end:w #1 ;
  }
```

(*End definition for* `\str_range:Nnn` *and others. These functions are documented on page 53.*)

`\__str_range_normalize:nn`

This function converts an ⟨*index*⟩ argument into an explicit position in the string (a result of 0 denoting "out of bounds"). Expects two explicit integer arguments: the ⟨*index*⟩ `#1` and the string count `#2`. If `#1` is negative, replace it by `#1` + `#2` + 1, then limit to the range [0, `#2`].

```
\cs_new:Npn \__str_range_normalize:nn #1#2
  {
    \int_eval:n
      {
        \if_int_compare:w #1 < 0 \exp_stop_f:
          \if_int_compare:w #1 < -#2 \exp_stop_f:
            0
          \else:
            #1 + #2 + 1
          \fi:
        \else:
          \if_int_compare:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

(*End definition for* `\__str_range_normalize:nn`.)

`\__str_collect_delimit_by_q_stop:w`
`\__str_collect_loop:wn`
`\__str_collect_loop:wnNNNNNNN`
`\__str_collect_end:wn`
`\__str_collect_end:nnnnnnnnw`

Collects `max(#1,0)` characters, and removes everything else until `\q_stop`. This is somewhat similar to `\__str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `\__str_collect_end:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```
\cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
  { \__str_collect_loop:wn #1 ; { } }
\cs_new:Npn \__str_collect_loop:wn #1 ;
  {
    \if_int_compare:w #1 > 7 \exp_stop_f:
      \exp_after:wN \__str_collect_loop:wnNNNNNNN
    \else:
      \exp_after:wN \__str_collect_end:wn
    \fi:
    #1 ;
  }
\cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
  {
    \exp_after:wN \__str_collect_loop:wn
    \__int_value:w \__int_eval:w #1 - 7 ;
    { #2 #3#4#5#6#7#8#9 }
  }
\cs_new:Npn \__str_collect_end:wn #1 ;
  {
    \exp_after:wN \__str_collect_end:nnnnnnnnw
    \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f: #1 \else: 0 \fi: \exp_stop_f:
```

```
    \or: \or: \or: \or: \or: \or: \fi:
  }
\cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
  { #1#2#3#4#5#6#7#8 }
```

(*End definition for* `\__str_collect_delimit_by_q_stop:w` *and others.*)

6.4 Counting characters

`\str_count_spaces:N`
`\str_count_spaces:c`
`\str_count_spaces:n`
`\__str_count_spaces_loop:w`

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing X⟨*number*⟩, and that ⟨*number*⟩ is added to the sum of 9 that precedes, to adjust the result.

```
\cs_new:Npn \str_count_spaces:N
  { \exp_args:No \str_count_spaces:n }
\cs_generate_variant:Nn \str_count_spaces:N { c }
\cs_new:Npn \str_count_spaces:n #1
  {
    \int_eval:n
      {
        \exp_after:wN \__str_count_spaces_loop:w
        \tl_to_str:n {#1} ~
        X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
        \q_stop
      }
  }
\cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
  {
    \if_meaning:w X #9
      \use_i_delimit_by_q_stop:nw
    \fi:
    9 + \__str_count_spaces_loop:w
  }
```

(*End definition for* `\str_count_spaces:N`*,* `\str_count_spaces:n`*, and* `\__str_count_spaces_loop:w`*. These functions are documented on page 52.*)

`\str_count:N`
`\str_count:c`
`\str_count:n`
`\str_count_ignore_spaces:n`
`\__str_count:n`
`\__str_count_aux:n`
`\__str_count_loop:NNNNNNNNN`

To count characters in a string we could first escape all spaces using `\__str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `\__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```
\cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
\cs_generate_variant:Nn \str_count:N { c }
\cs_new:Npn \str_count:n #1
  {
    \__str_count_aux:n
      {
        \str_count_spaces:n {#1}
        + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
      }
  }
\cs_new:Npn \__str_count:n #1
  {
    \__str_count_aux:n
      { \__str_count_loop:NNNNNNNNN #1 }
  }
\cs_new:Npn \str_count_ignore_spaces:n #1
  {
    \__str_count_aux:n
      { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
  }
\cs_new:Npn \__str_count_aux:n #1
  {
    \int_eval:n
      {
        #1
        { X 8 } { X 7 } { X 6 }
        { X 5 } { X 4 } { X 3 }
        { X 2 } { X 1 } { X 0 }
        \q_stop
      }
  }
\cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_meaning:w X #9
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    9 + \__str_count_loop:NNNNNNNNN
  }
```

(*End definition for* \str_count:N *and others. These functions are documented on page 52.*)

6.5 The first character in a string

\str_head:N
\str_head:c
\str_head:n
\str_head_ignore_spaces:n
__str_head:w

The _ignore_spaces variant applies \tl_to_str:n then grabs the first item, thus skipping spaces. As usual, \str_head:N expands its argument and hands it to \str_head:n. To circumvent the fact that TEX skips spaces when grabbing undelimited macro parameters, __str_head:w takes an argument delimited by a space. If #1 starts with a non-space character, \use_i_delimit_by_q_stop:nw leaves that in the input stream. On the other hand, if #1 starts with a space, the __str_head:w takes an empty argument, and the single (initially braced) space in the definition of __str_head:w makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of \str_head:n gives an empty result after passing through \use_i_delimit_by_q_stop:nw.

```
\cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
\cs_generate_variant:Nn \str_head:N { c }
\cs_new:Npn \str_head:n #1
  {
    \exp_after:wN \__str_head:w
    \tl_to_str:n {#1}
    { { } } ~ \q_stop
  }
\cs_new:Npn \__str_head:w #1 ~ %
  { \use_i_delimit_by_q_stop:nw #1 { ~ } }
\cs_new:Npn \str_head_ignore_spaces:n #1
  {
    \exp_after:wN \use_i_delimit_by_q_stop:nw
    \tl_to_str:n {#1} { } \q_stop
  }
```

(*End definition for* `\str_head:N` *and others. These functions are documented on page 52.*)

\str_tail:N
\str_tail:c
\str_tail:n
\str_tail_ignore_spaces:n
__str_tail_auxi:w
__str_tail_auxii:w

Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:`. This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:`. The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `\__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```
\cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
\cs_generate_variant:Nn \str_tail:N { c }
\cs_new:Npn \str_tail:n #1
  {
    \exp_after:wN \__str_tail_auxi:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \q_stop
  }
\cs_new:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
\cs_new:Npn \str_tail_ignore_spaces:n #1
  {
    \exp_after:wN \__str_tail_auxii:w
    \tl_to_str:n {#1} \q_mark \q_mark \q_stop
  }
\cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }
```

(*End definition for* `\str_tail:N` *and others. These functions are documented on page 52.*)

6.6 String manipulation

\str_fold_case:n
\str_fold_case:V
\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case:f
__str_change_case:nn
__str_change_case_aux:nn
__str_change_case_result:n
__str_change_case_output:nw
__str_change_case_output:fw
__str_change_case_end:nw
__str_change_case_loop:nw
__str_change_case_space:n
__str_change_case_char:nN
__str_lookup_lower:N
__str_lookup_upper:N
__str_lookup_fold:N

Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.

```
\cs_new:Npn \str_fold_case:n  #1 { \__str_change_case:nn {#1} { fold } }
\cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
\cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
\cs_generate_variant:Nn \str_fold_case:n  { V }
\cs_generate_variant:Nn \str_lower_case:n { f }
\cs_generate_variant:Nn \str_upper_case:n { f }
\cs_new:Npn \__str_change_case:nn #1
  {
    \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
      { \tl_to_str:n {#1} }
  }
\cs_new:Npn \__str_change_case_aux:nn #1#2
  {
    \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
      \__str_change_case_result:n { }
  }
\cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
  { #2 \__str_change_case_result:n { #3 #1 } }
\cs_generate_variant:Nn  \__str_change_case_output:nw { f }
\cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2 { #2 }
\cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
  {
    \tl_if_head_is_space:nTF {#2}
      { \__str_change_case_space:n }
      { \__str_change_case_char:nN }
    {#1} #2 \q_recursion_stop
  }
\use:x
  { \cs_new:Npn \exp_not:N \__str_change_case_space:n ##1 \c_space_tl }
  {
    \__str_change_case_output:nw { ~ }
    \__str_change_case_loop:nw {#1}
  }
\cs_new:Npn \__str_change_case_char:nN #1#2
  {
    \quark_if_recursion_tail_stop_do:Nn #2
      { \__str_change_case_end:wn }
    \cs_if_exist:cTF { c__unicode_ #1 _ #2 _tl }
      {
        \__str_change_case_output:fw
          { \tl_to_str:c { c__unicode_ #1 _ #2 _tl } }
      }
      { \__str_change_case_char_aux:nN {#1} #2 }
    \__str_change_case_loop:nw {#1}
  }
```

For Unicode engines there's a look up to see if the current character has a valid one-to-one case change mapping. That's not needed for 8-bit engines: as they don't have \utex_char:D all of the changes they can make are hard-coded and so already picked up above.

```
\cs_if_exist:NTF \utex_char:D
  {
    \cs_new:Npn \__str_change_case_char_aux:nN #1#2
      {
        \int_compare:nNnTF { \use:c { __str_lookup_ #1 :N } #2 } = { 0 }
          { \__str_change_case_output:nw {#2} }
          {
            \__str_change_case_output:fw
              { \utex_char:D \use:c { __str_lookup_ #1 :N } #2 ~ }
          }
      }
    \cs_new_protected:Npn \__str_lookup_lower:N #1 { \tex_lccode:D `#1 }
    \cs_new_protected:Npn \__str_lookup_upper:N #1 { \tex_uccode:D ‘#1 }
    \cs_new_eq:NN \__str_lookup_fold:N \__str_lookup_lower:N
  }
  {
    \cs_new:Npn \__str_change_case_char_aux:nN #1#2
      { \__str_change_case_output:nw {#2} }
  }
```

(*End definition for* `\str_fold_case:n` *and others. These functions are documented on page 55.*)

\c_ampersand_str
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str

For all of those strings, use `\cs_to_str:N` to get characters with the correct category code without worries

```
\str_const:Nx \c_ampersand_str   { \cs_to_str:N \& }
\str_const:Nx \c_atsign_str      { \cs_to_str:N \@ }
\str_const:Nx \c_backslash_str   { \cs_to_str:N \\ }
\str_const:Nx \c_left_brace_str  { \cs_to_str:N \{ }
\str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
\str_const:Nx \c_circumflex_str  { \cs_to_str:N \^ }
\str_const:Nx \c_colon_str       { \cs_to_str:N \: }
\str_const:Nx \c_dollar_str      { \cs_to_str:N \$ }
\str_const:Nx \c_hash_str        { \cs_to_str:N \# }
\str_const:Nx \c_percent_str     { \cs_to_str:N \% }
\str_const:Nx \c_tilde_str       { \cs_to_str:N \~ }
\str_const:Nx \c_underscore_str  { \cs_to_str:N \_ }
```

(*End definition for* `\c_ampersand_str` *and others. These variables are documented on page 56.*)

\l_tmpa_str
\l_tmpb_str
\g_tmpa_str
\g_tmpb_str

Scratch strings.

```
\str_new:N \l_tmpa_str
\str_new:N \l_tmpb_str
\str_new:N \g_tmpa_str
\str_new:N \g_tmpb_str
```

(*End definition for* `\l_tmpa_str` *and others. These variables are documented on page 56.*)

6.7 Viewing strings

\str_show:n
\str_show:N
\str_show:c

Displays a string on the terminal.

```
\cs_new_eq:NN \str_show:n \tl_show:n
\cs_new_eq:NN \str_show:N \tl_show:N
\cs_generate_variant:Nn \str_show:N { c }
```

(*End definition for* `\str_show:n` *and* `\str_show:N`*. These functions are documented on page 55.*)

6.8 Unicode data for case changing

```
⟨@@=unicode⟩
```

Case changing both for strings and "text" requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

The data required for cross-module manipulations is loaded here: currently this means for `str` and `tl` functions. As such, the prefix used is not `str` but rather `unicode`.

For performance (as the entire data set must be read during each run) and as this code comes somewhat early in the load process, there is quite a bit of low-level code here.

As only the data needs to remain at the end of this process, everything is set up inside a group.

```
\group_begin:
```

A read stream is needed. The I/O module is not yet in place *and* we do not want to use up a stream. We therefore use a known free one in format mode or look for the next free one in package mode (covers plain, LaTeX 2$_\varepsilon$ and ConTeXt MkII and MkIV).

```
⟨*initex⟩
  \tex_chardef:D \g__unicode_data_ior = 0 \scan_stop:
⟨/initex⟩
⟨*package⟩
  \tex_chardef:D \g__unicode_data_ior
    \etex_numexpr:D
      \cs_if_exist:NTF \lastallocatedread
        { \lastallocatedread }
        {
          \cs_if_exist:NTF \c_syst_last_allocated_read
            { \c_syst_last_allocated_read }
            { \tex_count:D 16 ~ }
        }
        + 1
    \scan_stop:
⟨/package⟩
```

Set up to read each file. As they use C-style comments, there is a need to deal with #. At the same time, spaces are important so they need to be picked up as they are important. Beyond that, the current category code scheme works fine. With no I/O loop available, hard-code one that works quickly.

```
  \cs_set_protected:Npn \__unicode_map_inline:n #1
    {
      \group_begin:
        \tex_catcode:D `\# = 12 \scan_stop:
        \tex_catcode:D `\  = 10 \scan_stop:
        \tex_openin:D \g__unicode_data_ior = #1 \scan_stop:
        \cs_if_exist:NT \utex_char:D
          { \__unicode_map_loop: }
        \tex_closein:D \g__unicode_data_ior
      \group_end:
    }
  \cs_set_protected:Npn \__unicode_map_loop:
    {
      \tex_ifeof:D \g__unicode_data_ior
        \exp_after:wN \use_none:n
      \else:
        \exp_after:wN \use:n
      \fi:
        {
          \tex_read:D \g__unicode_data_ior to \l__unicode_tmp_tl
          \if_meaning:w \c_empty_tl \l__unicode_tmp_tl
          \else:
            \exp_after:wN \__unicode_parse:w \l__unicode_tmp_tl \q_stop
          \fi:
          \__unicode_map_loop:
        }
    }
```

The lead-off parser for each line is common for all of the files. If the line starts with a # it's a comment. There's one special comment line to look out for in SpecialCasing.txt as we want to ignore everything after it. As this line does not appear in any other sources and the test is quite quick (there are relatively few comment lines), it can be present in all of the passes.

```
\cs_set_protected:Npn \__unicode_parse:w #1#2 \q_stop
  {
    \reverse_if:N \if:w \c_hash_str #1
      \__unicode_parse_auxi:w #1#2 \q_stop
    \else:
      \if_int_compare:w \__str_if_eq_x:nn
        { \exp_not:n {#2} } { ~Conditional~Mappings~ } = 0 \exp_stop_f:
        \cs_set_protected:Npn \__unicode_parse:w ##1 \q_stop { }
      \fi:
    \fi:
  }
```

Storing each exception is always done in the same way: create a constant token list which expands to exactly the mapping. These have the category codes "now" (so should be letters) but are later detokenized for string use.

```
\cs_set_protected:Npn \__unicode_store:nnnnn #1#2#3#4#5
  {
    \tl_const:cx { c__unicode_ #2 _ \utex_char:D "#1 _tl }
      {
        \utex_char:D "#3 ~
        \utex_char:D "#4 ~
        \tl_if_blank:nF {#5}
          { \utex_char:D "#5 }
      }
  }
```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains are all be covered by the TEX data).

```
\cs_set_protected:Npn \__unicode_parse_auxi:w
  #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
  { \__unicode_parse_auxii:w #1 ; }
\cs_set_protected:Npn \__unicode_parse_auxii:w
  #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
  {
    \tl_if_blank:nF {#7}
      {
        \if_int_compare:w \__str_if_eq_x:nn { #5 ~ } {#7} = 0 \exp_stop_f:
        \else:
          \tl_const:cx
            { c__unicode_mixed_ \utex_char:D "#1 _tl }
            { \utex_char:D "#7 }
        \fi:
      }
  }
\__unicode_map_inline:n { UnicodeData.txt }
```

The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```
\cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
  {
    \if_int_compare:w \__str_if_eq_x:nn {#2} { C } = 0 \exp_stop_f:
      \if_int_compare:w \tex_lccode:D "#1 = "#3 \scan_stop:
      \else:
        \tl_const:cx
          { c__unicode_fold_ \utex_char:D "#1 _tl }
          { \utex_char:D "#3 ~ }
      \fi:
    \else:
      \if_int_compare:w \__str_if_eq_x:nn {#2} { F } = 0 \exp_stop_f:
        \__unicode_parse_auxii:w #1 ~ #3 ~ \q_stop
      \fi:
    \fi:
  }
\cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
  { \__unicode_store:nnnnn {#1} { fold } {#2} {#3} {#4} }
\__unicode_map_inline:n { CaseFolding.txt }
```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider.

```
\cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
  {
    \use:n { \__unicode_parse_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
    \use:n { \__unicode_parse_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
    \if_int_compare:w \__str_if_eq_x:nn {#3} {#4} = 0 \exp_stop_f:
    \else:
      \use:n { \__unicode_parse_auxii:w #1 ~ mixed ~ #3 ~ } ~ \q_stop
    \fi:
  }
\cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
  {
    \tl_if_empty:nF {#4}
      { \__unicode_store:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\__unicode_map_inline:n { SpecialCasing.txt }
```

For the 8-bit engines, the above does nothing but there is some set up needed. There is no expandable character generator primitive so some alternative is needed. As we've not used up hash space for the above, we can go for the fast approach here of one name per letter. Keeping folding and lower casing separate makes the use later a bit easier.

```
\cs_if_exist:NF \utex_char:D
  {
    \cs_set_protected:Npn \__unicode_tmp:NN #1#2
      {
        \if_meaning:w \q_recursion_tail #2
          \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
        \fi:
        \tl_const:cn { c__unicode_fold_  #1 _tl } {#2}
        \tl_const:cn { c__unicode_lower_ #1 _tl } {#2}
        \tl_const:cn { c__unicode_upper_ #2 _tl } {#1}
        \__unicode_tmp:NN
      }
```

```
        \__unicode_tmp:NN
          AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
          ? \q_recursion_tail \q_recursion_stop
      }
```

All done: tidy up.

```
\group_end:
⟨/initex | package⟩
```

7 l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```
⟨*initex | package⟩
⟨@@=seq⟩
```

A sequence is a control sequence whose top-level expansion is of the form "\s__seq __seq_item:n {⟨*item*$_1$⟩} . . . __seq_item:n {⟨*item*$_n$⟩}", with a leading scan mark followed by n items of the same form. An earlier implementation used the structure "\seq_elt:w ⟨*item*$_1$⟩ \seq_elt_end: . . . \seq_elt:w ⟨*item*$_n$⟩ \seq_elt_end:". This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items.

\s__seq The variable is defined in the l3quark module, loaded later.

(*End definition for* \s__seq*.*)

__seq_item:n The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
\cs_new:Npn \__seq_item:n
  {
    \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
    \use_none:n
  }
```

(*End definition for* __seq_item:n*.*)

\l__seq_internal_a_tl
\l__seq_internal_b_tl Scratch space for various internal uses.

```
\tl_new:N \l__seq_internal_a_tl
\tl_new:N \l__seq_internal_b_tl
```

(*End definition for* \l__seq_internal_a_tl *and* \l__seq_internal_b_tl*.*)

__seq_tmp:w Scratch function for internal use.

```
\cs_new_eq:NN \__seq_tmp:w ?
```

(*End definition for* __seq_tmp:w*.*)

\c_empty_seq A sequence with no item, following the structure mentioned above.

```
\tl_const:Nn \c_empty_seq { \s__seq }
```

(*End definition for* \c_empty_seq*. This variable is documented on page 67.*)

7.1 Allocation and initialisation

\seq_new:N
\seq_new:c

Sequences are initialized to \c_empty_seq.

```
\cs_new_protected:Npn \seq_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_seq
  }
\cs_generate_variant:Nn \seq_new:N { c }
```

(*End definition for* \seq_new:N*. This function is documented on page 58.*)

\seq_clear:N
\seq_clear:c
\seq_gclear:N
\seq_gclear:c

Clearing a sequence is similar to setting it equal to the empty one.

```
\cs_new_protected:Npn \seq_clear:N  #1
  { \seq_set_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_clear:N  { c }
\cs_new_protected:Npn \seq_gclear:N #1
  { \seq_gset_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_gclear:N { c }
```

(*End definition for* \seq_clear:N *and* \seq_gclear:N*. These functions are documented on page 58.*)

\seq_clear_new:N
\seq_clear_new:c
\seq_gclear_new:N
\seq_gclear_new:c

Once again we copy code from the token list functions.

```
\cs_new_protected:Npn \seq_clear_new:N  #1
  { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_clear_new:N  { c }
\cs_new_protected:Npn \seq_gclear_new:N #1
  { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(*End definition for* \seq_clear_new:N *and* \seq_gclear_new:N*. These functions are documented on page 58.*)

\seq_set_eq:NN
\seq_set_eq:cN
\seq_set_eq:Nc
\seq_set_eq:cc
\seq_gset_eq:NN
\seq_gset_eq:cN
\seq_gset_eq:Nc
\seq_gset_eq:cc

Copying a sequence is the same as copying the underlying token list.

```
\cs_new_eq:NN \seq_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \seq_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \seq_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \seq_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* \seq_set_eq:NN *and* \seq_gset_eq:NN*. These functions are documented on page 58.*)

\seq_set_from_clist:NN
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

Setting a sequence from a comma-separated list is done using a simple mapping.

```
\cs_new_protected:Npn \seq_set_from_clist:NN #1#2
  {
    \tl_set:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
  {
    \tl_set:Nx #1
```

```
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
  {
    \tl_gset:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
  {
    \tl_gset:Nx #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_generate_variant:Nn \seq_set_from_clist:NN  {     Nc }
\cs_generate_variant:Nn \seq_set_from_clist:NN  { c , cc }
\cs_generate_variant:Nn \seq_set_from_clist:Nn  { c      }
\cs_generate_variant:Nn \seq_gset_from_clist:NN {     Nc }
\cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
\cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(*End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 58.*)

`\seq_set_split:Nnn`
`\seq_set_split:NnV`
`\seq_gset_split:Nnn`
`\seq_gset_split:NnV`
`\__seq_set_split:NNnn`
`\__seq_set_split_auxi:w`
`\__seq_set_split_auxii:w`
`\__seq_set_split_end:`

When the separator is empty, everything is very simple, just map `\__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `\__seq_set_split_auxi:w \prg_do_nothing:` ⟨*item with spaces*⟩ `\__seq_set_split_end:`. Then, x-expansion causes `\__seq_set_split_auxi:w` to trim spaces, and leaves its result as `\__seq_set_split_auxii:w` ⟨*trimmed item*⟩ `\__seq_set_split_end:`. This is then converted to the l3seq internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```
\cs_new_protected:Npn \seq_set_split:Nnn
  { \__seq_set_split:NNnn \tl_set:Nx }
\cs_new_protected:Npn \seq_gset_split:Nnn
  { \__seq_set_split:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
  {
    \tl_if_empty:nTF {#3}
      {
        \tl_set:Nn \l__seq_internal_a_tl
          { \tl_map_function:nN {#4} \__seq_wrap_item:n }
      }
      {
        \tl_set:Nn \l__seq_internal_a_tl
          {
            \__seq_set_split_auxi:w \prg_do_nothing:
            #4
            \__seq_set_split_end:
          }
        \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
          {
```

```
      \__seq_set_split_end:
      \__seq_set_split_auxi:w \prg_do_nothing:
     }
    \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
   }
  #1 #2 { \s__seq \l__seq_internal_a_tl }
 }
\cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
 {
  \exp_not:N \__seq_set_split_auxii:w
  \exp_args:No \tl_trim_spaces:n {#1}
  \exp_not:N \__seq_set_split_end:
 }
\cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
 { \__seq_wrap_item:n {#1} }
\cs_generate_variant:Nn \seq_set_split:Nnn  { NnV }
\cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }
```

(*End definition for* `\seq_set_split:Nnn` *and others. These functions are documented on page 59.*)

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops `f`-expansion.

```
\cs_new_protected:Npn \seq_concat:NNN #1#2#3
 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_generate_variant:Nn \seq_concat:NNN  { ccc }
\cs_generate_variant:Nn \seq_gconcat:NNN { ccc }
```

(*End definition for* `\seq_concat:NNN` *and* `\seq_gconcat:NNN`*. These functions are documented on page 59.*)

\seq_if_exist_p:N
\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
 { TF , T , F , p }
\prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
 { TF , T , F , p }
```

(*End definition for* `\seq_if_exist:NTF`*. This function is documented on page 59.*)

7.2 Appending data to either end

\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:Nv
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:cv
\seq_put_left:co
\seq_put_left:cx
\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:Nv
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:cv
\seq_gput_left:co
\seq_gput_left:cx
__seq_put_left_aux:w

When adding to the left of a sequence, remove `\s__seq`. This is done by `\__seq_put_left_aux:w`, which also stops `f`-expansion.

```
\cs_new_protected:Npn \seq_put_left:Nn #1#2
 {
  \tl_set:Nx #1
   {
    \exp_not:n { \s__seq \__seq_item:n {#2} }
    \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
   }
 }
\cs_new_protected:Npn \seq_gput_left:Nn #1#2
 {
    \tl_gset:Nx #1
      {
        \exp_not:n { \s__seq \__seq_item:n {#2} }
        \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
      }
  }
\cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\cs_generate_variant:Nn \seq_put_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_left:Nn  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_gput_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_left:Nn  { c , cV , cv , co , cx }
```

(*End definition for* `\seq_put_left:Nn`*,* `\seq_gput_left:Nn`*, and* `\__seq_put_left_aux:w`*. These functions are documented on page 59.*)

\seq_put_right:Nn
\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\seq_gput_right:cx

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_put_right:Nn #1#2
  { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\cs_new_protected:Npn \seq_gput_right:Nn #1#2
  { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\cs_generate_variant:Nn \seq_gput_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_put_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

(*End definition for* `\seq_put_right:Nn` *and* `\seq_gput_right:Nn`*. These functions are documented on page 59.*)

7.3 Modifying sequences

__seq_wrap_item:n

This function converts its argument to a proper sequence item in an x-expansion context.

```
\cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(*End definition for* `\__seq_wrap_item:n`*.*)

\l__seq_remove_seq

An internal sequence for the removal routines.

```
\seq_new:N \l__seq_remove_seq
```

(*End definition for* `\l__seq_remove_seq`*.*)

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
__seq_remove_duplicates:NN

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \seq_remove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_set_eq:NN }
\cs_new_protected:Npn \seq_gremove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
\cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
  {
    \seq_clear:N \l__seq_remove_seq
    \seq_map_inline:Nn #2
      {
        \seq_if_in:NnF \l__seq_remove_seq {##1}
          { \seq_put_right:Nn \l__seq_remove_seq {##1} }
      }
    #1 #2 \l__seq_remove_seq
  }
\cs_generate_variant:Nn \seq_remove_duplicates:N  { c }
\cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(*End definition for* `\seq_remove_duplicates:N`*,* `\seq_gremove_duplicates:N`*, and* `\__seq_remove_duplicates:NN`*. These functions are documented on page 62.*)

`\seq_remove_all:Nn`
`\seq_remove_all:cn`
`\seq_gremove_all:Nn`
`\seq_gremove_all:cn`
`\__seq_remove_all_aux:NNn`

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\__seq_pop_right:NNN`, using a "flexible" x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The x-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) ensures that nothing is lost.

```
\cs_new_protected:Npn \seq_remove_all:Nn
  { \__seq_remove_all_aux:NNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gremove_all:Nn
  { \__seq_remove_all_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
  {
    \__seq_push_item_def:n
      {
        \str_if_eq:nnT {##1} {#3}
          {
            \if_false: { \fi: }
            \tl_set:Nn \l__seq_internal_b_tl {##1}
            #1 #2
               { \if_false: } \fi:
                 \exp_not:o {#2}
                 \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
                   { \use_none:nn }
          }
        \__seq_wrap_item:n {##1}
      }
    \tl_set:Nn \l__seq_internal_a_tl {#3}
    #1 #2 {#2}
    \__seq_pop_item_def:
  }
\cs_generate_variant:Nn \seq_remove_all:Nn  { c }
\cs_generate_variant:Nn \seq_gremove_all:Nn { c }
```

(*End definition for* `\seq_remove_all:Nn`*,* `\seq_gremove_all:Nn`*, and* `\__seq_remove_all_aux:NNn`*. These functions are documented on page 62.*)

`\seq_reverse:N`
`\seq_reverse:c`
`\seq_greverse:N`
`\seq_greverse:c`
`\__seq_reverse:NN`
`\__seq_reverse_item:nwn`

Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\exp_stop_f:` marker.

```
\cs_new_protected:Npn \seq_reverse:N #1
  {
    \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
  }
```

```
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
  {
    #2 \exp_stop_f:
    \@@_item:n {#1}
  }
```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, TEX's usual tail recursion does not take place in this case: since the following `\__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, TEX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `\__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TEX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```
\cs_new_protected:Npn \seq_reverse:N
  { \__seq_reverse:NN \tl_set:Nx }
\cs_new_protected:Npn \seq_greverse:N
  { \__seq_reverse:NN \tl_gset:Nx }
\cs_new_protected:Npn \__seq_reverse:NN #1 #2
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
    #1 #2 { #2 \exp_not:n { } }
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
  {
    #2
    \exp_not:n { \__seq_item:n {#1} #3 }
  }
\cs_generate_variant:Nn \seq_reverse:N  { c }
\cs_generate_variant:Nn \seq_greverse:N { c }
```

(*End definition for `\seq_reverse:N` and others. These functions are documented on page 62.*)

\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn

Implemented in l3sort.

(*End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 62.*)

7.4 Sequence conditionals

\seq_if_empty_p:N
\seq_if_empty_p:c
\seq_if_empty:NTF
\seq_if_empty:cTF

Similar to token lists, we compare with the empty sequence.

```
\prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_seq
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

```
\cs_generate_variant:Nn \seq_if_empty_p:N { c }
\cs_generate_variant:Nn \seq_if_empty:NT { c }
\cs_generate_variant:Nn \seq_if_empty:NF { c }
\cs_generate_variant:Nn \seq_if_empty:NTF { c }
```

(*End definition for* `\seq_if_empty:NTF`*. This function is documented on page 62.*)

`\seq_if_in:NnTF`
`\seq_if_in:NVTF`
`\seq_if_in:NvTF`
`\seq_if_in:NoTF`
`\seq_if_in:NxTF`
`\seq_if_in:cnTF`
`\seq_if_in:cVTF`
`\seq_if_in:cvTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`
`\__seq_if_in:`

The approach here is to define `\__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `\__seq_item:n` is preserved in nested situations.

```
\prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
  { T , F , TF }
  {
    \group_begin:
      \tl_set:Nn \l__seq_internal_a_tl {#2}
      \cs_set_protected:Npn \__seq_item:n ##1
        {
          \tl_set:Nn \l__seq_internal_b_tl {##1}
          \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
            \exp_after:wN \__seq_if_in:
          \fi:
        }
      #1
    \group_end:
    \prg_return_false:
    \__prg_break_point:
  }
\cs_new:Npn \__seq_if_in:
  { \__prg_break:n { \group_end: \prg_return_true: } }
\cs_generate_variant:Nn \seq_if_in:NnT  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnT  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnF  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnF  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnTF {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
```

(*End definition for* `\seq_if_in:NnTF` *and* `\__seq_if_in:`*. These functions are documented on page 62.*)

7.5 Recovering data from sequences

`\__seq_pop:NNNN`
`\__seq_pop_TF:NNNN`

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```
\cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      \tl_set:Nn #4 { \q_no_value }
    \else:
      #1#2#3#4
    \fi:
  }
\cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      % \tl_set:Nn #4 { \q_no_value }
      \prg_return_false:
    \else:
      #1#2#3#4
      \prg_return_true:
    \fi:
  }
```

(*End definition for* __seq_pop:NNNN *and* __seq_pop_TF:NNNN*.*)

\seq_get_left:NN
\seq_get_left:cN
__seq_get_left:wnw

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `\__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```
\cs_new_protected:Npn \seq_get_left:NN #1#2
  {
    \tl_set:Nx #2
      {
        \exp_after:wN \__seq_get_left:wnw
        #1 \__seq_item:n { \q_no_value } \q_stop
      }
  }
\cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
  { \exp_not:n {#2} }
\cs_generate_variant:Nn \seq_get_left:NN { c }
```

(*End definition for* \seq_get_left:NN *and* __seq_get_left:wnw*. These functions are documented on page 59.*)

\seq_pop_left:NN
\seq_pop_left:cN
\seq_gpop_left:NN
\seq_gpop_left:cN
__seq_pop_left:NNN
__seq_pop_left:wnwNNN

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```
\cs_new_protected:Npn \seq_pop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
\cs_new_protected:Npn \seq_gpop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
\cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
  { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
\cs_new_protected:Npn \__seq_pop_left:wnwNNN
    #1 \__seq_item:n #2#3 \q_stop #4#5#6
  {
    #4 #5 { #1 #3 }
    \tl_set:Nn #6 {#2}
  }
\cs_generate_variant:Nn \seq_pop_left:NN  { c }
\cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(*End definition for* \seq_pop_left:NN *and others. These functions are documented on page 60.*)

\seq_get_right:NN
\seq_get_right:cN
__seq_get_right_loop:nn

First remove `\s__seq` and prepend `\q_no_value`, then take two arguments at a time. Before the right-hand end of the sequence, this is a brace group followed by `\__seq_item:n`, both removed by `\use_none:nn`. At the end of the sequence, the two question marks are taken by `\use_none:nn`, and the assignment is placed before the right-most item. In the next iteration, `\__seq_get_right_loop:nn` receives two empty arguments, and `\use_none:nn` stops the loop.

```
\cs_new_protected:Npn \seq_get_right:NN #1#2
  {
    \exp_after:wN \use_i_ii:nnn
    \exp_after:wN \__seq_get_right_loop:nn
    \exp_after:wN \q_no_value
    #1
    { ?? \tl_set:Nn #2 }
    { } { }
  }
\cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
  {
    \use_none:nn #2 {#1}
    \__seq_get_right_loop:nn
  }
\cs_generate_variant:Nn \seq_get_right:NN { c }
```

(*End definition for* `\seq_get_right:NN` *and* `\__seq_get_right_loop:nn`. *These functions are documented on page 60.*)

`\seq_pop_right:NN`
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
`\__seq_pop_right:NNN`
`\__seq_pop_right_loop:nn`

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a "flexible length" way to set a token list variable. This is supplied by the `{ \if_false: } \fi:` ... `\if_false: { \fi: }` construct. Using an `x`-type expansion and a "non-expanding" definition for `\__seq_item:n`, the left-most $n-1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```
\cs_new_protected:Npn \seq_pop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
\cs_new_protected:Npn \seq_gpop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \scan_stop:
    #1 #2
      { \if_false: } \fi: \s__seq
        \exp_after:wN \use_i:nnn
        \exp_after:wN \__seq_pop_right_loop:nn
        #2
        {
          \if_false: { \fi: }
          \tl_set:Nx #3
        }
        { } \use_none:nn
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_pop_right_loop:nn #1#2
  {
    #2 { \exp_not:n {#1} }
    \__seq_pop_right_loop:nn
  }
\cs_generate_variant:Nn \seq_pop_right:NN  { c }
\cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(*End definition for* `\seq_pop_right:NN` *and others. These functions are documented on page 60.*)

\seq_get_left:NNTF
\seq_get_left:cNTF
\seq_get_right:NNTF
\seq_get_right:cNTF

Getting from the left or right with a check on the results. The first argument to `\__seq_pop_TF:NNNN` is left unused.

```
\prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
\prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
\cs_generate_variant:Nn \seq_get_left:NNT   { c }
\cs_generate_variant:Nn \seq_get_left:NNF   { c }
\cs_generate_variant:Nn \seq_get_left:NNTF  { c }
\cs_generate_variant:Nn \seq_get_right:NNT  { c }
\cs_generate_variant:Nn \seq_get_right:NNF  { c }
\cs_generate_variant:Nn \seq_get_right:NNTF { c }
```

(*End definition for* `\seq_get_left:NNTF` *and* `\seq_get_right:NNTF`*. These functions are documented on page 61.*)

\seq_pop_left:NNTF
\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

More or less the same for popping.

```
\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
\cs_generate_variant:Nn \seq_pop_left:NNT    { c }
\cs_generate_variant:Nn \seq_pop_left:NNF    { c }
\cs_generate_variant:Nn \seq_pop_left:NNTF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNT   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNTF  { c }
\cs_generate_variant:Nn \seq_pop_right:NNT   { c }
\cs_generate_variant:Nn \seq_pop_right:NNF   { c }
\cs_generate_variant:Nn \seq_pop_right:NNTF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNT  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNTF { c }
```

(*End definition for* `\seq_pop_left:NNTF` *and others. These functions are documented on page 61.*)

\seq_item:Nn
\seq_item:cn
__seq_item:wNn
__seq_item:nN
__seq_item:nnn

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the stop code `{ ? \__prg_break: } { }` is used by the auxiliary, terminating the loop and returning nothing at all.

```
\cs_new:Npn \seq_item:Nn #1
  { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
```

```
\cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
  {
    \exp_args:Nf \__seq_item:nnn
      { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
    #1
    { ? \__prg_break: } { }
    \__prg_break_point:
  }
\cs_new:Npn \__seq_item:nN #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
      {#1}
  }
\cs_new:Npn \__seq_item:nnn #1#2#3
  {
    \use_none:n #2
    \int_compare:nNnTF {#1} = 1
      { \__prg_break:n { \exp_not:n {#3} } }
      { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \seq_item:Nn { c }
```

(*End definition for* `\seq_item:Nn` *and others. These functions are documented on page 60.*)

7.6 Mapping to sequences

\seq_map_break:
\seq_map_break:n

To break a function, the special token `\__prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```
\cs_new:Npn \seq_map_break:
  { \__prg_map_break:Nn \seq_map_break: { } }
\cs_new:Npn \seq_map_break:n
  { \__prg_map_break:Nn \seq_map_break: }
```

(*End definition for* `\seq_map_break:` *and* `\seq_map_break:n`*. These functions are documented on page 63.*)

\seq_map_function:NN
\seq_map_function:cN
__seq_map_function:NNn

The idea here is to apply the code of `#2` to each item in the sequence without altering the definition of `\__seq_item:n`. This is done as by noting that every odd token in the sequence must be `\__seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, `#2` is instead `? \seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```
\cs_new:Npn \seq_map_function:NN #1#2
  {
    \exp_after:wN \use_i_ii:nnn
    \exp_after:wN \__seq_map_function:NNn
    \exp_after:wN #2
    #1
    { ? \seq_map_break: } { }
    \__prg_break_point:Nn \seq_map_break: { }
  }
\cs_new:Npn \__seq_map_function:NNn #1#2#3
  {
    \use_none:n #2
    #1 {#3}
    \__seq_map_function:NNn #1
  }
\cs_generate_variant:Nn \seq_map_function:NN { c }
```

(*End definition for* `\seq_map_function:NN` *and* `\__seq_map_function:NNn`*. These functions are documented on page 62.*)

`\__seq_push_item_def:n`
`\__seq_push_item_def:x`
`\__seq_push_item_def:`
`\__seq_pop_item_def:`

The definition of `\__seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```
\cs_new_protected:Npn \__seq_push_item_def:n
  {
    \__seq_push_item_def:
    \cs_gset:Npn \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:x
  {
    \__seq_push_item_def:
    \cs_gset:Npx \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:
  {
    \int_gincr:N \g__prg_map_int
    \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
      \__seq_item:n
  }
\cs_new_protected:Npn \__seq_pop_item_def:
  {
    \cs_gset_eq:Nc \__seq_item:n
      { __prg_map_ \int_use:N \g__prg_map_int :w }
    \int_gdecr:N \g__prg_map_int
  }
```

(*End definition for* `\__seq_push_item_def:n`*,* `\__seq_push_item_def:`*, and* `\__seq_pop_item_def:`*.*)

`\seq_map_inline:Nn`
`\seq_map_inline:cn`

The idea here is that `\__seq_item:n` is already "applied" to each item in a sequence, and so an in-line mapping is just a case of redefining `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_map_inline:Nn #1#2
  {
    \__seq_push_item_def:n {#2}
    #1
    \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(*End definition for* `\seq_map_inline:Nn`*. This function is documented on page 63.*)

`\seq_map_variable:NNn`
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

```
\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
  {
    \__seq_push_item_def:x
```

```
      {
        \tl_set:Nn \exp_not:N #2 {##1}
        \exp_not:n {#3}
      }
    #1
    \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_variable:NNn {     Nc }
\cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(*End definition for* `\seq_map_variable:NNn`*. This function is documented on page 63.*)

\seq_count:N
\seq_count:c
__seq_count:n

Counting the items in a sequence is done using the same approach as for other count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics.

```
\cs_new:Npn \seq_count:N #1
  {
    \int_eval:n
      {
        0
        \seq_map_function:NN #1 \__seq_count:n
      }
  }
\cs_new:Npn \__seq_count:n #1 { + 1 }
\cs_generate_variant:Nn \seq_count:N { c }
```

(*End definition for* `\seq_count:N` *and* `\__seq_count:n`*. These functions are documented on page 64.*)

7.7 Using sequences

\seq_use:Nnnn
\seq_use:cnnn
__seq_use:NNnNnn
__seq_use_setup:w
__seq_use:nwwwwnwn
__seq_use:nwwn
\seq_use:Nn
\seq_use:cn

See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `\__seq_item:n` as a delimiter rather than commas. We also need to add `\__seq_item:n` at various places, and `\s__seq`.

```
\cs_new:Npn \seq_use:Nnnn #1#2#3#4
  {
    \seq_if_exist:NTF #1
      {
        \int_case:nnF { \seq_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
            { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
          }
          {
            \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
            \q_mark { \__seq_use:nwwwwnwn {#3} }
            \q_mark { \__seq_use:nwwn {#4} }
            \q_stop { }
          }
      }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \seq_use:Nnnn { c }
\cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
\cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwwnwn { } }
\cs_new:Npn \__seq_use:nwwwwnwn
    #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
    \q_mark #6#7 \q_stop #8
  {
    #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
    \q_mark {#6} #7 \q_stop { #8 #1 #2 }
  }
\cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \q_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \seq_use:Nn #1#2
  { \seq_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \seq_use:Nn { c }
```

(*End definition for* `\seq_use:Nnnn` *and others. These functions are documented on page 64.*)

7.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn`
`\seq_push:NV`
`\seq_push:Nv`
`\seq_push:No`
`\seq_push:Nx`
`\seq_push:cn`
`\seq_push:cV`
`\seq_push:cV`
`\seq_push:co`
`\seq_push:cx`
`\seq_gpush:Nn`
`\seq_gpush:NV`
`\seq_gpush:Nv`
`\seq_gpush:No`
`\seq_gpush:Nx`
`\seq_gpush:cn`
`\seq_gpush:cV`
`\seq_gpush:cv`
`\seq_gpush:co`
`\seq_gpush:cx`

Pushing to a sequence is the same as adding on the left.

```
\cs_new_eq:NN \seq_push:Nn  \seq_put_left:Nn
\cs_new_eq:NN \seq_push:NV  \seq_put_left:NV
\cs_new_eq:NN \seq_push:Nv  \seq_put_left:Nv
\cs_new_eq:NN \seq_push:No  \seq_put_left:No
\cs_new_eq:NN \seq_push:Nx  \seq_put_left:Nx
\cs_new_eq:NN \seq_push:cn  \seq_put_left:cn
\cs_new_eq:NN \seq_push:cV  \seq_put_left:cV
\cs_new_eq:NN \seq_push:cv  \seq_put_left:cv
\cs_new_eq:NN \seq_push:co  \seq_put_left:co
\cs_new_eq:NN \seq_push:cx  \seq_put_left:cx
\cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
\cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx
```

(*End definition for* `\seq_push:Nn` *and* `\seq_gpush:Nn`*. These functions are documented on page 66.*)

`\seq_get:NN`
`\seq_get:cN`
`\seq_pop:NN`
`\seq_pop:cN`
`\seq_gpop:NN`
`\seq_gpop:cN`

In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```
\cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
```

(*End definition for* \seq_get:NN*,* \seq_pop:NN*, and* \seq_gpop:NN*. These functions are documented on page 65.*)

\seq_get:NNTF
\seq_get:cNTF
\seq_pop:NNTF
\seq_pop:cNTF
\seq_gpop:NNTF
\seq_gpop:cNTF

More copies.

```
\prg_new_eq_conditional:NNn \seq_get:NN  \seq_get_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_get:cN  \seq_get_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:NN  \seq_pop_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:cN  \seq_pop_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
```

(*End definition for* \seq_get:NNTF*,* \seq_pop:NNTF*, and* \seq_gpop:NNTF*. These functions are documented on page 65.*)

7.9 Viewing sequences

\seq_show:N
\seq_show:c

Apply the general __msg_show_variable:NNNnn.

```
\cs_new_protected:Npn \seq_show:N #1
  {
    \__msg_show_variable:NNNnn #1
      \seq_if_exist:NTF \seq_if_empty:NTF { seq }
      { \seq_map_function:NN #1 \__msg_show_item:n }
  }
\cs_generate_variant:Nn \seq_show:N { c }
```

(*End definition for* \seq_show:N*. This function is documented on page 68.*)

\seq_log:N
\seq_log:c

Redirect output of \seq_show:N to the log.

```
\cs_new_protected:Npn \seq_log:N
  { \__msg_log_next: \seq_show:N }
\cs_generate_variant:Nn \seq_log:N { c }
```

(*End definition for* \seq_log:N*. This function is documented on page 68.*)

7.10 Scratch sequences

\l_tmpa_seq
\l_tmpb_seq
\g_tmpa_seq
\g_tmpb_seq

Temporary comma list variables.

```
\seq_new:N \l_tmpa_seq
\seq_new:N \l_tmpb_seq
\seq_new:N \g_tmpa_seq
\seq_new:N \g_tmpb_seq
```

(*End definition for* \l_tmpa_seq *and others. These variables are documented on page 68.*)

```
⟨/initex | package⟩
```

8 l3int implementation

```
⟨*initex | package⟩
⟨@@=int⟩
```

The following test files are used for this code: m3int001,m3int002,m3int03.

`\c_max_register_int`

Done in l3basics.

(*End definition for* `\c_max_register_int`. *This variable is documented on page 79.*)

`\__int_to_roman:w`
`\if_int_compare:w`

Done in l3basics.

(*End definition for* `\__int_to_roman:w` *and* `\if_int_compare:w`.)

`\or:`

Done in l3basics.

(*End definition for* `\or:`. *This function is documented on page 80.*)

`\__int_value:w`
`\__int_eval:w`
`\__int_eval_end:`
`\if_int_odd:w`
`\if_case:w`

Here are the remaining primitives for number comparisons and expressions.

```
\cs_new_eq:NN \__int_value:w      \tex_number:D
\cs_new_eq:NN \__int_eval:w       \etex_numexpr:D
\cs_new_eq:NN \__int_eval_end:    \tex_relax:D
\cs_new_eq:NN \if_int_odd:w     \tex_ifodd:D
\cs_new_eq:NN \if_case:w        \tex_ifcase:D
```

(*End definition for* `\__int_value:w` *and others.*)

8.1 Integer expressions

`\int_eval:n`

Wrapper for `\__int_eval:w`: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_eval:n } }
\cs_new:Npn \int_eval:n #1
  { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
```

(*End definition for* `\int_eval:n`. *This function is documented on page 69.*)

`\int_abs:n`
`\__int_abs:N`
`\int_max:nn`
`\int_min:nn`
`\__int_maxmin:wwN`

Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_abs:n } }
\cs_new:Npn \int_abs:n #1
  {
    \__int_value:w \exp_after:wN \__int_abs:N
      \__int_value:w \__int_eval:w #1 \__int_eval_end:
    \exp_stop_f:
  }
\cs_new:Npn \__int_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_max:nn }
    { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_max:nn }
  }
```

```
\cs_set:Npn \int_max:nn #1#2
  {
    \__int_value:w \exp_after:wN \__int_maxmin:wwN
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
      >
    \exp_stop_f:
  }
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_min:nn }
    { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_min:nn }
  }
\cs_set:Npn \int_min:nn #1#2
  {
    \__int_value:w \exp_after:wN \__int_maxmin:wwN
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
      <
    \exp_stop_f:
  }
\cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
  {
    \if_int_compare:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
  }
```

(*End definition for* `\int_abs:n` *and others. These functions are documented on page 69.*)

`\int_div_truncate:nn`
`\int_div_round:nn`
`\int_mod:nn`
`\__int_div_truncate:NwNw`
`\__int_mod:ww`

As `\__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (*e.g.*, `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that $0/0$ is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|\texttt{#3#4}|-1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε-TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_div_truncate:nn }
    { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_div_truncate:nn }
  }
\cs_new:Npn \int_div_truncate:nn #1#2
  {
    \__int_value:w \__int_eval:w
      \exp_after:wN \__int_div_truncate:NwNw
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
    \__int_eval_end:
  }
```

```
\cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
  {
    \if_meaning:w 0 #1
      0
    \else:
      (
        #1#2
        \if_meaning:w - #1 + \else: - \fi:
        ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
      )
    \fi:
    / #3#4
  }
```

For the sake of completeness:

```
\cs_new:Npn \int_div_round:nn #1#2
  { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_mod:nn }
    { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_mod:nn }
  }
\cs_new:Npn \int_mod:nn #1#2
  {
    \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
    \__int_eval_end:
  }
\cs_new:Npn \__int_mod:ww #1; #2;
  { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(*End definition for* `\int_div_truncate:nn` *and others. These functions are documented on page 70.*)

8.2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

Two ways to do this: one for the format and one for the LaTeX 2ε package. In plain TeX, `\newcount` (and other allocators) are `\outer`: to allow the code here to work in "generic" mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```
⟨*package⟩
\cs_new_protected:Npn \int_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newcount \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \int_new:N { c }
```

(*End definition for* `\int_new:N`*. This function is documented on page 70.*)

\int_const:Nn
\int_const:cn
__int_constdef:Nw
\c__max_constdef_int

As stated, most constants can be defined as \chardef or \mathchardef but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_const:Nn } }
\cs_new_protected:Npn \int_const:Nn #1#2
  {
    \int_compare:nNnTF {#2} < \c_zero
      {
        \int_new:N #1
        \int_gset:Nn #1 {#2}
      }
      {
        \int_compare:nNnTF {#2} > \c__max_constdef_int
          {
            \int_new:N #1
            \int_gset:Nn #1 {#2}
          }
          {
            \__chk_if_free_cs:N #1
            \tex_global:D \__int_constdef:Nw #1 =
              \__int_eval:w #2 \__int_eval_end:
          }
      }
  }
\cs_generate_variant:Nn \int_const:Nn { c }
\if_int_odd:w 0
  \cs_if_exist:NT \luatex_luatexversion:D  { 1 }
  \cs_if_exist:NT \uptex_disablecjktoken:D
    { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
  \cs_if_exist:NT \xetex_XeTeXversion:D    { 1 } ~
    \cs_if_exist:NTF \uptex_disablecjktoken:D
      { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
      { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
    \__int_constdef:Nw \c__max_constdef_int 1114111 ~
\else:
  \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
  \tex_mathchardef:D \c__max_constdef_int 32767 ~
\fi:
```

(*End definition for \int_const:Nn, __int_constdef:Nw, and \c__max_constdef_int. These functions are documented on page 70.*)

\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c

Functions that reset an ⟨*integer*⟩ register to zero.

```
\cs_new_protected:Npn \int_zero:N  #1 { #1 = \c_zero }
\cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\cs_generate_variant:Nn \int_zero:N  { c }
\cs_generate_variant:Nn \int_gzero:N { c }
```

(*End definition for \int_zero:N and \int_gzero:N. These functions are documented on page 70.*)

\int_zero_new:N
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \int_zero_new:N  #1
  { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
```

```
\cs_new_protected:Npn \int_gzero_new:N #1
  { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
\cs_generate_variant:Nn \int_zero_new:N  { c }
\cs_generate_variant:Nn \int_gzero_new:N { c }
```

(*End definition for* \int_zero_new:N *and* \int_gzero_new:N*. These functions are documented on page 70.*)

\int_set_eq:NN
\int_set_eq:cN
\int_set_eq:Nc
\int_set_eq:cc
\int_gset_eq:NN
\int_gset_eq:cN
\int_gset_eq:Nc
\int_gset_eq:cc

Setting equal means using one integer inside the set function of another.

```
\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \int_set_eq:NN {       c }
\cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \int_gset_eq:NN {       c }
\cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
```

(*End definition for* \int_set_eq:NN *and* \int_gset_eq:NN*. These functions are documented on page 70.*)

\int_if_exist_p:N
\int_if_exist_p:c
\int_if_exist:NTF
\int_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* \int_if_exist:NTF*. This function is documented on page 71.*)

8.3 Setting and incrementing integers

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Adding and subtracting to and from a counter …

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_add:Nn } }
\cs_new_protected:Npn \int_add:Nn #1#2
  { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_sub:Nn } }
\cs_new_protected:Npn \int_sub:Nn #1#2
  { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gadd:Nn
  { \tex_global:D \int_add:Nn }
\cs_new_protected:Npn \int_gsub:Nn
  { \tex_global:D \int_sub:Nn }
\cs_generate_variant:Nn \int_add:Nn  { c }
\cs_generate_variant:Nn \int_gadd:Nn { c }
\cs_generate_variant:Nn \int_sub:Nn  { c }
\cs_generate_variant:Nn \int_gsub:Nn { c }
```

(*End definition for* \int_add:Nn *and others. These functions are documented on page 71.*)

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

```
\cs_new_protected:Npn \int_incr:N #1
  { \tex_advance:D #1 \c_one }
\cs_new_protected:Npn \int_decr:N #1
  { \tex_advance:D #1 - \c_one }
\cs_new_protected:Npn \int_gincr:N
```

```
  { \tex_global:D \int_incr:N }
\cs_new_protected:Npn \int_gdecr:N
  { \tex_global:D \int_decr:N }
\cs_generate_variant:Nn \int_incr:N  { c }
\cs_generate_variant:Nn \int_decr:N  { c }
\cs_generate_variant:Nn \int_gincr:N { c }
\cs_generate_variant:Nn \int_gdecr:N { c }
```

(*End definition for* `\int_incr:N` *and others. These functions are documented on page 71.*)

\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn

As integers are register-based TeX issues an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_set:Nn } }
\cs_new_protected:Npn \int_set:Nn #1#2
  { #1 ~ \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
\cs_generate_variant:Nn \int_set:Nn  { c }
\cs_generate_variant:Nn \int_gset:Nn { c }
```

(*End definition for* `\int_set:Nn` *and* `\int_gset:Nn`*. These functions are documented on page 71.*)

8.4 Using integers

\int_use:N
\int_use:c

Here is how counters are accessed:

```
\cs_new_eq:NN \int_use:N \tex_the:D
```

We hand-code this for some speed gain:

```
%\cs_generate_variant:Nn \int_use:N { c }
\cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* `\int_use:N`*. This function is documented on page 71.*)

8.5 Integer expression conditionals

__prg_compare_error:
__prg_compare_error:Nw

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `\__prg_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `\__prg_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
\cs_new_protected:Npn \__prg_compare_error:
  {
    \if_int_compare:w \c_zero \c_zero \fi:
    =
    \__prg_compare_error:
  }
\cs_new:Npn \__prg_compare_error:Nw
    #1#2 \q_stop
  {
    { }
    \c_zero \fi:
```

```
    \__msg_kernel_expandable_error:nnn
      { kernel } { unknown-comparison } {#1}
    \prg_return_false:
  }
```

(*End definition for* `\__prg_compare_error:` *and* `\__prg_compare_error:Nw`.)

\int_compare_p:n
\int_compare:nTF
__int_compare:w
__int_compare:Nw
__int_compare:NNw
__int_compare:nnN
__int_compare_end_=:NNw
__int_compare_=:NNw
__int_compare_<:NNw
__int_compare_>:NNw
__int_compare_==:NNw
__int_compare_!=:NNw
__int_compare_<=:NNw
__int_compare_>=:NNw

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `\__int_compare:Nw` reads one ⟨*operand*⟩ and one ⟨*comparison*⟩ symbol, and leaves roughly

```
⟨operand⟩ \prg_return_false: \fi:
\reverse_if:N \if_int_compare:w ⟨operand⟩ ⟨comparison⟩
\__int_compare:Nw
```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the ⟨*comparisons*⟩ is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\__int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let TeX evaluate this left hand side of the (in)equality using `\__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `\__prg_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `\__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\q_stop` used to grab the entire argument when necessary.

```
\prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__int_compare:w
    \__int_value:w \__int_eval:w #1 \__prg_compare_error:
  }
\cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
  {
    \exp_after:wN \if_false: \__int_value:w
      \__int_compare:Nw #1 e { = nd_ } \q_stop
  }
```

The goal here is to find an ⟨*operand*⟩ and a ⟨*comparison*⟩. The ⟨*operand*⟩ is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `\__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `\__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `\__prg_compare_error:Nw` raises an error.

```
\cs_new:Npn \__int_compare:Nw #1#2 \q_stop
```

```
  {
    \exp_after:wN \__int_compare:NNw
      \__int_to_roman:w - 0 #2 \q_mark
    #1#2 \q_stop
  }
\cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
  {
    \etex_unexpanded:D
    \use:c
      {
        __int_compare_ \token_to_str:N #1
        \if_meaning:w = #2 =  \fi:
        :NNw
      }
      \__prg_compare_error:Nw #1
  }
```

When the last ⟨*operand*⟩ is seen, __int_compare:NNw receives e and =nd_ as arguments, hence calling __int_compare_end_=:NNw to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls __int_compare:nnN where #1 is \if_int_compare:w or \reverse_if:N \if_int_compare:w, #2 is the ⟨*operand*⟩, and #3 is one of <, =, or >. As announced earlier, we leave the ⟨*operand*⟩ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return false. Otherwise, we apply the conditional #1 to the ⟨*operand*⟩ #2 and the comparison #3, and call __int_compare:Nw to look for additional operands, after evaluating the following expression.

```
\cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
  {
    {#3} \exp_stop_f:
    \prg_return_false: \else: \prg_return_true: \fi:
  }
\cs_new:Npn \__int_compare:nnN #1#2#3
  {
        {#2} \exp_stop_f:
      \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
  }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding __prg_compare_error:Nw ⟨*token*⟩ responsible for error detection.

```
\cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
\cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
\cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
  { \__int_compare:nnN { \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
  { \__int_compare:nnN { \if_int_compare:w } {#3} > }
\cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
  { \__int_compare:nnN { \if_int_compare:w } {#3} < }
```

(*End definition for* `\int_compare:nTF` *and others. These functions are documented on page 72.*)

`\int_compare_p:nNn`
`\int_compare:nNnTF`

More efficient but less natural in typing.

```
\__debug_patch_conditional_args:nNNpnn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_compare:nNn }
    { \__int_eval_end: #2 }
    { \__debug_chk_expr:nNnN {#3} \__int_eval:w { } \int_compare:nNn }
  }
\prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\int_compare:nNnTF`*. This function is documented on page 72.*)

`\int_case:nn`
`\int_case:nnTF`
`\__int_case:nnTF`
`\__int_case:nw`
`\__int_case_end:nw`

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nn(TF)` as described in l3basics.

```
\cs_new:Npn \int_case:nnTF #1
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
  }
\cs_new:Npn \int_case:nnT #1#2#3
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \int_case:nnF #1#2
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \int_case:nn #1#2
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__int_case:nnTF #1#2#3#4
  { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__int_case:nw #1#2#3
  {
    \int_compare:nNnTF {#1} = {#2}
      { \__int_case_end:nw {#3} }
      { \__int_case:nw {#1} }
  }
\cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\int_case:nnTF` *and others. These functions are documented on page 73.*)

\int_if_odd_p:n
\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF

A predicate function.

```
\__debug_patch_conditional_args:nNNpnn
  { { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_odd:n } }
\prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\__debug_patch_conditional_args:nNNpnn
  { { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_even:n } }
\prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\int_if_odd:nTF` *and* `\int_if_even:nTF`*. These functions are documented on page 73.*)

8.6 Integer expression loops

\int_while_do:nn
\int_until_do:nn
\int_do_while:nn
\int_do_until:nn

These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```
\cs_new:Npn \int_while_do:nn #1#2
  {
    \int_compare:nT {#1}
      {
        #2
        \int_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_until_do:nn #1#2
  {
    \int_compare:nF {#1}
      {
        #2
        \int_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_do_while:nn #1#2
  {
    #2
    \int_compare:nT {#1}
      { \int_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \int_do_until:nn #1#2
```

```
  {
    #2
    \int_compare:nF {#1}
      { \int_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\int_while_do:nn` *and others. These functions are documented on page 74.*)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```
\cs_new:Npn \int_while_do:nNnn #1#2#3#4
  {
    \int_compare:nNnT {#1} #2 {#3}
      {
        #4
        \int_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_until_do:nNnn #1#2#3#4
  {
    \int_compare:nNnF {#1} #2 {#3}
      {
        #4
        \int_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_do_while:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnT {#1} #2 {#3}
      { \int_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \int_do_until:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnF {#1} #2 {#3}
      { \int_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\int_while_do:nNnn` *and others. These functions are documented on page 74.*)

8.7 Integer step functions

`\int_step_function:nnnN`
`\__int_step:wwwN`
`\__int_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__int_eval:w { } \int_step_function:nnnN }
    { \__debug_chk_expr:nNnN {#2} \__int_eval:w { } \int_step_function:nnnN }
    { \__debug_chk_expr:nNnN {#3} \__int_eval:w { } \int_step_function:nnnN }
  }
\cs_new:Npn \int_step_function:nnnN #1#2#3
```

```
  {
    \exp_after:wN \__int_step:wwwN
    \__int_value:w \__int_eval:w #1 \exp_after:wN ;
    \__int_value:w \__int_eval:w #2 \exp_after:wN ;
    \__int_value:w \__int_eval:w #3 ;
  }
\cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
  {
    \int_compare:nNnTF {#2} > \c_zero
      { \__int_step:NnnnN > }
      {
        \int_compare:nNnTF {#2} = \c_zero
          {
            \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
            \use_none:nnnn
          }
          { \__int_step:NnnnN < }
      }
      {#1} {#2} {#3} #4
  }
\cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
  {
    \int_compare:nNnF {#2} #1 {#4}
      {
        #5 {#2}
        \exp_args:NNf \__int_step:NnnnN
          #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
      }
  }
```

(*End definition for* `\int_step_function:nnnN`*,* `\__int_step:wwwN`*, and* `\__int_step:NnnnN`*. These functions are documented on page 75.*)

`\int_step_inline:nnnn`
`\int_step_variable:nnnNn`
`\__int_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\__prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```
\cs_new_protected:Npn \int_step_inline:nnnn
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_protected:Npn
      { __prg_map_ \int_use:N \g__prg_map_int :w }
  }
\cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_protected:Npx
      { __prg_map_ \int_use:N \g__prg_map_int :w }
      {#1}{#2}{#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \int_step_function:nnnN {#3} {#4} {#5} #2
    \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
  }
```

(*End definition for* `\int_step_inline:nnnn`*,* `\int_step_variable:nnnNn`*, and* `\__int_step:NNnnnn`*. These functions are documented on page 75.*)

8.8 Formatting integers

`\int_to_arabic:n`

Nothing exciting here.

```
\cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(*End definition for* `\int_to_arabic:n`*. This function is documented on page 75.*)

`\int_to_symbols:nnn`
`\__int_to_symbols:nnnn`

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
\cs_new:Npn \int_to_symbols:nnn #1#2#3
  {
    \int_compare:nNnTF {#1} > {#2}
      {
        \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
          {
            \int_case:nn
              { 1 + \int_mod:nn { #1 - 1 } {#2} }
              {#3}
          }
          {#1} {#2} {#3}
      }
      { \int_case:nn {#1} {#3} }
  }
\cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
  {
    \exp_args:Nf \int_to_symbols:nnn
      { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
    #1
  }
```

(*End definition for* `\int_to_symbols:nnn` *and* `\__int_to_symbols:nnnn`*. These functions are documented on page 76.*)

`\int_to_alph:n`
`\int_to_Alph:n`

These both use the above function with input functions that make sense for the alphabet in English.

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        {  3 } { c }
        {  4 } { d }
        {  5 } { e }
        {  6 } { f }
        {  7 } { g }
        {  8 } { h }
        {  9 } { i }
        { 10 } { j }
        { 11 } { k }
        { 12 } { l }
        { 13 } { m }
        { 14 } { n }
        { 15 } { o }
        { 16 } { p }
        { 17 } { q }
        { 18 } { r }
        { 19 } { s }
        { 20 } { t }
        { 21 } { u }
        { 22 } { v }
        { 23 } { w }
        { 24 } { x }
        { 25 } { y }
        { 26 } { z }
      }
  }
\cs_new:Npn \int_to_Alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { A }
        {  2 } { B }
        {  3 } { C }
        {  4 } { D }
        {  5 } { E }
        {  6 } { F }
        {  7 } { G }
        {  8 } { H }
        {  9 } { I }
        { 10 } { J }
        { 11 } { K }
        { 12 } { L }
        { 13 } { M }
        { 14 } { N }
        { 15 } { O }
        { 16 } { P }
        { 17 } { Q }
        { 18 } { R }
        { 19 } { S }
        { 20 } { T }
        { 21 } { U }
        { 22 } { V }
        { 23 } { W }
        { 24 } { X }
        { 25 } { Y }
        { 26 } { Z }
      }
  }
```

(*End definition for* `\int_to_alph:n` *and* `\int_to_Alph:n`*. These functions are documented on page 76.*)

`\int_to_base:nn`
`\int_to_Base:nn`
`\__int_to_base:nn`
`\__int_to_Base:nn`
`\__int_to_base:nnN`
`\__int_to_Base:nnN`
`\__int_to_base:nnnN`
`\__int_to_Base:nnnN`
`\__int_to_letter:n`
`\__int_to_Letter:n`

Converting from base ten (`#1`) to a second base (`#2`) starts with computing `#1`: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either `-` or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

```
\cs_new:Npn \int_to_base:nn #1
  { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\cs_new:Npn \int_to_Base:nn #1
  { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\cs_new:Npn \__int_to_base:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
  }
\cs_new:Npn \__int_to_Base:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
  }
```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in `#1` is checked to see if it is less than the new base (`#2`). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```
\cs_new:Npn \__int_to_base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_base:nnnN
          { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_base:nnN
```

```
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
\cs_new:Npn \__int_to_Base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_Base:nnnN
          { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_Base:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```
\cs_new:Npn \__int_to_letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
         a
    \or: b
    \or: c
    \or: d
    \or: e
    \or: f
    \or: g
    \or: h
    \or: i
    \or: j
    \or: k
    \or: l
    \or: m
    \or: n
    \or: o
    \or: p
    \or: q
    \or: r
    \or: s
    \or: t
    \or: u
    \or: v
    \or: w
    \or: x
    \or: y
    \or: z
    \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
\cs_new:Npn \__int_to_Letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
         A
    \or: B
    \or: C
    \or: D
    \or: E
    \or: F
    \or: G
    \or: H
    \or: I
    \or: J
    \or: K
    \or: L
    \or: M
    \or: N
    \or: O
    \or: P
    \or: Q
    \or: R
    \or: S
    \or: T
    \or: U
    \or: V
    \or: W
    \or: X
    \or: Y
    \or: Z
    \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
```

(*End definition for* `\int_to_base:nn` *and others. These functions are documented on page 77.*)

`\int_to_bin:n`
`\int_to_hex:n`
`\int_to_Hex:n`
`\int_to_oct:n`

Wrappers around the generic function.

```
\cs_new:Npn \int_to_bin:n #1
  { \int_to_base:nn {#1} { 2 } }
\cs_new:Npn \int_to_hex:n #1
  { \int_to_base:nn {#1} { 16 } }
\cs_new:Npn \int_to_Hex:n #1
  { \int_to_Base:nn {#1} { 16 } }
```

```
\cs_new:Npn \int_to_oct:n #1
  { \int_to_base:nn {#1} { 8 } }
```

(*End definition for* `\int_to_bin:n` *and others. These functions are documented on page 76.*)

`\int_to_roman:n`
`\int_to_Roman:n`
`\__int_to_roman:N`
`\__int_to_roman:N`
`\__int_to_roman_i:w`
`\__int_to_roman_v:w`
`\__int_to_roman_x:w`
`\__int_to_roman_l:w`
`\__int_to_roman_c:w`
`\__int_to_roman_d:w`
`\__int_to_roman_m:w`
`\__int_to_roman_Q:w`
`\__int_to_Roman_i:w`
`\__int_to_Roman_v:w`
`\__int_to_Roman_x:w`
`\__int_to_Roman_l:w`
`\__int_to_Roman_c:w`
`\__int_to_Roman_d:w`
`\__int_to_Roman_m:w`
`\__int_to_Roman_Q:w`

The `\__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the `Q`.

```
\cs_new:Npn \int_to_roman:n #1
  {
    \exp_after:wN \__int_to_roman:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_roman:N #1
  {
    \use:c { __int_to_roman_ #1 :w }
    \__int_to_roman:N
  }
\cs_new:Npn \int_to_Roman:n #1
  {
    \exp_after:wN \__int_to_Roman_aux:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_Roman_aux:N #1
  {
    \use:c { __int_to_Roman_ #1 :w }
    \__int_to_Roman_aux:N
  }
\cs_new:Npn \__int_to_roman_i:w { i }
\cs_new:Npn \__int_to_roman_v:w { v }
\cs_new:Npn \__int_to_roman_x:w { x }
\cs_new:Npn \__int_to_roman_l:w { l }
\cs_new:Npn \__int_to_roman_c:w { c }
\cs_new:Npn \__int_to_roman_d:w { d }
\cs_new:Npn \__int_to_roman_m:w { m }
\cs_new:Npn \__int_to_roman_Q:w #1 { }
\cs_new:Npn \__int_to_Roman_i:w { I }
\cs_new:Npn \__int_to_Roman_v:w { V }
\cs_new:Npn \__int_to_Roman_x:w { X }
\cs_new:Npn \__int_to_Roman_l:w { L }
\cs_new:Npn \__int_to_Roman_c:w { C }
\cs_new:Npn \__int_to_Roman_d:w { D }
\cs_new:Npn \__int_to_Roman_m:w { M }
\cs_new:Npn \__int_to_Roman_Q:w #1 { }
```

(*End definition for* `\int_to_roman:n` *and others. These functions are documented on page 77.*)

8.9 Converting from other formats to integers

`\__int_pass_signs:wn`
`\__int_pass_signs_end:wn`

Called as `\__int_pass_signs:wn` ⟨*signs and digits*⟩ `\q_stop {`⟨*code*⟩`}`, this function leaves in the input stream any sign it finds, then inserts the ⟨*code*⟩ before the first non-sign token (and removes `\q_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```
\cs_new:Npn \__int_pass_signs:wn #1
  {
    \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
      \exp_after:wN \__int_pass_signs:wn
    \else:
      \exp_after:wN \__int_pass_signs_end:wn
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__int_pass_signs_end:wn #1 \q_stop #2 { #2 #1 }
```

(*End definition for `\__int_pass_signs:wn` and `\__int_pass_signs_end:wn`.*)

\int_from_alph:n
__int_from_alph:nN
__int_from_alph:N

First take care of signs then loop through the input using the recursion quarks. The __int_from_alph:nN auxiliary collects in its first argument the value obtained so far, and the auxiliary __int_from_alph:N converts one letter to an expression which evaluates to the correct number.

```
\cs_new:Npn \int_from_alph:n #1
  {
    \int_eval:n
      {
        \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
          \q_stop { \__int_from_alph:nN { 0 } }
        \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new:Npn \__int_from_alph:nN #1#2
  {
    \quark_if_recursion_tail_stop_do:Nn #2 {#1}
    \exp_args:Nf \__int_from_alph:nN
      { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
  }
\cs_new:Npn \__int_from_alph:N #1
  { ‘#1 - \int_compare:nNnTF { ‘#1 } < { 91 } { 64 } { 96 } }
```

(*End definition for `\int_from_alph:n`, `\__int_from_alph:nN`, and `\__int_from_alph:N`. These functions are documented on page 77.*)

\int_from_base:nn
__int_from_base:nnN
__int_from_base:N

Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of __int_from_base:nnN. To convert a single character, __int_from_base:N checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use \int_eval:n, hence is not safe for general use.

```
\cs_new:Npn \int_from_base:nn #1#2
  {
    \int_eval:n
      {
        \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
          \q_stop { \__int_from_base:nnN { 0 } {#2} }
        \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new:Npn \__int_from_base:nnN #1#2#3
  {
    \quark_if_recursion_tail_stop_do:Nn #3 {#1}
    \exp_args:Nf \__int_from_base:nnN
      { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
      {#2}
  }
\cs_new:Npn \__int_from_base:N #1
  {
    \int_compare:nNnTF { `#1 } < { 58 }
      {#1}
      { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 55 } { 87 } }
  }
```

(*End definition for* `\int_from_base:nn`*,* `\__int_from_base:nnN`*, and* `\__int_from_base:N`*. These functions are documented on page 78.*)

\int_from_bin:n
\int_from_hex:n
\int_from_oct:n

Wrappers around the generic function.

```
\cs_new:Npn \int_from_bin:n #1
  { \int_from_base:nn {#1} { 2 } }
\cs_new:Npn \int_from_hex:n #1
  { \int_from_base:nn {#1} { 16 } }
\cs_new:Npn \int_from_oct:n #1
  { \int_from_base:nn {#1} { 8 } }
```

(*End definition for* `\int_from_bin:n`*,* `\int_from_hex:n`*, and* `\int_from_oct:n`*. These functions are documented on page 77.*)

\c__int_from_roman_i_int
\c__int_from_roman_v_int
\c__int_from_roman_x_int
\c__int_from_roman_l_int
\c__int_from_roman_c_int
\c__int_from_roman_d_int
\c__int_from_roman_m_int
\c__int_from_roman_I_int
\c__int_from_roman_V_int
\c__int_from_roman_X_int
\c__int_from_roman_L_int
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

Constants used to convert from Roman numerals to integers.

```
\int_const:cn { c__int_from_roman_i_int } { 1 }
\int_const:cn { c__int_from_roman_v_int } { 5 }
\int_const:cn { c__int_from_roman_x_int } { 10 }
\int_const:cn { c__int_from_roman_l_int } { 50 }
\int_const:cn { c__int_from_roman_c_int } { 100 }
\int_const:cn { c__int_from_roman_d_int } { 500 }
\int_const:cn { c__int_from_roman_m_int } { 1000 }
\int_const:cn { c__int_from_roman_I_int } { 1 }
\int_const:cn { c__int_from_roman_V_int } { 5 }
\int_const:cn { c__int_from_roman_X_int } { 10 }
\int_const:cn { c__int_from_roman_L_int } { 50 }
\int_const:cn { c__int_from_roman_C_int } { 100 }
\int_const:cn { c__int_from_roman_D_int } { 500 }
\int_const:cn { c__int_from_roman_M_int } { 1000 }
```

(*End definition for* `\c__int_from_roman_i_int` *and others.*)

\int_from_roman:n
__int_from_roman:NN
__int_from_roman_error:w

The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by TEX. If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by -1.

```
\cs_new:Npn \int_from_roman:n #1
  {
    \int_eval:n
      {
        (
          0
          \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
          \q_recursion_tail \q_recursion_tail \q_recursion_stop
        )
      }
  }
\cs_new:Npn \__int_from_roman:NN #1#2
  {
    \quark_if_recursion_tail_stop:N #1
    \int_if_exist:cF { c__int_from_roman_ #1 _int }
      { \__int_from_roman_error:w }
    \quark_if_recursion_tail_stop_do:Nn #2
      { + \use:c { c__int_from_roman_ #1 _int } }
    \int_if_exist:cF { c__int_from_roman_ #2 _int }
      { \__int_from_roman_error:w }
    \int_compare:nNnTF
      { \use:c { c__int_from_roman_ #1 _int } }
      <
      { \use:c { c__int_from_roman_ #2 _int } }
      {
        + \use:c { c__int_from_roman_ #2 _int }
        - \use:c { c__int_from_roman_ #1 _int }
        \__int_from_roman:NN
      }
      {
        + \use:c { c__int_from_roman_ #1 _int }
        \__int_from_roman:NN #2
      }
  }
\cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
  { #2 * 0 - 1 }
```

(*End definition for* `\int_from_roman:n`*,* `\__int_from_roman:NN`*, and* `\__int_from_roman_error:w`*. These functions are documented on page 78.*)

8.10 Viewing integer

\int_show:N
\int_show:c
__int_show:nN

Diagnostics.

```
\cs_new_eq:NN \int_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \int_show:N { c }
```

(*End definition for* `\int_show:N` *and* `\__int_show:nN`*. These functions are documented on page 78.*)

\int_show:n

We don't use the TEX primitive `\showthe` to show integer expressions: this gives a more unified output.

```
\cs_new_protected:Npn \int_show:n
  { \__msg_show_wrap:Nn \int_eval:n }
```

(*End definition for* `\int_show:n`*. This function is documented on page 78.*)

\int_log:N
\int_log:c

Diagnostics.

```
\cs_new_eq:NN \int_log:N \__kernel_register_log:N
\cs_generate_variant:Nn \int_log:N { c }
```

(*End definition for* `\int_log:N`*. This function is documented on page 78.*)

\int_log:n

Redirect output of \int_show:n to the log.

```
\cs_new_protected:Npn \int_log:n
  { \__msg_log_next: \int_show:n }
```

(*End definition for* \int_log:n. *This function is documented on page 78.*)

8.11 Constant integers

\c_zero

Again, in l3basics

(*End definition for* \c_zero. *This variable is documented on page 79.*)

\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen

Low-number values not previously defined.

```
\int_const:Nn \c_one      {  1 }
\int_const:Nn \c_two      {  2 }
\int_const:Nn \c_three    {  3 }
\int_const:Nn \c_four     {  4 }
\int_const:Nn \c_five     {  5 }
\int_const:Nn \c_six      {  6 }
\int_const:Nn \c_seven    {  7 }
\int_const:Nn \c_eight    {  8 }
\int_const:Nn \c_nine     {  9 }
\int_const:Nn \c_ten      { 10 }
\int_const:Nn \c_eleven   { 11 }
\int_const:Nn \c_twelve   { 12 }
\int_const:Nn \c_thirteen { 13 }
\int_const:Nn \c_fourteen { 14 }
\int_const:Nn \c_fifteen  { 15 }
\int_const:Nn \c_sixteen  { 16 }
```

(*End definition for* \c_one *and others. These variables are documented on page 79.*)

\c_thirty_two

One middling value.

```
\int_const:Nn \c_thirty_two { 32 }
```

(*End definition for* \c_thirty_two. *This variable is documented on page 79.*)

\c_two_hundred_fifty_five
\c_two_hundred_fifty_six

Two classic mid-range integer constants.

```
\int_const:Nn \c_two_hundred_fifty_five { 255 }
\int_const:Nn \c_two_hundred_fifty_six  { 256 }
```

(*End definition for* \c_two_hundred_fifty_five *and* \c_two_hundred_fifty_six. *These variables are documented on page 79.*)

\c_one_hundred
\c_one_thousand
\c_ten_thousand

Simple runs of powers of ten.

```
\int_const:Nn \c_one_hundred  {   100 }
\int_const:Nn \c_one_thousand {  1000 }
\int_const:Nn \c_ten_thousand { 10000 }
```

(*End definition for* \c_one_hundred*,* \c_one_thousand*, and* \c_ten_thousand. *These variables are documented on page 79.*)

\c_max_int

The largest number allowed is $2^{31} - 1$

```
\int_const:Nn \c_max_int { 2 147 483 647 }
```

(*End definition for* \c_max_int. *This variable is documented on page 79.*)

\c_max_char_int

The largest character code is 1114111 (hexadecimal 10FFFF) in XƎTEX and LuaTEX and 255 in other engines. In many places pTEX and upTEX support larger character codes but for instance the values of \lccode are restricted to $[0, 255]$.

```
\int_const:Nn \c_max_char_int
  {
    \if_int_odd:w 0
      \cs_if_exist:NT \luatex_luatexversion:D  { 1 }
      \cs_if_exist:NT \xetex_XeTeXversion:D    { 1 } ~
      "10FFFF
    \else:
      "FF
    \fi:
  }
```

(*End definition for* \c_max_char_int*. This variable is documented on page 79.*)

8.12 Scratch integers

\l_tmpa_int
\l_tmpb_int
\g_tmpa_int
\g_tmpb_int

We provide two local and two global scratch counters, maybe we need more or less.

```
\int_new:N \l_tmpa_int
\int_new:N \l_tmpb_int
\int_new:N \g_tmpa_int
\int_new:N \g_tmpb_int
```

(*End definition for* \l_tmpa_int *and others. These variables are documented on page 79.*)

8.13 Deprecated

\c_minus_one

The actual allocation mechanism is in l3alloc; it requires \c_one to be defined. In package mode, reuse \m@ne. We also store in two global token lists some code for \debug_deprecation_on: and \debug_deprecation_off:. For the latter, we need to locally set \c_minus_one back to the constant hence use a private name. We use \tex_let:D directly because \c_minus_one (as all deprecated commands) is made outer by \debug_deprecation_on:.

```
⟨package⟩\cs_gset_eq:NN \c__deprecation_minus_one \m@ne
⟨initex⟩\int_const:Nn \c__deprecation_minus_one { -1 }
\cs_new_eq:NN \c_minus_one \c__deprecation_minus_one
\__debug:TF
  {
    \tl_gput_right:Nn \g__debug_deprecation_on_tl
      { \__deprecation_error:Nnn \c_minus_one { -1 } { 2018-12-31 } }
    \tl_gput_right:Nn \g__debug_deprecation_off_tl
      { \tex_let:D \c_minus_one \c__deprecation_minus_one }
  }
  { }
```

(*End definition for* \c_minus_one*.*)

```
⟨/initex | package⟩
```

9 l3intarray implementation

```
⟨*initex | package⟩
⟨@@=intarray⟩
```

9.1 Allocating arrays

\g__intarray_font_int

Used to assign one font per array.

```
\int_new:N \g__intarray_font_int
```

(*End definition for* \g__intarray_font_int*.*)

__intarray_new:Nn

Declare #1 to be a font (arbitrarily cmr10 at a never-used size). Store the array's size as the \hyphenchar of that font and make sure enough \fontdimen are allocated, by setting the last one. Then clear any \fontdimen that cmr10 starts with. It seems LuaTeX's cmr10 has an extra \fontdimen parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such).

```
\cs_new_protected:Npn \__intarray_new:Nn #1#2
  {
    \__chk_if_free_cs:N #1
    \int_gincr:N \g__intarray_font_int
    \tex_global:D \tex_font:D #1 = cmr10~at~ \g__intarray_font_int sp \scan_stop:
    \tex_hyphenchar:D #1 = \int_eval:n {#2} \scan_stop:
    \int_compare:nNnT { \tex_hyphenchar:D #1 } > 0
      { \tex_fontdimen:D \tex_hyphenchar:D #1 #1 = 0 sp \scan_stop: }
    \int_step_inline:nnnn { 1 } { 1 } { 8 }
      { \tex_fontdimen:D ##1 #1 = 0 sp \scan_stop: }
  }
```

(*End definition for* __intarray_new:Nn*.*)

__intarray_count:N

Size of an array.

```
\cs_new:Npn \__intarray_count:N #1 { \tex_the:D \tex_hyphenchar:D #1 }
```

(*End definition for* __intarray_count:N*.*)

9.2 Array items

__intarray_gset:Nnn
__intarray_gset_fast:Nnn
__intarray_gset_aux:Nnn

Set the appropriate \fontdimen. The slow version checks the position and value are within bounds.

```
\cs_new_protected:Npn \__intarray_gset_fast:Nnn #1#2#3
  { \tex_fontdimen:D \int_eval:n {#2} #1 = \int_eval:n {#3} sp \scan_stop: }
\cs_new_protected:Npn \__intarray_gset:Nnn #1#2#3
  {
    \exp_args:Nff \__intarray_gset_aux:Nnn #1
      { \int_eval:n {#2} } { \int_eval:n {#3} }
  }
\cs_new_protected:Npn \__intarray_gset_aux:Nnn #1#2#3
  {
    \int_compare:nTF { 1 <= #2 <= \__intarray_count:N #1 }
      {
        \int_compare:nTF { - \c_max_dim <= \int_abs:n {#3} <= \c_max_dim }
          { \__intarray_gset_fast:Nnn #1 {#2} {#3} }
          {
            \__msg_kernel_error:nnxxxx { kernel } { overflow }
              { \token_to_str:N #1 } {#2} {#3}
              { \int_compare:nNnT {#3} < 0 { - } \__int_value:w \c_max_dim }
            \__intarray_gset_fast:Nnn #1 {#2}
              { \int_compare:nNnT {#3} < 0 { - } \c_max_dim }
          }
      }
      {
        \__msg_kernel_error:nnxxx { kernel } { out-of-bounds }
          { \token_to_str:N #1 } {#2} { \__intarray_count:N #1 }
      }
  }
```

(*End definition for* `\__intarray_gset:Nnn`*,* `\__intarray_gset_fast:Nnn`*, and* `\__intarray_gset_aux:Nnn`*.*)

`\__intarray_item:Nn`
`\__intarray_item_fast:Nn`
`\__intarray_item_aux:Nn`

Get the appropriate `\fontdimen` and perform bound checks if requested.

```
\cs_new:Npn \__intarray_item_fast:Nn #1#2
  { \__int_value:w \tex_fontdimen:D \int_eval:n {#2} #1 }
\cs_new:Npn \__intarray_item:Nn #1#2
  { \exp_args:Nf \__intarray_item_aux:Nn #1 { \int_eval:n {#2} } }
\cs_new:Npn \__intarray_item_aux:Nn #1#2
  {
    \int_compare:nTF { 1 <= #2 <= \__intarray_count:N #1 }
      { \__intarray_item_fast:Nn #1 {#2} }
      {
        \__msg_kernel_expandable_error:nnnnn { kernel } { out-of-bounds }
          { \token_to_str:N #1 } {#2} { \__intarray_count:N #1 }
        0
      }
  }
```

(*End definition for* `\__intarray_item:Nn`*,* `\__intarray_item_fast:Nn`*, and* `\__intarray_item_aux:Nn`*.*)

```
⟨/initex | package⟩
```

10 l3flag implementation

```
⟨*initex | package⟩
```

```
⟨@@=flag⟩
```

The following test files are used for this code: m3flag001.

10.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form `\flag` ⟨*name*⟩ ⟨*integer*⟩ to `\relax` for $0 \leq \langle integer\rangle < h$. When a flag is raised, a "trap" function `\flag` ⟨*name*⟩ is called. The existence of this function is also used to test for the existence of a flag.

`\flag_new:n`

For each flag, we define a "trap" function, which by default simply increases the flag by 1 by letting the appropriate control sequence to `\relax`. This can be done expandably!

```
\cs_new_protected:Npn \flag_new:n #1
  {
    \cs_new:cpn { flag~#1 } ##1 ;
```

```
      { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
  }
```

(*End definition for* `\flag_new:n`. *This function is documented on page 83.*)

`\flag_clear:n`
`\__flag_clear:wn`

Undefine control sequences, starting from the `0` flag, upwards, until reaching an undefined control sequence. We don't use `\cs_undefine:c` because that would act globally. When the option `check-declarations` is used, check for the function defined by `\flag_new:n`.

```
\__debug_patch:nnNNpn
  { \exp_args:Nc \__debug_chk_var_exist:N { flag~#1 } } { }
\cs_new_protected:Npn \flag_clear:n #1 { \__flag_clear:wn 0 ; {#1} }
\cs_new_protected:Npn \__flag_clear:wn #1 ; #2
  {
    \if_cs_exist:w flag~#2~#1 \cs_end:
      \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
      \exp_after:wN \__flag_clear:wn
      \__int_value:w \__int_eval:w 1 + #1
    \else:
      \use_i:nnn
    \fi:
    ; {#2}
  }
```

(*End definition for* `\flag_clear:n` *and* `\__flag_clear:wn`. *These functions are documented on page 83.*)

`\flag_clear_new:n`

As for other datatypes, clear the ⟨*flag*⟩ or create a new one, as appropriate.

```
\cs_new_protected:Npn \flag_clear_new:n #1
  { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }
```

(*End definition for* `\flag_clear_new:n`. *This function is documented on page 83.*)

`\flag_show:n`
`\flag_log:n`

Show the height (terminal or log file) using appropriate l3msg auxiliaries.

```
\cs_new_protected:Npn \flag_show:n #1
  {
    \exp_args:Nc \__msg_show_variable:NNNnn { flag~#1 } \cs_if_exist:NTF ? { }
      { > ~ flag ~ #1 ~ height = \flag_height:n {#1} }
  }
\cs_new_protected:Npn \flag_log:n
  { \__msg_log_next: \flag_show:n }
```

(*End definition for* `\flag_show:n` *and* `\flag_log:n`. *These functions are documented on page 83.*)

10.2 Expandable flag commands

`\__flag_chk_exist:n`

Analogue of `\__debug_chk_var_exist:N` for flags, and with an expandable error. We need to add checks by hand because flags are not implemented in terms of other variables. Not all functions need to be patched since some are defined in terms of others.

```
⟨*package⟩
\tex_ifodd:D \l@expl@enable@debug@bool
  \cs_new:Npn \__flag_chk_exist:n #1
    {
      \flag_if_exist:nF {#1}
        {
            \__msg_kernel_expandable_error:nnn
              { kernel } { bad-variable } { flag~#1~ }
          }
      }
\fi:
⟨/package⟩
```

(*End definition for* `\__flag_chk_exist:n`.)

`\flag_if_exist_p:n`
`\flag_if_exist:nTF`

A flag exist if the corresponding trap `\flag ⟨flag name⟩:n` is defined.

```
\prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
  {
    \cs_if_exist:cTF { flag~#1 }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* `\flag_if_exist:nTF`. *This function is documented on page 84.*)

`\flag_if_raised_p:n`
`\flag_if_raised:nTF`

Test if the flag has a non-zero height, by checking the `0` control sequence.

```
\__debug_patch_conditional:nNNpnn { \__flag_chk_exist:n {#1} }
\prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
  {
    \if_cs_exist:w flag~#1~0 \cs_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\flag_if_raised:nTF`. *This function is documented on page 84.*)

`\flag_height:n`
`\__flag_height_loop:wn`
`\__flag_height_end:wn`

Extract the value of the flag by going through all of the control sequences starting from `0`.

```
\__debug_patch:nnNNpn { \__flag_chk_exist:n {#1} } { }
\cs_new:Npn \flag_height:n #1 { \__flag_height_loop:wn 0; {#1} }
\cs_new:Npn \__flag_height_loop:wn #1 ; #2
  {
    \if_cs_exist:w flag~#2~#1 \cs_end:
      \exp_after:wN \__flag_height_loop:wn \__int_value:w \__int_eval:w 1 +
    \else:
      \exp_after:wN \__flag_height_end:wn
    \fi:
    #1 ; {#2}
  }
\cs_new:Npn \__flag_height_end:wn #1 ; #2 {#1}
```

(*End definition for* `\flag_height:n`, `\__flag_height_loop:wn`, *and* `\__flag_height_end:wn`. *These functions are documented on page 84.*)

`\flag_raise:n`

Simply apply the trap to the height, after expanding the latter.

```
\cs_new:Npn \flag_raise:n #1
  {
    \cs:w flag~#1 \exp_after:wN \cs_end:
    \__int_value:w \flag_height:n {#1} ;
  }
```

(*End definition for* `\flag_raise:n`*. This function is documented on page 84.*)

```
⟨/initex | package⟩
```

11 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```
⟨*initex | package⟩
```

11.1 Quarks

```
⟨@@=quark⟩
```

`\quark_new:N`

Allocate a new quark.

```
\cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }
```

(*End definition for* `\quark_new:N`*. This function is documented on page 85.*)

`\q_nil`
`\q_mark`
`\q_no_value`
`\q_stop`

Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
\quark_new:N \q_nil
\quark_new:N \q_mark
\quark_new:N \q_no_value
\quark_new:N \q_stop
```

(*End definition for* `\q_nil` *and others. These variables are documented on page 86.*)

`\q_recursion_tail`
`\q_recursion_stop`

Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
\quark_new:N \q_recursion_tail
\quark_new:N \q_recursion_stop
```

(*End definition for* `\q_recursion_tail` *and* `\q_recursion_stop`*. These variables are documented on page 87.*)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
\cs_new:Npn \quark_if_recursion_tail_stop:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
```

(*End definition for* `\quark_if_recursion_tail_stop:N` *and* `\quark_if_recursion_tail_stop_do:Nn`*. These functions are documented on page 87.*)

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on
__quark_if_recursion_tail:w

See `\quark_if_nil:nTF` for the details. Expanding `\__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if `#1` is exactly `\q_recursion_tail`.

```
\cs_new:Npn \quark_if_recursion_tail_stop:n #1
  {
    \tl_if_empty:oTF
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      { \use_none_delimit_by_q_recursion_stop:w }
      { }
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
  {
    \tl_if_empty:oTF
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      { \use_i_delimit_by_q_recursion_stop:nw }
      { \use_none:n }
  }
\cs_new:Npn \__quark_if_recursion_tail:w
    #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(*End definition for* `\quark_if_recursion_tail_stop:n`*,* `\quark_if_recursion_tail_stop_do:nn`*, and* `\__quark_if_recursion_tail:w`*. These functions are documented on page 87.*)

__quark_if_recursion_tail_break:NN
__quark_if_recursion_tail_break:nN

Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using `#2`.

```
\cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN #2
    \fi:
  }
\cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
  {
    \tl_if_empty:oTF
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      {#2}
      { }
  }
```

(*End definition for* `\__quark_if_recursion_tail_break:NN` *and* `\__quark_if_recursion_tail_break:nN`*.*)

\quark_if_nil_p:N
\quark_if_nil:N*TF*
\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:N*TF*
\quark_if_no_value:c*TF*

Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if `#1` is wrongly given a string like `aabc` instead of a single token.[9]

[9] It may still loop in special circumstances however!

```
\prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
  {
    \if_meaning:w \q_nil #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
  {
    \if_meaning:w \q_no_value #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \quark_if_no_value_p:N { c }
\cs_generate_variant:Nn \quark_if_no_value:NT  { c }
\cs_generate_variant:Nn \quark_if_no_value:NF  { c }
\cs_generate_variant:Nn \quark_if_no_value:NTF { c }
```

(*End definition for* `\quark_if_nil:NTF` *and* `\quark_if_no_value:NTF`*. These functions are documented on page 86.*)

`\quark_if_nil_p:n`
`\quark_if_nil_p:V`
`\quark_if_nil_p:o`
`\quark_if_nil:nTF`
`\quark_if_nil:VTF`
`\quark_if_nil:oTF`
`\quark_if_no_value_p:n`
`\quark_if_no_value:nTF`
`\__quark_if_nil:w`
`\__quark_if_no_value:w`

Let us explain `\quark_if_nil:n(TF)`. Expanding `\__quark_if_nil:w` once is safe thanks to the trailing `\q_nil ??!`. The result of expanding once is empty if and only if both delimited arguments `#1` and `#2` are empty and `#3` is delimited by the last tokens `?!`. Thanks to the leading `{}`, the argument `#1` is empty if and only if the argument of `\quark_if_nil:n` starts with `\q_nil`. The argument `#2` is empty if and only if this `\q_nil` is followed immediately by `?` or by `{}?`, coming either from the trailing tokens in the definition of `\quark_if_nil:n`, or from its argument. In the first case, `\__quark_if_nil:w` is followed by `{}\q_nil {}? !\q_nil ??!`, hence `#3` is delimited by the final `?!`, and the test returns `true` as wanted. In the second case, the result is not empty since the first `?!` in the definition of `\quark_if_nil:n` stop `#3`.

```
\prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
  {
    \__tl_if_empty_return:o
      { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
  }
\cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
\prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
  {
    \__tl_if_empty_return:o
      { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
  }
\cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
\cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
\cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
\cs_generate_variant:Nn \quark_if_nil:nT  { V , o }
\cs_generate_variant:Nn \quark_if_nil:nF  { V , o }
```

(*End definition for* `\quark_if_nil:nTF` *and others. These functions are documented on page 86.*)

\q__tl_act_mark
\q__tl_act_stop

These private quarks are needed by l3tl, but that is loaded before the quark module, hence their definition is deferred.

```
\quark_new:N \q__tl_act_mark
\quark_new:N \q__tl_act_stop
```

(*End definition for* \q__tl_act_mark *and* \q__tl_act_stop.)

11.2 Scan marks

```
⟨@@=scan⟩
```

\g__scan_marks_tl

The list of all scan marks currently declared.

```
\tl_new:N \g__scan_marks_tl
```

(*End definition for* \g__scan_marks_tl.)

__scan_new:N

Check whether the variable is already a scan mark, then declare it to be equal to \scan_stop: globally.

```
\cs_new_protected:Npn \__scan_new:N #1
  {
    \tl_if_in:NnTF \g__scan_marks_tl { #1 }
      {
        \__msg_kernel_error:nnx { kernel } { scanmark-already-defined }
          { \token_to_str:N #1 }
      }
      {
        \tl_gput_right:Nn \g__scan_marks_tl {#1}
        \cs_new_eq:NN #1 \scan_stop:
      }
  }
```

(*End definition for* __scan_new:N.)

\s__stop

We only declare one scan mark here, more can be defined by specific modules.

```
\__scan_new:N \s__stop
```

(*End definition for* \s__stop.)

__use_none_delimit_by_s__stop:w

Similar to \use_none_delimit_by_q_stop:w.

```
\cs_new:Npn \__use_none_delimit_by_s__stop:w #1 \s__stop { }
```

(*End definition for* __use_none_delimit_by_s__stop:w.)

\s__seq

This private scan mark is needed by l3seq, but that is loaded before the quark module, hence its definition is deferred.

```
\__scan_new:N \s__seq
```

(*End definition for* \s__seq.)

```
⟨/initex | package⟩
```

12 l3prg implementation

The following test files are used for this code: m3prg001.lvt,m3prg002.lvt,m3prg003.lvt.

```
⟨*initex | package⟩
```

12.1 Primitive conditionals

\if_bool:N
\if_predicate:w

Those two primitive TEX conditionals are synonyms.

```
\cs_new_eq:NN \if_bool:N      \tex_ifodd:D
\cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(*End definition for* `\if_bool:N` *and* `\if_predicate:w`*. These functions are documented on page 97.*)

12.2 Defining a set of conditional functions

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\prg_return_true:
\prg_return_false:

These are all defined in l3basics, as they are needed "early". This is just a reminder!

(*End definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page 90.*)

12.3 The boolean data type

```
⟨@@=bool⟩
```

\bool_new:N
\bool_new:c

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_new:N { c }
```

(*End definition for* `\bool_new:N`*. This function is documented on page 92.*)

\bool_set_true:N
\bool_set_true:c
\bool_gset_true:N
\bool_gset_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c

Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on TEX registers.

```
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_set_true:N #1
  { \cs_set_eq:NN #1 \c_true_bool }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_set_false:N #1
  { \cs_set_eq:NN #1 \c_false_bool }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_gset_true:N #1
  { \cs_gset_eq:NN #1 \c_true_bool }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_gset_false:N #1
  { \cs_gset_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_set_true:N   { c }
\cs_generate_variant:Nn \bool_set_false:N  { c }
\cs_generate_variant:Nn \bool_gset_true:N  { c }
\cs_generate_variant:Nn \bool_gset_false:N { c }
```

(*End definition for* `\bool_set_true:N` *and others. These functions are documented on page 92.*)

\bool_set_eq:NN
\bool_set_eq:cN
\bool_set_eq:Nc
\bool_set_eq:cc
\bool_gset_eq:NN
\bool_gset_eq:cN
\bool_gset_eq:Nc
\bool_gset_eq:cc

The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.

```
\cs_new_eq:NN \bool_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
\cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
```

(*End definition for* `\bool_set_eq:NN` *and* `\bool_gset_eq:NN`*. These functions are documented on page 93.*)

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`

This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`. Again, we include some checking code.

```
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_set:Nn #1#2
  { \tex_chardef:D #1 = \bool_if_p:n {#2} }
\__debug_patch:nnNNpn { \__debug_chk_var_exist:N #1 } { }
\cs_new_protected:Npn \bool_gset:Nn #1#2
  { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
\cs_generate_variant:Nn \bool_set:Nn  { c }
\cs_generate_variant:Nn \bool_gset:Nn { c }
```

(*End definition for* `\bool_set:Nn` *and* `\bool_gset:Nn`*. These functions are documented on page 93.*)

`\bool_if_p:N`
`\bool_if_p:c`
`\bool_if:NTF`
`\bool_if:cTF`

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
  {
    \if_bool:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \bool_if_p:N { c }
\cs_generate_variant:Nn \bool_if:NT  { c }
\cs_generate_variant:Nn \bool_if:NF  { c }
\cs_generate_variant:Nn \bool_if:NTF { c }
```

(*End definition for* `\bool_if:NTF`*. This function is documented on page 93.*)

`\bool_show:N`
`\bool_show:c`
`\bool_show:n`
`\__bool_to_str:n`

Show the truth value of the boolean, as `true` or `false`.

```
\cs_new_protected:Npn \bool_show:N #1
  {
    \__msg_show_variable:NNNnn #1 \bool_if_exist:NTF ? { }
      { > ~ \token_to_str:N #1 = \__bool_to_str:n {#1} }
  }
\cs_new_protected:Npn \bool_show:n
  { \__msg_show_wrap:Nn \__bool_to_str:n }
\cs_new:Npn \__bool_to_str:n #1
  { \bool_if:nTF {#1} { true } { false } }
\cs_generate_variant:Nn \bool_show:N { c }
```

(*End definition for* `\bool_show:N`*,* `\bool_show:n`*, and* `\__bool_to_str:n`*. These functions are documented on page 93.*)

`\bool_log:N`
`\bool_log:c`
`\bool_log:n`

Redirect output of `\bool_show:N` to the log.

```
\cs_new_protected:Npn \bool_log:N
  { \__msg_log_next: \bool_show:N }
\cs_new_protected:Npn \bool_log:n
  { \__msg_log_next: \bool_show:n }
\cs_generate_variant:Nn \bool_log:N { c }
```

(*End definition for* `\bool_log:N` *and* `\bool_log:n`*. These functions are documented on page 93.*)

`\l_tmpa_bool`
`\l_tmpb_bool`
`\g_tmpa_bool`
`\g_tmpb_bool`

A few booleans just if you need them.

```
\bool_new:N \l_tmpa_bool
\bool_new:N \l_tmpb_bool
\bool_new:N \g_tmpa_bool
\bool_new:N \g_tmpb_bool
```

(*End definition for* `\l_tmpa_bool` *and others. These variables are documented on page 93.*)

`\bool_if_exist_p:N`
`\bool_if_exist_p:c`
`\bool_if_exist:NTF`
`\bool_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* `\bool_if_exist:NTF`*. This function is documented on page 93.*)

12.4 Boolean expressions

`\bool_if_p:n`
`\bool_if:nTF`

Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical "Not", `&&` for logical "And" and `||` for logical "Or". However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the `!` and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value ⟨*true*⟩ or ⟨*false*⟩.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

⟨*true*⟩And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*false*⟩And Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return ⟨*false*⟩.

⟨*true*⟩Or Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return ⟨*true*⟩.

⟨*false*⟩Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*true*⟩Close Current truth value is true, Close seen, return ⟨*true*⟩.

⟨*false*⟩Close Current truth value is false, Close seen, return ⟨*false*⟩.

```
\prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_if_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* \bool_if:nTF. *This function is documented on page 95.*)

\bool_if_p:n First issue a \group_align_safe_begin: as we are using && as syntax shorthand for the And operation and we need to hide it for TEX. This group is closed after __bool_get_next:NN returns \c_true_bool or \c_false_bool. That function requires the trailing parenthesis to know where the expression ends.

```
\cs_new:Npn \bool_if_p:n #1
  {
    \group_align_safe_begin:
    \exp_after:wN
    \group_align_safe_end:
    \exp:w \exp_end_continue_f:w % (
    \__bool_get_next:NN \use_i:nnnn #1 )
  }
```

(*End definition for* \bool_if_p:n. *This function is documented on page 95.*)

__bool_get_next:NN The GetNext operation. Its first argument is \use_i:nnnn, \use_ii:nnnn, \use_iii:nnnn, or \use_iv:nnnn (we call these "states"). In the first state, this function eventually expand to the truth value \c_true_bool or \c_false_bool of the expression which follows until the next unmatched closing parenthesis. For instance "__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)" (including the closing parenthesis) expands to \c_true_bool. In the second state (after a !) the logic is reversed. We call these two states "normal" and the next two "skipping". In the third state (after \c_true_bool||) it always returns \c_true_bool. In the fourth state (after \c_false_bool&&) it always returns \c_false_bool and also stops when encountering ||, not only parentheses. This code itself is a switch: if what follows is neither ! nor (, we assume it is a predicate.

```
\cs_new:Npn \__bool_get_next:NN #1#2
  {
    \use:c
      {
        __bool_
        \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
        :Nw
      }
      #1 #2
  }
```

(*End definition for* __bool_get_next:NN.)

__bool_!:Nw The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after \c_true_bool|| or \c_false_bool&& the ! is ignored.

```
\cs_new:cpn { __bool_!:Nw } #1#2
  {
    \exp_after:wN \__bool_get_next:NN
    #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
  }
```

(*End definition for* `\__bool_!:Nw`.)

`\__bool_(:Nw`

The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```
\cs_new:cpn { __bool_(:Nw } #1#2
  {
    \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
    \__int_value:w \__bool_get_next:NN \use_i:nnnn
  }
```

(*End definition for* `\__bool_(:Nw`.)

`\__bool_p:Nw`

If what follows GetNext is neither `!` nor `(`, evaluate the predicate using the primitive `\__int_value:w`. The canonical `true` and `false` values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```
\cs_new:cpn { __bool_p:Nw } #1
  { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \__int_value:w }
```

(*End definition for* `\__bool_p:Nw`.)

`\__bool_choose:NNN`
`\__bool_)_0:`
`\__bool_)_1:`
`\__bool_)_2:`
`\__bool_&_0:`
`\__bool_&_1:`
`\__bool_&_2:`
`\__bool_|_0:`
`\__bool_|_1:`
`\__bool_|_2:`

The arguments are `#1`: a function such as `\use_i:nnnn`, `#2`: 0 or 1 encoding the current truth value, `#3`: the next operation, And, Or or Close. We distinguish three cases according to a combination of `#1` and `#2`. Case 2 is when `#1` is `\use_iii:nnnn` (state 3), namely after `\c_true_bool ||`. Case 1 is when `#1` is `\use_i:nnnn` and `#2` is `true` or when `#1` is `\use_ii:nnnn` and `#2` is `false`, for instance for `!\c_false_bool`. Case 0 includes the same with `true`/`false` interchanged and the case where `#1` is `\use_iv:nnnn` namely after `\c_false_bool &&`.

When seeing `)` the current subexpression is done, leave the appropriate boolean. When seeing `&` in case 0 go into state 4, equivalent to having seen `\c_false_bool &&`. In case 1, namely when the argument is `true` and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing `|` in case 0, continue in a normal state; in particular stop skipping for `\c_false_bool &&` because that binds more tightly than `||`. In the other two cases start skipping for `\c_true_bool ||`.

```
\cs_new:Npn \__bool_choose:NNN #1#2#3
  {
    \use:c
      {
        __bool_ \token_to_str:N #3 _
        #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
      }
  }
\cs_new:cpn { __bool_)_0: } { \c_false_bool }
\cs_new:cpn { __bool_)_1: } { \c_true_bool }
\cs_new:cpn { __bool_)_2: } { \c_true_bool }
\cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
```

```
\cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
\cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }
\cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
\cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
\cs_new:cpn { __bool_|_2: } | { \__bool_get_next:NN \use_iii:nnnn }
```

(*End definition for* `\__bool_choose:NNN` *and others.*)

`\bool_lazy_all_p:n`
`\bool_lazy_all:nTF`
`\__bool_lazy_all:n`

Go through the list of expressions, stopping whenever an expression is `false`. If the end is reached without finding any `false` expression, then the result is `true`.

```
\prg_new_conditional:Npnn \bool_lazy_all:n #1 { p , T , F , TF }
  { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
\cs_new:Npn \__bool_lazy_all:n #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_true: }
    \bool_if:nF {#1}
      { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_false: } }
    \__bool_lazy_all:n
  }
```

(*End definition for* `\bool_lazy_all:nTF` *and* `\__bool_lazy_all:n`*. These functions are documented on page 95.*)

`\bool_lazy_and_p:nn`
`\bool_lazy_and:nnTF`

Only evaluate the second expression if the first is `true`.

```
\prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
  {
    \bool_if:nTF {#1}
      { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
      { \prg_return_false: }
  }
```

(*End definition for* `\bool_lazy_and:nnTF`*. This function is documented on page 95.*)

`\bool_lazy_any_p:n`
`\bool_lazy_any:nTF`
`\__bool_lazy_any:n`

Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

```
\prg_new_conditional:Npnn \bool_lazy_any:n #1 { p , T , F , TF }
  { \__bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
\cs_new:Npn \__bool_lazy_any:n #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_false: }
    \bool_if:nT {#1}
      { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_true: } }
    \__bool_lazy_any:n
  }
```

(*End definition for* `\bool_lazy_any:nTF` *and* `\__bool_lazy_any:n`*. These functions are documented on page 95.*)

`\bool_lazy_or_p:nn`
`\bool_lazy_or:nnTF`

Only evaluate the second expression if the first is `false`.

```
\prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
  {
    \bool_if:nTF {#1}
      { \prg_return_true: }
      { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
  }
```

(*End definition for* \bool_lazy_or:nnTF. *This function is documented on page 95.*)

\bool_not_p:n

The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
\cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(*End definition for* \bool_not_p:n. *This function is documented on page 95.*)

\bool_xor_p:nn

Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```
\cs_new:Npn \bool_xor_p:nn #1#2
  {
    \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
      \c_false_bool
      \c_true_bool
  }
```

(*End definition for* \bool_xor_p:nn. *This function is documented on page 96.*)

12.5 Logical loops

\bool_while_do:Nn
\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn

A `while` loop where the boolean is tested before executing the statement. The "while" version executes the code as long as the boolean is true; the "until" version executes the code as long as the boolean is false.

```
\cs_new:Npn \bool_while_do:Nn #1#2
  { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
\cs_new:Npn \bool_until_do:Nn #1#2
  { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_while_do:Nn { c }
\cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(*End definition for* \bool_while_do:Nn *and* \bool_until_do:Nn. *These functions are documented on page 96.*)

\bool_do_while:Nn
\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn

A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```
\cs_new:Npn \bool_do_while:Nn #1#2
  { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
\cs_new:Npn \bool_do_until:Nn #1#2
  { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_do_while:Nn { c }
\cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(*End definition for* \bool_do_while:Nn *and* \bool_do_until:Nn. *These functions are documented on page 96.*)

\bool_while_do:nn
\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn

Loop functions with the test either before or after the first body expansion.

```
\cs_new:Npn \bool_while_do:nn #1#2
  {
    \bool_if:nT {#1}
      {
        #2
        \bool_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_while:nn #1#2
  {
    #2
    \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \bool_until_do:nn #1#2
  {
    \bool_if:nF {#1}
      {
        #2
        \bool_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_until:nn #1#2
  {
    #2
    \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2}  }
  }
```

(*End definition for* `\bool_while_do:nn` *and others. These functions are documented on page 97.*)

12.6 Producing multiple copies

```
⟨@@=prg⟩
```

\prg_replicate:nn
__prg_replicate:N
__prg_replicate_first:N
__prg_replicate_
__prg_replicate_0:n
__prg_replicate_1:n
__prg_replicate_2:n
__prg_replicate_3:n
__prg_replicate_4:n
__prg_replicate_5:n
__prg_replicate_6:n
__prg_replicate_7:n
__prg_replicate_8:n
__prg_replicate_9:n
__prg_replicate_first_-:n
__prg_replicate_first_0:n
__prg_replicate_first_1:n
__prg_replicate_first_2:n
__prg_replicate_first_3:n
__prg_replicate_first_4:n
__prg_replicate_first_5:n
__prg_replicate_first_6:n
__prg_replicate_first_7:n
__prg_replicate_first_8:n
__prg_replicate_first_9:n

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input `25` result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {`⟨*code*⟩`} }`. An alternative approach is to create a string of `m`'s with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \prg_replicate:nn #1
  {
    \exp:w
      \exp_after:wN \__prg_replicate_first:N
        \__int_value:w \__int_eval:w #1 \__int_eval_end:
```

```
      \cs_end:
  }
\cs_new:Npn \__prg_replicate:N #1
  { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
\cs_new:Npn \__prg_replicate_first:N #1
  { \cs:w __prg_replicate_first_ #1 :n \__prg_replicate:N }
```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

```
\cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
\cs_new:cpn { __prg_replicate_0:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
\cs_new:cpn { __prg_replicate_1:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
\cs_new:cpn { __prg_replicate_2:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
\cs_new:cpn { __prg_replicate_3:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
\cs_new:cpn { __prg_replicate_4:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
\cs_new:cpn { __prg_replicate_5:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_6:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_7:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_8:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_9:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
```

Users shouldn't ask for something to be replicated once or even not at all but…

```
\cs_new:cpn { __prg_replicate_first_-:n } #1
  {
    \exp_end:
    \__msg_kernel_expandable_error:nn { kernel } { negative-replication }
  }
\cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
\cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
\cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
\cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
\cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_9:n } #1 { \exp_end: #1#1#1#1#1#1#1#1#1 }
```

(*End definition for* `\prg_replicate:nn` *and others. These functions are documented on page 97.*)

12.7 Detecting TeX's mode

`\mode_if_vertical_p:`
`\mode_if_vertical:TF`

For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
\prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
  { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_vertical:TF`. *This function is documented on page 97.*)

\mode_if_horizontal_p:
\mode_if_horizontal:TF

For testing horizontal mode.

```
\prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
  { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_horizontal:TF`. *This function is documented on page 97.*)

\mode_if_inner_p:
\mode_if_inner:TF

For testing inner mode.

```
\prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
  { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_inner:TF`. *This function is documented on page 97.*)

\mode_if_math_p:
\mode_if_math:TF

For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
\prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
  { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_math:TF`. *This function is documented on page 97.*)

12.8 Internal programming functions

\group_align_safe_begin:
\group_align_safe_end:

TeX's alignment structures present many problems. As Knuth says himself in *TeX: The Program*: "It's sort of a miracle whenever `\halign` or `\valign` work, [...]" One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&`$_4$ giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that TeX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The TeXbook*... We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
\cs_new:Npn \group_align_safe_begin:
  { \if_int_compare:w \if_false: { \fi: `} = \c_zero \fi: }
\cs_new:Npn \group_align_safe_end:
  { \if_int_compare:w `{ = \c_zero } \fi: }
```

(*End definition for* `\group_align_safe_begin:` *and* `\group_align_safe_end:`.)

```
⟨@@=prg⟩
```

\g__prg_map_int

A nesting counter for mapping.

```
\int_new:N \g__prg_map_int
```

(*End definition for* `\g__prg_map_int`.)

__prg_break_point:Nn
__prg_map_break:Nn

These are defined in l3basics, as they are needed "early". This is just a reminder that is the case!

(*End definition for* __prg_break_point:Nn *and* __prg_map_break:Nn*.*)

__prg_break_point:
__prg_break:
__prg_break:n

Also done in l3basics as in format mode these are needed within l3alloc.

(*End definition for* __prg_break_point: *,* __prg_break: *, and* __prg_break:n*.*)

```
⟨/initex | package⟩
```

13 l3clist implementation

The following test files are used for this code: m3clist002.

```
⟨*initex | package⟩
⟨@@=clist⟩
```

\c_empty_clist

An empty comma list is simply an empty token list.

```
\cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(*End definition for* \c_empty_clist*. This variable is documented on page 107.*)

\l__clist_internal_clist

Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before \clist_new:N

```
\tl_new:N \l__clist_internal_clist
```

(*End definition for* \l__clist_internal_clist*.*)

__clist_tmp:w

A temporary function for various purposes.

```
\cs_new_protected:Npn \__clist_tmp:w { }
```

(*End definition for* __clist_tmp:w*.*)

13.1 Allocation and initialisation

\clist_new:N
\clist_new:c

Internally, comma lists are just token lists.

```
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:c \tl_new:c
```

(*End definition for* \clist_new:N*. This function is documented on page 99.*)

\clist_const:Nn
\clist_const:cn
\clist_const:Nx
\clist_const:cx

Creating and initializing a constant comma list is done in a way similar to \clist_set:Nn and \clist_gset:Nn, being careful to strip spaces.

```
\cs_new_protected:Npn \clist_const:Nn #1#2
  { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(*End definition for* \clist_const:Nn*. This function is documented on page 99.*)

\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

Clearing comma lists is just the same as clearing token lists.

```
\cs_new_eq:NN \clist_clear:N  \tl_clear:N
\cs_new_eq:NN \clist_clear:c  \tl_clear:c
\cs_new_eq:NN \clist_gclear:N \tl_gclear:N
\cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(*End definition for* `\clist_clear:N` *and* `\clist_gclear:N`*. These functions are documented on page 99.*)

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

Once again a copy from the token list functions.

```
\cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \clist_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(*End definition for* `\clist_clear_new:N` *and* `\clist_gclear_new:N`*. These functions are documented on page 99.*)

`\clist_set_eq:NN`
`\clist_set_eq:cN`
`\clist_set_eq:Nc`
`\clist_set_eq:cc`
`\clist_gset_eq:NN`
`\clist_gset_eq:cN`
`\clist_gset_eq:Nc`
`\clist_gset_eq:cc`

Once again, these are simple copies from the token list functions.

```
\cs_new_eq:NN \clist_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \clist_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \clist_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \clist_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\clist_set_eq:NN` *and* `\clist_gset_eq:NN`*. These functions are documented on page 100.*)

`\clist_set_from_seq:NN`
`\clist_set_from_seq:cN`
`\clist_set_from_seq:Nc`
`\clist_set_from_seq:cc`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:cN`
`\clist_gset_from_seq:Nc`
`\clist_gset_from_seq:cc`
`\__clist_set_from_seq:NNNN`
`\__clist_wrap_item:n`
`\__clist_set_from_seq:w`

Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with `\exp_not:n`, and a comma. Items which contain a comma or a space are surrounded by an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\cs_new_protected:Npn \clist_set_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_clear:N  \tl_set:Nx  }
\cs_new_protected:Npn \clist_gset_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
  {
    \seq_if_empty:NTF #4
      { #1 #3 }
      {
        #2 #3
          {
            \exp_last_unbraced:Nf \use_none:n
              { \seq_map_function:NN #4 \__clist_wrap_item:n }
          }
      }
  }
\cs_new:Npn \__clist_wrap_item:n #1
  {
    ,
    \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
      { \exp_not:n   {#1}   }
      { \exp_not:n { {#1} } }
  }
\cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
\cs_generate_variant:Nn \clist_set_from_seq:NN  {     Nc }
```

```
\cs_generate_variant:Nn \clist_set_from_seq:NN  { c , cc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN {     Nc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(*End definition for* \clist_set_from_seq:NN *and others. These functions are documented on page 100.*)

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
__clist_concat:NNNN

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
\cs_new_protected:Npn \clist_concat:NNN
  { \__clist_concat:NNNN \tl_set:Nx }
\cs_new_protected:Npn \clist_gconcat:NNN
  { \__clist_concat:NNNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
  {
    #1 #2
      {
        \exp_not:o #3
        \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
        \exp_not:o #4
      }
  }
\cs_generate_variant:Nn \clist_concat:NNN  { ccc }
\cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(*End definition for* \clist_concat:NNN*,* \clist_gconcat:NNN*, and* __clist_concat:NNNN*. These functions are documented on page 100.*)

\clist_if_exist_p:N
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* \clist_if_exist:NTF*. This function is documented on page 100.*)

13.2 Removing spaces around items

__clist_trim_spaces_generic:nw
__clist_trim_spaces_generic:nn

__clist_trim_spaces_generic:nw {⟨*code*⟩} \q_mark ⟨*item*⟩ ,

This expands to the ⟨*code*⟩, followed by a brace group containing the ⟨*item*⟩, with leading and trailing spaces removed. The calling function is responsible for inserting \q_mark in front of the ⟨*item*⟩, as well as testing for the end of the list. We reuse a l3tl internal function, whose first argument must start with \q_mark. That trims the item #2, then feeds the result (after having to do an o-type expansion) to __clist_trim_spaces_generic:nn which places the ⟨*code*⟩ in front of the ⟨*trimmed item*⟩.

```
\cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
  {
    \__tl_trim_spaces:nn {#2}
      { \exp_args:No \__clist_trim_spaces_generic:nn } {#1}
  }
\cs_new:Npn \__clist_trim_spaces_generic:nn #1#2 { #2 {#1} }
```

(*End definition for* __clist_trim_spaces_generic:nw *and* __clist_trim_spaces_generic:nn*.*)

__clist_trim_spaces:n
__clist_trim_spaces:nn

The first argument of __clist_trim_spaces:nn is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```
\cs_new:Npn \__clist_trim_spaces:n #1
  {
    \__clist_trim_spaces_generic:nw
      { \__clist_trim_spaces:nn { } }
      \q_mark #1 ,
    \q_recursion_tail, \q_recursion_stop
  }
\cs_new:Npn \__clist_trim_spaces:nn #1 #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \tl_if_empty:nTF {#2}
      {
        \__clist_trim_spaces_generic:nw
          { \__clist_trim_spaces:nn {#1} } \q_mark
      }
      {
        #1 \exp_not:n {#2}
        \__clist_trim_spaces_generic:nw
          { \__clist_trim_spaces:nn { , } } \q_mark
      }
  }
```

(*End definition for* __clist_trim_spaces:n *and* __clist_trim_spaces:nn*.*)

13.3 Adding data to comma lists

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:No
\clist_gset:Nx
\clist_gset:cn
\clist_gset:cV
\clist_gset:co
\clist_gset:cx

```
\cs_new_protected:Npn \clist_set:Nn #1#2
  { \tl_set:Nx #1 { \__clist_trim_spaces:n {#2} } }
\cs_new_protected:Npn \clist_gset:Nn #1#2
  { \tl_gset:Nx #1 { \__clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_set:Nn  { NV , No , Nx , c , cV , co , cx }
\cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
```

(*End definition for* \clist_set:Nn *and* \clist_gset:Nn*. These functions are documented on page 100.*)

\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
__clist_put_left:NNNn

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```
\cs_new_protected:Npn \clist_put_left:Nn
  { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected:Npn \clist_gput_left:Nn
  { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 \l__clist_internal_clist #3
  }
\cs_generate_variant:Nn \clist_put_left:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_left:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_left:Nn {     NV , No , Nx }
```

```
\cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_left:Nn`*,* `\clist_gput_left:Nn`*, and* `\__clist_put_left:NNNn`*. These functions are documented on page 100.*)

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
__clist_put_right:NNNn

```
\cs_new_protected:Npn \clist_put_right:Nn
  { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected:Npn \clist_gput_right:Nn
  { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 #3 \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_put_right:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_right:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_right:Nn {     NV , No , Nx }
\cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_right:Nn`*,* `\clist_gput_right:Nn`*, and* `\__clist_put_right:NNNn`*. These functions are documented on page 101.*)

13.4 Comma lists as stacks

\clist_get:NN
\clist_get:cN
__clist_get:wN

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```
\cs_new_protected:Npn \clist_get:NN #1#2
  {
    \if_meaning:w #1 \c_empty_clist
      \tl_set:Nn #2 { \q_no_value }
    \else:
      \exp_after:wN \__clist_get:wN #1 , \q_stop #2
    \fi:
  }
\cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
  { \tl_set:Nn #3 {#1} }
\cs_generate_variant:Nn \clist_get:NN { c }
```

(*End definition for* `\clist_get:NN` *and* `\__clist_get:wN`*. These functions are documented on page 105.*)

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
__clist_pop:NNN
__clist_pop:wwNNN
__clist_pop:wN

An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\__clist_pop:wwNNN` is a comma list ending in a comma and `\q_mark`, unless the original clist contained exactly one item: then the argument is just `\q_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as `#2`, ensuring that the result can safely be an empty comma list.

```
\cs_new_protected:Npn \clist_pop:NN
  { \__clist_pop:NNN \tl_set:Nx }
\cs_new_protected:Npn \clist_gpop:NN
  { \__clist_pop:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_pop:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \tl_set:Nn #3 { \q_no_value }
    \else:
      \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
    \fi:
  }
\cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
  {
    \tl_set:Nn #5 {#1}
    #3 #4
      {
        \__clist_pop:wN \prg_do_nothing:
          #2 \exp_not:o
          , \q_mark \use_none:n
        \q_stop
      }
  }
\cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
\cs_generate_variant:Nn \clist_pop:NN  { c }
\cs_generate_variant:Nn \clist_gpop:NN { c }
```

(*End definition for* \clist_pop:NN *and others. These functions are documented on page 105.*)

\clist_get:NN*TF*
\clist_get:cN*TF*
\clist_pop:NN*TF*
\clist_pop:cN*TF*
\clist_gpop:NN*TF*
\clist_gpop:cN*TF*
__clist_pop_TF:NNN

The same, as branching code: very similar to the above.

```
\prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
  {
    \if_meaning:w #1 \c_empty_clist
      \prg_return_false:
    \else:
      \exp_after:wN \__clist_get:wN #1 , \q_stop #2
      \prg_return_true:
    \fi:
  }
\cs_generate_variant:Nn \clist_get:NNT  { c }
\cs_generate_variant:Nn \clist_get:NNF  { c }
\cs_generate_variant:Nn \clist_get:NNTF { c }
\prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
\cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \prg_return_false:
    \else:
      \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
      \prg_return_true:
    \fi:
  }
\cs_generate_variant:Nn \clist_pop:NNT   { c }
\cs_generate_variant:Nn \clist_pop:NNF   { c }
\cs_generate_variant:Nn \clist_pop:NNTF  { c }
\cs_generate_variant:Nn \clist_gpop:NNT  { c }
\cs_generate_variant:Nn \clist_gpop:NNF  { c }
\cs_generate_variant:Nn \clist_gpop:NNTF { c }
```

(*End definition for* \clist_get:NNTF *and others. These functions are documented on page 105.*)

\clist_push:Nn
\clist_push:NV
\clist_push:No
\clist_push:Nx
\clist_push:cn
\clist_push:cV
\clist_push:co
\clist_push:cx
\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:Nx
\clist_gpush:cn
\clist_gpush:cV
\clist_gpush:co
\clist_gpush:cx

Pushing to a comma list is the same as adding on the left.

```
\cs_new_eq:NN \clist_push:Nn  \clist_put_left:Nn
\cs_new_eq:NN \clist_push:NV  \clist_put_left:NV
\cs_new_eq:NN \clist_push:No  \clist_put_left:No
\cs_new_eq:NN \clist_push:Nx  \clist_put_left:Nx
\cs_new_eq:NN \clist_push:cn  \clist_put_left:cn
\cs_new_eq:NN \clist_push:cV  \clist_put_left:cV
\cs_new_eq:NN \clist_push:co  \clist_put_left:co
\cs_new_eq:NN \clist_push:cx  \clist_put_left:cx
\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
```

(*End definition for* `\clist_push:Nn` *and* `\clist_gpush:Nn`*. These functions are documented on page 106.*)

13.5 Modifying comma lists

\l__clist_internal_remove_clist

An internal comma list for the removal routines.

```
\clist_new:N \l__clist_internal_remove_clist
```

(*End definition for* `\l__clist_internal_remove_clist`*.*)

\clist_remove_duplicates:N
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
__clist_remove_duplicates:NN

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \clist_remove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\cs_new_protected:Npn \clist_gremove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
\cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
  {
    \clist_clear:N \l__clist_internal_remove_clist
    \clist_map_inline:Nn #2
      {
        \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
          { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
      }
    #1 #2 \l__clist_internal_remove_clist
  }
\cs_generate_variant:Nn \clist_remove_duplicates:N  { c }
\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(*End definition for* `\clist_remove_duplicates:N`*,* `\clist_gremove_duplicates:N`*, and* `\__clist_remove_duplicates:NN`*. These functions are documented on page 101.*)

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn
__clist_remove_all:NNn
__clist_remove_all:w
__clist_remove_all:

The method used here is very similar to `\tl_replace_all:Nnn`. Build a function delimited by the ⟨*item*⟩ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the ⟨*item*⟩. The loop is controlled by the argument grabbed by `\__clist_remove_all:w`: when the item was found, the \q_mark delimiter used is the one inserted by __clist_tmp:w, and \use_none_delimit_by_q_stop:w is deleted. At the end, the final ⟨*item*⟩ is grabbed, and the argument of __clist_tmp:w contains \q_mark: in that case, __clist_remove_all:w removes the second \q_mark (inserted by __clist_tmp:w), and lets \use_none_delimit_by_q_stop:w act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn't remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```
\cs_new_protected:Npn \clist_remove_all:Nn
  { \__clist_remove_all:NNn \tl_set:Nx }
\cs_new_protected:Npn \clist_gremove_all:Nn
  { \__clist_remove_all:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
  {
    \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
      {
        ##1
        , \q_mark , \use_none_delimit_by_q_stop:w ,
        \__clist_remove_all:
      }
    #1 #2
      {
        \exp_after:wN \__clist_remove_all:
        #2 , \q_mark , #3 , \q_stop
      }
    \clist_if_empty:NF #2
      {
        #1 #2
          {
            \exp_args:No \exp_not:o
              { \exp_after:wN \use_none:n #2 }
          }
      }
  }
\cs_new:Npn \__clist_remove_all:
  { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
\cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
\cs_generate_variant:Nn \clist_remove_all:Nn  { c }
\cs_generate_variant:Nn \clist_gremove_all:Nn { c }
```

(*End definition for \clist_remove_all:Nn and others. These functions are documented on page 101.*)

\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

Use \clist_reverse:n in an x-expanding assignment. The extra work that \clist_reverse:n does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

```
\cs_new_protected:Npn \clist_reverse:N #1
  { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
\cs_new_protected:Npn \clist_greverse:N #1
  { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
\cs_generate_variant:Nn \clist_reverse:N { c }
\cs_generate_variant:Nn \clist_greverse:N { c }
```

(*End definition for* `\clist_reverse:N` *and* `\clist_greverse:N`*. These functions are documented on page 101.*)

`\clist_reverse:n`
`\__clist_reverse:wwNww`
`\__clist_reverse_end:ww`

The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of `?` followed by zero or more instances of "$\langle item\rangle$`,`". We start from a comma list "$\langle item_1\rangle$`,...,`$\langle item_n\rangle$". During the loop, the auxiliary `\__clist_reverse:wwNww` receives "`?`$\langle item_i\rangle$" as `#1`, "$\langle item_{i+1}\rangle$`,...,`$\langle item_n\rangle$" as `#2`, `\__clist_reverse:wwNww` as `#3`, what remains until `\q_stop` as `#4`, and "$\langle item_{i-1}\rangle$`,...,`$\langle item_1\rangle$`,`" as `#5`. The auxiliary moves `#1` just before `#5`, with a comma, and calls itself (`#3`). After the last item is moved, `\__clist_reverse:wwNww` receives "`\q_mark \__clist_reverse:wwNww !`" as its argument `#1`, thus `\__clist_reverse_end:ww` as its argument `#3`. This second auxiliary cleans up until the marker `!`, removes the trailing comma (introduced when the first item was moved after `\q_stop`), and leaves its argument `#1` within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```
\cs_new:Npn \clist_reverse:n #1
  {
    \__clist_reverse:wwNww ? #1 ,
      \q_mark \__clist_reverse:wwNww ! ,
      \q_mark \__clist_reverse_end:ww
      \q_stop ? \q_mark
  }
\cs_new:Npn \__clist_reverse:wwNww
    #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
  { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
\cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
  { \exp_not:o { \use_none:n #2 } }
```

(*End definition for* `\clist_reverse:n`*,* `\__clist_reverse:wwNww`*, and* `\__clist_reverse_end:ww`*. These functions are documented on page 101.*)

`\clist_sort:Nn`
`\clist_sort:cn`
`\clist_gsort:Nn`
`\clist_gsort:cn`

Implemented in l3sort.

(*End definition for* `\clist_sort:Nn` *and* `\clist_gsort:Nn`*. These functions are documented on page 102.*)

13.6 Comma list conditionals

`\clist_if_empty_p:N`
`\clist_if_empty_p:c`
`\clist_if_empty:NTF`
`\clist_if_empty:cTF`

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
  { p , T , F , TF }
```

(*End definition for* `\clist_if_empty:NTF`*. This function is documented on page 102.*)

`\clist_if_empty_p:n`
`\clist_if_empty:nTF`
`\__clist_if_empty_n:w`
`\__clist_if_empty_n:wNw`

As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.

```
\prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
```

```
  {
    \__clist_if_empty_n:w ? #1
    , \q_mark \prg_return_false:
    , \q_mark \prg_return_true:
    \q_stop
  }
\cs_new:Npn \__clist_if_empty_n:w #1 ,
  {
    \tl_if_empty:oTF { \use_none:nn #1 ? }
      { \__clist_if_empty_n:w ? }
      { \__clist_if_empty_n:wNw }
  }
\cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}
```

(*End definition for `\clist_if_empty:nTF`, `\__clist_if_empty_n:w`, and `\__clist_if_empty_n:wNw`. These functions are documented on page 102.*)

`\clist_if_in:NnTF`
`\clist_if_in:NVTF`
`\clist_if_in:NoTF`
`\clist_if_in:cnTF`
`\clist_if_in:cVTF`
`\clist_if_in:coTF`
`\clist_if_in:nnTF`
`\clist_if_in:nVTF`
`\clist_if_in:noTF`
`\__clist_if_in_return:nn`

See description of the `\tl_if_in:Nn` function for details. We simply surround the comma list, and the item, with commas.

```
\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T   , F , TF }
  {
    \exp_args:No \__clist_if_in_return:nn #1 {#2}
  }
\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T   , F , TF }
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
  }
\cs_new_protected:Npn \__clist_if_in_return:nn #1#2
  {
    \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
    \tl_if_empty:oTF
      { \__clist_tmp:w ,#1, {} {} ,#2, }
      { \prg_return_false: } { \prg_return_true: }
  }
\cs_generate_variant:Nn \clist_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnT  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnF  {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:nnT  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnF  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnTF {     nV , no }
```

(*End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `\__clist_if_in_return:nn`. These functions are documented on page 102.*)

13.7 Mapping to comma lists

`\clist_map_function:NN`
`\clist_map_function:cN`
`\__clist_map_function:Nw`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by \q_recursion_tail. The auxiliary function __clist_map_function:Nw is used directly in \clist_map_inline:Nn. Change with care.

```
\cs_new:Npn \clist_map_function:NN #1#2
  {
    \clist_if_empty:NF #1
      {
        \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
          , \q_recursion_tail ,
        \__prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new:Npn \__clist_map_function:Nw #1#2 ,
  {
    \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
    #1 {#2}
    \__clist_map_function:Nw #1
  }
\cs_generate_variant:Nn \clist_map_function:NN { c }
```

(*End definition for* \clist_map_function:NN *and* __clist_map_function:Nw*. These functions are documented on page 103.*)

\clist_map_function:nN
__clist_map_function_n:Nn
__clist_map_unbrace:Nw

The n-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on __clist_trim_spaces_generic:nw. The auxiliary __clist_map_function_n:Nn receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by __clist_map_unbrace:Nw.

```
\cs_new:Npn \clist_map_function:nN #1#2
  {
    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
    \q_mark #1, \q_recursion_tail,
    \__prg_break_point:Nn \clist_map_break: { }
  }
\cs_new:Npn \__clist_map_function_n:Nn #1 #2
  {
    \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
    \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
    \q_mark
  }
\cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

(*End definition for* \clist_map_function:nN *,* __clist_map_function_n:Nn *, and* __clist_map_unbrace:Nw*. These functions are documented on page 103.*)

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

Inline mapping is done by creating a suitable function "on the fly": this is done globally to avoid any issues with TeX's groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the n version simply by storing the comma-list in a variable. We don't need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```
\cs_new_protected:Npn \clist_map_inline:Nn #1#2
  {
    \clist_if_empty:NF #1
      {
        \int_gincr:N \g__prg_map_int
        \cs_gset_protected:cpn
          { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
        \exp_last_unbraced:Nco \__clist_map_function:Nw
          { __prg_map_ \int_use:N \g__prg_map_int :w }
          #1 , \q_recursion_tail ,
        \__prg_break_point:Nn \clist_map_break:
          { \int_gdecr:N \g__prg_map_int }
      }
  }
\cs_new_protected:Npn \clist_map_inline:nn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_inline:Nn \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(*End definition for* `\clist_map_inline:Nn` *and* `\clist_map_inline:nn`*. These functions are documented on page 103.*)

\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn
__clist_map_variable:Nnw

As for other comma-list mappings, filter out the case of an empty list. Same approach as `\clist_map_function:Nn`, additionally we store each item in the given variable. As for inline mappings, space trimming for the `n` variant is done by storing the comma list in a variable.

```
\cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
  {
    \clist_if_empty:NF #1
      {
        \exp_args:Nno \use:nn
          { \__clist_map_variable:Nnw #2 {#3} }
          #1
          , \q_recursion_tail , \q_recursion_stop
        \__prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new_protected:Npn \clist_map_variable:nNn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_variable:NNn \l__clist_internal_clist
  }
\cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
  {
    \tl_set:Nn #1 {#3}
    \quark_if_recursion_tail_stop:N #1
    \use:n {#2}
    \__clist_map_variable:Nnw #1 {#2}
  }
\cs_generate_variant:Nn \clist_map_variable:NNn { c }
```

(*End definition for* `\clist_map_variable:NNn` *,* `\clist_map_variable:nNn` *, and* `\__clist_map_variable:Nnw`*. These functions are documented on page 103.*)

\clist_map_break:
\clist_map_break:n

The break statements use the general __prg_map_break:Nn mechanism.

```
\cs_new:Npn \clist_map_break:
  { \__prg_map_break:Nn \clist_map_break: { } }
\cs_new:Npn \clist_map_break:n
  { \__prg_map_break:Nn \clist_map_break: }
```

(*End definition for* \clist_map_break: *and* \clist_map_break:n*. These functions are documented on page 103.*)

\clist_count:N
\clist_count:c
\clist_count:n
__clist_count:n
__clist_count:w

Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics. In the case of an n-type comma-list, we could of course use \clist_map_function:nN, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not {}, hence the extra spaces).

```
\cs_new:Npn \clist_count:N #1
  {
    \int_eval:n
      {
        0
        \clist_map_function:NN #1 \__clist_count:n
      }
  }
\cs_generate_variant:Nn \clist_count:N { c }
\cs_new:Npx \clist_count:n #1
  {
    \exp_not:N \int_eval:n
      {
        0
        \exp_not:N \__clist_count:w \c_space_tl
        #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
\cs_new:Npn \__clist_count:n #1 { + 1 }
\cs_new:Npx \__clist_count:w #1 ,
  {
    \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
    \exp_not:N \tl_if_blank:nF {#1} { + 1 }
    \exp_not:N \__clist_count:w \c_space_tl
  }
```

(*End definition for* \clist_count:N *and others. These functions are documented on page 104.*)

13.8 Using comma lists

\clist_use:Nnnn
\clist_use:cnnn
__clist_use:wwn
__clist_use:nwwwwnwn
__clist_use:nwwn
\clist_use:Nn
\clist_use:cn

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the ⟨*separator between two*⟩ in the middle.

Otherwise, __clist_use:nwwwwnwn takes the following arguments; 1: a ⟨*separator*⟩, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a ⟨*continuation*⟩ function (use_ii or use_iii with its ⟨*separator*⟩ argument), 7: junk, and 8: the temporary result, which is built in a brace group following \q_stop. The ⟨*separator*⟩ and the first of the three items are placed in the result, then we use the

⟨*continuation*⟩, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\q_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\q_mark` is taken as a third item, and now the second `\q_mark` serves as a delimiter to `use_ii`, switching to the other ⟨*continuation*⟩, `use_iii`, which uses the ⟨*separator between final two*⟩.

```
\cs_new:Npn \clist_use:Nnnn #1#2#3#4
  {
    \clist_if_exist:NTF #1
      {
        \int_case:nnF { \clist_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
            { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
          }
          {
            \exp_after:wN \__clist_use:nwwwwnwn
            \exp_after:wN { \exp_after:wN } #1 ,
            \q_mark , { \__clist_use:nwwwwnwn {#3} }
            \q_mark , { \__clist_use:nwwn {#4} }
            \q_stop { }
          }
      }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \clist_use:Nnnn { c }
\cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
\cs_new:Npn \__clist_use:nwwwwnwn
    #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
  { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
\cs_new:Npn \__clist_use:nwwn #1#2 , #3 \q_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \clist_use:Nn #1#2
  { \clist_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \clist_use:Nn { c }
```

(*End definition for* `\clist_use:Nnnn` *and others. These functions are documented on page 104.*)

13.9 Using a single item

`\clist_item:Nn`
`\clist_item:cn`
`\__clist_item:nnnN`
`\__clist_item:ffoN`
`\__clist_item:ffnN`
`\__clist_item_N_loop:nw`

To avoid needing to test the end of the list at each step, we first compute the ⟨*length*⟩ of the list. If the item number is 0, less than −⟨*length*⟩, or more than ⟨*length*⟩, the result is empty. If it is negative, but not less than −⟨*length*⟩, add ⟨*length*⟩ + 1 to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```
\cs_new:Npn \clist_item:Nn #1#2
  {
    \__clist_item:ffoN
      { \clist_count:N #1 }
      { \int_eval:n {#2} }
      #1
      \__clist_item_N_loop:nw
  }
\cs_new:Npn \__clist_item:nnnN #1#2#3#4
  {
    \int_compare:nNnTF {#2} < 0
      {
        \int_compare:nNnTF {#2} < { - #1 }
          { \use_none_delimit_by_q_stop:w }
          { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
      }
      {
        \int_compare:nNnTF {#2} > {#1}
          { \use_none_delimit_by_q_stop:w }
          { #4 {#2} }
      }
    { } , #3 , \q_stop
  }
\cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
\cs_new:Npn \__clist_item_N_loop:nw #1 #2,
  {
    \int_compare:nNnTF {#1} = 0
      { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
      { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \clist_item:Nn { c }
```

(*End definition for* `\clist_item:Nn`*,* `\__clist_item:nnnN`*, and* `\__clist_item_N_loop:nw`*. These functions are documented on page 106.*)

\clist_item:nn
__clist_item_n:nw
__clist_item_n_loop:nw
__clist_item_n_end:n
__clist_item_n_strip:w

This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```
\cs_new:Npn \clist_item:nn #1#2
  {
    \__clist_item:ffnN
      { \clist_count:n {#1} }
      { \int_eval:n {#2} }
      {#1}
      \__clist_item_n:nw
  }
\cs_new:Npn \__clist_item_n:nw #1
  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
\cs_new:Npn \__clist_item_n_loop:nw #1 #2,
  {
    \exp_args:No \tl_if_blank:nTF {#2}
      { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
      {
        \int_compare:nNnTF {#1} = 0
          { \exp_args:No \__clist_item_n_end:n {#2} }
          {
            \exp_args:Nf \__clist_item_n_loop:nw
              { \int_eval:n { #1 - 1 } }
              \prg_do_nothing:
          }
      }
  }
\cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
  {
    \__tl_trim_spaces:nn { \q_mark #1 }
      { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
  }
\cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(*End definition for* `\clist_item:nn` *and others. These functions are documented on page 106.*)

13.10 Viewing comma lists

`\clist_show:N`
`\clist_show:c`
`\clist_show:n`

Apply the general `\__msg_show_variable:NNNnn`. In the case of an `n`-type comma-list, we must do things by hand, using the same message `show-clist` as for an `N`-type comma-list but with an empty name (first argument).

```
\cs_new_protected:Npn \clist_show:N #1
  {
    \__msg_show_variable:NNNnn #1
      \clist_if_exist:NTF \clist_if_empty:NTF { clist }
      { \clist_map_function:NN #1 \__msg_show_item:n }
  }
\cs_new_protected:Npn \clist_show:n #1
  {
    \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-clist }
      { } { \clist_if_empty:nF {#1} { ? } } { } { }
    \__msg_show_wrap:n
      { \clist_map_function:nN {#1} \__msg_show_item:n }
  }
\cs_generate_variant:Nn \clist_show:N { c }
```

(*End definition for* `\clist_show:N` *and* `\clist_show:n`*. These functions are documented on page 106.*)

`\clist_log:N`
`\clist_log:c`
`\clist_log:n`

Redirect output of `\clist_show:N` and `\clist_show:n` to the log.

```
\cs_new_protected:Npn \clist_log:N
  { \__msg_log_next: \clist_show:N }
\cs_new_protected:Npn \clist_log:n
  { \__msg_log_next: \clist_show:n }
\cs_generate_variant:Nn \clist_log:N { c }
```

(*End definition for* `\clist_log:N` *and* `\clist_log:n`*. These functions are documented on page 107.*)

13.11 Scratch comma lists

`\l_tmpa_clist`
`\l_tmpb_clist`
`\g_tmpa_clist`
`\g_tmpb_clist`

Temporary comma list variables.

```
\clist_new:N \l_tmpa_clist
\clist_new:N \l_tmpb_clist
\clist_new:N \g_tmpa_clist
\clist_new:N \g_tmpb_clist
```

(*End definition for* `\l_tmpa_clist` *and others. These variables are documented on page 107.*)

```
⟨/initex | package⟩
```

14 l3token implementation

⟨*initex | package⟩

⟨@@=char⟩

14.1 Manipulating and interrogating character tokens

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n

Simple wrappers around the primitives.

```
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new_protected:Npn \char_set_catcode:nn #1#2
  {
    \tex_catcode:D \__int_eval:w #1 \__int_eval_end:
      = \__int_eval:w #2 \__int_eval_end:
  }
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \char_value_catcode:n #1
  { \tex_the:D \tex_catcode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_show_value_catcode:n #1
  { \__msg_show_wrap:n { > ~ \char_value_catcode:n {#1} } }
```

(*End definition for* \char_set_catcode:nn *,* \char_value_catcode:n *, and* \char_show_value_catcode:n*. These functions are documented on page 111.*)

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

```
\cs_new_protected:Npn \char_set_catcode_escape:N #1
  { \char_set_catcode:nn { `#1 } { 0 } }
\cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  { \char_set_catcode:nn { `#1 } { 1 } }
\cs_new_protected:Npn \char_set_catcode_group_end:N #1
  { \char_set_catcode:nn { `#1 } { 2 } }
\cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  { \char_set_catcode:nn { `#1 } { 3 } }
\cs_new_protected:Npn \char_set_catcode_alignment:N #1
  { \char_set_catcode:nn { `#1 } { 4 } }
\cs_new_protected:Npn \char_set_catcode_end_line:N #1
  { \char_set_catcode:nn { `#1 } { 5 } }
\cs_new_protected:Npn \char_set_catcode_parameter:N #1
  { \char_set_catcode:nn { `#1 } { 6 } }
\cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  { \char_set_catcode:nn { `#1 } { 7 } }
\cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  { \char_set_catcode:nn { `#1 } { 8 } }
\cs_new_protected:Npn \char_set_catcode_ignore:N #1
  { \char_set_catcode:nn { `#1 } { 9 } }
\cs_new_protected:Npn \char_set_catcode_space:N #1
  { \char_set_catcode:nn { `#1 } { 10 } }
\cs_new_protected:Npn \char_set_catcode_letter:N #1
  { \char_set_catcode:nn { `#1 } { 11 } }
\cs_new_protected:Npn \char_set_catcode_other:N #1
  { \char_set_catcode:nn { `#1 } { 12 } }
\cs_new_protected:Npn \char_set_catcode_active:N #1
  { \char_set_catcode:nn { `#1 } { 13 } }
\cs_new_protected:Npn \char_set_catcode_comment:N #1
  { \char_set_catcode:nn { `#1 } { 14 } }
```

```
\cs_new_protected:Npn \char_set_catcode_invalid:N #1
  { \char_set_catcode:nn { ‘#1 } { 15 } }
```

(*End definition for* `\char_set_catcode_escape:N` *and others. These functions are documented on page 110.*)

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

```
\cs_new_protected:Npn \char_set_catcode_escape:n #1
  { \char_set_catcode:nn {#1} { 0 } }
\cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  { \char_set_catcode:nn {#1} { 1 } }
\cs_new_protected:Npn \char_set_catcode_group_end:n #1
  { \char_set_catcode:nn {#1} { 2 } }
\cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  { \char_set_catcode:nn {#1} { 3 } }
\cs_new_protected:Npn \char_set_catcode_alignment:n #1
  { \char_set_catcode:nn {#1} { 4 } }
\cs_new_protected:Npn \char_set_catcode_end_line:n #1
  { \char_set_catcode:nn {#1} { 5 } }
\cs_new_protected:Npn \char_set_catcode_parameter:n #1
  { \char_set_catcode:nn {#1} { 6 } }
\cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  { \char_set_catcode:nn {#1} { 7 } }
\cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
  { \char_set_catcode:nn {#1} { 8 } }
\cs_new_protected:Npn \char_set_catcode_ignore:n #1
  { \char_set_catcode:nn {#1} { 9 } }
\cs_new_protected:Npn \char_set_catcode_space:n #1
  { \char_set_catcode:nn {#1} { 10 } }
\cs_new_protected:Npn \char_set_catcode_letter:n #1
  { \char_set_catcode:nn {#1} { 11 } }
\cs_new_protected:Npn \char_set_catcode_other:n #1
  { \char_set_catcode:nn {#1} { 12 } }
\cs_new_protected:Npn \char_set_catcode_active:n #1
  { \char_set_catcode:nn {#1} { 13 } }
\cs_new_protected:Npn \char_set_catcode_comment:n #1
  { \char_set_catcode:nn {#1} { 14 } }
\cs_new_protected:Npn \char_set_catcode_invalid:n #1
  { \char_set_catcode:nn {#1} { 15 } }
```

(*End definition for* `\char_set_catcode_escape:n` *and others. These functions are documented on page 110.*)

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n

Pretty repetitive, but necessary!

```
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new_protected:Npn \char_set_mathcode:nn #1#2
  {
    \tex_mathcode:D \__int_eval:w #1 \__int_eval_end:
    = \__int_eval:w #2 \__int_eval_end:
  }
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \char_value_mathcode:n #1
  { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_show_value_mathcode:n #1
  { \__msg_show_wrap:n { > ~ \char_value_mathcode:n {#1} } }
```

```
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new_protected:Npn \char_set_lccode:nn #1#2
  {
    \tex_lccode:D \__int_eval:w #1 \__int_eval_end:
    = \__int_eval:w #2 \__int_eval_end:
  }
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \char_value_lccode:n #1
  { \tex_the:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_show_value_lccode:n #1
  { \__msg_show_wrap:n { > ~ \char_value_lccode:n {#1} } }
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new_protected:Npn \char_set_uccode:nn #1#2
  {
    \tex_uccode:D \__int_eval:w #1 \__int_eval_end:
    = \__int_eval:w #2 \__int_eval_end:
  }
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \char_value_uccode:n #1
  { \tex_the:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_show_value_uccode:n #1
  { \__msg_show_wrap:n { > ~ \char_value_uccode:n {#1} } }
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new_protected:Npn \char_set_sfcode:nn #1#2
  {
    \tex_sfcode:D \__int_eval:w #1 \__int_eval_end:
    = \__int_eval:w #2 \__int_eval_end:
  }
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \char_value_sfcode:n #1
  { \tex_the:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_show_value_sfcode:n #1
  { \__msg_show_wrap:n { > ~ \char_value_sfcode:n {#1} } }
```

(*End definition for* `\char_set_mathcode:nn` *and others. These functions are documented on page 112.*)

\l_char_active_seq
\l_char_special_seq

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for "special" characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```
\seq_new:N \l_char_special_seq
\seq_set_split:Nnn \l_char_special_seq { }
  { \  \" \# \$ \% \& \\ \^ \_ \{ \} \~ }
\seq_new:N \l_char_active_seq
\seq_set_split:Nnn \l_char_active_seq { }
  { \" \$ \& \^ \_ \~ }
```

(*End definition for* `\l_char_active_seq` *and* `\l_char_special_seq`*. These variables are documented on page 112.*)

14.2 Creating character tokens

\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_set_protected:Npn \__char_tmp:nN #1#2
    {
      \cs_new_protected:cpn { #1 :nN } ##1
        {
          \group_begin:
            \char_set_lccode:nn { `\^^@ } { ##1 }
          \tex_lowercase:D { \group_end: #2 ^^@ }
        }
      \cs_new_protected:cpx { #1 :NN } ##1
        { \exp_not:c { #1 : nN } { `##1 } }
    }
  \__char_tmp:nN { char_set_active_eq }  \cs_set_eq:NN
  \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
\group_end:
\cs_generate_variant:Nn \char_set_active_eq:NN  { Nc }
\cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
\cs_generate_variant:Nn \char_set_active_eq:nN  { nc }
\cs_generate_variant:Nn \char_gset_active_eq:nN { nc }
```

(*End definition for* `\char_set_active_eq:NN` *and others. These functions are documented on page 108.*)

`\char_generate:nn`
`\__char_generate:nn`
`\__char_generate_aux:nn`
`\__char_generate_aux:nnw`
`\l__char_tmp_tl`
`\c__char_max_int`
`\__char_generate_invalid_catcode:`

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XƎTeX, LuaTeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```
\__debug_patch_args:nNNpn { { (#1) } { (#2) } }
\cs_new:Npn \char_generate:nn #1#2
  {
    \exp:w \exp_after:wN \__char_generate_aux:w
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
  }
\cs_new:Npn \__char_generate:nn #1#2
  {
    \exp:w \exp_after:wN
      \__char_generate_aux:nnw \exp_after:wN
        { \__int_value:w \__int_eval:w #1 } {#2} \exp_end:
  }
```

Before doing any actual conversion, first some special case filtering. The `\Ucharcat` primitive cannot make active chars, so that is turned off here: if the primitive gets altered then the code is already in place for 8-bit engines and will kick in for LuaTeX too. Spaces are also banned here as LuaTeX emulation only makes normal (charcode 32 spaces. However, `^^@` is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```
\cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
  {
    \if_int_compare:w #2 = 13 \exp_stop_f:
      \__msg_kernel_expandable_error:nn { kernel } { char-active }
    \else:
      \if_int_compare:w #2 = 10 \exp_stop_f:
        \if_int_compare:w #1 =  0 \exp_stop_f:
```

```
     \__msg_kernel_expandable_error:nn { kernel } { char-null-space }
    \else:
     \__msg_kernel_expandable_error:nn { kernel } { char-space }
    \fi:
   \else:
    \if_int_odd:w 0
      \if_int_compare:w #2 < 1  \exp_stop_f: 1 \fi:
      \if_int_compare:w #2 = 5  \exp_stop_f: 1 \fi:
      \if_int_compare:w #2 = 9  \exp_stop_f: 1 \fi:
      \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
     \__msg_kernel_expandable_error:nn { kernel }
      { char-invalid-catcode }
    \else:
     \if_int_odd:w 0
      \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
      \if_int_compare:w #1 > \c__char_max_int 1 \fi: \exp_stop_f:
      \__msg_kernel_expandable_error:nn { kernel }
       { char-out-of-range }
     \else:
      \__char_generate_aux:nnw {#1} {#2}
     \fi:
    \fi:
   \fi:
  \fi:
  \exp_end:
 }
\tl_new:N \l__char_tmp_tl
```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and recent XeTeX releases there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression for speed. The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much.

```
\group_begin:
⟨*package⟩
 \char_set_catcode_active:N \^^L
 \cs_set:Npn ^^L { }
⟨/package⟩
 \char_set_catcode_other:n { 0 }
 \if_int_odd:w 0
   \cs_if_exist:NT \luatex_directlua:D { 1 }
   \cs_if_exist:NT \utex_charcat:D     { 1 } \exp_stop_f:
  \int_const:Nn \c__char_max_int { 1114111 }
  \cs_if_exist:NTF \luatex_directlua:D
   {
    \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
     {
      #3
      \exp_after:wN \exp_end:
      \luatex_directlua:D { l3kernel.charcat(#1, #2) }
     }
   }
   {
    \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
          {
            #3
            \exp_after:wN \exp_end:
            \utex_charcat:D  #1 ~ #2 ~
          }
      }
  \else:
```

For engines where `\Ucharcat` isn't available (or emulated) then we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then `x`-type expanded together into the desired form.

```
      \int_const:Nn \c__char_max_int { 255 }
      \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
      \char_set_catcode_group_begin:n { 0 } % {
      \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
      \char_set_catcode_group_end:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
      \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
      \char_set_catcode_math_toggle:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
```

As TEX is very unhappy if if finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. TEX is happy if the token is hidden inside `\unexpanded` (which needs to be the primitive). The expansion chain here is required so that the conditional gets cleaned up correctly (other code assumes there is exactly one token to skip during the clean-up).

```
      \char_set_catcode_alignment:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl
        {
          \or:
            \etex_unexpanded:D \exp_after:wN
              { \exp_after:wN ^^@ \exp_after:wN }
        }
      \tl_put_right:Nn \l__char_tmp_tl { \or: }
      \char_set_catcode_parameter:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
      \char_set_catcode_math_superscript:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
      \char_set_catcode_math_subscript:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
      \tl_put_right:Nn \l__char_tmp_tl { \or: }
```

For making spaces, there needs to be an `o`-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space. We also set up active tokens although they are (currently) filtered out by the interface layer (`\Ucharcat` cannot make active tokens).

```
      \char_set_catcode_space:n { 0 }
      \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^^@ }
      \char_set_catcode_letter:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
      \char_set_catcode_other:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
      \char_set_catcode_active:n { 0 }
      \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. In package mode, `^^L` is awkward hence this is done in three parts. Notice that at this stage `^^@` is active.

```
      \cs_set_protected:Npn \__char_tmp:n #1
        {
          \char_set_lccode:nn { 0 } {#1}
          \char_set_lccode:nn { 32 } {#1}
          \exp_args:Nx \tex_lowercase:D
            {
              \tl_const:Nn
                \exp_not:c { c__char_ \__int_to_roman:w #1 _tl }
                { \exp_not:o \l__char_tmp_tl }
            }
        }
⟨*package⟩
      \int_step_function:nnnN { 0 }  { 1 } { 11 }  \__char_tmp:n
      \group_begin:
        \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
        \__char_tmp:n { 12 }
      \group_end:
      \int_step_function:nnnN { 13 } { 1 } { 255 } \__char_tmp:n
⟨/package⟩
⟨*initex⟩
      \int_step_function:nnnN { 0 }  { 1 } { 255 }  \__char_tmp:n
⟨/initex⟩
      \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
        {
          #3
          \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_end:
          \exp_after:wN \exp_after:wN
          \if_case:w #2
            \exp_last_unbraced:Nv \exp_stop_f:
              { c__char_ \__int_to_roman:w #1 _tl }
          \fi:
        }
  \fi:
\group_end:
```

(*End definition for* `\char_generate:nn` *and others. These functions are documented on page 109.*)

`\c_catcode_other_space_tl`

Create a space with category code 12: an “other” space.

```
\tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { `\  } { 12 } }
```

(*End definition for* `\c_catcode_other_space_tl`*. This function is documented on page 109.*)

14.3 Generic tokens

```
⟨@@=token⟩
```

`\token_to_meaning:N`
`\token_to_meaning:c`
`\token_to_str:N`
`\token_to_str:c`

These are all defined in l3basics, as they are needed “early”. This is just a reminder!

(*End definition for* `\token_to_meaning:N` *and* `\token_to_str:N`*. These functions are documented on page 113.*)

`\token_new:Nn`

Creates a new token.

```
\cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }
```

(*End definition for* `\token_new:Nn`*. This function is documented on page 113.*)

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`
`\c_catcode_letter_token`
`\c_catcode_other_token`

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that's not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `\__chk_if_free_cs:N` check.

```
\group_begin:
  \__chk_if_free_cs:N \c_group_begin_token
  \tex_global:D \tex_let:D \c_group_begin_token {
  \__chk_if_free_cs:N \c_group_end_token
  \tex_global:D \tex_let:D \c_group_end_token }
  \char_set_catcode_math_toggle:N \*
  \cs_new_eq:NN \c_math_toggle_token *
  \char_set_catcode_alignment:N \*
  \cs_new_eq:NN \c_alignment_token *
  \cs_new_eq:NN \c_parameter_token #
  \cs_new_eq:NN \c_math_superscript_token ^
  \char_set_catcode_math_subscript:N \*
  \cs_new_eq:NN \c_math_subscript_token *
  \__chk_if_free_cs:N \c_space_token
  \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
  \cs_new_eq:NN \c_catcode_letter_token a
  \cs_new_eq:NN \c_catcode_other_token 1
\group_end:
```

(*End definition for* `\c_group_begin_token` *and others. These functions are documented on page 113.*)

`\c_catcode_active_tl`

Not an implicit token!

```
\group_begin:
  \char_set_catcode_active:N \*
  \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
\group_end:
```

(*End definition for* `\c_catcode_active_tl`*. This variable is documented on page 113.*)

14.4 Token conditionals

`\token_if_group_begin_p:N`
`\token_if_group_begin:NTF`

Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
\prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_begin_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_group_begin:NTF`*. This function is documented on page 114.*)

\token_if_group_end_p:N
\token_if_group_end:NTF

Check if token is a end group token. We use the constant \c_group_end_token for this.

```
\prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_end_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* \token_if_group_end:NTF*. This function is documented on page 114.*)

\token_if_math_toggle_p:N
\token_if_math_toggle:NTF

Check if token is a math shift token. We use the constant \c_math_toggle_token for this.

```
\prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_toggle_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* \token_if_math_toggle:NTF*. This function is documented on page 114.*)

\token_if_alignment_p:N
\token_if_alignment:NTF

Check if token is an alignment tab token. We use the constant \c_alignment_token for this.

```
\prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_alignment_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* \token_if_alignment:NTF*. This function is documented on page 114.*)

\token_if_parameter_p:N
\token_if_parameter:NTF

Check if token is a parameter token. We use the constant \c_parameter_token for this. We have to trick TeX a bit to avoid an error message: within a group we prevent \c_parameter_token from behaving like a macro parameter character. The definitions of \prg_new_conditional:Npnn are global, so they remain after the group.

```
\group_begin:
\cs_set_eq:NN \c_parameter_token \scan_stop:
\prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_parameter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\group_end:
```

(*End definition for* \token_if_parameter:NTF*. This function is documented on page 114.*)

\token_if_math_superscript_p:N
token_if_math_superscript:NTF

Check if token is a math superscript token. We use the constant \c_math_superscript_token for this.

```
\prg_new_conditional:Npnn \token_if_math_superscript:N #1
  { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_superscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* \token_if_math_superscript:NTF*. This function is documented on page 114.*)

\token_if_math_subscript_p:N
\token_if_math_subscript:N*TF*

Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.

```
\prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_subscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_math_subscript:NTF. This function is documented on page 114.*)

\token_if_space_p:N
\token_if_space:N*TF*

Check if token is a space token. We use the constant \c_space_token for this.

```
\prg_new_conditional:Npnn \token_if_space:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_space_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_space:NTF. This function is documented on page 114.*)

\token_if_letter_p:N
\token_if_letter:N*TF*

Check if token is a letter token. We use the constant \c_catcode_letter_token for this.

```
\prg_new_conditional:Npnn \token_if_letter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_letter:NTF. This function is documented on page 115.*)

\token_if_other_p:N
\token_if_other:N*TF*

Check if token is an other char token. We use the constant \c_catcode_other_token for this.

```
\prg_new_conditional:Npnn \token_if_other:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_other_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_other:NTF. This function is documented on page 115.*)

\token_if_active_p:N
\token_if_active:N*TF*

Check if token is an active char token. We use the constant \c_catcode_active_tl for this. A technical point is that \c_catcode_active_tl is in fact a macro expanding to \exp_not:N *, where * is active.

```
\prg_new_conditional:Npnn \token_if_active:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_active:NTF. This function is documented on page 115.*)

\token_if_eq_meaning_p:NN
\token_if_eq_meaning:NN*TF*

Check if the tokens #1 and #2 have same meaning.

```
\prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T ,  F , TF }
  {
    \if_meaning:w  #1  #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_meaning:NNTF`*. This function is documented on page 115.*)

`\token_if_eq_catcode_p:NN`
`\token_if_eq_catcode:NNTF`

Check if the tokens `#1` and `#2` have same category code.

```
\prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_catcode:NNTF`*. This function is documented on page 115.*)

`\token_if_eq_charcode_p:NN`
`\token_if_eq_charcode:NNTF`

Check if the tokens `#1` and `#2` have same character code.

```
\prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_charcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_charcode:NNTF`*. This function is documented on page 115.*)

`\token_if_macro_p:N`
`\token_if_macro:NTF`
`\__token_if_macro_p:w`

When a token is a macro, `\token_to_meaning:N` always outputs something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. However, this can fail the five `\...mark` primitives, whose meaning has the form `...mark:`⟨*user material*⟩. The problem is that the ⟨*user material*⟩ can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in LaTeX3) before the string `macro:`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character ` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m`...

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```
\use:x
  {
    \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
      { p , T ,  F , TF }
      {
        \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
        \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
          \exp_not:N \q_stop
      }
    \cs_new:Npn \exp_not:N  \__token_if_macro_p:w
      ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
  }
      {
        \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = 0 \exp_stop_f:
            \prg_return_true:
        \else:
            \prg_return_false:
        \fi:
      }
```

(*End definition for* `\token_if_macro:NTF` *and* `\__token_if_macro_p:w`*. These functions are documented on page 115.*)

`\token_if_cs_p:N`
`\token_if_cs:NTF`

Check if token has same catcode as a control sequence. This follows the same pattern as for `\token_if_letter:N` *etc.* We use `\scan_stop:` for this.

```
\prg_new_conditional:Npnn \token_if_cs:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \scan_stop:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_cs:NTF`*. This function is documented on page 115.*)

`\token_if_expandable_p:N`
`\token_if_expandable:NTF`

Check if token is expandable. We use the fact that TEX temporarily converts `\exp_not:N` ⟨*token*⟩ into `\scan_stop:` if ⟨*token*⟩ is expandable. An `undefined` token is not considered as expandable. No problem nesting the conditionals, since the third `#1` is only skipped if it is non-expandable (hence not part of TEX's conditional apparatus).

```
\prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T ,  F , TF }
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

(*End definition for* `\token_if_expandable:NTF`*. This function is documented on page 115.*)

`\__token_delimit_by_char":w`
`\__token_delimit_by_count:w`
`\__token_delimit_by_dimen:w`
`\__token_delimit_by_macro:w`
`\__token_delimit_by_muskip:w`
`\__token_delimit_by_skip:w`
`\__token_delimit_by_toks:w`

These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\q_stop`, and returns the first one and its delimiter. This result is eventually compared to another string.

```
\group_begin:
\cs_set_protected:Npn \__token_tmp:w #1
  {
    \use:x
      {
        \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
            ####1 \tl_to_str:n {#1} ####2 \exp_not:N \q_stop
          { ####1 \tl_to_str:n {#1} }
      }
  }
\__token_tmp:w { char" }
\__token_tmp:w { count }
\__token_tmp:w { dimen }
\__token_tmp:w { macro }
\__token_tmp:w { muskip }
\__token_tmp:w { skip }
\__token_tmp:w { toks }
\group_end:
```

(*End definition for* `\__token_delimit_by_char":w` *and others.*)

\token_if_chardef_p:N
\token_if_chardef:NTF
\token_if_mathchardef_p:N
\token_if_mathchardef:NTF
\token_if_long_macro_p:N
\token_if_long_macro:NTF
\token_if_protected_macro_p:N
\token_if_protected_macro:NTF
\token_if_protected_long_macro_p:N
\token_if_protected_long_macro:NTF
\token_if_dim_register_p:N
\token_if_dim_register:NTF
\token_if_int_register_p:N
\token_if_int_register:NTF
\token_if_muskip_register_p:N
\token_if_muskip_register:NTF
\token_if_skip_register_p:N
\token_if_skip_register:NTF
\token_if_toks_register_p:N
\token_if_toks_register:NTF

Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\__str_if_eq_x_return:nn` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within `x`-expansion. The temporary function `\__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first five conditionals, `\cs_if_exist:cT` turns out to be `false`, and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two TeX primitives which would wrongly be recognized as registers otherwise. Despite using TeX's primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not TeX conditionals).

```
\group_begin:
\cs_set_protected:Npn \__token_tmp:w #1#2#3
  {
    \use:x
      {
        \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
          { p , T ,  F , TF }
          {
            \cs_if_exist:cT { tex_ #2 :D }
              {
                \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }
                \exp_not:N \prg_return_false:
                \exp_not:N \else:
                \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 def:D }
                \exp_not:N \prg_return_false:
                \exp_not:N \else:
              }
            \exp_not:N \__str_if_eq_x_return:nn
              {
                \exp_not:N \exp_after:wN
                \exp_not:c { __token_delimit_by_ #2 :w }
                \exp_not:N \token_to_meaning:N ####1
                ? \tl_to_str:n {#2} \exp_not:N \q_stop
              }
              { \exp_not:n {#3} }
            \cs_if_exist:cT { tex_ #2 :D }
              {
                \exp_not:N \fi:
                \exp_not:N \fi:
              }
          }
      }
  }
\__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
\__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
\__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
\__token_tmp:w { protected_macro } { macro }
  { \tl_to_str:n { \protected } macro }
\__token_tmp:w { protected_long_macro } { macro }
  { \token_to_str:N \protected \tl_to_str:n { \long } macro }
\__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
\__token_tmp:w { int_register } { count } { \token_to_str:N \count }
\__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
\__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
\__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
\group_end:
```

(*End definition for* `\token_if_chardef:NTF` *and others. These functions are documented on page 116.*)

\token_if_primitive_p:N
\token_if_primitive:NTF
__token_if_primitive:NNw
__token_if_primitive_space:w
__token_if_primitive_nullfont:N
__token_if_primitive_loop:N
__token_if_primitive:Nw
__token_if_primitive_undefined:N

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., `the letter A`), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form ⟨*letters*⟩`:`⟨*user material*⟩. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than `` `A `` (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or \tex_undefined:D, and if it is " or a digit, then the token is not a primitive.

```
\tex_chardef:D \c__token_A_int = ‘A ~ %
\use:x
  {
    \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
      { p , T , F , TF }
      {
        \exp_not:N \token_if_macro:NTF ##1
          \exp_not:N \prg_return_false:
          {
            \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
            \exp_not:N \token_to_meaning:N ##1
              \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
          }
      }
    \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
      ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
      {
        \exp_not:N \tl_if_empty:oTF
          { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
          {
            \exp_not:N \__token_if_primitive_loop:N ##3
              \c_colon_str \exp_not:N \q_stop
          }
          { \exp_not:N \__token_if_primitive_nullfont:N }
      }
  }
\cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
\cs_new:Npn \__token_if_primitive_nullfont:N #1
  {
    \if_meaning:w \tex_nullfont:D #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__token_if_primitive_loop:N #1
  {
    \if_int_compare:w ‘#1 < \c__token_A_int %
      \exp_after:wN \__token_if_primitive:Nw
      \exp_after:wN #1
    \else:
      \exp_after:wN \__token_if_primitive_loop:N
    \fi:
  }
\cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
  {
    \if:w : #1
      \exp_after:wN \__token_if_primitive_undefined:N
    \else:
      \prg_return_false:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new:Npn \__token_if_primitive_undefined:N #1
  {
    \if_cs_exist:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\token_if_primitive:NTF` *and others. These functions are documented on page 117.*)

14.5 Peeking ahead at the next token

```
⟨@@=peek⟩
```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

`\l_peek_token`
`\g_peek_token`

Storage tokens which are publicly documented: the token peeked.

```
\cs_new_eq:NN \l_peek_token ?
\cs_new_eq:NN \g_peek_token ?
```

(*End definition for* `\l_peek_token` *and* `\g_peek_token`*. These variables are documented on page 117.*)

`\l__peek_search_token`

The token to search for as an implicit token: *cf.* `\l__peek_search_tl`.

```
\cs_new_eq:NN \l__peek_search_token ?
```

(*End definition for* `\l__peek_search_token`*.*)

`\l__peek_search_tl`

The token to search for as an explicit token: *cf.* `\l__peek_search_token`.

```
\tl_new:N \l__peek_search_tl
```

(*End definition for* `\l__peek_search_tl`*.*)

`\__peek_true:w`
`\__peek_true_aux:w`
`\__peek_false:w`
`\__peek_tmp:w`

Functions used by the branching and space-stripping code.

```
\cs_new:Npn \__peek_true:w  { }
\cs_new:Npn \__peek_true_aux:w  { }
\cs_new:Npn \__peek_false:w { }
\cs_new:Npn \__peek_tmp:w { }
```

(*End definition for* `\__peek_true:w` *and others.*)

`\peek_after:Nw`
`\peek_gafter:Nw`

Simple wrappers for `\futurelet`: no arguments absorbed here.

```
\cs_new_protected:Npn \peek_after:Nw
  { \tex_futurelet:D \l_peek_token }
\cs_new_protected:Npn \peek_gafter:Nw
  { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(*End definition for* \peek_after:Nw *and* \peek_gafter:Nw*. These functions are documented on page 117.*)

__peek_true_remove:w

A function to remove the next token and then regain control.

```
\cs_new_protected:Npn \__peek_true_remove:w
  {
    \tex_afterassignment:D \__peek_true_aux:w
    \cs_set_eq:NN \__peek_tmp:w
  }
```

(*End definition for* __peek_true_remove:w*.*)

__peek_token_generic_aux:NNNTF

The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, `#1` is `\__peek_true_remove:w` when removing the token and `\__peek_true_aux:w` otherwise.

```
\cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
  {
    \group_align_safe_begin:
    \cs_set_eq:NN \l__peek_search_token #3
    \tl_set:Nn \l__peek_search_tl {#3}
    \cs_set:Npx \__peek_true_aux:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#4}
      }
    \cs_set_eq:NN \__peek_true:w #1
    \cs_set:Npx \__peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#5}
      }
    \peek_after:Nw #2
  }
```

(*End definition for* __peek_token_generic_aux:NNNTF*.*)

__peek_token_generic:NN*TF*
__peek_token_remove_generic:NN*TF*

For token removal there needs to be a call to the auxiliary function which does the work.

```
\cs_new_protected:Npn \__peek_token_generic:NNTF
  { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
\cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
  { \__peek_token_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
  { \__peek_token_generic:NNTF #1 #2 { } {#3} }
\cs_new_protected:Npn \__peek_token_remove_generic:NNTF
  { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
\cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* __peek_token_generic:NNTF *and* __peek_token_remove_generic:NNTF*.*)

\__peek_execute_branches_meaning:

The meaning test is straight forward.

```
\cs_new:Npn \__peek_execute_branches_meaning:
  {
    \if_meaning:w \l_peek_token \l__peek_search_token
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End definition for* \__peek_execute_branches_meaning:*.*)

\__peek_execute_branches_catcode:
\__peek_execute_branches_charcode:
\__peek_execute_branches_catcode_aux:
\__peek_execute_branches_catcode_auxii:N
\__peek_execute_branches_catcode_auxiii:

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which are only given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TEX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non-LATEX3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```
\cs_new:Npn \__peek_execute_branches_catcode:
  { \if_catcode:w \__peek_execute_branches_catcode_aux: }
\cs_new:Npn \__peek_execute_branches_charcode:
  { \if_charcode:w \__peek_execute_branches_catcode_aux: }
\cs_new:Npn \__peek_execute_branches_catcode_aux:
  {
        \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__peek_execute_branches_catcode_auxii:N
          \exp_after:wN \exp_not:N
        \else:
          \exp_after:wN \__peek_execute_branches_catcode_auxiii:
        \fi:
  }
\cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
  {
        \exp_not:N #1
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
    #1
  }
\cs_new:Npn \__peek_execute_branches_catcode_auxiii:
  {
        \exp_not:N \l_peek_token
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End definition for* `\__peek_execute_branches_catcode:` *and others.*)

`\__peek_ignore_spaces_execute_branches:`

This function removes one space token at a time, and calls `\__peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non-LaTeX3 packages). Spaces are removed using a side-effect of `f`-expansion: `\exp:w \exp_end_continue_f:w` removes one space.

```
\cs_new_protected:Npn \__peek_ignore_spaces_execute_branches:
  {
    \if_meaning:w \l_peek_token \c_space_token
      \exp_after:wN \peek_after:Nw
      \exp_after:wN \__peek_ignore_spaces_execute_branches:
      \exp:w \exp_end_continue_f:w
    \else:
      \exp_after:wN \__peek_execute_branches:
    \fi:
  }
```

(*End definition for* `\__peek_ignore_spaces_execute_branches:`*.*)

`\__peek_def:nnnn`
`\__peek_def:nnnnn`

The public functions themselves cannot be defined using `\prg_new_conditional:Npnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```
\group_begin:
  \cs_set:Npn \__peek_def:nnnn #1#2#3#4
    {
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
    }
  \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
    {
      \cs_new_protected:cpx { #1 #5 }
        {
          \tl_if_empty:nF {#2}
            { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
          \exp_not:c { #3 #5 }
          \exp_not:n {#4}
        }
    }
```

(*End definition for* `\__peek_def:nnnn` *and* `\__peek_def:nnnnn`*.*)

`\peek_catcode:NTF`
`\peek_catcode_ignore_spaces:NTF`
`\peek_catcode_remove:NTF`
`\peek_catcode_remove_ignore_spaces:NTF`

With everything in place the definitions can take place. First for category codes.

```
\__peek_def:nnnn { peek_catcode:N }
  { }
  { __peek_token_generic:NN }
  { \__peek_execute_branches_catcode: }
\__peek_def:nnnn { peek_catcode_ignore_spaces:N }
  { \__peek_execute_branches_catcode: }
  { __peek_token_generic:NN }
  { \__peek_ignore_spaces_execute_branches: }
\__peek_def:nnnn { peek_catcode_remove:N }
  { }
  { __peek_token_remove_generic:NN }
  { \__peek_execute_branches_catcode: }
\__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
  { \__peek_execute_branches_catcode: }
  { __peek_token_remove_generic:NN }
  { \__peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_catcode:NTF` *and others. These functions are documented on page 117.*)

`\peek_charcode:NTF`
`\peek_charcode_ignore_spaces:NTF`
`\peek_charcode_remove:NTF`
`\peek_charcode_remove_ignore_spaces:NTF`

Then for character codes.

```
\__peek_def:nnnn { peek_charcode:N }
  { }
  { __peek_token_generic:NN }
  { \__peek_execute_branches_charcode: }
\__peek_def:nnnn { peek_charcode_ignore_spaces:N }
  { \__peek_execute_branches_charcode: }
  { __peek_token_generic:NN }
  { \__peek_ignore_spaces_execute_branches: }
\__peek_def:nnnn { peek_charcode_remove:N }
  { }
  { __peek_token_remove_generic:NN }
  { \__peek_execute_branches_charcode: }
\__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
  { \__peek_execute_branches_charcode: }
  { __peek_token_remove_generic:NN }
  { \__peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_charcode:NTF` *and others. These functions are documented on page 118.*)

`\peek_meaning:NTF`
`\peek_meaning_ignore_spaces:NTF`
`\peek_meaning_remove:NTF`
`\peek_meaning_remove_ignore_spaces:NTF`

Finally for meaning, with the group closed to remove the temporary definition functions.

```
\__peek_def:nnnn { peek_meaning:N }
  { }
  { __peek_token_generic:NN }
  { \__peek_execute_branches_meaning: }
\__peek_def:nnnn { peek_meaning_ignore_spaces:N }
  { \__peek_execute_branches_meaning: }
  { __peek_token_generic:NN }
  { \__peek_ignore_spaces_execute_branches: }
\__peek_def:nnnn { peek_meaning_remove:N }
  { }
  { __peek_token_remove_generic:NN }
```

```
    { \__peek_execute_branches_meaning: }
  \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
    { \__peek_execute_branches_meaning: }
    { __peek_token_remove_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
\group_end:
```

(*End definition for* `\peek_meaning:NTF` *and others. These functions are documented on page 119.*)

14.6 Decomposing a macro definition

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
__peek_get_prefix_arg_replacement:wN

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```
\exp_args:Nno \use:nn
  { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
  { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
  { #4 {#1} {#2} {#3} }
\cs_new:Npn \token_get_prefix_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_i:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_arg_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_ii:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_replacement_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_iii:nnn
      }
      { \scan_stop: }
  }
```

(*End definition for* `\token_get_prefix_spec:N` *and others. These functions are documented on page 120.*)

```
⟨/initex | package⟩
```

15 l3prop implementation

The following test files are used for this code: m3prop001, m3prop002, m3prop003, m3prop004, m3show001.

```
⟨*initex | package⟩
```

```
⟨@@=prop⟩
```

A property list is a macro whose top-level expansion is of the form

```
\s__prop \__prop_pair:wn ⟨key1⟩ \s__prop {⟨value1⟩}
...
\__prop_pair:wn ⟨keyn⟩ \s__prop {⟨valuen⟩}
```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `\__prop_pair:wn` can be used to map through the property list.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
\__scan_new:N \s__prop
```

(*End definition for* `\s__prop`*.*)

`\__prop_pair:wn` The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```
\cs_new:Npn \__prop_pair:wn #1 \s__prop #2
  { \__msg_kernel_expandable_error:nn { kernel } { misused-prop } }
```

(*End definition for* `\__prop_pair:wn`*.*)

`\l__prop_internal_tl` Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

```
\tl_new:N \l__prop_internal_tl
```

(*End definition for* `\l__prop_internal_tl`*.*)

`\c_empty_prop` An empty prop.

```
\tl_const:Nn \c_empty_prop { \s__prop }
```

(*End definition for* `\c_empty_prop`*. This variable is documented on page 128.*)

15.1 Allocation and initialisation

`\prop_new:N`
`\prop_new:c`
Property lists are initialized with the value `\c_empty_prop`.

```
\cs_new_protected:Npn \prop_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_prop
  }
\cs_generate_variant:Nn \prop_new:N { c }
```

(*End definition for* `\prop_new:N`*. This function is documented on page 123.*)

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

The same idea for clearing.

```
\cs_new_protected:Npn \prop_clear:N  #1
  { \prop_set_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_clear:N  { c }
\cs_new_protected:Npn \prop_gclear:N #1
  { \prop_gset_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_gclear:N { c }
```

(*End definition for* `\prop_clear:N` *and* `\prop_gclear:N`*. These functions are documented on page 123.*)

\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c

Once again a simple variation of the token list functions.

```
\cs_new_protected:Npn \prop_clear_new:N  #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_clear_new:N  { c }
\cs_new_protected:Npn \prop_gclear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(*End definition for* `\prop_clear_new:N` *and* `\prop_gclear_new:N`*. These functions are documented on page 123.*)

\prop_set_eq:NN
\prop_set_eq:cN
\prop_set_eq:Nc
\prop_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:cN
\prop_gset_eq:Nc
\prop_gset_eq:cc

These are simply copies from the token list functions.

```
\cs_new_eq:NN \prop_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \prop_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \prop_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \prop_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\prop_set_eq:NN` *and* `\prop_gset_eq:NN`*. These functions are documented on page 123.*)

\l_tmpa_prop
\l_tmpb_prop
\g_tmpa_prop
\g_tmpb_prop

We can now initialize the scratch variables.

```
\prop_new:N \l_tmpa_prop
\prop_new:N \l_tmpb_prop
\prop_new:N \g_tmpa_prop
\prop_new:N \g_tmpb_prop
```

(*End definition for* `\l_tmpa_prop` *and others. These variables are documented on page 128.*)

15.2 Accessing data in property lists

__prop_split:NnTF
__prop_split_aux:NnTF
__prop_split_aux:w

This function is used by most of the module, and hence must be fast. It receives a ⟨*property list*⟩, a ⟨*key*⟩, a ⟨*true code*⟩ and a ⟨*false code*⟩. The aim is to split the ⟨*property list*⟩ at the given ⟨*key*⟩ into the ⟨*extract*$_1$⟩ before the key–value pair, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*$_2$⟩ after the key–value pair. This is done using a delimited function, whose definition is as follows, where the ⟨*key*⟩ is turned into a string.

```
\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn ⟨key⟩ \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {⟨true code⟩} {⟨false code⟩} }
```

If the ⟨*key*⟩ is present in the property list, __prop_split_aux:w's #1 is the part before the ⟨*key*⟩, #2 is the ⟨*value*⟩, #3 is the part after the ⟨*key*⟩, #4 is \use_i:nn, and #5 is additional tokens that we do not care about. The ⟨*true code*⟩ is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 __prop_pair:wn ⟨*key*⟩ \s__prop {#2} #3.

If the ⟨*key*⟩ is not there, then the ⟨*function*⟩ is \use_ii:nn, which keeps the ⟨*false code*⟩.

```
\cs_new_protected:Npn \__prop_split:NnTF #1#2
  { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
\cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
  {
    \cs_set:Npn \__prop_split_aux:w ##1
      \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
      { ##4 {#3} {#4} }
    \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
      \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
  }
\cs_new:Npn \__prop_split_aux:w { }
```

(*End definition for __prop_split:NnTF, __prop_split_aux:NnTF, and __prop_split_aux:w.*)

\prop_remove:Nn
\prop_remove:NV
\prop_remove:cn
\prop_remove:cV
\prop_gremove:Nn
\prop_gremove:NV
\prop_gremove:cn
\prop_gremove:cV

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```
\cs_new_protected:Npn \prop_remove:Nn #1#2
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_set:Nn #1 { ##1 ##3 } }
      { }
  }
\cs_new_protected:Npn \prop_gremove:Nn #1#2
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_gset:Nn #1 { ##1 ##3 } }
      { }
  }
\cs_generate_variant:Nn \prop_remove:Nn  {     NV }
\cs_generate_variant:Nn \prop_remove:Nn  { c , cV }
\cs_generate_variant:Nn \prop_gremove:Nn {     NV }
\cs_generate_variant:Nn \prop_gremove:Nn { c , cV }
```

(*End definition for \prop_remove:Nn and \prop_gremove:Nn. These functions are documented on page 125.*)

\prop_get:NnN
\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:coN

Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to \q_no_value.

```
\cs_new_protected:Npn \prop_get:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_set:Nn #3 {##2} }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \prop_get:NnN {     NV , No }
\cs_generate_variant:Nn \prop_get:NnN { c , cV , co }
```

(*End definition for* \prop_get:NnN. *This function is documented on page 124.*)

\prop_pop:NnN
\prop_pop:NoN
\prop_pop:cnN
\prop_pop:coN
\prop_gpop:NnN
\prop_gpop:NoN
\prop_gpop:cnN
\prop_gpop:coN

Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save \q_no_value in the token list.

```
\cs_new_protected:Npn \prop_pop:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_set:Nn #1 { ##1 ##3 }
      }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_gpop:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_gset:Nn #1 { ##1 ##3 }
      }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \prop_pop:NnN  {     No }
\cs_generate_variant:Nn \prop_pop:NnN  { c , co }
\cs_generate_variant:Nn \prop_gpop:NnN {     No }
\cs_generate_variant:Nn \prop_gpop:NnN { c , co }
```

(*End definition for* \prop_pop:NnN *and* \prop_gpop:NnN. *These functions are documented on page 124.*)

\prop_item:Nn
\prop_item:cn
__prop_item_Nn:nwwn

Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one ⟨*key*⟩–⟨*value*⟩ pair at a time: the arguments of __prop_item_Nn:nwn are the ⟨*key*⟩ we are looking for, a ⟨*key*⟩ of the property list, and its associated value. The ⟨*keys*⟩ are compared (as strings). If they match, the ⟨*value*⟩ is returned, within \exp_not:n. The loop terminates even if the ⟨*key*⟩ is missing, and yields an empty value, because we have appended the appropriate ⟨*key*⟩–⟨*empty value*⟩ pair to the property list.

```
\cs_new:Npn \prop_item:Nn #1#2
  {
    \exp_last_unbraced:Noo \__prop_item_Nn:nwwn { \tl_to_str:n {#2} } #1
      \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
    \__prg_break_point:
  }
\cs_new:Npn \__prop_item_Nn:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \str_if_eq_x:nnTF {#1} {#3}
      { \__prg_break:n { \exp_not:n {#4} } }
      { \__prop_item_Nn:nwwn {#1} }
  }
\cs_generate_variant:Nn \prop_item:Nn { c }
```

(*End definition for* \prop_item:Nn *and* __prop_item_Nn:nwwn. *These functions are documented on page 125.*)

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*
\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, \prg_return_true: is used after the assignments.

```
\prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_set:Nn #1 { ##1 ##3 }
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_gset:Nn #1 { ##1 ##3 }
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \prop_pop:NnNT   { c }
\cs_generate_variant:Nn \prop_pop:NnNF   { c }
\cs_generate_variant:Nn \prop_pop:NnNTF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNT  { c }
\cs_generate_variant:Nn \prop_gpop:NnNF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNTF { c }
```

(*End definition for \prop_pop:NnNTF and \prop_gpop:NnNTF. These functions are documented on page 126.*)

\prop_put:Nnn
\prop_put:NnV
\prop_put:Nno
\prop_put:Nnx
\prop_put:NVn
\prop_put:NVV
\prop_put:Non
\prop_put:Noo
\prop_put:cnn
\prop_put:cnV
\prop_put:cno
\prop_put:cnx
\prop_put:cVn
\prop_put:cVV
\prop_put:con
\prop_put:coo
\prop_gput:Nnn
\prop_gput:NnV
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:NVn
\prop_gput:NVV
\prop_gput:Non
\prop_gput:Noo
\prop_gput:cnn
\prop_gput:cnV
\prop_gput:cno
\prop_gput:cnx
\prop_gput:cVn
\prop_gput:cVV

Since the branches of __prop_split:NnTF are used as the replacement text of an internal macro, and since the ⟨*key*⟩ and new ⟨*value*⟩ may contain arbitrary tokens, it is not safe to include them in the argument of __prop_split:NnTF. We thus start by storing in \l__prop_internal_tl tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in __prop_split:NnTF. If the ⟨*key*⟩ was absent, append the new key–value to the list. Otherwise concatenate the extracts ##1 and ##3 with the new key–value pair \l__prop_internal_tl. The updated entry is placed at the same spot as the original ⟨*key*⟩ in the property list, preserving the order of entries.

```
\cs_new_protected:Npn \prop_put:Nnn  { \__prop_put:NNnn \tl_set:Nx }
\cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
  {
    \tl_set:Nn \l__prop_internal_tl
      {
        \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
        \s__prop { \exp_not:n {#4} }
      }
    \__prop_split:NnTF #2 {#3}
      { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
  }
\cs_generate_variant:Nn \prop_put:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_put:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
\cs_generate_variant:Nn \prop_gput:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_gput:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
```

(*End definition for* `\prop_put:Nnn`*,* `\prop_gput:Nnn`*, and* `\__prop_put:NNnn`*. These functions are documented on page 124.*)

`\prop_put_if_new:Nnn`
`\prop_put_if_new:cnn`
`\prop_gput_if_new:Nnn`
`\prop_gput_if_new:cnn`
`\__prop_put_if_new:NNnn`

Adding conditionally also splits. If the key is already present, the three brace groups given by `\__prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```
\cs_new_protected:Npn \prop_put_if_new:Nnn
  { \__prop_put_if_new:NNnn \tl_set:Nx }
\cs_new_protected:Npn \prop_gput_if_new:Nnn
  { \__prop_put_if_new:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
  {
    \tl_set:Nn \l__prop_internal_tl
      {
        \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
        \s__prop \exp_not:n { {#4} }
      }
    \__prop_split:NnTF #2 {#3}
      { }
      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
  }
\cs_generate_variant:Nn \prop_put_if_new:Nnn  { c }
\cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }
```

(*End definition for* `\prop_put_if_new:Nnn`*,* `\prop_gput_if_new:Nnn`*, and* `\__prop_put_if_new:NNnn`*. These functions are documented on page 124.*)

15.3 Property list conditionals

`\prop_if_exist_p:N`
`\prop_if_exist_p:c`
`\prop_if_exist:NTF`
`\prop_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* `\prop_if_exist:NTF`*. This function is documented on page 125.*)

`\prop_if_empty_p:N`
`\prop_if_empty_p:c`
`\prop_if_empty:NTF`
`\prop_if_empty:cTF`

Same test as for token lists.

```
\prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
  {
    \tl_if_eq:NNTF #1 \c_empty_prop
      \prg_return_true: \prg_return_false:
  }
\cs_generate_variant:Nn \prop_if_empty_p:N { c }
```

```
\cs_generate_variant:Nn \prop_if_empty:NT { c }
\cs_generate_variant:Nn \prop_if_empty:NF { c }
\cs_generate_variant:Nn \prop_if_empty:NTF { c }
```

(*End definition for* `\prop_if_empty:NTF`. *This function is documented on page 125.*)

\prop_if_in_p:Nn
\prop_if_in_p:NV
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:co
\prop_if_in:Nn*TF*
\prop_if_in:NV*TF*
\prop_if_in:No*TF*
\prop_if_in:cn*TF*
\prop_if_in:cV*TF*
\prop_if_in:co*TF*
__prop_if_in:nwwn
__prop_if_in:N

Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```
\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
  {
    \@@_split:NnTF #1 {#2}
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

but `\__prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq_x:nn`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq_x:nn`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `\__prop_if_in:nwwn` is most often empty. When the ⟨*key*⟩ is found in the list, `\__prop_if_in:N` receives `\__prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```
\prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
  {
    \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
      \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
      \q_recursion_tail
    \__prg_break_point:
  }
\cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \str_if_eq_x:nnTF {#1} {#3}
      { \__prop_if_in:N }
      { \__prop_if_in:nwwn {#1} }
  }
\cs_new:Npn \__prop_if_in:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
    \__prg_break:
  }
\cs_generate_variant:Nn \prop_if_in_p:Nn {     NV , No }
\cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnT  { c , cV , co }
```

```
\cs_generate_variant:Nn \prop_if_in:NnF  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }
```

(*End definition for* `\prop_if_in:NnTF`*,* `\__prop_if_in:nwwn`*, and* `\__prop_if_in:N`*. These functions are documented on page 125.*)

15.4 Recovering values from property lists with branching

`\prop_get:NnNTF`
`\prop_get:NVNTF`
`\prop_get:NoNTF`
`\prop_get:cnNTF`
`\prop_get:cVNTF`
`\prop_get:coNTF`

Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```
\prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \prop_get:NnNT  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNF  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNTF {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNT  { c , cV , co }
\cs_generate_variant:Nn \prop_get:NnNF  { c , cV , co }
\cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }
```

(*End definition for* `\prop_get:NnNTF`*. This function is documented on page 126.*)

15.5 Mapping to property lists

`\prop_map_function:NN`
`\prop_map_function:Nc`
`\prop_map_function:cN`
`\prop_map_function:cc`
`\__prop_map_function:Nwwn`

The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`. A special case to note is when the key #3 is empty: then `\q_recursion_tail` is compared to `\exp_after:wN`, also different. Note that #2 is empty, except at the first iteration, where it is `\s__prop`.

```
\cs_new:Npn \prop_map_function:NN #1#2
  {
    \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
      \__prop_pair:wn \q_recursion_tail \s__prop { }
    \__prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \if_meaning:w \q_recursion_tail #3
      \exp_after:wN \prop_map_break:
    \fi:
    #1 {#3} {#4}
    \__prop_map_function:Nwwn #1
  }
\cs_generate_variant:Nn \prop_map_function:NN {     Nc }
\cs_generate_variant:Nn \prop_map_function:NN { c , cc }
```

(*End definition for* `\prop_map_function:NN` *and* `\__prop_map_function:Nwwn`*. These functions are documented on page 126.*)

`\prop_map_inline:Nn`
`\prop_map_inline:cn`

Mapping in line requires a nesting level counter. Store the current definition of `\__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `\__prop_pair:wn` ⟨*key*⟩ `\s__prop {`⟨*value*⟩`}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

```
\cs_new_protected:Npn \prop_map_inline:Nn #1#2
  {
    \cs_gset_eq:cN
      { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
    \int_gincr:N \g__prg_map_int
    \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
    #1
    \__prg_break_point:Nn \prop_map_break:
      {
        \int_gdecr:N \g__prg_map_int
        \cs_gset_eq:Nc \__prop_pair:wn
          { __prg_map_ \int_use:N \g__prg_map_int :wn }
      }
  }
\cs_generate_variant:Nn \prop_map_inline:Nn { c }
```

(*End definition for* `\prop_map_inline:Nn`*. This function is documented on page 126.*)

`\prop_map_break:`
`\prop_map_break:n`

The break statements are based on the general `\__prg_map_break:Nn`.

```
\cs_new:Npn \prop_map_break:
  { \__prg_map_break:Nn \prop_map_break: { } }
\cs_new:Npn \prop_map_break:n
  { \__prg_map_break:Nn \prop_map_break: }
```

(*End definition for* `\prop_map_break:` *and* `\prop_map_break:n`*. These functions are documented on page 127.*)

15.6 Viewing property lists

`\prop_show:N`
`\prop_show:c`

Apply the general `\__msg_show_variable:NNNnn`. Contrarily to sequences and comma lists, we use `\__msg_show_item:nn` to format both the key and the value for each pair.

```
\cs_new_protected:Npn \prop_show:N #1
  {
    \__msg_show_variable:NNNnn #1
      \prop_if_exist:NTF \prop_if_empty:NTF { prop }
      { \prop_map_function:NN #1 \__msg_show_item:nn }
  }
\cs_generate_variant:Nn \prop_show:N { c }
```

(*End definition for* `\prop_show:N`*. This function is documented on page 127.*)

`\prop_log:N`
`\prop_log:c`

Redirect output of `\prop_show:N` to the log.

```
\cs_new_protected:Npn \prop_log:N
  { \__msg_log_next: \prop_show:N }
\cs_generate_variant:Nn \prop_log:N { c }
```

(*End definition for* \prop_log:N. *This function is documented on page 127.*)

```
⟨/initex | package⟩
```

16 l3msg implementation

```
⟨*initex | package⟩
⟨@@=msg⟩
```

\l__msg_internal_tl A general scratch for the module.

```
\tl_new:N \l__msg_internal_tl
```

(*End definition for* \l__msg_internal_tl.)

\l__msg_line_context_bool Controls whether the line context is shown as part of the decoration of all (non-expandable) messages.

```
\bool_new:N \l__msg_line_context_bool
```

(*End definition for* \l__msg_line_context_bool.)

16.1 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

\c__msg_text_prefix_tl \c__msg_more_text_prefix_tl Locations for the text of messages.

```
\tl_const:Nn \c__msg_text_prefix_tl      { msg~text~>~ }
\tl_const:Nn \c__msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(*End definition for* \c__msg_text_prefix_tl *and* \c__msg_more_text_prefix_tl.)

\msg_if_exist_p:nn \msg_if_exist:nnTF Test whether the control sequence containing the message text exists or not.

```
\prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* \msg_if_exist:nnTF. *This function is documented on page 130.*)

__chk_if_free_msg:nn This auxiliary is similar to __chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```
\__debug_patch:nnNNpn { }
  { \__debug_log:x { Defining~message~ #1 / #2 ~\msg_line_context: } }
\cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
  {
    \msg_if_exist:nnT {#1} {#2}
      {
        \__msg_kernel_error:nnxx { kernel } { message-already-defined }
          {#1} {#2}
      }
  }
```

(*End definition for* `\__chk_if_free_msg:nn`*.*)

`\msg_new:nnnn`
`\msg_new:nnn`
`\msg_gset:nnnn`
`\msg_gset:nnn`
`\msg_set:nnnn`
`\msg_set:nnn`

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
\cs_new_protected:Npn \msg_new:nnnn #1#2
  {
    \__chk_if_free_msg:nn {#1} {#2}
    \msg_gset:nnnn {#1} {#2}
  }
\cs_new_protected:Npn \msg_new:nnn #1#2#3
  { \msg_new:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
  {
    \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_set:nnn #1#2#3
  { \msg_set:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
  {
    \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_gset:nnn #1#2#3
  { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(*End definition for* `\msg_new:nnnn` *and others. These functions are documented on page 129.*)

16.2 Messages: support functions and text

`\c__msg_coding_error_text_tl`
`\c__msg_continue_text_tl`
`\c__msg_critical_text_tl`
`\c__msg_fatal_text_tl`
`\c__msg_help_text_tl`
`\c__msg_no_info_text_tl`
`\c__msg_on_line_text_tl`
`\c__msg_return_text_tl`
`\c__msg_trouble_text_tl`

Simple pieces of text for messages.

```
\tl_const:Nn \c__msg_coding_error_text_tl
  {
    This~is~a~coding~error.
    \\ \\
  }
\tl_const:Nn \c__msg_continue_text_tl
  { Type~<return>~to~continue }
\tl_const:Nn \c__msg_critical_text_tl
  { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
\tl_const:Nn \c__msg_fatal_text_tl
  { This~is~a~fatal~error:~LaTeX~will~abort. }
\tl_const:Nn \c__msg_help_text_tl
  { For~immediate~help~type~H~<return> }
\tl_const:Nn \c__msg_no_info_text_tl
  {
    LaTeX~does~not~know~anything~more~about~this~error,~sorry.
    \c__msg_return_text_tl
  }
\tl_const:Nn \c__msg_on_line_text_tl { on~line }
```

```
\tl_const:Nn \c__msg_return_text_tl
  {
    \\ \\
    Try~typing~<return>~to~proceed.
    \\
    If~that~doesn't~work,~type~X~<return>~to~quit.
  }
\tl_const:Nn \c__msg_trouble_text_tl
  {
    \\ \\
    More~errors~will~almost~certainly~follow: \\
    the~LaTeX~run~should~be~aborted.
  }
```

(*End definition for* `\c__msg_coding_error_text_tl` *and others.*)

\msg_line_number:
\msg_line_context:

For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not `new`.

```
\cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
\cs_gset:Npn \msg_line_context:
  {
    \c__msg_on_line_text_tl
    \c_space_tl
    \msg_line_number:
  }
```

(*End definition for* `\msg_line_number:` *and* `\msg_line_context:`*. These functions are documented on page 130.*)

16.3 Showing messages: low level mechanism

\msg_interrupt:nnn

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`.

```
\cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      {
        \__msg_interrupt_wrap:nn { \\ \c__msg_no_info_text_tl }
          {#1 \\\\ #2 \\\\ \c__msg_continue_text_tl }
      }
      {
        \__msg_interrupt_wrap:nn { \\ #3 }
          {#1 \\\\ #2 \\\\ \c__msg_help_text_tl }
      }
  }
```

(*End definition for* `\msg_interrupt:nnn`*. This function is documented on page 134.*)

__msg_interrupt_wrap:nn
__msg_interrupt_more_text:n

First setup TeX's `\errhelp` register with the extra help `#1`, then build a nice-looking error message with `#2`. Everything is done using `x`-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion.

```
\cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
  {
    \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
    \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
  }
\cs_new_protected:Npn \__msg_interrupt_more_text:n #1
  {
    \exp_args:Nx \tex_errhelp:D
      {
        |'''''''''''''''''''''''''''''''''''''''''''''''
        #1 \iow_newline:
        |...............................................
      }
  }
```

(*End definition for* `\__msg_interrupt_wrap:nn` *and* `\__msg_interrupt_more_text:n`.)

`\__msg_interrupt_text:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made "invisible": TEX's own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {`⟨*dots*⟩`}` which fills the output with dots. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `\__iow_with:Nnn` auxiliary, defined in l3file, expects an ⟨*integer variable*⟩, an integer ⟨*value*⟩, and some ⟨*code*⟩. It runs the ⟨*code*⟩ after ensuring that the ⟨*integer variable*⟩ takes the given ⟨*value*⟩, then restores the former value of the ⟨*integer variable*⟩ if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is −1, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```
\group_begin:
  \char_set_lccode:nn {`\{} {`\ }
  \char_set_lccode:nn {`\}} {`\ }
  \char_set_lccode:nn {`\&} {`\!}
  \char_set_catcode_active:N \&
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \__msg_interrupt_text:n #1
      {
        \iow_term:x
          {
            \iow_newline:
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            \iow_newline:
```

```
            !
          }
        \__iow_with:Nnn \tex_newlinechar:D { `\^^J }
          {
            \__iow_with:Nnn \tex_errorcontextlines:D { -1 }
              {
                \group_begin:
                  \cs_set_protected:Npn &
                    {
                      \tex_errmessage:D
                        {
                          #1
                          \use_none:n
                            { ............................................ }
                        }
                    }
                  \exp_after:wN
                \group_end:
                &
              }
          }
      }
  }
```

(*End definition for* `\__msg_interrupt_text:n`*.*)

`\msg_log:n`
`\msg_term:n`

Printing to the log or terminal without a stop is rather easier. A bit of simple visual work sets things off nicely.

```
\cs_new_protected:Npn \msg_log:n #1
  {
    \iow_log:n { ................................................. }
    \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
    \iow_log:n { ................................................. }
  }
\cs_new_protected:Npn \msg_term:n #1
  {
    \iow_term:n { ************************************************* }
    \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
    \iow_term:n { ************************************************* }
  }
```

(*End definition for* `\msg_log:n` *and* `\msg_term:n`*. These functions are documented on page 135.*)

16.4 Displaying messages

LaTeX is handling error messages and so the TeX ones are disabled. This is already done by the LaTeX 2ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```
⟨*initex⟩
\int_gset:Nn \tex_errorcontextlines:D { -1 }
⟨/initex⟩
```

`\msg_fatal_text:n`
`\msg_critical_text:n`
`\msg_error_text:n`
`\msg_warning_text:n`
`\msg_info_text:n`

A function for issuing messages: both the text and order could in principle vary.

```
\cs_new:Npn \msg_fatal_text:n #1
```

```
  {
    Fatal~#1~error
    \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
  }
\cs_new:Npn \msg_critical_text:n #1
  {
    Critical~#1~error
    \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
  }
\cs_new:Npn \msg_error_text:n #1
  {
    #1~error
    \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
  }
\cs_new:Npn \msg_warning_text:n #1
  {
    #1~warning
    \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
  }
\cs_new:Npn \msg_info_text:n #1
  {
    #1~info
    \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
  }
```

(*End definition for* `\msg_fatal_text:n` *and others. These functions are documented on page 130.*)

\msg_see_documentation_text:n

Contextual footer information. The LaTeX module only comprises LaTeX3 code, so we refer to the LaTeX3 documentation rather than simply "LaTeX".

```
\cs_new:Npn \msg_see_documentation_text:n #1
  {
    \\ \\ See~the~
    \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
    documentation~for~further~information.
  }
```

(*End definition for* `\msg_see_documentation_text:n`*. This function is documented on page 131.*)

__msg_class_new:nn

```
\group_begin:
  \cs_set_protected:Npn \__msg_class_new:nn #1#2
    {
      \prop_new:c { l__msg_redirect_ #1 _prop }
      \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
          ##1##2##3##4##5##6 {#2}
      \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
                { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
                { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
            }
        }
      \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
      \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
      \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:N \exp_not:n
                { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
                {##3} {##4} {##5} {##6}
            }
        }
      \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
        { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
        { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
        { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
    }
```

(*End definition for* `\__msg_class_new:nn`.)

\msg_fatal:nnnnnn
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn

For fatal errors, after the error message TeX bails out.

```
\__msg_class_new:nn { fatal }
  {
    \msg_interrupt:nnn
      { \msg_fatal_text:n {#1} : ~ "#2" }
      {
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c__msg_fatal_text_tl }
    \tex_end:D
  }
```

(*End definition for* `\msg_fatal:nnnnnn` *and others. These functions are documented on page 131.*)

\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

Not quite so bad: just end the current file.

```
\__msg_class_new:nn { critical }
  {
    \msg_interrupt:nnn
      { \msg_critical_text:n {#1} : ~ "#2" }
      {
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c__msg_critical_text_tl }
    \tex_endinput:D
  }
```

(*End definition for* `\msg_critical:nnnnnn` *and others. These functions are documented on page 131.*)

\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn
__msg_error:cnnnnn
__msg_no_more_text:nnnn

For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text.

```
\__msg_class_new:nn { error }
  {
    \__msg_error:cnnnnn
      { \c__msg_more_text_prefix_tl #1 / #2 }
      {#3} {#4} {#5} {#6}
      {
        \msg_interrupt:nnn
          { \msg_error_text:n {#1} : ~ "#2" }
          {
            \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
            \msg_see_documentation_text:n {#1}
          }
     }
  }
\cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
  {
    \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
      { #6 { } }
      { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
  }
\cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }
```

(*End definition for* `\msg_error:nnnnnn` *and others. These functions are documented on page 132.*)

\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

Warnings are printed to the terminal.

```
\__msg_class_new:nn { warning }
  {
    \msg_term:n
      {
        \msg_warning_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_warning:nnnnnn` *and others. These functions are documented on page 132.*)

\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn

Information only goes into the log.

```
\__msg_class_new:nn { info }
  {
    \msg_log:n
      {
        \msg_info_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_info:nnnnnn` *and others. These functions are documented on page 132.*)

\msg_log:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn

“Log” data is very similar to information, but with no extras added.

```
  \__msg_class_new:nn { log }
    {
      \iow_wrap:nnnN
        { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
        { } { } \iow_log:n
    }
```

(*End definition for* \msg_log:nnnnnn *and others. These functions are documented on page 132.*)

\msg_none:nnnnnn
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_none:nnx
\msg_none:nn

The none message type is needed so that input can be gobbled.

```
  \__msg_class_new:nn { none } { }
```

(*End definition for* \msg_none:nnnnnn *and others. These functions are documented on page 133.*)

End the group to eliminate __msg_class_new:nn.

```
\group_end:
```

__msg_class_chk_exist:nT

Checking that a message class exists. We build this from \cs_if_free:cTF rather than \cs_if_exist:cTF because that avoids reading the second argument earlier than necessary.

```
\cs_new:Npn \__msg_class_chk_exist:nT #1
  {
    \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
      { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
  }
```

(*End definition for* __msg_class_chk_exist:nT*.*)

\l__msg_class_tl
\l__msg_current_class_tl

Support variables needed for the redirection system.

```
\tl_new:N \l__msg_class_tl
\tl_new:N \l__msg_current_class_tl
```

(*End definition for* \l__msg_class_tl *and* \l__msg_current_class_tl*.*)

\l__msg_redirect_prop

For redirection of individually-named messages

```
\prop_new:N \l__msg_redirect_prop
```

(*End definition for* \l__msg_redirect_prop*.*)

\l__msg_hierarchy_seq

During redirection, split the message name into a sequence with items {/module/submodule}, {/module}, and {}.

```
\seq_new:N \l__msg_hierarchy_seq
```

(*End definition for* \l__msg_hierarchy_seq*.*)

\l__msg_class_loop_seq

Classes encountered when following redirections to check for loops.

```
\seq_new:N \l__msg_class_loop_seq
```

(*End definition for* \l__msg_class_loop_seq*.*)

__msg_use:nnnnnnn
__msg_use_redirect_name:n
__msg_use_hierarchy:nwwN
__msg_use_redirect_module:n
__msg_use_code:

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to `\__msg_use_code:` is similar to `\tl_set:Nn`. The message is eventually produced with whatever `\l__msg_class_tl` is when `\__msg_use_code:` is called.

```
\cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
  {
    \msg_if_exist:nnTF {#2} {#3}
      {
        \__msg_class_chk_exist:nT {#1}
          {
            \tl_set:Nn \l__msg_current_class_tl {#1}
            \cs_set_protected:Npx \__msg_use_code:
              {
                \exp_not:n
                  {
                    \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
                      {#2} {#3} {#4} {#5} {#6} {#7}
                  }
              }
            \__msg_use_redirect_name:n { #2 / #3 }
          }
      }
      { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
  }
\cs_new_protected:Npn \__msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into `module/submodule/message` (with an arbitrary number of slashes), and store `{/module/submodule}`, `{/module}` and `{}` into `\l__msg_hierarchy_seq`. We then map through this sequence, applying the most specific redirection.

```
\cs_new_protected:Npn \__msg_use_redirect_name:n #1
  {
    \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
      { \__msg_use_code: }
      {
        \seq_clear:N \l__msg_hierarchy_seq
        \__msg_use_hierarchy:nwwN { }
          #1 \q_mark \__msg_use_hierarchy:nwwN
          /  \q_mark \use_none_delimit_by_q_stop:w
          \q_stop
        \__msg_use_redirect_module:n { }
      }
  }
\cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
  {
    \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
    #4 { #1 / #2 } #3 \q_mark #4
  }
```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `\__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, /module for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module ##1. The loop is interrupted after testing for a redirection for ##1 equal to the argument #1 (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as ##1.

```
\cs_new_protected:Npn \__msg_use_redirect_module:n #1
  {
    \seq_map_inline:Nn \l__msg_hierarchy_seq
      {
        \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
          {##1} \l__msg_class_tl
          {
            \seq_map_break:n
              {
                \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
                  { \__msg_use_code: }
                  {
                    \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
                    \__msg_use_redirect_module:n {##1}
                  }
              }
          }
          {
            \str_if_eq:nnT {##1} {#1}
              {
                \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
                \seq_map_break:n { \__msg_use_code: }
              }
          }
      }
  }
```

(*End definition for* __msg_use:nnnnnnn *and others.*)

\msg_redirect_name:nnn

Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```
\cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
      {
        \__msg_class_chk_exist:nT {#3}
          { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
      }
  }
```

(*End definition for* \msg_redirect_name:nnn*. This function is documented on page 134.*)

\msg_redirect_class:nn
\msg_redirect_module:nnn
__msg_redirect:nnn
__msg_redirect_loop_chk:nnn
__msg_redirect_loop_list:n

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in \l__msg_current_class_tl.

```
\cs_new_protected:Npn \msg_redirect_class:nn
  { \__msg_redirect:nnn { } }
\cs_new_protected:Npn \msg_redirect_module:nnn #1
  { \__msg_redirect:nnn { / #1 } }
\cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
  {
    \__msg_class_chk_exist:nT {#2}
      {
        \tl_if_empty:nTF {#3}
          { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
          {
            \__msg_class_chk_exist:nT {#3}
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
                \tl_set:Nn \l__msg_current_class_tl {#2}
                \seq_clear:N \l__msg_class_loop_seq
                \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
              }
          }
      }
  }
```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with \l__msg_class_tl, and keep track in \l__msg_class_loop_seq of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```
\cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
  {
    \seq_put_right:Nn \l__msg_class_loop_seq {#1}
    \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
      {
        \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
          {
            \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
                \__msg_kernel_warning:nnxxxx
                  { kernel } { message-redirect-loop }
                  { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
                  { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
                  {#3}
                  {
                    \seq_map_function:NN \l__msg_class_loop_seq
                      \__msg_redirect_loop_list:n
                    { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
                  }
              }
              { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
          }
      }
  }
\cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
\cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }
```

(*End definition for* `\msg_redirect_class:nn` *and others. These functions are documented on page 133.*)

16.5 Kernel-specific functions

__msg_kernel_new:nnnn
__msg_kernel_new:nnn
__msg_kernel_set:nnnn
__msg_kernel_set:nnn

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```
\cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
  { \msg_new:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
  { \msg_new:nnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
  { \msg_set:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
  { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(*End definition for* `\__msg_kernel_new:nnnn` *and others.*)

__msg_kernel_class_new:nN
__msg_kernel_class_new_aux:nN

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `\__msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```
\group_begin:
  \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
    { \__msg_kernel_class_new_aux:nN { kernel_ #1 } }
  \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
    {
      \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
                { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
                { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
            }
        }
      \cs_new_protected:cpx { __msg_ #1 :nnnnn } ##1##2##3##4##5
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { __msg_ #1 :nnnn } ##1##2##3##4
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnn } ##1##2##3
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nn } ##1##2
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
        {
          \use:x
```

```
            {
              \exp_not:N \exp_not:n
                { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
                {##3} {##4} {##5} {##6}
            }
        }
      \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
    }
```

(*End definition for* `\__msg_kernel_class_new:nN` *and* `\__msg_kernel_class_new_aux:nN`*.*)

__msg_kernel_fatal:nnnnnn
__msg_kernel_fatal:nnxxxx
__msg_kernel_fatal:nnnnn
__msg_kernel_fatal:nnxxx
__msg_kernel_fatal:nnnn
__msg_kernel_fatal:nnxx
__msg_kernel_fatal:nnn
__msg_kernel_fatal:nnx
__msg_kernel_fatal:nn
__msg_kernel_error:nnnnnn
__msg_kernel_error:nnxxxx
__msg_kernel_error:nnnnn
__msg_kernel_error:nnxxx
__msg_kernel_error:nnnn
__msg_kernel_error:nnxx
__msg_kernel_error:nnn
__msg_kernel_error:nnx
__msg_kernel_error:nn

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the "LaTeX" module name. Three functions are already defined by l3basics; we need to undefine them to avoid errors.

```
    \__msg_kernel_class_new:nN { fatal } \__msg_fatal_code:nnnnnn
    \cs_undefine:N \__msg_kernel_error:nnxx
    \cs_undefine:N \__msg_kernel_error:nnx
    \cs_undefine:N \__msg_kernel_error:nn
    \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn
```

(*End definition for* `\__msg_kernel_fatal:nnnnnn` *and others.*)

__msg_kernel_warning:nnnnnn
__msg_kernel_warning:nnxxxx
__msg_kernel_warning:nnnnn
__msg_kernel_warning:nnxxx
__msg_kernel_warning:nnnn
__msg_kernel_warning:nnxx
__msg_kernel_warning:nnn
__msg_kernel_warning:nnx
__msg_kernel_warning:nn
__msg_kernel_info:nnnnnn
__msg_kernel_info:nnxxxx
__msg_kernel_info:nnnnn
__msg_kernel_info:nnxxx
__msg_kernel_info:nnnn
__msg_kernel_info:nnxx
__msg_kernel_info:nnn
__msg_kernel_info:nnx
__msg_kernel_info:nn

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name "LaTeX".

```
    \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
    \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx
```

(*End definition for* `\__msg_kernel_warning:nnnnnn` *and others.*)

End the group to eliminate `\__msg_kernel_class_new:nN`.

```
  \group_end:
```

Error messages needed to actually implement the message system itself.

```
  \__msg_kernel_new:nnnn { kernel } { message-already-defined }
    { Message~'#2'~for~module~'#1'~already~defined. }
    {
      \c__msg_coding_error_text_tl
      LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
      by~the~module~'#1':~this~message~already~exists.
      \c__msg_return_text_tl
    }
  \__msg_kernel_new:nnnn { kernel } { message-unknown }
    { Unknown~message~'#2'~for~module~'#1'. }
    {
      \c__msg_coding_error_text_tl
      LaTeX~was~asked~to~display~a~message~called~'#2'\\
      by~the~module~'#1':~this~message~does~not~exist.
      \c__msg_return_text_tl
    }
  \__msg_kernel_new:nnnn { kernel } { message-class-unknown }
    { Unknown~message~class~'#1'. }
    {
      LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
      this~was~never~defined.
      \c__msg_return_text_tl
    }
  \__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
    {
      Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
      \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
    }
    {
      Adding~the~message~redirection~ {#1} ~=>~ {#2}
      \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
      created~an~infinite~loop\\\\
      \iow_indent:n { #4 \\\\ }
    }
```

Messages for earlier kernel modules.

```
\__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
  { Function~'#1'~cannot~be~defined~with~#2~arguments. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~a~function~'#1'~with~
    #2~arguments.~
    TeX~allows~between~0~and~9~arguments~for~a~single~function.
  }
\__msg_kernel_new:nnn { kernel } { char-active }
  { Cannot~generate~active~chars. }
\__msg_kernel_new:nnn { kernel } { char-invalid-catcode }
  { Invalid~catcode~for~char~generation. }
\__msg_kernel_new:nnn { kernel } { char-null-space }
  { Cannot~generate~null~char~as~a~space. }
\__msg_kernel_new:nnn { kernel } { char-out-of-range }
  { Charcode~requested~out~of~engine~range. }
\__msg_kernel_new:nnn { kernel } { char-space }
  { Cannot~generate~space~chars. }
\__msg_kernel_new:nnnn { kernel } { command-already-defined }
  { Control~sequence~#1~already~defined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
    but~this~name~has~already~been~used~elsewhere. \\ \\
    The~current~meaning~is:\\
    \ \ #2
  }
\__msg_kernel_new:nnnn { kernel } { command-not-defined }
  { Control~sequence~#1~undefined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\
    this~has~not~been~defined~yet.
  }
\__msg_kernel_new:nnn { kernel } { deprecated-command }
  {
    The~deprecated~command~'#2'~has~been~or~will~be~removed~on~#1.
    \tl_if_empty:nF {#3} { ~Use~instead~’#3’. }
  }
\__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
  { Empty~search~pattern. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~replace~an~empty~pattern~by~’#1’:~that~
    would~lead~to~an~infinite~loop!
  }
\__msg_kernel_new:nnnn { kernel } { out-of-registers }
  { No~room~for~a~new~#1. }
  {
    TeX~only~supports~\int_use:N \c_max_register_int \ %
    of~each~type.~All~the~#1~registers~have~been~used.~
    This~run~will~be~aborted~now.
  }
\__msg_kernel_new:nnnn { kernel } { non-base-function }
  { Function~’#1’~is~not~a~base~function }
  {
    \c__msg_coding_error_text_tl
    Functions~defined~through~\iow_char:N\\cs_new:Nn~must~have~
    a~signature~consisting~of~only~normal~arguments~’N’~and~’n’.~
    To~define~variants~use~\iow_char:N\\cs_generate_variant:Nn~
    and~to~define~other~functions~use~\iow_char:N\\cs_new:Npn.
  }
\__msg_kernel_new:nnnn { kernel } { missing-colon }
  { Function~’#1’~contains~no~’:’. }
  {
    \c__msg_coding_error_text_tl
    Code-level~functions~must~contain~’:’~to~separate~the~
    argument~specification~from~the~function~name.~This~is~
    needed~when~defining~conditionals~or~variants,~or~when~building~a~
    parameter~text~from~the~number~of~arguments~of~the~function.
  }
\__msg_kernel_new:nnnn { kernel } { overflow }
  { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
  {
    An~attempt~was~made~to~store~#3~at~position~#2~in~the~array~’#1’.~
    The~largest~allowed~value~#4~will~be~used~instead.
  }
\__msg_kernel_new:nnnn { kernel } { out-of-bounds }
  { Access~to~an~entry~beyond~an~array’s~bounds. }
  {
    An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
    array~’#1’,~but~this~array~has~entries~at~positions~from~1~to~#3.
  }
\__msg_kernel_new:nnnn { kernel } { protected-predicate }
  { Predicate~’#1’~must~be~expandable. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~’#1’~as~a~protected~predicate.~
    Only~expandable~tests~can~have~a~predicate~version.
  }
\__msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
```

```
  { Conditional~form~'#1'~for~function~'#2'~unknown. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
    the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
  }
\__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
  { Scan~mark~#1~already~defined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
    but~this~name~has~already~been~used~for~a~scan~mark.
  }
\__msg_kernel_new:nnnn { kernel } { variable-not-defined }
  { Variable~#1~undefined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
    been~defined~yet.
  }
\__msg_kernel_new:nnnn { kernel } { variant-too-long }
  { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~that~is~longer~than~
    the~signature~(part~after~the~colon)~of~'#2'.
  }
\__msg_kernel_new:nnnn { kernel } { invalid-variant }
  { Variant~form~'#1'~invalid~for~base~form~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
    from~type~'#3'~to~type~'#4'.
  }
```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```
⟨*package⟩
\bool_if:NTF \l@expl@enable@debug@bool
  {
    \__msg_kernel_new:nnnn { kernel } { debug }
      { The~debugging~option~'#1'~does~not~exist~\msg_line_context:. }
      {
        The~functions~'\iow_char:N\\debug_on:n'~and~
        '\iow_char:N\\debug_off:n'~only~accept~the~arguments~
        'check-declarations',~'deprecation',~'log-functions',~not~'#1'.
      }
    \__msg_kernel_new:nnn { kernel } { expr } { '#2'~in~#1 }
    \__msg_kernel_new:nnnn { kernel } { non-declared-variable }
      { The~variable~#1~has~not~been~declared~\msg_line_context:. }
      {
```

```
    Checking~is~active,~and~you~have~tried~do~so~something~like: \\
    \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
    without~first~having: \\
    \ \ \tl_new:N ~ #1  \\
    \\
    LaTeX~will~create~the~variable~and~continue.
   }
 }
 {
  \__msg_kernel_new:nnnn { kernel } { enable-debug }
   { To~use~'#1'~load~expl3~with~the~'enable-debug'~option. }
   {
    The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
    some~internal~functions~in~expl3~have~been~appropriately~
    defined.~This~only~happens~if~one~of~the~options~
    'enable-debug',~'check-declarations'~or~'log-functions'~was~
    given~when~loading~expl3.
   }
 }
⟨/package⟩
⟨*initex⟩
\__msg_kernel_new:nnnn { kernel } { enable-debug }
  { '#1'~cannot~be~used~in~format~mode. }
  {
    The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
    some~internal~functions~in~expl3~have~been~appropriately~
    defined.~This~only~happens~in~package~mode~(and~only~if~one~of~
    the~options~'enable-debug',~'check-declarations'~or~'log-functions'~
    was~given~when~loading~expl3.
  }
⟨/initex⟩
```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
\__msg_kernel_new:nnn { kernel } { bad-variable }
  { Erroneous~variable~#1 used! }
\__msg_kernel_new:nnn { kernel } { misused-sequence }
  { A~sequence~was~misused. }
\__msg_kernel_new:nnn { kernel } { misused-prop }
  { A~property~list~was~misused. }
\__msg_kernel_new:nnn { kernel } { negative-replication }
  { Negative~argument~for~\prg_replicate:nn. }
\__msg_kernel_new:nnn { kernel } { unknown-comparison }
  { Relation~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
\__msg_kernel_new:nnn { kernel } { zero-step }
  { Zero~step~size~for~step~function~#1. }
```

Messages used by the "show" functions.

```
\__msg_kernel_new:nnn { kernel } { show-clist }
  {
    The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
    \tl_if_empty:nTF {#2}
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
```

```
\__msg_kernel_new:nnn { kernel } { show-prop }
  {
    The~property~list~#1~
    \tl_if_empty:nTF {#2}
      { is~empty }
      { contains~the~pairs~(without~outer~braces): }
  }
\__msg_kernel_new:nnn { kernel } { show-seq }
  {
    The~sequence~#1~
    \tl_if_empty:nTF {#2}
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
\__msg_kernel_new:nnn { kernel } { show-streams }
  {
    \tl_if_empty:nTF {#2} { No~ } { The~following~ }
    \str_case:nn {#1}
      {
        { ior } { input ~ }
        { iow } { output ~ }
      }
    streams~are~
    \tl_if_empty:nTF {#2} { open } { in~use: }
  }
```

16.6 Expandable errors

__msg_expandable_error:n
__msg_expandable_error:w

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```
<argument> \LaTeX3 error:
                             The error message.
```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error:` ⟨*error message*⟩. Then `\__msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3~error:` from being globally equal to `\scan_stop:`.

```
\group_begin:
\cs_set_protected:Npn \__msg_tmp:w #1#2
  {
    \cs_new:Npn \__msg_expandable_error:n ##1
      {
        \exp:w
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__msg_expandable_error:w
        \exp_after:wN \exp_after:wN
        \exp_after:wN \exp_end:
        \use:n { #1 #2 ##1 } #2
      }
    \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
  }
\exp_args:Ncx \__msg_tmp:w { LaTeX3~error: }
  { \char_generate:nn { `\  } { 7 } }
\group_end:
```

(*End definition for* `\__msg_expandable_error:n` *and* `\__msg_expandable_error:w`*.*)

__msg_kernel_expandable_error:nnnnnn
__msg_kernel_expandable_error:nnnnn
__msg_kernel_expandable_error:nnnn
__msg_kernel_expandable_error:nnn
__msg_kernel_expandable_error:nn

The command built from the csname `\c_@@_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `\__msg_expandable_error:n`.

```
\cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nf \__msg_expandable_error:n
      {
        \exp_args:NNc \exp_after:wN \exp_stop_f:
          { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
          {#3} {#4} {#5} {#6}
      }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} {#4} {#5} { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} {#4} { } { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnn #1#2#3
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} { } { } { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} { } { } { } { }
  }
```

(*End definition for* `\__msg_kernel_expandable_error:nnnnnn` *and others.*)

16.7 Showing variables

Functions defined in this section are used for diagnostic functions in l3clist, l3file, l3prop, l3seq, xtemplate

\g__msg_log_next_bool
__msg_log_next:

```
\bool_new:N \g__msg_log_next_bool
\cs_new_protected:Npn \__msg_log_next:
  { \bool_gset_true:N \g__msg_log_next_bool }
```

(*End definition for* `\g__msg_log_next_bool` *and* `\__msg_log_next:`*.*)

__msg_show_pre:nnnnnn
__msg_show_pre:nnxxxx
__msg_show_pre:nnnnnV
__msg_show_pre_aux:n

Print the text of a message to the terminal or log file without formatting: short cuts around \iow_wrap:nnnN. The choice of terminal or log file is done by __msg_show_pre_aux:n.

```
\cs_new_protected:Npn \__msg_show_pre:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nx \iow_wrap:nnnN
      {
        \exp_not:c { \c__msg_text_prefix_tl #1 / #2 }
          { \tl_to_str:n {#3} }
          { \tl_to_str:n {#4} }
          { \tl_to_str:n {#5} }
          { \tl_to_str:n {#6} }
      }
      { } { } \__msg_show_pre_aux:n
  }
\cs_new_protected:Npn \__msg_show_pre:nnxxxx #1#2#3#4#5#6
  {
    \use:x
      { \exp_not:n { \__msg_show_pre:nnnnnn {#1} {#2} } {#3} {#4} {#5} {#6} }
  }
\cs_generate_variant:Nn \__msg_show_pre:nnnnnn { nnnnnV }
\cs_new_protected:Npn \__msg_show_pre_aux:n
  { \bool_if:NTF \g__msg_log_next_bool { \iow_log:n } { \iow_term:n } }
```

(*End definition for* __msg_show_pre:nnnnnn *and* __msg_show_pre_aux:n.)

__msg_show_variable:NNNnn

The arguments of __msg_show_variable:NNNnn are

- The ⟨*variable*⟩ to be shown as #1.
- An ⟨*if-exist*⟩ conditional #2 with NTF signature.
- An ⟨*if-empty*⟩ conditional #3 or other function with NTF signature (sometimes \use_ii:nnn).
- The ⟨*message*⟩ #4 to use.
- A construction #5 which produces the formatted string eventually passed to the \showtokens primitive. Typically this is a mapping of the form \seq_map_function:NN ⟨*variable*⟩ __msg_show_item:n.

If ⟨*if-exist*⟩ ⟨*variable*⟩ is false, throw an error and remember to reset \g__msg_log_next_bool, which is otherwise reset by __msg_show_wrap:n. If ⟨*message*⟩ is not empty, output the message LaTeX/kernel/show-⟨*message*⟩ with as its arguments the ⟨*variable*⟩, and either an empty second argument or ? depending on the result of ⟨*if-empty*⟩ ⟨*variable*⟩. Afterwards, show the contents of #5 using __msg_show_wrap:n or __msg_log_wrap:n.

```
\cs_new_protected:Npn \__msg_show_variable:NNNnn #1#2#3#4#5
  {
    #2 #1
      {
        \tl_if_empty:nF {#4}
          {
            \__msg_show_pre:nnxxxx { LaTeX / kernel } { show- #4 }
```

```
       { \token_to_str:N #1 } { #3 #1 { } { ? } } { } { }
     }
    \__msg_show_wrap:n {#5}
   }
   {
    \__msg_kernel_error:nnx { kernel } { variable-not-defined }
     { \token_to_str:N #1 }
    \bool_gset_false:N \g__msg_log_next_bool
   }
 }
```

(*End definition for* `\__msg_show_variable:NNNnn`.)

__msg_show_wrap:Nn

A short-hand used for `\int_show:n` and many other functions that passes to `\__msg_show_wrap:n` the result of applying `#1` (a function such as `\int_eval:n`) to the expression `#2`. The leading `>~` is needed by `\__msg_show_wrap:n`. The use of `x`-expansion ensures that `#1` is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```
\cs_new_protected:Npn \__msg_show_wrap:Nn #1#2
 {
  \exp_args:Nx \__msg_show_wrap:n
   {
    > ~ \exp_not:n { \tl_to_str:n {#2} } =
    \exp_not:N \tl_to_str:n { #1 {#2} }
   }
 }
```

(*End definition for* `\__msg_show_wrap:Nn`.)

__msg_show_wrap:n
__msg_show_wrap_aux:n
__msg_show_wrap_aux:w

The argument of `\__msg_show_wrap:n` is line-wrapped using `\iow_wrap:nnnN`. Everything before the first `>` in the wrapped text is removed, as well as an optional space following it (because of `f`-expansion). In order for line-wrapping to give the correct result, the first `>` must in fact appear at the beginning of a line and be followed by a space (or a line-break), so in practice, the argument of `\__msg_show_wrap:n` begins with `>~` or `\\>~`.

The line-wrapped text is then either sent to the log file through `\iow_log:x`, or shown in the terminal using the ε-TeX primitive `\showtokens` after removing a leading `>~` and trailing dot since those are added automatically by `\showtokens`. The trailing dot was included in the first place because its presence can affect line-wrapping. Note that the space after `>` is removed through `f`-expansion rather than by using an argument delimited by `>~` because the space may have been replaced by a line-break when line-wrapping.

A special case is that if the line-wrapped text is a single dot (in other words if the argument of `\__msg_show_wrap:n` `x`-expands to nothing) then no `>~` should be removed. This makes it unnecessary to check explicitly for emptyness when using for instance `\seq_map_function:NN` ⟨*seq var*⟩ `\__msg_show_item:n` as the argument of `\__msg_show_wrap:n`.

Finally, the token list `\l__msg_internal_tl` containing the result of all these manipulations is displayed to the terminal using `\etex_showtokens:D` and odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to __iow_with:Nnn ensure that the \newlinechar is set to 10 so that the \iow_newline: inserted by the line-wrapping code are correctly recognized by TeX, and that \errorcontextlines is -1 to avoid printing irrelevant context.

Note also that \g__msg_log_next_bool is only reset if that is necessary. This allows the user of an interactive prompt to insert tokens as a response to ε-TeX's \showtokens.

```
\cs_new_protected:Npn \__msg_show_wrap:n #1
  { \iow_wrap:nnnN { #1 . } { } { } \__msg_show_wrap_aux:n }
\cs_new_protected:Npn \__msg_show_wrap_aux:n #1
  {
    \tl_if_single:nTF {#1}
      { \tl_clear:N \l__msg_internal_tl }
      { \tl_set:Nf \l__msg_internal_tl { \__msg_show_wrap_aux:w #1 \q_stop } }
    \bool_if:NTF \g__msg_log_next_bool
      {
        \iow_log:x { > ~ \l__msg_internal_tl . }
        \bool_gset_false:N \g__msg_log_next_bool
      }
      {
        \__iow_with:Nnn \tex_newlinechar:D { 10 }
          {
            \__iow_with:Nnn \tex_errorcontextlines:D { -1 }
              {
                \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
                  { \exp_after:wN \l__msg_internal_tl }
              }
          }
      }
  }
\cs_new:Npn \__msg_show_wrap_aux:w #1 > #2 . \q_stop {#2}
```

(*End definition for* __msg_show_wrap:n*,* __msg_show_wrap_aux:n*, and* __msg_show_wrap_aux:w*.*)

__msg_show_item:n
__msg_show_item:nn
__msg_show_item_unbraced:nn

Each item in the variable is formatted using one of the following functions.

```
\cs_new:Npn \__msg_show_item:n #1
  {
    \\ > \ \ \{ \tl_to_str:n {#1} \}
  }
\cs_new:Npn \__msg_show_item:nn #1#2
  {
    \\ > \ \ \{ \tl_to_str:n {#1} \}
    \ \ => \ \ \{ \tl_to_str:n {#2} \}
  }
\cs_new:Npn \__msg_show_item_unbraced:nn #1#2
  {
    \\ > \ \ \tl_to_str:n {#1}
    \ \ => \ \ \tl_to_str:n {#2}
  }
```

(*End definition for* __msg_show_item:n*,* __msg_show_item:nn*, and* __msg_show_item_unbraced:nn*.*)

```
⟨/initex | package⟩
```

17 l3file implementation

The following test files are used for this code: m3file001*.*

```
⟨*initex | package⟩
⟨@@=file⟩
```

17.1 File operations

\g_file_curr_dir_str
\g_file_curr_ext_str
\g_file_curr_name_str

The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the run. In LaTeX 2ε package mode the current file name is collected from \@currname.

```
\str_new:N \g_file_curr_dir_str
\str_new:N \g_file_curr_ext_str
\str_new:N \g_file_curr_name_str
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \str_gset:Nx \g_file_curr_name_str { \tex_jobname:D }
  }
⟨/initex⟩
⟨*package⟩
\cs_if_exist:NT \@currname
  { \str_gset_eq:NN \g_file_curr_name_str \@currname }
⟨/package⟩
```

(*End definition for* \g_file_curr_dir_str*,* \g_file_curr_ext_str*, and* \g_file_curr_name_str*. These variables are documented on page 139.*)

\g__file_stack_seq

The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by LaTeX 2ε (we must be in the preamble and loaded using \usepackage or \RequirePackage). As LaTeX 2ε doesn't store directory and name separately, we stick to the same convention here.

```
\seq_new:N \g__file_stack_seq
⟨*package⟩
\group_begin:
  \cs_set_protected:Npn \__file_tmp:w #1#2#3
    {
      \tl_if_blank:nTF {#1}
        {
          \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \q_stop { { } {##2} {  } }
          \seq_gput_right:Nx \g__file_stack_seq
            {
              \exp_after:wN \__file_tmp:w \tex_jobname:D
                " \tex_jobname:D " \q_stop
            }
        }
        {
          \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
          \__file_tmp:w
        }
    }
  \cs_if_exist:NT \@currnamestack
```

```
    { \exp_after:wN \__file_tmp:w \@currnamestack }
\group_end:
⟨/package⟩
```

(*End definition for* \g__file_stack_seq.)

\g__file_record_seq The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! In format mode, this is done at the very start of the TeX run. In package mode we will eventually copy the contents of \@filelist.

```
\seq_new:N \g__file_record_seq
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \seq_gput_right:NV \g__file_record_seq \g_file_curr_name_str
  }
⟨/initex⟩
```

(*End definition for* \g__file_record_seq.)

\l__file_tmp_tl Used as a short-term scratch variable.

```
\tl_new:N \l__file_tmp_tl
```

(*End definition for* \l__file_tmp_tl.)

\l__file_base_name_str
\l__file_full_name_str
For storing the basename and full path whilst passing data internally.

```
\str_new:N \l__file_base_name_str
\str_new:N \l__file_full_name_str
```

(*End definition for* \l__file_base_name_str *and* \l__file_full_name_str.)

\l__file_dir_str
\l__file_ext_str
\l__file_name_str
Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```
\str_new:N \l__file_dir_str
\str_new:N \l__file_ext_str
\str_new:N \l__file_name_str
```

(*End definition for* \l__file_dir_str, \l__file_ext_str, *and* \l__file_name_str.)

\l_file_search_path_seq The current search path.

```
\seq_new:N \l_file_search_path_seq
```

(*End definition for* \l_file_search_path_seq. *This variable is documented on page 139.*)

\l__file_tmp_seq Scratch space for comma list conversion in package mode.

```
⟨*package⟩
\seq_new:N \l__file_tmp_seq
⟨/package⟩
```

(*End definition for* \l__file_tmp_seq.)

__file_name_sanitize:nN
__file_name_quote:nN
__file_name_sanitize_aux:n

For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by `lualatexquotejobname`: check for balanced `"`, and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```
\cs_new_protected:Npn \__file_name_sanitize:nN #1#2
  {
    \group_begin:
      \seq_map_inline:Nn \l_char_active_seq
        {
          \tl_set:Nx \l__file_tmp_tl { \iow_char:N ##1 }
          \char_set_active_eq:NN ##1 \l__file_tmp_tl
        }
      \tl_set:Nx \l__file_tmp_tl {#1}
      \tl_set:Nx \l__file_tmp_tl
        { \tl_to_str:N \l__file_tmp_tl }
    \exp_args:NNNV \group_end:
    \str_set:Nn #2 \l__file_tmp_tl
  }
\cs_new_protected:Npn \__file_name_quote:nN #1#2
  {
    \str_set:Nx #2 {#1}
    \int_if_even:nF
      { 0 \tl_map_function:NN #2 \__file_name_quote_aux:n }
      {
        \__msg_kernel_error:nnx
          { kernel } { unbalanced-quote-in-filename } {#2}
      }
    \tl_remove_all:Nn #2 { " }
    \tl_if_in:NnT #2 { ~ }
      { \str_set:Nx #2 { " \exp_not:V #2 " } }
  }
\cs_new:Npn \__file_name_quote_aux:n #1
  { \token_if_eq_charcode:NNT #1 " { + 1 } }
```

(*End definition for* `\__file_name_sanitize:nN`*,* `\__file_name_quote:nN`*, and* `\__file_name_sanitize_aux:n`*.*)

\file_get_full_name:nN
\file_get_full_name:VN
__file_get_full_name_search:nN

The way to test if a file exists is to try to open it: if it does not exist then TeX reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to `\__prg_break_point:`. If nothing is found, `#2` is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with `.tex` extension in that directory, and if it exists we include the `.tex` extension in the result.

```
\cs_new_protected:Npn \file_get_full_name:nN #1#2
  {
    \__file_name_sanitize:nN {#1} \l__file_base_name_str
    \__file_get_full_name_search:nN { } \use:n
    \seq_map_inline:Nn \l_file_search_path_seq
      { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
⟨*package⟩
    \cs_if_exist:NT \input@path
      {
```

```
    \tl_map_inline:Nn \input@path
     { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
   }
⟨/package⟩
  \str_clear:N \l__file_full_name_str
  \__prg_break_point:
  \str_if_empty:NF \l__file_full_name_str
   {
    \exp_args:NV \file_parse_full_name:nNNN \l__file_full_name_str
     \l__file_dir_str \l__file_name_str \l__file_ext_str
    \str_if_empty:NT \l__file_ext_str
     {
      \__ior_open:No \g__file_internal_ior
       { \l__file_full_name_str .tex }
      \ior_if_eof:NF \g__file_internal_ior
       { \str_put_right:Nn \l__file_full_name_str { .tex } }
     }
   }
  \str_set_eq:NN #2 \l__file_full_name_str
  \ior_close:N \g__file_internal_ior
 }
\cs_generate_variant:Nn \file_get_full_name:nN { V }
\cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
 {
  \__file_name_quote:nN
   { \tl_to_str:n {#1} \l__file_base_name_str }
   \l__file_full_name_str
  \__ior_open:No \g__file_internal_ior \l__file_full_name_str
  \ior_if_eof:NF \g__file_internal_ior { #2 { \__prg_break: } }
 }
```

(*End definition for* `\file_get_full_name:nN` *and* `\__file_get_full_name_search:nN`*. These functions are documented on page 140.*)

`\file_if_exist:nTF`

The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```
\prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
 {
  \file_get_full_name:nN {#1} \l__file_full_name_str
  \str_if_empty:NTF \l__file_full_name_str
   { \prg_return_false: }
   { \prg_return_true: }
 }
```

(*End definition for* `\file_if_exist:nTF`*. This function is documented on page 139.*)

`\__file_missing:n`

An error message for a missing file, also used in `\ior_open:Nn`.

```
\cs_new_protected:Npn \__file_missing:n #1
 {
  \__file_name_sanitize:nN {#1} \l__file_base_name_str
  \__msg_kernel_error:nnx { kernel } { file-not-found }
   { \l__file_base_name_str }
 }
```

(*End definition for* `\__file_missing:n`*.*)

`\file_input:n`
`\__file_input:n`
`\__file_input:V`
`\__file_input_push:n`
`\__file_input_pop:`
`\__file_input_pop:nnn`

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```
\cs_new_protected:Npn \file_input:n #1
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      { \__file_missing:n {#1} }
      { \__file_input:V \l__file_full_name_str }
  }
\cs_new_protected:Npn \__file_input:n #1
  {
⟨*initex⟩
    \seq_gput_right:Nn \g__file_record_seq {#1}
⟨/initex⟩
⟨*package⟩
    \clist_if_exist:NTF \@filelist
      { \@addtofilelist {#1} }
      { \seq_gput_right:Nn \g__file_record_seq {#1} }
⟨/package⟩
    \__file_input_push:n {#1}
    \tex_input:D #1 \c_space_tl
    \__file_input_pop:
  }
\cs_generate_variant:Nn \__file_input:n { V }
```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```
\cs_new_protected:Npn \__file_input_push:n #1
  {
    \seq_gpush:Nx \g__file_stack_seq
      {
        { \g_file_curr_dir_str }
        { \g_file_curr_name_str }
        { \g_file_curr_ext_str }
      }
    \file_parse_full_name:nNNN {#1}
      \l__file_dir_str \l__file_name_str \l__file_ext_str
    \str_gset_eq:NN \g_file_curr_dir_str  \l__file_dir_str
    \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
    \str_gset_eq:NN \g_file_curr_ext_str  \l__file_ext_str
  }
\cs_new_protected:Npn \__file_input_pop:
  {
    \seq_gpop:NN \g__file_stack_seq \l__file_tmp_tl
    \exp_after:wN \__file_input_pop:nnn \l__file_tmp_tl
  }
\cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
  {
    \str_gset:Nn \g_file_curr_dir_str  {#1}
    \str_gset:Nn \g_file_curr_name_str {#2}
    \str_gset:Nn \g_file_curr_ext_str  {#3}
  }
```

(End definition for `\file_input:n` *and others. These functions are documented on page 140.)*

`\file_parse_full_name:nNNN`
`\__file_parse_full_name_auxi:w`
`\__file_parse_full_name_split:nNNNTF`

Parsing starts by stripping off any surrounding quotes. Then find the directory `#4` by splitting at the last `/`. (The auxiliary returns `true`/`false` depending on whether it found the delimiter.) We correct for the case of a file in the root `/`, as in that case we wish to keep the trailing (and only) slash. Then split the base name `#5` at the last dot. If there was indeed a dot, `#5` contains the name and `#6` the extension without the dot, which we add back for convenience. In the special case of no extension given, the auxiliary stored the name into `#6`, we just have to move it to `#5`.

```
\cs_new_protected:Npn \file_parse_full_name:nNNN #1#2#3#4
  {
    \exp_after:wN \__file_parse_full_name_auxi:w
      \tl_to_str:n { #1 " #1 " } \q_stop #2#3#4
  }
\cs_new_protected:Npn \__file_parse_full_name_auxi:w #1 " #2 " #3 \q_stop #4#5#6
  {
    \__file_parse_full_name_split:nNNNTF {#2} / #4 #5
      { \str_if_empty:NT #4 { \str_set:Nn #4 { / } } }
      { }
    \exp_args:No \__file_parse_full_name_split:nNNNTF {#5} . #5 #6
      { \str_put_left:Nn #6 { . } }
      {
        \str_set_eq:NN #5 #6
        \str_clear:N #6
      }
  }
\cs_new_protected:Npn \__file_parse_full_name_split:nNNNTF #1#2#3#4
  {
    \cs_set_protected:Npn \__file_tmp:w ##1 ##2 #2 ##3 \q_stop
      {
        \tl_if_empty:nTF {##3}
          {
            \str_set:Nn #4 {##2}
            \tl_if_empty:nTF {##1}
              {
                \str_clear:N #3
                \use_ii:nn
              }
              {
                \str_set:Nx #3 { \str_tail:n {##1} }
                \use_i:nn
              }
          }
          { \__file_tmp:w { ##1 #2 ##2 } ##3 \q_stop }
      }
    \__file_tmp:w { } #1 #2 \q_stop
  }
```

(End definition for `\file_parse_full_name:nNNN`*,* `\__file_parse_full_name_auxi:w`*, and* `\__file_parse_full_name_split:nNNNTF`*. These functions are documented on page 140.)*

`\file_show_list:`
`\file_log_list:`
`\__file_list_aux:n`

A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it \AtBeginDocument into \g__file_record_seq), turning it to a string (this does not affect the commas of this comma list). The message system is a bit finnicky (it can only display results that start with >~ and end with a dot) so that constrains the possible markup. The advantage is that we get terminal and log outputs for free.

```
\cs_new_protected:Npn \file_show_list:
  {
    \seq_clear:N \l__file_tmp_seq
⟨*package⟩
    \clist_if_exist:NT \@filelist
      {
        \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
      }
⟨/package⟩
    \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
    \seq_remove_duplicates:N \l__file_tmp_seq
    \__msg_show_wrap:n
      {
        >~File~List~< \\
        \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n
        . . . . . . . . . . . .
      }
  }
\cs_new:Npn \__file_list_aux:n #1 { #1 \\ }
\cs_new_protected:Npn \file_log_list:
  { \__msg_log_next: \file_show_list: }
```

(*End definition for* \file_show_list:*,* \file_log_list:*, and* __file_list_aux:n*. These functions are documented on page 140.*)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in \@filelist must be turned to strings before being added to \g__file_record_seq.

```
⟨*package⟩
\AtBeginDocument
  {
    \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
      { \tl_to_str:N \@filelist }
    \seq_gconcat:NNN \g__file_record_seq \g__file_record_seq \l__file_tmp_seq
  }
⟨/package⟩
```

17.2 Input operations

```
⟨@@=ior⟩
```

17.2.1 Variables and constants

\c_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
\int_const:Nn \c_term_ior { 16 }
```

(*End definition for* \c_term_ior*. This variable is documented on page 147.*)

\g__ior_streams_seq

A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to LaTeX 2ε as they are needed (initially none are needed), so the starting point varies!

```
\seq_new:N \g__ior_streams_seq
⟨*initex⟩
\seq_gset_split:Nnn \g__ior_streams_seq { , }
  { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
⟨/initex⟩
```

(*End definition for* \g__ior_streams_seq*.*)

\l__ior_stream_tl

Used to recover the raw stream number from the stack.

```
\tl_new:N \l__ior_stream_tl
```

(*End definition for* \l__ior_stream_tl*.*)

\g__ior_streams_prop

The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For LaTeX 2ε and plain TeX this data is stored in \count16: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConTeXt, we need to look at \count38 but there is no subtraction: like the original plain TeX/LaTeX 2ε mechanism it holds the value of the *last* stream allocated.

```
\prop_new:N \g__ior_streams_prop
⟨*package⟩
\int_step_inline:nnnn
  { 0 }
  { 1 }
  {
    \cs_if_exist:NTF \normalend
      { \tex_count:D 38 ~ }
      {
        \tex_count:D 16 ~ %
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
  }
⟨/package⟩
```

(*End definition for* \g__ior_streams_prop*.*)

17.2.2 Stream management

\ior_new:N
\ior_new:c

Reserving a new stream is done by defining the name as equal to using the terminal.

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
```

(*End definition for* \ior_new:N*. This function is documented on page 141.*)

\ior_open:Nn
\ior_open:cn

Use the conditional version, with an error if the file is not found.

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \ior_open:NnF #1 {#2} { \__file_missing:n {#2} } }
\cs_generate_variant:Nn \ior_open:Nn { c }
```

(*End definition for* `\ior_open:Nn`*. This function is documented on page 141.*)

`\ior_open:NnTF`
`\ior_open:cnTF`

An auxiliary searches for the file in the TEX, LATEX 2ε and LATEX3 paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  {
    \file_get_full_name:nN {#2} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      { \prg_return_false: }
      {
        \__ior_open:No #1 \l__file_full_name_str
        \prg_return_true:
      }
  }
\cs_generate_variant:Nn \ior_open:NnT  { c }
\cs_generate_variant:Nn \ior_open:NnF  { c }
\cs_generate_variant:Nn \ior_open:NnTF { c }
```

(*End definition for* `\ior_open:NnTF`*. This function is documented on page 141.*)

`\__ior_new:N`

In package mode, streams are reserved using `\newread` before they can be managed by ior. To prevent ior from being affected by redefinitions of `\newread` (such as done by the third-party package morewrites), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain TEX's `\newread` being `\outer`.

```
⟨*package⟩
\exp_args:NNf \cs_new_protected:Npn \__ior_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
⟨/package⟩
```

(*End definition for* `\__ior_new:N`.)

`\__ior_open:Nn`
`\__ior_open:No`
`\__ior_open_stream:Nn`

The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask plain TEX or LATEX 2ε for a new stream and use that number (after a bit of conversion).

```
\cs_new_protected:Npn \__ior_open:Nn #1#2
  {
    \ior_close:N #1
    \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
      { \__ior_open_stream:Nn #1 {#2} }
⟨*initex⟩
      { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
⟨/initex⟩
⟨*package⟩
      {
        \__ior_new:N #1
        \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
        \__ior_open_stream:Nn #1 {#2}
      }
⟨/package⟩
  }
\cs_generate_variant:Nn \__ior_open:Nn { No }
\cs_new_protected:Npn \__ior_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
    \prop_gput:NVn \g__ior_streams_prop #1 {#2}
    \tex_openin:D #1 #2 \scan_stop:
  }
```

(*End definition for* `\__ior_open:Nn` *and* `\__ior_open_stream:Nn`*.*)

`\ior_close:N`
`\ior_close:c`

Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range $[0, 15]$), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \int_compare:nT { -1 < #1 < \c_term_ior }
      {
        \tex_closein:D #1
        \prop_gremove:NV \g__ior_streams_prop #1
        \seq_if_in:NVF \g__ior_streams_seq #1
          { \seq_gpush:NV \g__ior_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_ior
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
```

(*End definition for* `\ior_close:N`*. This function is documented on page 141.*)

`\ior_show_list:`
`\ior_log_list:`
`\__ior_list:Nn`

Show the property lists, but with some “pretty printing”. See the l3msg module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (in fact a question mark) otherwise. The code of the message `show-streams` takes care of translating `ior`/`iow` to English. The list of streams is formatted using `\__msg_show_item_unbraced:nn`.

```
\cs_new_protected:Npn \ior_show_list:
  { \__ior_list:Nn \g__ior_streams_prop { ior } }
\cs_new_protected:Npn \ior_log_list:
  { \__msg_log_next: \ior_show_list: }
\cs_new_protected:Npn \__ior_list:Nn #1#2
  {
    \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-streams }
      {#2} { \prop_if_empty:NF #1 { ? } } { } { }
    \__msg_show_wrap:n
      { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
  }
```

(*End definition for* `\ior_show_list:`*,* `\ior_log_list:`*, and* `\__ior_list:Nn`*. These functions are documented on page 141.*)

17.2.3 Reading input

\if_eof:w

The primitive conditional

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*End definition for* `\if_eof:w`.)

\ior_if_eof_p:N
\ior_if_eof:N*TF*

To test if some particular input stream is exhausted the following conditional is provided. The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```
\prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \int_compare:nTF { -1 < #1 < \c_term_ior }
          {
            \if_eof:w #1
              \prg_return_true:
            \else:
              \prg_return_false:
            \fi:
          }
          { \prg_return_true: }
      }
      { \prg_return_true: }
  }
```

(*End definition for* `\ior_if_eof:NTF`. *This function is documented on page 144.*)

\ior_get:NN

And here we read from files.

```
\cs_new_protected:Npn \ior_get:NN #1#2
  { \tex_read:D #1 to #2 }
```

(*End definition for* `\ior_get:NN`. *This function is documented on page 142.*)

\ior_str_get:NN

Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```
\cs_new_protected:Npn \ior_str_get:NN #1#2
  {
    \use:x
      {
        \int_set:Nn \tex_endlinechar:D { -1 }
        \exp_not:n { \etex_readline:D #1 to #2 }
        \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
      }
  }
```

(*End definition for* `\ior_str_get:NN`. *This function is documented on page 142.*)

\ior_map_break:
\ior_map_break:n

Usual map breaking functions.

```
\cs_new:Npn \ior_map_break:
  { \__prg_map_break:Nn \ior_map_break: { } }
\cs_new:Npn \ior_map_break:n
  { \__prg_map_break:Nn \ior_map_break: }
```

(*End definition for* `\ior_map_break:` *and* `\ior_map_break:n`*. These functions are documented on page 143.*)

`\ior_map_inline:Nn`
`\ior_str_map_inline:Nn`
`\__ior_map_inline:NNn`
`\__ior_map_inline:NNNn`
`\__ior_map_inline_loop:NNN`
`\l__ior_internal_tl`

Mapping to an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N`. This mapping cannot be nested with twice the same stream, as the stream has only one "current line".

```
\cs_new_protected:Npn \ior_map_inline:Nn
  { \__ior_map_inline:NNn \ior_get:NN }
\cs_new_protected:Npn \ior_str_map_inline:Nn
  { \__ior_map_inline:NNn \ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_inline:NNn
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:Nc \__ior_map_inline:NNNn
      { __prg_map_ \int_use:N \g__prg_map_int :n }
  }
\cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
  {
    \cs_gset_protected:Npn #1 ##1 {#4}
    \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
    \__prg_break_point:Nn \ior_map_break:
      { \int_gdecr:N \g__prg_map_int }
  }
\cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l__ior_internal_tl
    \ior_if_eof:NF #3
      {
        \exp_args:No #1 \l__ior_internal_tl
        \__ior_map_inline_loop:NNN #1#2#3
      }
  }
\tl_new:N  \l__ior_internal_tl
```

(*End definition for* `\ior_map_inline:Nn` *and others. These functions are documented on page 143.*)

`\g__file_internal_ior`

Needed by the higher-level code, but cannot be created until here.

```
\ior_new:N \g__file_internal_ior
```

(*End definition for* `\g__file_internal_ior`*.*)

17.3 Output operations

```
⟨@@=iow⟩
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

17.3.1 Variables and constants

`\c_log_iow`
`\c_term_iow`

Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTEX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```
\int_const:Nn \c_log_iow  { -1 }
\int_const:Nn \c_term_iow
  {
    \cs_if_exist:NTF \luatex_directlua:D
      {
        \int_compare:nNnTF \luatex_luatexversion:D > { 80 }
          { 128 }
          { 16 }
      }
      { 16 }
  }
```

(*End definition for* \c_log_iow *and* \c_term_iow*. These variables are documented on page 147.*)

\g__iow_streams_seq A list of the currently-available output streams to be used as a stack.

```
\seq_new:N \g__iow_streams_seq
⟨*initex⟩
\use:x
  {
    \exp_not:n { \seq_gset_split:Nnn \g__iow_streams_seq { } }
      {
        \int_step_function:nnnN { 0 } { 1 } { \c_term_iow }
          \prg_do_nothing:
      }
  }
⟨/initex⟩
```

(*End definition for* \g__iow_streams_seq*.*)

\l__iow_stream_tl Used to recover the raw stream number from the stack.

```
\tl_new:N \l__iow_stream_tl
```

(*End definition for* \l__iow_stream_tl*.*)

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

```
\prop_new:N \g__iow_streams_prop
⟨*package⟩
\int_step_inline:nnnn
  { 0 }
  { 1 }
  {
    \cs_if_exist:NTF \normalend
      { \tex_count:D 39 ~ }
      {
        \tex_count:D 17 ~
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
  }
⟨/package⟩
```

(*End definition for* \g__iow_streams_prop*.*)

17.4 Stream management

\iow_new:N
\iow_new:c

Reserving a new stream is done by defining the name as equal to writing to the terminal: odd but at least consistent.

```
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*End definition for* \iow_new:N*. This function is documented on page 141.*)

__iow_new:N

As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

```
⟨*package⟩
\exp_args:NNf \cs_new_protected:Npn \__iow_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
⟨/package⟩
```

(*End definition for* __iow_new:N*.*)

\iow_open:Nn
\iow_open:cn
__iow_open_stream:Nn
__iow_open_stream:NV

The same idea as for reading, but without the path and without the need to allow for a conditional version.

```
\cs_new_protected:Npn \iow_open:Nn #1#2
  {
    \__file_name_sanitize:nN {#2} \l__file_base_name_str
    \iow_close:N #1
    \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
      { \__iow_open_stream:NV #1 \l__file_base_name_str }
⟨*initex⟩
      { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
⟨/initex⟩
⟨*package⟩
      {
        \__iow_new:N #1
        \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
        \__iow_open_stream:NV #1 \l__file_base_name_str
      }
⟨/package⟩
  }
\cs_generate_variant:Nn \iow_open:Nn { c }
\cs_new_protected:Npn \__iow_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
    \prop_gput:NVn \g__iow_streams_prop #1 {#2}
    \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
  }
\cs_generate_variant:Nn \__iow_open_stream:Nn { NV }
```

(*End definition for* \iow_open:Nn *and* __iow_open_stream:Nn*. These functions are documented on page 141.*)

\iow_close:N
\iow_close:c

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
\cs_new_protected:Npn \iow_close:N #1
  {
    \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
      {
        \tex_immediate:D \tex_closeout:D #1
        \prop_gremove:NV \g__iow_streams_prop #1
        \seq_if_in:NVF \g__iow_streams_seq #1
          { \seq_gpush:NV \g__iow_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_iow
      }
  }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*End definition for* \iow_close:N. *This function is documented on page 141.*)

\iow_show_list:
\iow_log_list:
__iow_list:Nn

Done as for input, but with a copy of the auxiliary so the name is correct.

```
\cs_new_protected:Npn \iow_show_list:
  { \__iow_list:Nn \g__iow_streams_prop { iow } }
\cs_new_protected:Npn \iow_log_list:
  { \__msg_log_next: \iow_show_list: }
\cs_new_eq:NN \__iow_list:Nn \__ior_list:Nn
```

(*End definition for* \iow_show_list: , \iow_log_list: , *and* __iow_list:Nn. *These functions are documented on page 141.*)

17.4.1 Deferred writing

\iow_shipout_x:Nn
\iow_shipout_x:Nx
\iow_shipout_x:cn
\iow_shipout_x:cx

First the easy part, this is the primitive, which expects its argument to be braced.

```
\cs_new_protected:Npn \iow_shipout_x:Nn #1#2
  { \tex_write:D #1 {#2} }
\cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
```

(*End definition for* \iow_shipout_x:Nn. *This function is documented on page 145.*)

\iow_shipout:Nn
\iow_shipout:Nx
\iow_shipout:cn
\iow_shipout:cx

With ε-TeX available deferred writing without expansion is easy.

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
```

(*End definition for* \iow_shipout:Nn. *This function is documented on page 145.*)

17.4.2 Immediate writing

__iow_with:Nnn
__iow_with_aux:nNnn

If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```
\cs_new_protected:Npn \__iow_with:Nnn #1#2
  {
    \int_compare:nNnTF {#1} = {#2}
      { \use:n }
      { \exp_args:No \__iow_with_aux:nNnn { \int_use:N #1 } #1 {#2} }
  }
\cs_new_protected:Npn \__iow_with_aux:nNnn #1#2#3#4
  {
    \int_set:Nn #2 {#3}
    #4
    \int_set:Nn #2 {#1}
  }
```

(*End definition for* __iow_with:Nnn *and* __iow_with_aux:nNnn.)

\iow_now:Nn
\iow_now:Nx
\iow_now:cn
\iow_now:cx

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `\__iow_with:Nnn` to support formats such as plain TeX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TeX looks at the value of the `\newlinechar` at shipout time in those cases.

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  {
    \__iow_with:Nnn \tex_newlinechar:D { `\^^J }
      { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
  }
\cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(*End definition for* \iow_now:Nn. *This function is documented on page 144.*)

\iow_log:n
\iow_log:x
\iow_term:n
\iow_term:x

Writing to the log and the terminal directly are relatively easy.

```
\cs_set_protected:Npn \iow_log:x  { \iow_now:Nx \c_log_iow  }
\cs_new_protected:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
\cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
```

(*End definition for* \iow_log:n *and* \iow_term:n. *These functions are documented on page 144.*)

17.4.3 Special characters for writing

\iow_newline:

Global variable holding the character that forces a new line when something is written to an output stream.

```
\cs_new:Npn \iow_newline: { ^^J }
```

(*End definition for* \iow_newline:. *This function is documented on page 145.*)

\iow_char:N

Function to write any escaped char to an output stream.

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End definition for* \iow_char:N. *This function is documented on page 145.*)

17.4.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

\l_iow_line_count_int

This is the "raw" number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TeXLive and MikTeX.

```
\int_new:N  \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }
```

(*End definition for* \l_iow_line_count_int. *This variable is documented on page 146.*)

\l__iow_newline_tl

The token list inserted to produce a new line, with the ⟨*run-on text*⟩.

```
\tl_new:N \l__iow_newline_tl
```

(*End definition for* `\l__iow_newline_tl`*.*)

`\l__iow_line_target_int`

This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
\int_new:N \l__iow_line_target_int
```

(*End definition for* `\l__iow_line_target_int`*.*)

`\__iow_set_indent:n`
`\__iow_unindent:w`
`\l__iow_one_indent_tl`
`\l__iow_one_indent_int`

The `one_indent` variables hold one indentation marker and its length. The `\__iow_unindent:w` auxiliary removes one indentation. The function `\__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int
\cs_new:Npn \__iow_unindent:w { }
\cs_new_protected:Npn \__iow_set_indent:n #1
  {
    \tl_set:Nx \l__iow_one_indent_tl
      { \exp_args:No \__str_to_other_fast:n { \tl_to_str:n {#1} } }
    \int_set:Nn \l__iow_one_indent_int { \str_count:N \l__iow_one_indent_tl }
    \exp_last_unbraced:NNo
      \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
  }
\exp_args:Nx \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(*End definition for* `\__iow_set_indent:n` *and others.*)

`\l__iow_indent_tl`
`\l__iow_indent_int`

The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```
\tl_new:N \l__iow_indent_tl
\int_new:N \l__iow_indent_int
```

(*End definition for* `\l__iow_indent_tl` *and* `\l__iow_indent_int`*.*)

`\l__iow_line_tl`
`\l__iow_line_part_tl`

These hold the current line of text and a partial line to be added to it, respectively.

```
\tl_new:N \l__iow_line_tl
\tl_new:N \l__iow_line_part_tl
```

(*End definition for* `\l__iow_line_tl` *and* `\l__iow_line_part_tl`*.*)

`\l__iow_line_break_bool`

Indicates whether the line was broken precisely at a chunk boundary.

```
\bool_new:N \l__iow_line_break_bool
```

(*End definition for* `\l__iow_line_break_bool`*.*)

`\l__iow_wrap_tl`

Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
\tl_new:N \l__iow_wrap_tl
```

(*End definition for* `\l__iow_wrap_tl`*.*)

\c__iow_wrap_marker_tl
\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl

Every special action of the wrapping code is starts with the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that "word", the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```
\group_begin:
  \int_set:Nn \tex_escapechar:D { -1 }
  \tl_const:Nx \c__iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
\group_end:
\tl_map_inline:nn
  { { end } { newline } { indent } { unindent } }
  {
    \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
      {
        \c__iow_wrap_marker_tl
        #1
        \c_catcode_other_space_tl
      }
  }
```

(*End definition for* `\c__iow_wrap_marker_tl` *and others.*)

\iow_indent:n
__iow_indent:n
__iow_indent_error:n

We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_indent:n` when valid and otherwise to `\__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```
\cs_new_protected:Npn \iow_indent:n #1
  {
    \__msg_kernel_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
\cs_new:Npx \__iow_indent:n #1
  {
    \c__iow_wrap_indent_marker_tl
    #1
    \c__iow_wrap_unindent_marker_tl
  }
\cs_new:Npn \__iow_indent_error:n #1
  {
    \__msg_kernel_expandable_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
```

(*End definition for* `\iow_indent:n`, `\__iow_indent:n`, *and* `\__iow_indent_error:n`. *These functions are documented on page 146.*)

\iow_wrap:nnnN

The main wrapping function works as follows. First give `\\`, `\␣` and other formatting commands the correct definition for messages and perform the given setup `#3`. The definition of `\␣` uses an "other" space rather than a normal space, because the latter might be absorbed by TeX to end a number or other `f`-type expansions.

```
\cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
  {
    \group_begin:
      \int_set:Nn \tex_escapechar:D { -1 }
      \cs_set:Npx \{ { \token_to_str:N \{ }
      \cs_set:Npx \# { \token_to_str:N \# }
      \cs_set:Npx \} { \token_to_str:N \} }
      \cs_set:Npx \% { \token_to_str:N \% }
      \cs_set:Npx \~ { \token_to_str:N \~ }
      \int_set:Nn \tex_escapechar:D { 92 }
      \cs_set_eq:NN \\ \c__iow_wrap_newline_marker_tl
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
      #3
```

Then fully-expand the input: in package mode, the expansion uses LaTeX 2$_\varepsilon$'s `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```
⟨package⟩      \cs_set_eq:NN \protect \token_to_str:N
      \tl_set:Nx \l__iow_wrap_tl {#1}
      \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```
      \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
      \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
      \int_set:Nn \l__iow_line_target_int
        { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }
```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the "other" spaces back to normal spaces. The `f`-expansion removes a leading space from `\l__iow_wrap_tl`.

```
      \__iow_wrap_do:
    \exp_args:NNf \group_end:
    #4 { \tl_to_str:N \l__iow_wrap_tl }
  }
```

(*End definition for* `\iow_wrap:nnnN`*. This function is documented on page 146.*)

`\__iow_wrap_do:`
`\__iow_wrap_start:w`

Escape spaces. Set up a few variables, in particular the initial value of `\l__iow_wrap_tl`: the space stops the `f`-expansion of the main wrapping function and `\use_none:n` removes a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```
\cs_new_protected:Npn \__iow_wrap_do:
  {
    \tl_set:Nx \l__iow_wrap_tl
      {
        \exp_args:No \__str_to_other_fast:n \l__iow_wrap_tl
        \c__iow_wrap_end_marker_tl
      }
    \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
  }
\cs_new_protected:Npn \__iow_wrap_start:w
  {
    \bool_set_false:N \l__iow_line_break_bool
    \tl_clear:N \l__iow_line_tl
    \tl_clear:N \l__iow_line_part_tl
    \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
    \int_zero:N \l__iow_indent_int
    \tl_clear:N \l__iow_indent_tl
    \__iow_wrap_chunk:nw { \l_iow_line_count_int }
  }
```

(*End definition for* `\__iow_wrap_do:` *and* `\__iow_wrap_start:w`.)

__iow_wrap_chunk:nw
__iow_wrap_next:nw

The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `\__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (`#1`) before the `\__iow_wrap_end_chunk:w` auxiliary.

```
\cs_set_protected:Npn \__iow_tmp:w #1#2
  {
    \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
      {
        \tl_if_empty:nTF {##2}
          {
            \tl_clear:N \l__iow_line_part_tl
            \__iow_wrap_next:nw {##1}
          }
          {
            \tl_if_empty:NTF \l__iow_line_tl
              {
                \__iow_wrap_line:nw
                  { \l__iow_indent_tl }
                  ##1 - \l__iow_indent_int ;
              }
              { \__iow_wrap_line:nw { } ##1 ; }
            ##2 #1
            \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \q_stop
          }
      }
    \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
      { \use:c { __iow_wrap_##2:n } {##1} }
  }
\exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
```

(*End definition for* `\__iow_wrap_chunk:nw` *and* `\__iow_wrap_next:nw`.)

__iow_wrap_line:nw
__iow_wrap_line_loop:w
__iow_wrap_line_aux:Nw
__iow_wrap_line_end:NnnnnnnnN
__iow_wrap_line_end:nw
__iow_wrap_end_chunk:w

This is followed by `{`⟨*string*⟩`}` ⟨*intexpr*⟩ `;`. It stores the ⟨*string*⟩ and up to ⟨*intexpr*⟩ characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k, then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments `#2`–`#9` of the `line_loop` auxiliary or as one of the arguments `#2`–`#8` of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```
\cs_new_protected:Npn \__iow_wrap_line:nw #1
  {
    \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
    #1
    \exp_after:wN \__iow_wrap_line_loop:w
    \__int_value:w \__int_eval:w
  }
\cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 < 8 \exp_stop_f:
      \__iow_wrap_line_aux:Nw #1
    \fi:
    #2 #3 #4 #5 #6 #7 #8 #9
    \exp_after:wN \__iow_wrap_line_loop:w
    \__int_value:w \__int_eval:w #1 - 8 ;
  }
\cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
  {
    #2
    \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \exp_after:wN
    \if_case:w #1 \exp_stop_f:
         \prg_do_nothing:
    \or: \use_none:n
    \or: \use_none:nn
    \or: \use_none:nnn
    \or: \use_none:nnnn
    \or: \use_none:nnnnn
    \or: \use_none:nnnnnn
    \or: \use_none:nnnnnnn
    \fi:
    { } { } { } { } { } { } { } #3
  }
\cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    #2 #3 #4 #5 #6 #7 #8
    \use_none:nnnnn \__int_eval:w 8 - ; #9
    \token_if_eq_charcode:NNTF \c_space_token #9
      { \__iow_wrap_line_end:nw { } }
      { \if_false: { \fi: } \__iow_wrap_break:w #9 }
  }
\cs_new:Npn \__iow_wrap_line_end:nw #1
  {
    \if_false: { \fi: }
    \__iow_wrap_store_do:n {#1}
    \__iow_wrap_next_line:w
  }
\cs_new:Npn \__iow_wrap_end_chunk:w
    #1 \__int_eval:w #2 - #3 ; #4#5 \q_stop
  {
    \if_false: { \fi: }
    \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
  }
```

(*End definition for* `\__iow_wrap_line:nw` *and others.*)

`\__iow_wrap_break:w`
`\__iow_wrap_break_first:w`
`\__iow_wrap_break_none:w`
`\__iow_wrap_break_loop:w`
`\__iow_wrap_break_end:w`

Functions here are defined indirectly: `\__iow_tmp:w` is eventually called with an "other" space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves "words" (delimited by spaces) until it hits the trailing space: then its argument `##3` is `? \__iow_wrap_break_end:w` instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls `\__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including `##4`, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_break:w
      {
        \tex_edef:D \l__iow_line_part_tl
          { \if_false: } \fi:
            \exp_after:wN \__iow_wrap_break_first:w
            \l__iow_line_part_tl
            #1
            { ? \__iow_wrap_break_end:w }
            \q_mark
      }
    \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
      {
        \use_none:nn ##2 \__iow_wrap_break_none:w
        \__iow_wrap_break_loop:w ##1 #1 ##2
      }
    \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \q_mark ##4 #1
      {
        \tl_if_empty:NTF \l__iow_line_tl
          { ##2 ##4 \__iow_wrap_line_end:nw { } }
          { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
      }
    \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
      {
        \use_none:n ##3
        ##1 #1
        \__iow_wrap_break_loop:w ##2 #1 ##3
      }
    \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \q_mark
      { ##1 \__iow_wrap_line_end:nw { } ##3 }
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End definition for* `\__iow_wrap_break:w` *and others.*)

__iow_wrap_next_line:w

The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```
\cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \q_stop
  {
    \tl_clear:N \l__iow_line_tl
    \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
      {
        \tl_clear:N \l__iow_line_part_tl
        \bool_set_true:N \l__iow_line_break_bool
        \__iow_wrap_next:nw { \l__iow_line_target_int }
      }
      {
        \__iow_wrap_line:nw
          { \l__iow_indent_tl }
          \l__iow_line_target_int - \l__iow_indent_int ;
          #1 #2 \q_stop
      }
  }
```

(*End definition for* `\__iow_wrap_next_line:w`.)

__iow_wrap_indent:
__iow_wrap_unindent:

These functions are called after a chunk has been wrapped, when encountering `indent`/`unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```
\cs_new_protected:Npn \__iow_wrap_indent:n #1
  {
    \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
\cs_new_protected:Npn \__iow_wrap_unindent:n #1
  {
    \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \tl_set:Nx \l__iow_indent_tl
      { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
```

(*End definition for* `\__iow_wrap_indent:` *and* `\__iow_wrap_unindent:`*.*)

`\__iow_wrap_newline:`
`\__iow_wrap_end:`

These functions are called after a chunk has been line-wrapped, when encountering a `newline`/`end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```
\cs_new_protected:Npn \__iow_wrap_newline:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
    \__iow_wrap_chunk:nw { \l__iow_line_target_int }
  }
\cs_new_protected:Npn \__iow_wrap_end:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
  }
```

(*End definition for* `\__iow_wrap_newline:` *and* `\__iow_wrap_end:`*.*)

`\__iow_wrap_store_do:n`

First add the last line part to the line, then append it to `\l__iow_wrap_tl` with the appropriate new line (with “run-on” text), possibly with its last space removed (`#1` is empty or `\__iow_wrap_trim:N`).

```
\cs_new_protected:Npn \__iow_wrap_store_do:n #1
  {
    \tl_set:Nx \l__iow_line_tl
      { \l__iow_line_tl \l__iow_line_part_tl }
    \tl_set:Nx \l__iow_wrap_tl
      {
        \l__iow_wrap_tl
        \l__iow_newline_tl
        #1 \l__iow_line_tl
      }
    \tl_clear:N \l__iow_line_tl
  }
```

(*End definition for* `\__iow_wrap_store_do:n`*.*)

`\__iow_wrap_trim:N`
`\__iow_wrap_trim:w`

Remove one trailing “other” space from the argument.

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_trim:N ##1
      { \tl_if_empty:NF ##1 { \exp_after:wN \__iow_wrap_trim:w ##1 \q_stop } }
    \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \q_stop {##1}
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End definition for* `\__iow_wrap_trim:N` *and* `\__iow_wrap_trim:w`*.*)

17.5 Messages

```
\__msg_kernel_new:nnnn { kernel } { file-not-found }
  { File~'#1'~not~found. }
  {
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
  { Input~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
    All~16~are~currently~in~use,~and~something~wanted~to~open~
    another~one.
  }
\__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
  { Output~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
    All~16~are~currently~in~use,~and~something~wanted~to~open~
    another~one.
  }
\__msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
  { Unbalanced~quotes~in~file~name~'#1'. }
  {
    File~names~must~contain~balanced~numbers~of~quotes~(").
  }
\__msg_kernel_new:nnnn { kernel } { iow-indent }
  { Only~#1 (arg~1)~allows~#2 }
  {
    The~command~#2 can~only~be~used~in~messages~
    which~will~be~wrapped~using~#1.~
    It~was~called~with~argument~'#3'.
  }
```

17.6 Deprecated functions

\g_file_current_name_tl

For removal after 2018-12-31. Contrarily to most other deprecated commands this is expandable so we need to put code by hand in two token lists. We use \tex_def:D directly because \g_file_current_name_tl is made outer by \debug_deprecation_on:.

```
\tl_new:N \g_file_current_name_tl
\tl_gset:Nn \g_file_current_name_tl { \g_file_curr_name_str }
\__debug:TF
  {
    \tl_gput_right:Nn \g__debug_deprecation_on_tl
      {
        \__deprecation_error:Nnn \g_file_current_name_tl
          { \g_file_curr_name_str } { 2018-12-31 }
      }
    \tl_gput_right:Nn \g__debug_deprecation_off_tl
      { \tex_def:D \g_file_current_name_tl { \g_file_curr_name_str } }
  }
  { }
```

(*End definition for* `\g_file_current_name_tl`*.*)

`\file_path_include:n`
`\file_path_remove:n`

Wrapper functions to manage the search path.

```
\__debug_deprecation:nnNNpn { 2018-12-31 }
  { \seq_put_right:Nn \l_file_search_path_seq }
\cs_new_protected:Npn \file_path_include:n #1
  {
    \__file_name_sanitize:nN {#1} \l__file_full_name_str
    \seq_if_in:NVF \l_file_search_path_seq \l__file_full_name_str
      { \seq_put_right:NV \l_file_search_path_seq \l__file_full_name_str }
  }
\__debug_deprecation:nnNNpn { 2018-12-31 }
  { \seq_remove_all:Nn \l_file_search_path_seq }
\cs_new_protected:Npn \file_path_remove:n #1
  {
    \__file_name_sanitize:nN {#1} \l__file_full_name_str
    \seq_remove_all:NV \l_file_search_path_seq \l__file_full_name_str
  }
```

(*End definition for* `\file_path_include:n` *and* `\file_path_remove:n`*.*)

`\file_add_path:nN`

For removal after 2018-12-31.

```
\__debug_deprecation:nnNNpn { 2018-12-31 } { \file_get_full_name:nN }
\cs_new_protected:Npn \file_add_path:nN #1#2
  {
    \file_get_full_name:nN {#1} #2
    \str_if_empty:NT #2
      { \tl_set:Nn #2 { \q_no_value } }
  }
```

(*End definition for* `\file_add_path:nN`*.*)

`\ior_get_str:NN`

For removal after 2017-12-31.

```
\__debug_deprecation:nnNNpn { 2017-12-31 } { \ior_str_get:NN }
\cs_new_protected:Npn \ior_get_str:NN      { \ior_str_get:NN }
```

(*End definition for* `\ior_get_str:NN`*.*)

`\file_list:`

Renamed to `\file_log_list:`. For removal after 2018-12-31.

```
\__debug_deprecation:nnNNpn { 2018-12-31 } { \file_log_list: }
\cs_new_protected:Npn \file_list:          { \file_log_list: }
```

(*End definition for* `\file_list:`*.*)

`\ior_list_streams:`
`\ior_log_streams:`
`\iow_list_streams:`
`\iow_log_streams:`

These got a more consistent naming.

```
\__debug_deprecation:nnNNpn { 2018-12-31 } { \ior_show_list: }
\cs_new_protected:Npn \ior_list_streams:   { \ior_show_list: }
\__debug_deprecation:nnNNpn { 2018-12-31 } { \ior_log_list: }
\cs_new_protected:Npn \ior_log_streams:    { \ior_log_list: }
\__debug_deprecation:nnNNpn { 2018-12-31 } { \iow_show_list: }
\cs_new_protected:Npn \iow_list_streams:   { \iow_show_list: }
\__debug_deprecation:nnNNpn { 2018-12-31 } { \iow_log_list: }
\cs_new_protected:Npn \iow_log_streams:    { \iow_log_list: }
```

(*End definition for* `\ior_list_streams:` *and others.*)

```
⟨/initex | package⟩
```

18 l3skip implementation

```
⟨*initex | package⟩
⟨@@=dim⟩
```

18.1 Length primitives renamed

\if_dim:w
__dim_eval:w
__dim_eval_end:

Primitives renamed.

```
\cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\cs_new_eq:NN \__dim_eval:w      \etex_dimexpr:D
\cs_new_eq:NN \__dim_eval_end:   \tex_relax:D
```

(*End definition for* `\if_dim:w`*,* `\__dim_eval:w`*, and* `\__dim_eval_end:`*. These functions are documented on page 162.*)

18.2 Creating and initialising dim variables

\dim_new:N
\dim_new:c

Allocating ⟨*dim*⟩ registers ...

```
⟨*package⟩
\cs_new_protected:Npn \dim_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newdimen \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \dim_new:N { c }
```

(*End definition for* `\dim_new:N`*. This function is documented on page 149.*)

\dim_const:Nn
\dim_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \dim_const:Nn #1
  {
    \dim_new:N #1
    \dim_gset:Nn #1
  }
\cs_generate_variant:Nn \dim_const:Nn { c }
```

(*End definition for* `\dim_const:Nn`*. This function is documented on page 149.*)

\dim_zero:N
\dim_zero:c
\dim_gzero:N
\dim_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\cs_generate_variant:Nn \dim_zero:N  { c }
\cs_generate_variant:Nn \dim_gzero:N { c }
```

(*End definition for* `\dim_zero:N` *and* `\dim_gzero:N`*. These functions are documented on page 149.*)

\dim_zero_new:N
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \dim_zero_new:N  #1
  { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\cs_new_protected:Npn \dim_gzero_new:N #1
  { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
\cs_generate_variant:Nn \dim_zero_new:N  { c }
\cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(*End definition for* `\dim_zero_new:N` *and* `\dim_gzero_new:N`*. These functions are documented on page 149.*)

`\dim_if_exist_p:N`
`\dim_if_exist_p:c`
`\dim_if_exist:NTF`
`\dim_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* `\dim_if_exist:NTF`*. This function is documented on page 149.*)

18.3 Setting dim variables

`\dim_set:Nn`
`\dim_set:cn`
`\dim_gset:Nn`
`\dim_gset:cn`

Setting dimensions is easy enough.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_set:Nn } }
\cs_new_protected:Npn \dim_set:Nn #1#2
  { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
\cs_generate_variant:Nn \dim_set:Nn  { c }
\cs_generate_variant:Nn \dim_gset:Nn { c }
```

(*End definition for* `\dim_set:Nn` *and* `\dim_gset:Nn`*. These functions are documented on page 150.*)

`\dim_set_eq:NN`
`\dim_set_eq:cN`
`\dim_set_eq:Nc`
`\dim_set_eq:cc`
`\dim_gset_eq:NN`
`\dim_gset_eq:cN`
`\dim_gset_eq:Nc`
`\dim_gset_eq:cc`

All straightforward.

```
\cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \dim_set_eq:NN {        c }
\cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \dim_gset_eq:NN {        c }
\cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
```

(*End definition for* `\dim_set_eq:NN` *and* `\dim_gset_eq:NN`*. These functions are documented on page 150.*)

`\dim_add:Nn`
`\dim_add:cn`
`\dim_gadd:Nn`
`\dim_gadd:cn`
`\dim_sub:Nn`
`\dim_sub:cn`
`\dim_gsub:Nn`
`\dim_gsub:cn`

Using `by` here deals with the (incorrect) case `\dimen123`.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_add:Nn } }
\cs_new_protected:Npn \dim_add:Nn #1#2
  { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\cs_generate_variant:Nn \dim_add:Nn  { c }
\cs_generate_variant:Nn \dim_gadd:Nn { c }
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_sub:Nn } }
\cs_new_protected:Npn \dim_sub:Nn #1#2
  { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
\cs_generate_variant:Nn \dim_sub:Nn  { c }
\cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(*End definition for* `\dim_add:Nn` *and others. These functions are documented on page 150.*)

18.4 Utilities for dimension calculations

\dim_abs:n
__dim_abs:N
\dim_max:nn
\dim_min:nn
__dim_maxmin:wwN

Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_abs:n } }
\cs_new:Npn \dim_abs:n #1
  {
    \exp_after:wN \__dim_abs:N
    \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
  }
\cs_new:Npn \__dim_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_max:nn }
    { \__debug_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_max:nn }
  }
\cs_new:Npn \dim_max:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      >
    \__dim_eval_end:
  }
\__debug_patch_args:nNNpn
  {
    { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_min:nn }
    { \__debug_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_min:nn }
  }
\cs_new:Npn \dim_min:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      <
    \__dim_eval_end:
  }
\cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
  {
    \if_dim:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
  }
```

(*End definition for* `\dim_abs:n` *and others. These functions are documented on page 150.*)

\dim_ratio:nn
__dim_ratio:n

With dimension expressions, something like `10 pt * ( 5 pt / 10 pt )` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\__int_value:w` forces everything into `sp`, avoiding any decimal parts.

```
\cs_new:Npn \dim_ratio:nn #1#2
```

```
  { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
\cs_new:Npn \__dim_ratio:n #1
  { \__int_value:w \__dim_eval:w (#1) \__dim_eval_end: }
```

(*End definition for* `\dim_ratio:nn` *and* `\__dim_ratio:n`*. These functions are documented on page 151.*)

18.5 Dimension expression conditionals

`\dim_compare_p:nNn`
`\dim_compare:nNnTF`

Simple comparison.

```
\__debug_patch_conditional_args:nNNpnn
  {
    { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_compare:nNn }
    { \__dim_eval_end: #2 }
    { \__debug_chk_expr:nNnN {#3} \__dim_eval:w { } \dim_compare:nNn }
  }
\prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\dim_compare:nNnTF`*. This function is documented on page 151.*)

`\dim_compare_p:n`
`\dim_compare:nTF`
`\__dim_compare:w`
`\__dim_compare:wNN`
`\__dim_compare_=:w`
`\__dim_compare_!:w`
`\__dim_compare_<:w`
`\__dim_compare_>:w`

This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `\__prg_compare_error:`. Just like for integers, the looping auxiliary `\__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three "simple" relations `<`, `=`, and `>`.

```
\prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__dim_compare:w
    \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
  }
\cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
  {
    \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
      \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__dim_compare:wNN #1 }
  { \tl_to_str:n {pt} }
  #2#3
  {
      \if_meaning:w = #3
        \use:c { __dim_compare_#2:w }
      \fi:
        #1 pt \exp_stop_f:
      \prg_return_false:
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    \reverse_if:N \if_dim:w #1 pt #2
      \exp_after:wN \__dim_compare:wNN
      \dim_use:N \__dim_eval:w #3
  }
\cs_new:cpn { __dim_compare_ ! :w }
    #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
\cs_new:cpn { __dim_compare_ = :w }
    #1 \__dim_eval:w = { #1 \__dim_eval:w }
\cs_new:cpn { __dim_compare_ < :w }
    #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
\cs_new:cpn { __dim_compare_ > :w }
    #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
\cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
  { #1 \prg_return_false: \else: \prg_return_true: \fi: }
```

(*End definition for* `\dim_compare:nTF` *and others. These functions are documented on page 152.*)

`\dim_case:nn`
`\dim_case:nnTF`
`\__dim_case:nnTF`
`\__dim_case:nw`
`\__dim_case_end:nw`

For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nn(TF)` as described in l3basics.

```
\cs_new:Npn \dim_case:nnTF #1
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
  }
\cs_new:Npn \dim_case:nnT #1#2#3
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \dim_case:nnF #1#2
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \dim_case:nn #1#2
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__dim_case:nnTF #1#2#3#4
  { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__dim_case:nw #1#2#3
  {
    \dim_compare:nNnTF {#1} = {#2}
      { \__dim_case_end:nw {#3} }
      { \__dim_case:nw {#1} }
  }
\cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\dim_case:nnTF` *and others. These functions are documented on page 153.*)

18.6 Dimension expression loops

`\dim_while_do:nn`
`\dim_until_do:nn`
`\dim_do_while:nn`
`\dim_do_until:nn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_new:Npn \dim_while_do:nn #1#2
```

```
  {
    \dim_compare:nT {#1}
      {
        #2
        \dim_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \dim_until_do:nn #1#2
  {
    \dim_compare:nF {#1}
      {
        #2
        \dim_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \dim_do_while:nn #1#2
  {
    #2
    \dim_compare:nT {#1}
      { \dim_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \dim_do_until:nn #1#2
  {
    #2
    \dim_compare:nF {#1}
      { \dim_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\dim_while_do:nn` *and others. These functions are documented on page 154.*)

`\dim_while_do:nNnn`
`\dim_until_do:nNnn`
`\dim_do_while:nNnn`
`\dim_do_until:nNnn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_new:Npn \dim_while_do:nNnn #1#2#3#4
  {
    \dim_compare:nNnT {#1} #2 {#3}
      {
        #4
        \dim_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \dim_until_do:nNnn #1#2#3#4
  {
  \dim_compare:nNnF {#1} #2 {#3}
    {
      #4
      \dim_until_do:nNnn {#1} #2 {#3} {#4}
    }
  }
\cs_new:Npn \dim_do_while:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnT {#1} #2 {#3}
      { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
  }
```

```
\cs_new:Npn \dim_do_until:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnF {#1} #2 {#3}
      { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\dim_while_do:nNnn` *and others. These functions are documented on page 154.*)

18.7 Using dim expressions and variables

`\dim_eval:n`

Evaluating a dimension expression expandably.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_eval:n } }
\cs_new:Npn \dim_eval:n #1
  { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_eval:n`*. This function is documented on page 154.*)

`\dim_use:N`
`\dim_use:c`

Accessing a ⟨*dim*⟩.

```
\cs_new_eq:NN \dim_use:N \tex_the:D
```

We hand-code this for some speed gain:

```
%\cs_generate_variant:Nn \dim_use:N { c }
\cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* `\dim_use:N`*. This function is documented on page 154.*)

`\dim_to_decimal:n`
`\__dim_to_decimal:w`

A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_to_decimal:n } }
\cs_new:Npn \dim_to_decimal:n #1
  {
    \exp_after:wN
      \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \__dim_to_decimal:w
      ##1 . ##2 \tl_to_str:n { pt }
  }
      {
        \int_compare:nNnTF {#2} > { 0 }
          { #1 . #2 }
          { #1 }
      }
```

(*End definition for* `\dim_to_decimal:n` *and* `\__dim_to_decimal:w`*. These functions are documented on page 155.*)

\dim_to_decimal_in_bp:n

Conversion to big points is done using a scaling inside __dim_eval:w as ε-TEX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```
\cs_new:Npn \dim_to_decimal_in_bp:n #1
  { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }
```

(*End definition for* \dim_to_decimal_in_bp:n. *This function is documented on page 155.*)

\dim_to_decimal_in_sp:n

Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_to_decimal_in_sp:n } }
\cs_new:Npn \dim_to_decimal_in_sp:n #1
  { \int_eval:n { \__dim_eval:w #1 \__dim_eval_end: } }
```

(*End definition for* \dim_to_decimal_in_sp:n. *This function is documented on page 155.*)

\dim_to_decimal_in_unit:nn

An analogue of \dim_ratio:nn that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```
\cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
  {
    \dim_to_decimal:n
      {
        1pt *
        \dim_ratio:nn {#1} {#2}
      }
  }
```

(*End definition for* \dim_to_decimal_in_unit:nn. *This function is documented on page 155.*)

\dim_to_fp:n

Defined in l3fp-convert, documented here.

(*End definition for* \dim_to_fp:n. *This function is documented on page 156.*)

18.8 Viewing dim variables

\dim_show:N
\dim_show:c

Diagnostics.

```
\cs_new_eq:NN  \dim_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \dim_show:N { c }
```

(*End definition for* \dim_show:N. *This function is documented on page 156.*)

\dim_show:n

Diagnostics. We don't use the TEX primitive \showthe to show dimension expressions: this gives a more unified output.

```
\cs_new_protected:Npn \dim_show:n
  { \__msg_show_wrap:Nn \dim_eval:n }
```

(*End definition for* \dim_show:n. *This function is documented on page 156.*)

\dim_log:N
\dim_log:c
\dim_log:n

Diagnostics. Redirect output of \dim_show:n to the log.

```
\cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\cs_new_eq:NN \dim_log:c \__kernel_register_log:c
\cs_new_protected:Npn \dim_log:n
  { \__msg_log_next: \dim_show:n }
```

(*End definition for* \dim_log:N *and* \dim_log:n. *These functions are documented on page 156.*)

18.9 Constant dimensions

\c_zero_dim
\c_max_dim

Constant dimensions.

```
\dim_const:Nn \c_zero_dim { 0 pt }
\dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(*End definition for* `\c_zero_dim` *and* `\c_max_dim`*. These variables are documented on page 156.*)

18.10 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
\g_tmpa_dim
\g_tmpb_dim

We provide two local and two global scratch registers, maybe we need more or less.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

(*End definition for* `\l_tmpa_dim` *and others. These variables are documented on page 156.*)

18.11 Creating and initialising skip variables

\skip_new:N
\skip_new:c

Allocation of a new internal registers.

```
⟨*package⟩
\cs_new_protected:Npn \skip_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newskip \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \skip_new:N { c }
```

(*End definition for* `\skip_new:N`*. This function is documented on page 157.*)

\skip_const:Nn
\skip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \skip_const:Nn #1
  {
    \skip_new:N #1
    \skip_gset:Nn #1
  }
\cs_generate_variant:Nn \skip_const:Nn { c }
```

(*End definition for* `\skip_const:Nn`*. This function is documented on page 157.*)

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\cs_generate_variant:Nn \skip_zero:N  { c }
\cs_generate_variant:Nn \skip_gzero:N { c }
```

(*End definition for* `\skip_zero:N` *and* `\skip_gzero:N`*. These functions are documented on page 157.*)

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \skip_zero_new:N  #1
  { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\cs_new_protected:Npn \skip_gzero_new:N #1
  { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
\cs_generate_variant:Nn \skip_zero_new:N  { c }
\cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(*End definition for* \skip_zero_new:N *and* \skip_gzero_new:N*. These functions are documented on page 157.*)

\skip_if_exist_p:N
\skip_if_exist_p:c
\skip_if_exist:N*TF*
\skip_if_exist:c*TF*

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* \skip_if_exist:NTF*. This function is documented on page 157.*)

18.12 Setting skip variables

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Much the same as for dimensions.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \etex_glueexpr:D { } \skip_set:Nn } }
\cs_new_protected:Npn \skip_set:Nn #1#2
  { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
\cs_generate_variant:Nn \skip_set:Nn  { c }
\cs_generate_variant:Nn \skip_gset:Nn { c }
```

(*End definition for* \skip_set:Nn *and* \skip_gset:Nn*. These functions are documented on page 157.*)

\skip_set_eq:NN
\skip_set_eq:cN
\skip_set_eq:Nc
\skip_set_eq:cc
\skip_gset_eq:NN
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \skip_set_eq:NN {      c }
\cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \skip_gset_eq:NN {      c }
\cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
```

(*End definition for* \skip_set_eq:NN *and* \skip_gset_eq:NN*. These functions are documented on page 157.*)

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Using by here deals with the (incorrect) case \skip123.

```
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \etex_glueexpr:D { } \skip_add:Nn } }
\cs_new_protected:Npn \skip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\cs_generate_variant:Nn \skip_add:Nn  { c }
\cs_generate_variant:Nn \skip_gadd:Nn { c }
\__debug_patch_args:nNNpn
  { {#1} { \__debug_chk_expr:nNnN {#2} \etex_glueexpr:D { } \skip_sub:Nn } }
\cs_new_protected:Npn \skip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
```

```
\cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
\cs_generate_variant:Nn \skip_sub:Nn  { c }
\cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(*End definition for* `\skip_add:Nn` *and others. These functions are documented on page 157.*)

18.13 Skip expression conditionals

\skip_if_eq_p:nn
\skip_if_eq:nnTF

Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
\prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if_int_compare:w
      \__str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
      = 0 \exp_stop_f:
        \prg_return_true:
    \else:
        \prg_return_false:
    \fi:
  }
```

(*End definition for* `\skip_if_eq:nnTF`*. This function is documented on page 158.*)

\skip_if_finite_p:n
\skip_if_finite:nTF
__skip_if_finite:wwNw

With ε-TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```
\cs_set_protected:Npn \__cs_tmp:w #1
  {
    \__debug_patch_conditional_args:nNNpnn
      {
        {
          \__debug_chk_expr:nNnN
            {##1} \etex_glueexpr:D { } \skip_if_finite:n
        }
      }
    \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
      {
        \exp_after:wN \__skip_if_finite:wwNw
        \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
        #1 ; \prg_return_true: \q_stop
      }
    \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
  }
\exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }
```

(*End definition for* `\skip_if_finite:nTF` *and* `\__skip_if_finite:wwNw`*. These functions are documented on page 158.*)

18.14 Using skip expressions and variables

\skip_eval:n

Evaluating a skip expression expandably.

```
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \etex_glueexpr:D { } \skip_eval:n } }
\cs_new:Npn \skip_eval:n #1
  { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }
```

(*End definition for* `\skip_eval:n`*. This function is documented on page 158.*)

\skip_use:N
\skip_use:c

Accessing a ⟨*skip*⟩.

```
\cs_new_eq:NN \skip_use:N \tex_the:D
%\cs_generate_variant:Nn \skip_use:N { c }
\cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* `\skip_use:N`*. This function is documented on page 158.*)

18.15 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Inserting skips.

```
\cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \etex_glueexpr:D { } \skip_horizontal:n } }
\cs_new:Npn \skip_horizontal:n #1
  { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
\__debug_patch_args:nNNpn
  { { \__debug_chk_expr:nNnN {#1} \etex_glueexpr:D { } \skip_vertical:n } }
\cs_new:Npn \skip_vertical:n #1
  { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
\cs_generate_variant:Nn \skip_horizontal:N { c }
\cs_generate_variant:Nn \skip_vertical:N { c }
```

(*End definition for* `\skip_horizontal:N` *and others. These functions are documented on page 159.*)

18.16 Viewing skip variables

\skip_show:N
\skip_show:c

Diagnostics.

```
\cs_new_eq:NN  \skip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \skip_show:N { c }
```

(*End definition for* `\skip_show:N`*. This function is documented on page 158.*)

\skip_show:n

Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
\cs_new_protected:Npn \skip_show:n
  { \__msg_show_wrap:Nn \skip_eval:n }
```

(*End definition for* `\skip_show:n`*. This function is documented on page 159.*)

\skip_log:N
\skip_log:c
\skip_log:n

Diagnostics. Redirect output of `\skip_show:n` to the log.

```
\cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\cs_new_eq:NN \skip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \skip_log:n
  { \__msg_log_next: \skip_show:n }
```

(*End definition for* `\skip_log:N` *and* `\skip_log:n`*. These functions are documented on page 159.*)

18.17 Constant skips

\c_zero_skip
\c_max_skip

Skips with no rubber component are just dimensions but need to terminate correctly.

```
\skip_const:Nn \c_zero_skip { \c_zero_dim }
\skip_const:Nn \c_max_skip { \c_max_dim }
```

(*End definition for* \c_zero_skip *and* \c_max_skip*. These functions are documented on page 159.*)

18.18 Scratch skips

\l_tmpa_skip
\l_tmpb_skip
\g_tmpa_skip
\g_tmpb_skip

We provide two local and two global scratch registers, maybe we need more or less.

```
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip
```

(*End definition for* \l_tmpa_skip *and others. These variables are documented on page 159.*)

18.19 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

And then we add muskips.

```
⟨*package⟩
\cs_new_protected:Npn \muskip_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newmuskip \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \muskip_new:N { c }
```

(*End definition for* \muskip_new:N*. This function is documented on page 160.*)

\muskip_const:Nn
\muskip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \muskip_const:Nn #1
  {
    \muskip_new:N #1
    \muskip_gset:Nn #1
  }
\cs_generate_variant:Nn \muskip_const:Nn { c }
```

(*End definition for* \muskip_const:Nn*. This function is documented on page 160.*)

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \muskip_zero:N #1
  { #1 \c_zero_muskip }
\cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
\cs_generate_variant:Nn \muskip_zero:N  { c }
\cs_generate_variant:Nn \muskip_gzero:N { c }
```

(*End definition for* \muskip_zero:N *and* \muskip_gzero:N*. These functions are documented on page 160.*)

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \muskip_zero_new:N  #1
  { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\cs_new_protected:Npn \muskip_gzero_new:N #1
  { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
\cs_generate_variant:Nn \muskip_zero_new:N  { c }
\cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(*End definition for* `\muskip_zero_new:N` *and* `\muskip_gzero_new:N`*. These functions are documented on page 160.*)

\muskip_if_exist_p:N
\muskip_if_exist_p:c
\muskip_if_exist:N*TF*
\muskip_if_exist:c*TF*

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End definition for* `\muskip_if_exist:NTF`*. This function is documented on page 160.*)

18.20 Setting muskip variables

\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn

This should be pretty familiar.

```
\__debug_patch_args:nNNpn
  {
    {#1}
    {
      \__debug_chk_expr:nNnN {#2} \etex_muexpr:D
        { \etex_mutoglue:D } \muskip_set:Nn
    }
  }
\cs_new_protected:Npn \muskip_set:Nn #1#2
  { #1 ~ \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
\cs_generate_variant:Nn \muskip_set:Nn  { c }
\cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(*End definition for* `\muskip_set:Nn` *and* `\muskip_gset:Nn`*. These functions are documented on page 161.*)

\muskip_set_eq:NN
\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \muskip_set_eq:NN {      c }
\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \muskip_gset_eq:NN {      c }
\cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
```

(*End definition for* `\muskip_set_eq:NN` *and* `\muskip_gset_eq:NN`*. These functions are documented on page 161.*)

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Using `by` here deals with the (incorrect) case `\muskip123`.

```
\__debug_patch_args:nNNpn
  {
    {#1}
    {
      \__debug_chk_expr:nNnN {#2} \etex_muexpr:D
        { \etex_mutoglue:D } \muskip_add:Nn
    }
  }
\cs_new_protected:Npn \muskip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
\cs_generate_variant:Nn \muskip_add:Nn  { c }
\cs_generate_variant:Nn \muskip_gadd:Nn { c }
\__debug_patch_args:nNNpn
  {
    {#1}
    {
      \__debug_chk_expr:nNnN {#2} \etex_muexpr:D
        { \etex_mutoglue:D } \muskip_sub:Nn
    }
  }
\cs_new_protected:Npn \muskip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
\cs_generate_variant:Nn \muskip_sub:Nn  { c }
\cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(*End definition for* `\muskip_add:Nn` *and others. These functions are documented on page 160.*)

18.21 Using muskip expressions and variables

\muskip_eval:n

Evaluating a muskip expression expandably.

```
\__debug_patch_args:nNNpn
  {
    {
      \__debug_chk_expr:nNnN {#1} \etex_muexpr:D
        { \etex_mutoglue:D } \muskip_eval:n
    }
  }
\cs_new:Npn \muskip_eval:n #1
  { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(*End definition for* `\muskip_eval:n`*. This function is documented on page 161.*)

\muskip_use:N
\muskip_use:c

Accessing a ⟨*muskip*⟩.

```
\cs_new_eq:NN \muskip_use:N \tex_the:D
\cs_generate_variant:Nn \muskip_use:N { c }
```

(*End definition for* `\muskip_use:N`*. This function is documented on page 161.*)

18.22 Viewing muskip variables

\muskip_show:N
\muskip_show:c

Diagnostics.

```
\cs_new_eq:NN  \muskip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \muskip_show:N { c }
```

(*End definition for* `\muskip_show:N`*. This function is documented on page 161.*)

\muskip_show:n

Diagnostics. We don't use the TEX primitive \showthe to show muskip expressions: this gives a more unified output.

```
\cs_new_protected:Npn \muskip_show:n
  { \__msg_show_wrap:Nn \muskip_eval:n }
```

(*End definition for* \muskip_show:n*. This function is documented on page 161.*)

\muskip_log:N
\muskip_log:c
\muskip_log:n

Diagnostics. Redirect output of \muskip_show:n to the log.

```
\cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \muskip_log:n
  { \__msg_log_next: \muskip_show:n }
```

(*End definition for* \muskip_log:N *and* \muskip_log:n*. These functions are documented on page 162.*)

18.23 Constant muskips

\c_zero_muskip
\c_max_muskip

Constant muskips given by their value.

```
\muskip_const:Nn \c_zero_muskip { 0 mu }
\muskip_const:Nn \c_max_muskip  { 16383.99999 mu }
```

(*End definition for* \c_zero_muskip *and* \c_max_muskip*. These functions are documented on page 162.*)

18.24 Scratch muskips

\l_tmpa_muskip
\l_tmpb_muskip
\g_tmpa_muskip
\g_tmpb_muskip

We provide two local and two global scratch registers, maybe we need more or less.

```
\muskip_new:N \l_tmpa_muskip
\muskip_new:N \l_tmpb_muskip
\muskip_new:N \g_tmpa_muskip
\muskip_new:N \g_tmpb_muskip
```

(*End definition for* \l_tmpa_muskip *and others. These variables are documented on page 162.*)

```
⟨/initex | package⟩
```

19 l3keys Implementation

```
⟨*initex | package⟩
```

19.1 Low-level interface

The low-level key parser is based heavily on keyval, but with a number of additional "safety" requirements and with the idea that the parsed list of key–value pairs can be processed in a variety of ways. The net result is that this code needs around twice the amount of time as keyval to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level expl3 interfaces.

```
⟨@@=keyval⟩
```

\l__keyval_key_tl
\l__keyval_value_tl

The current key name and value.

```
\tl_new:N \l__keyval_key_tl
\tl_new:N \l__keyval_value_tl
```

(*End definition for* \l__keyval_key_tl *and* \l__keyval_value_tl*.*)

\l__keyval_sanitise_tl

A token list variable for dealing with awkward category codes in the input.

```
\tl_new:N \l__keyval_sanitise_tl
```

(*End definition for* \l__keyval_sanitise_tl.)

\keyval_parse:NNn

The main function starts off by normalising category codes in package mode. That's relatively "expensive" so is skipped (hopefully) in format mode. We then hand off to the parser. The use of \q_mark here prevents loss of braces from the key argument. This particular quark is chosen as it fits in with __tl_trim_spaces:nn and allows a performance enhancement as the token can be carried through. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
\cs_new_protected:Npn \keyval_parse:NNn #1#2#3
  {
⟨*initex⟩
    \__keyval_loop:NNw #1#2 \q_mark #3 , \q_recursion_tail ,
⟨/initex⟩
⟨*package⟩
    \tl_set:Nn \l__keyval_sanitise_tl {#3}
    \__keyval_sanitise_equals:
    \__keyval_sanitise_comma:
    \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
      \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
⟨/package⟩
  }
```

(*End definition for* \keyval_parse:NNn. *This function is documented on page 176.*)

__keyval_sanitise_equals:
__keyval_sanitise_comma:
__keyval_sanitise_equals_auxi:w
__keyval_sanitise_equals_auxii:w
__keyval_sanitise_comma_auxi:w
__keyval_sanitise_comma_auxii:w
__keyval_sanitise_aux:w

A reasonably fast search and replace set up specifically for the active tokens. The nature of the input is known so everything is hard-coded. With only two tokens to cover, the speed gain from using dedicated functions is worth it.

```
⟨*package⟩
\group_begin:
  \char_set_catcode_active:n { `\= }
  \char_set_catcode_active:n { `\, }
  \cs_new_protected:Npn \__keyval_sanitise_equals:
    {
      \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
        \q_mark = \q_nil =
      \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
    }
    \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
      {
        \tl_set:Nn \l__keyval_sanitise_tl {#1}
        \__keyval_sanitise_equals_auxii:w
      }
    \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
      {
        \if_meaning:w \q_nil #1 \scan_stop:
        \else:
          \tl_set:Nx \l__keyval_sanitise_tl
            {
              \exp_not:o \l__keyval_sanitise_tl
              \token_to_str:N =
              \exp_not:n {#1}
            }
          \exp_after:wN \__keyval_sanitise_equals_auxii:w
        \fi:
      }
  \cs_new_protected:Npn \__keyval_sanitise_comma:
    {
      \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
        \q_mark , \q_nil ,
      \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
    }
    \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
      {
        \tl_set:Nn \l__keyval_sanitise_tl {#1}
        \__keyval_sanitise_comma_auxii:w
      }
    \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
      {
        \if_meaning:w \q_nil #1 \scan_stop:
        \else:
          \tl_set:Nx \l__keyval_sanitise_tl
            {
              \exp_not:o \l__keyval_sanitise_tl
              \token_to_str:N ,
              \exp_not:n {#1}
            }
          \exp_after:wN \__keyval_sanitise_comma_auxii:w
        \fi:
      }
\group_end:
\cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
  { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
⟨/package⟩
```

(*End definition for* `\__keyval_sanitise_equals:` *and others.*)

__keyval_loop:NNw

A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```
\cs_new_protected:Npn \__keyval_loop:NNw #1#2#3 ,
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
      \use_none:n #3 \prg_do_nothing:
    \else:
      \__keyval_split:NNw #1#2#3 == \q_stop
      \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
        \exp_after:wN \q_mark
    \fi:
  }
```

(*End definition for* `\__keyval_loop:NNw`.)

__keyval_split:NNw
__keyval_split_value:NNw
__keyval_split_tidy:w
__keyval_action:

The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front of the key and value. Doing this using an assignment is marginally faster than an an expansion chain.

```
\cs_new_protected:Npn \__keyval_split:NNw #1#2#3 =
  {
    \__keyval_def:Nn \l__keyval_key_tl {#3}
    \if_meaning:w \l__keyval_key_tl \c_empty_tl
      \exp_after:wN \__keyval_split_tidy:w
    \else:
      \exp_after:wN \__keyval_split_value:NNw \exp_after:wN #1 \exp_after:wN #2
        \exp_after:wN \q_mark
    \fi:
  }
\cs_new_protected:Npn \__keyval_split_value:NNw #1#2#3 = #4 \q_stop
  {
    \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
      \cs_set:Npx \__keyval_action:
        { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
    \else:
      \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #4 }
        \scan_stop:
        \__keyval_def:Nn \l__keyval_value_tl {#3}
        \cs_set:Npx \__keyval_action:
          {
            \exp_not:N #2
              { \exp_not:o \l__keyval_key_tl }
              { \exp_not:o \l__keyval_value_tl }
          }
      \else:
        \cs_set:Npn \__keyval_action:
          { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
      \fi:
    \fi:
    \__keyval_action:
  }
\cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
  {
    \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #1 }
      \scan_stop:
    \else:
      \exp_after:wN \__keyval_empty_key:
    \fi:
  }
\cs_new:Npn \__keyval_action: { }
\cs_new_protected:Npn \__keyval_empty_key:
  { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
```

(*End definition for* `\__keyval_split:NNw` *and others.*)

__keyval_def:Nn
__keyval_def_aux:n
__keyval_def_aux:w

First trim spaces off, then potentially remove a set of braces. By using the internal interface `\__tl_trim_spaces:nn` we can take advantage of the fact it needs a leading

\q_mark in this process. The \exp_after:wN removes the quark, the delimited argument deals with any braces.

```
\cs_new_protected:Npn \__keyval_def:Nn #1#2
  { \tl_set:Nx #1 { \__tl_trim_spaces:nn {#2} \__keyval_def_aux:n } }
\cs_new:Npn \__keyval_def_aux:n #1
  { \exp_after:wN \__keyval_def_aux:w #1 \q_stop }
\cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }
```

(*End definition for __keyval_def:Nn, __keyval_def_aux:n, and __keyval_def_aux:w.*)

One message for the low level parsing system.

```
\__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
  { Misplaced~equals~sign~in~key-value~input~\msg_line_number: }
  {
    LaTeX~is~attempting~to~parse~some~key-value~input~but~found~
    two~equals~signs~not~separated~by~a~comma.
  }
```

19.2 Constants and variables

```
⟨@@=keys⟩
```

\c__keys_code_root_tl
\c__keys_default_root_tl
\c__keys_groups_root_tl
\c__keys_inherit_root_tl
\c__keys_type_root_tl
\c__keys_validate_root_tl

Various storage areas for the different data which make up keys.

```
\tl_const:Nn \c__keys_code_root_tl     { key~code~>~ }
\tl_const:Nn \c__keys_default_root_tl  { key~default~>~ }
\tl_const:Nn \c__keys_groups_root_tl   { key~groups~>~ }
\tl_const:Nn \c__keys_inherit_root_tl  { key~inherit~>~ }
\tl_const:Nn \c__keys_type_root_tl     { key~type~>~ }
\tl_const:Nn \c__keys_validate_root_tl { key~validate~>~ }
```

(*End definition for \c__keys_code_root_tl and others.*)

\c__keys_props_root_tl

The prefix for storing properties.

```
\tl_const:Nn \c__keys_props_root_tl { key~prop~>~ }
```

(*End definition for \c__keys_props_root_tl.*)

\l_keys_choice_int
\l_keys_choice_tl

Publicly accessible data on which choice is being used when several are generated as a set.

```
\int_new:N \l_keys_choice_int
\tl_new:N \l_keys_choice_tl
```

(*End definition for \l_keys_choice_int and \l_keys_choice_tl. These variables are documented on page 170.*)

\l__keys_groups_clist

Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
\clist_new:N \l__keys_groups_clist
```

(*End definition for \l__keys_groups_clist.*)

\l_keys_key_tl

The name of a key itself: needed when setting keys.

```
\tl_new:N \l_keys_key_tl
```

(*End definition for \l_keys_key_tl. This variable is documented on page 172.*)

\l__keys_module_tl

The module for an entire set of keys.

```
\tl_new:N \l__keys_module_tl
```

(*End definition for* `\l__keys_module_tl`.)

\l__keys_no_value_bool

A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```
\bool_new:N \l__keys_no_value_bool
```

(*End definition for* `\l__keys_no_value_bool`.)

\l__keys_only_known_bool

Used to track if only "known" keys are being set.

```
\bool_new:N \l__keys_only_known_bool
```

(*End definition for* `\l__keys_only_known_bool`.)

\l_keys_path_tl

The "path" of the current key is stored here: this is available to the programmer and so is public.

```
\tl_new:N \l_keys_path_tl
```

(*End definition for* `\l_keys_path_tl`. *This variable is documented on page 172.*)

\l__keys_property_tl

The "property" begin set for a key at definition time is stored here.

```
\tl_new:N \l__keys_property_tl
```

(*End definition for* `\l__keys_property_tl`.)

\l__keys_selective_bool
\l__keys_filtered_bool

Two flags for using key groups: one to indicate that "selective" setting is active, a second to specify which type ("opt-in" or "opt-out").

```
\bool_new:N \l__keys_selective_bool
\bool_new:N \l__keys_filtered_bool
```

(*End definition for* `\l__keys_selective_bool` *and* `\l__keys_filtered_bool`.)

\l__keys_selective_seq

The list of key groups being filtered in or out during selective setting.

```
\seq_new:N \l__keys_selective_seq
```

(*End definition for* `\l__keys_selective_seq`.)

\l__keys_unused_clist

Used when setting only some keys to store those left over.

```
\tl_new:N \l__keys_unused_clist
```

(*End definition for* `\l__keys_unused_clist`.)

\l_keys_value_tl

The value given for a key: may be empty if no value was given.

```
\tl_new:N \l_keys_value_tl
```

(*End definition for* `\l_keys_value_tl`. *This variable is documented on page 172.*)

\l__keys_tmp_bool

Scratch space.

```
\bool_new:N \l__keys_tmp_bool
```

(*End definition for* `\l__keys_tmp_bool`.)

19.3 The key defining mechanism

\keys_define:nn
__keys_define:nnn
__keys_define:onn

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```
\cs_new_protected:Npn \keys_define:nn
  { \__keys_define:onn \l__keys_module_tl }
\cs_new_protected:Npn \__keys_define:nnn #1#2#3
  {
    \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
    \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
    \tl_set:Nn \l__keys_module_tl {#1}
  }
\cs_generate_variant:Nn \__keys_define:nnn { o }
```

(*End definition for* \keys_define:nn *and* __keys_define:nnn*. These functions are documented on page 165.*)

__keys_define:n
__keys_define:nn
__keys_define_aux:nn

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
\cs_new_protected:Npn \__keys_define:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_define_aux:nn {#1} { }
  }
\cs_new_protected:Npn \__keys_define:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_define_aux:nn {#1} {#2}
  }
\cs_new_protected:Npn \__keys_define_aux:nn #1#2
  {
    \__keys_property_find:n {#1}
    \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
      { \__keys_define_code:n {#2} }
      {
         \tl_if_empty:NF \l__keys_property_tl
           {
             \__msg_kernel_error:nnxx { kernel } { property-unknown }
              { \l__keys_property_tl } { \l_keys_path_tl }
           }
      }
  }
```

(*End definition for* __keys_define:n *,* __keys_define:nn *, and* __keys_define_aux:nn*.*)

__keys_property_find:n
__keys_property_find:w

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```
\cs_new_protected:Npn \__keys_property_find:n #1
  {
    \tl_set:Nx \l__keys_property_tl { \__keys_remove_spaces:n {#1} }
    \exp_after:wN \__keys_property_find:w \l__keys_property_tl . . \q_stop {#1}
  }
\cs_new_protected:Npn \__keys_property_find:w #1 . #2 . #3 \q_stop #4
  {
    \tl_if_blank:nTF {#3}
      {
        \tl_clear:N \l__keys_property_tl
        \__msg_kernel_error:nnn { kernel } { key-no-property } {#4}
      }
      {
        \str_if_eq:nnTF {#3} { . }
          {
            \tl_set:Nx \l_keys_path_tl
              {
                \tl_if_empty:NF \l__keys_module_tl
                  { \l__keys_module_tl  / }
                #1
              }
            \tl_set:Nn \l__keys_property_tl { . #2 }
          }
          {
            \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / #1 . #2 }
            \__keys_property_search:w #3 \q_stop
          }
      }
  }
\cs_new_protected:Npn \__keys_property_search:w #1 . #2 \q_stop
  {
    \str_if_eq:nnTF {#2} { . }
      {
        \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl }
        \tl_set:Nn \l__keys_property_tl { . #1 }
      }
      {
        \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . #1 }
        \__keys_property_search:w #2 \q_stop
      }
  }
```

(*End definition for* `\__keys_property_find:n` *and* `\__keys_property_find:w`.)

`\__keys_define_code:n`
`\__keys_define_code:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it was missing the earlier tests would have failed.

```
\cs_new_protected:Npn \__keys_define_code:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \exp_after:wN \__keys_define_code:w
          \l__keys_property_tl \q_stop
          { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
          {
            \__msg_kernel_error:nnxx { kernel }
              { property-requires-value } { \l__keys_property_tl }
```

```
       { \l_keys_path_tl }
      }
   }
   { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
 }
\use:x
 {
  \cs_new:Npn \exp_not:N \__keys_define_code:w
   ##1 \c_colon_str ##2 \exp_not:N \q_stop
 }
 { \tl_if_empty:nTF {#2} }
```

(*End definition for* __keys_define_code:n *and* __keys_define_code:w.)

19.4 Turning properties into actions

__keys_bool_set:Nn
__keys_bool_set:cn

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or g for global.

```
\cs_new_protected:Npn \__keys_bool_set:Nn #1#2
 {
  \bool_if_exist:NF #1 { \bool_new:N #1 }
  \__keys_choice_make:
  \__keys_cmd_set:nx { \l_keys_path_tl / true }
   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
  \__keys_cmd_set:nx { \l_keys_path_tl / false }
   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
  \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
   {
    \__msg_kernel_error:nnx { kernel } { boolean-values-only }
     { \l_keys_key_tl }
   }
  \__keys_default_set:n { true }
 }
\cs_generate_variant:Nn \__keys_bool_set:Nn { c }
```

(*End definition for* __keys_bool_set:Nn.)

__keys_bool_set_inverse:Nn
__keys_bool_set_inverse:cn

Inverse boolean setting is much the same.

```
\cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
 {
  \bool_if_exist:NF #1 { \bool_new:N #1 }
  \__keys_choice_make:
  \__keys_cmd_set:nx { \l_keys_path_tl / true }
   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
  \__keys_cmd_set:nx { \l_keys_path_tl / false }
   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
  \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
   {
    \__msg_kernel_error:nnx { kernel } { boolean-values-only }
     { \l_keys_key_tl }
   }
  \__keys_default_set:n { true }
 }
\cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
```

(*End definition for* `\__keys_bool_set_inverse:Nn`.)

`\__keys_choice_make:`
`\__keys_multichoice_make:`
`\__keys_choice_make:N`
`\__keys_choice_make_aux:N`

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

```
\cs_new_protected:Npn \__keys_choice_make:
  { \__keys_choice_make:N \__keys_choice_find:n }
\cs_new_protected:Npn \__keys_multichoice_make:
  { \__keys_choice_make:N \__keys_multichoice_find:n }
\cs_new_protected:Npn \__keys_choice_make:N #1
  {
    \cs_if_exist:cTF
      { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl }
      {
        \str_if_eq_x:nnTF
          { \exp_not:v { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl } }
          { choice }
          {
            \__msg_kernel_error:nnxx { kernel } { nested-choice-key }
              { \l_keys_path_tl } { \__keys_parent:o \l_keys_path_tl }
          }
          { \__keys_choice_make_aux:N #1 }
      }
      { \__keys_choice_make_aux:N #1 }
  }
\cs_new_protected:Npn \__keys_choice_make_aux:N #1
  {
    \cs_set_nopar:cpn { \c__keys_type_root_tl \l_keys_path_tl } { choice }
    \__keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \__msg_kernel_error:nnxx { kernel } { key-choice-unknown }
          { \l_keys_path_tl } {##1}
      }
  }
```

(*End definition for* `\__keys_choice_make:` *and others.*)

`\__keys_choices_make:nn`
`\__keys_multichoices_make:nn`
`\__keys_choices_make:Nnn`

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```
\cs_new_protected:Npn \__keys_choices_make:nn
  { \__keys_choices_make:Nnn \__keys_choice_make: }
\cs_new_protected:Npn \__keys_multichoices_make:nn
  { \__keys_choices_make:Nnn \__keys_multichoice_make: }
\cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
  {
    #1
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#2}
      {
        \int_incr:N \l_keys_choice_int
        \__keys_cmd_set:nx { \l_keys_path_tl / \__keys_remove_spaces:n {##1} }
          {
            \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
            \int_set:Nn \exp_not:N \l_keys_choice_int
              { \int_use:N \l_keys_choice_int }
            \exp_not:n {#3}
          }
      }
  }
```

(*End definition for* `\__keys_choices_make:nn` *,* `\__keys_multichoices_make:nn` *, and* `\__keys_choices_make:Nnn`*.*)

`\__keys_cmd_set:nn`
`\__keys_cmd_set:nx`
`\__keys_cmd_set:Vn`
`\__keys_cmd_set:Vo`

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```
\__debug_patch:nnNNpn
  {
    \cs_if_exist:cF { \c__keys_code_root_tl #1 }
      { \__debug_log:x { Defining~key~#1~\msg_line_context: } }
  }
  { }
\cs_new_protected:Npn \__keys_cmd_set:nn #1#2
  { \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 {#2} }
\cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
```

(*End definition for* `\__keys_cmd_set:nn`*.*)

`\__keys_default_set:n`

Setting a default value is easy. These are stored using `\cs_set:cpx` as this avoids any worries about whether a token list exists.

```
\cs_new_protected:Npn \__keys_default_set:n #1
  {
    \tl_if_empty:nTF {#1}
      {
        \cs_set_eq:cN
          { \c__keys_default_root_tl \l_keys_path_tl }
          \tex_undefined:D
      }
      {
        \cs_set:cpx
          { \c__keys_default_root_tl \l_keys_path_tl }
          { \exp_not:n {#1} }
      }
  }
```

(*End definition for* `\__keys_default_set:n`*.*)

`\__keys_groups_set:n`

Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```
\cs_new_protected:Npn \__keys_groups_set:n #1
  {
    \clist_set:Nn \l__keys_groups_clist {#1}
    \clist_if_empty:NTF \l__keys_groups_clist
      {
        \cs_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
          \tex_undefined:D
      }
      {
        \cs_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
          \l__keys_groups_clist
      }
  }
```

(*End definition for* `\__keys_groups_set:n`.)

`\__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```
\cs_new_protected:Npn \__keys_inherit:n #1
  {
    \__keys_undefine:
    \cs_set_nopar:cpn { \c__keys_inherit_root_tl \l_keys_path_tl } {#1}
  }
```

(*End definition for* `\__keys_inherit:n`.)

`\__keys_initialise:n` A set up for initialisation: just run the code if it exists.

```
\cs_new_protected:Npn \__keys_initialise:n #1
  {
    \cs_if_exist_use:cT { \c__keys_code_root_tl \l_keys_path_tl } { {#1} }
  }
```

(*End definition for* `\__keys_initialise:n`.)

`\__keys_meta_make:n`
`\__keys_meta_make:nn` To create a meta-key, simply set up to pass data through.

```
\cs_new_protected:Npn \__keys_meta_make:n #1
  {
    \__keys_cmd_set:Vo \l_keys_path_tl
      {
        \exp_after:wN \keys_set:nn
        \exp_after:wN { \l__keys_module_tl } {#1}
      }
  }
\cs_new_protected:Npn \__keys_meta_make:nn #1#2
  { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(*End definition for* `\__keys_meta_make:n` *and* `\__keys_meta_make:nn`.)

`\__keys_undefine:` Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```
\cs_new_protected:Npn \__keys_undefine:
  {
    \clist_map_inline:nn
      { code , default , groups , inherit , type , validate }
      {
        \cs_set_eq:cN
          { \tl_use:c { c__keys_ ##1 _root_tl } \l_keys_path_tl }
          \tex_undefined:D
      }
  }
```

(*End definition for* `\__keys_undefine:`.)

__keys_value_requirement:nn
__keys_validate_forbidden:
__keys_validate_required:
__keys_validate_cleanup:w

Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

```
\cs_new_protected:Npn \__keys_value_requirement:nn #1#2
  {
    \str_case:nnF {#2}
      {
        { true }
          {
            \cs_set_eq:cc
              { \c__keys_validate_root_tl \l_keys_path_tl }
              { __keys_validate_ #1 : }
          }
        { false }
          {
            \cs_if_eq:ccT
              { \c__keys_validate_root_tl \l_keys_path_tl }
              { __keys_validate_ #1 : }
              {
                \cs_set_eq:cN
                  { \c__keys_validate_root_tl \l_keys_path_tl }
                  \tex_undefined:D
              }
          }
      }
      {
        \__msg_kernel_error:nnx { kernel } { property-boolean-values-only }
          { .value_ #1 :n }
      }
  }
\cs_new_protected:Npn \__keys_validate_forbidden:
  {
    \bool_if:NF \l__keys_no_value_bool
      {
        \__msg_kernel_error:nnxx { kernel } { value-forbidden }
          { \l_keys_path_tl } { \l_keys_value_tl }
        \__keys_validate_cleanup:w
      }
  }
\cs_new_protected:Npn \__keys_validate_required:
  {
    \bool_if:NT \l__keys_no_value_bool
      {
        \__msg_kernel_error:nnx { kernel } { value-required }
          { \l_keys_path_tl }
        \__keys_validate_cleanup:w
      }
  }
\cs_new_protected:Npn \__keys_validate_cleanup:w #1 \cs_end: #2#3 { }
```

(*End definition for* __keys_value_requirement:nn *and others.*)

__keys_variable_set:NnnN
__keys_variable_set:cnnN

Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```
\cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
  {
    \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
    \__keys_cmd_set:nx { \l_keys_path_tl }
      {
        \exp_not:c { #2 _ #3 set:N #4 }
        \exp_not:N #1
        \exp_not:n  { {##1} }
      }
  }
\cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
```

(*End definition for* `\__keys_variable_set:NnnN`*.*)

19.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have "normal" argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
  { \__keys_bool_set:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
  { \__keys_bool_set:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
  { \__keys_bool_set:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
  { \__keys_bool_set:cn {#1} { g } }
```

(*End definition for* `.bool_set:N` *and* `.bool_gset:N`*. These functions are documented on page 166.*)

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { g } }
```

(*End definition for* `.bool_set_inverse:N` *and* `.bool_gset_inverse:N`*. These functions are documented on page 166.*)

`.choice:`

Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .choice: }
  { \__keys_choice_make: }
```

(*End definition for* `.choice:`*. This function is documented on page 166.*)

.choices:nn
.choices:Vn
.choices:on
.choices:xn

For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
  { \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
  { \exp_args:NV \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
  { \exp_args:No \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
  { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(*End definition for* `.choices:nn`. *This function is documented on page 166.*)

.code:n

Creating code is simply a case of passing through to the underlying `set` function.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
  { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(*End definition for* `.code:n`. *This function is documented on page 166.*)

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(*End definition for* `.clist_set:N` *and* `.clist_gset:N`. *These functions are documented on page 166.*)

.default:n
.default:V
.default:o
.default:x

Expansion is left to the internal functions.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
  { \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
  { \exp_args:NV \__keys_default_set:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
  { \exp_args:No \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
  { \exp_args:Nx \__keys_default_set:n {#1} }
```

(*End definition for* `.default:n`. *This function is documented on page 167.*)

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
  { \__keys_variable_set:NnnN #1 { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
  { \__keys_variable_set:cnnN {#1} { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
  { \__keys_variable_set:NnnN #1 { dim } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { dim } { g } n }
```

(*End definition for* `.dim_set:N` *and* `.dim_gset:N`. *These functions are documented on page 167.*)

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
  { \__keys_variable_set:NnnN #1 { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
  { \__keys_variable_set:cnnN {#1} { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
  { \__keys_variable_set:NnnN #1 { fp } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { fp } { g } n }
```

(*End definition for* `.fp_set:N` *and* `.fp_gset:N`*. These functions are documented on page 167.*)

.groups:n

A single property to create groups of keys.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
  { \__keys_groups_set:n {#1} }
```

(*End definition for* `.groups:n`*. This function is documented on page 167.*)

.inherit:n

Nothing complex: only one variant at the moment!

```
\cs_new_protected:cpn { \c__keys_props_root_tl .inherit:n } #1
  { \__keys_inherit:n {#1} }
```

(*End definition for* `.inherit:n`*. This function is documented on page 167.*)

.initial:n
.initial:V
.initial:o
.initial:x

The standard hand-off approach.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
  { \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
  { \exp_args:NV \__keys_initialise:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
  { \exp_args:No \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
  { \exp_args:Nx \__keys_initialise:n {#1} }
```

(*End definition for* `.initial:n`*. This function is documented on page 168.*)

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
  { \__keys_variable_set:NnnN #1 { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
  { \__keys_variable_set:cnnN {#1} { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
  { \__keys_variable_set:NnnN #1 { int } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(*End definition for* `.int_set:N` *and* `.int_gset:N`*. These functions are documented on page 168.*)

.meta:n

Making a meta is handled internally.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
  { \__keys_meta_make:n {#1} }
```

(*End definition for* `.meta:n`*. This function is documented on page 168.*)

`.meta:nn`

Meta with path: potentially lots of variants, but for the moment no so many defined.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
  { \__keys_meta_make:nn #1 }
```

(*End definition for* `.meta:nn`. *This function is documented on page 168.*)

`.multichoice:`
`.multichoices:nn`
`.multichoices:Vn`
`.multichoices:on`
`.multichoices:xn`

The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoice: }
  { \__keys_multichoice_make: }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
  { \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
  { \exp_args:NV \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
  { \exp_args:No \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
  { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(*End definition for* `.multichoice:` *and* `.multichoices:nn`. *These functions are documented on page 168.*)

`.skip_set:N`
`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
  { \__keys_variable_set:NnnN #1 { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
  { \__keys_variable_set:cnnN {#1} { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
  { \__keys_variable_set:NnnN #1 { skip } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(*End definition for* `.skip_set:N` *and* `.skip_gset:N`. *These functions are documented on page 168.*)

`.tl_set:N`
`.tl_set:c`
`.tl_gset:N`
`.tl_gset:c`
`.tl_set_x:N`
`.tl_set_x:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(*End definition for* `.tl_set:N` *and others. These functions are documented on page 168.*)

`.undefine:`

Another simple wrapper.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .undefine: }
  { \__keys_undefine: }
```

(*End definition for* .undefine:. *This function is documented on page 169.*)

.value_forbidden:n
.value_required:n

These are very similar, so both call the same function.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
  { \__keys_value_requirement:nn { forbidden } {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
  { \__keys_value_requirement:nn { required } {#1} }
```

(*End definition for* .value_forbidden:n *and* .value_required:n*. These functions are documented on page 169.*)

19.6 Setting keys

\keys_set:nn
\keys_set:nV
\keys_set:nv
\keys_set:no
__keys_set:nnn
__keys_set:onn

A simple wrapper again.

```
\cs_new_protected:Npn \keys_set:nn
  { \__keys_set:onn { \l__keys_module_tl } }
\cs_new_protected:Npn \__keys_set:nnn #1#2#3
  {
    \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
    \keyval_parse:NNn \__keys_set:n \__keys_set:nn {#3}
    \tl_set:Nn \l__keys_module_tl {#1}
  }
\cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
\cs_generate_variant:Nn \__keys_set:nnn { o }
```

(*End definition for* \keys_set:nn *and* __keys_set:nnn*. These functions are documented on page 172.*)

\keys_set_known:nnN
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:noN
__keys_set_known:nnnN
__keys_set_known:onnN
\keys_set_known:nn
\keys_set_known:nV
\keys_set_known:nv
\keys_set_known:no
__keys_keys_set_known:nn

Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of \l__keys_unused_clist is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a tl operation to set the clist here!

```
\cs_new_protected:Npn \keys_set_known:nnN
  { \__keys_set_known:onnN \l__keys_unused_clist }
\cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
  {
    \clist_clear:N \l__keys_unused_clist
    \keys_set_known:nn {#2} {#3}
    \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
    \tl_set:Nn \l__keys_unused_clist {#1}
  }
\cs_generate_variant:Nn \__keys_set_known:nnnN { o }
\cs_new_protected:Npn \keys_set_known:nn #1#2
  {
    \bool_if:NTF \l__keys_only_known_bool
      { \keys_set:nn }
      { \__keys_set_known:nn }
      {#1} {#2}
  }
\cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
\cs_new_protected:Npn \__keys_set_known:nn #1#2
  {
    \bool_set_true:N \l__keys_only_known_bool
    \keys_set:nn {#1} {#2}
    \bool_set_false:N \l__keys_only_known_bool
  }
```

(*End definition for* `\keys_set_known:nnN` *and others. These functions are documented on page 173.*)

\keys_set_filter:nnnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
__keys_set_filter:nnnnN
__keys_set_filter:onnnN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nno
__keys_set_filter:nnn
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
__keys_set_groups:nnn
__keys_set_selective:nnn
__keys_set_selective:nnnn
__keys_set_selective:onnn
__keys_set_selective:nn

The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here. We have a bit more shuffling to do to keep everything nestable.

```
\cs_new_protected:Npn \keys_set_filter:nnnN
  {  \__keys_set_filter:onnnN \l__keys_unused_clist }
\cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
\cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
  {
    \clist_clear:N \l__keys_unused_clist
    \keys_set_filter:nnn {#2} {#3} {#4}
    \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
    \tl_set:Nn \l__keys_unused_clist {#1}
  }
\cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
\cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
  {
    \bool_if:NTF \l__keys_filtered_bool
      { \__keys_set_selective:nnn }
      { \__keys_set_filter:nnn }
      {#1} {#2} {#3}
  }
\cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
\cs_new_protected:Npn \__keys_set_filter:nnn #1#2#3
  {
    \bool_set_true:N \l__keys_filtered_bool
    \__keys_set_selective:nnn {#1} {#2} {#3}
    \bool_set_false:N \l__keys_filtered_bool
  }
\cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
  {
    \bool_if:NTF \l__keys_filtered_bool
      { \__keys_set_groups:nnn }
      { \__keys_set_selective:nnn }
      {#1} {#2} {#3}
  }
\cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
\cs_new_protected:Npn \__keys_set_groups:nnn #1#2#3
  {
    \bool_set_false:N \l__keys_filtered_bool
    \__keys_set_selective:nnn {#1} {#2} {#3}
    \bool_set_true:N \l__keys_filtered_bool
  }
\cs_new_protected:Npn \__keys_set_selective:nnn
  { \__keys_set_selective:onnn \l__keys_selective_seq }
\cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
  {
    \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
    \bool_if:NTF \l__keys_selective_bool
      { \keys_set:nn }
      { \__keys_set_selective:nn }
      {#2} {#4}
    \tl_set:Nn \l__keys_selective_seq {#1}
  }
\cs_generate_variant:Nn \__keys_set_selective:nnnn { o }
\cs_new_protected:Npn \__keys_set_selective:nn #1#2
  {
    \bool_set_true:N \l__keys_selective_bool
    \keys_set:nn {#1} {#2}
    \bool_set_false:N \l__keys_selective_bool
  }
```

(*End definition for* `\keys_set_filter:nnnN` *and others. These functions are documented on page 174.*)

__keys_set:n
__keys_set:nn
__keys_set_aux:nnn
__keys_set_aux:onn
__keys_find_key_module:w
__keys_set_aux:
__keys_set_selective:

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```
\cs_new_protected:Npn \__keys_set:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_set_aux:onn \l__keys_module_tl {#1} { }
  }
\cs_new_protected:Npn \__keys_set:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_set_aux:onn \l__keys_module_tl {#1} {#2}
  }
```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```
\cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
  {
    \tl_set:Nx \l_keys_path_tl
      {
        \tl_if_blank:nF {#1}
          { #1 / }
        \__keys_remove_spaces:n {#2}
      }
    \tl_clear:N \l__keys_module_tl
    \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
    \__keys_value_or_default:n {#3}
    \bool_if:NTF \l__keys_selective_bool
      { \__keys_set_selective: }
      { \__keys_execute: }
    \tl_set:Nn \l__keys_module_tl {#1}
  }
\cs_generate_variant:Nn \__keys_set_aux:nnn { o }
\cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
  {
    \tl_if_blank:nTF {#2}
      { \tl_set:Nn \l_keys_key_tl {#1} }
      {
        \tl_put_right:Nx \l__keys_module_tl
          {
            \tl_if_empty:NF \l__keys_module_tl { / }
            #1
          }
        \__keys_find_key_module:w #2 \q_stop
      }
  }
```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```
\cs_new_protected:Npn \__keys_set_selective:
  {
    \cs_if_exist:cTF { \c__keys_groups_root_tl \l_keys_path_tl }
      {
        \clist_set_eq:Nc \l__keys_groups_clist
          { \c__keys_groups_root_tl \l_keys_path_tl }
        \__keys_check_groups:
      }
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_execute: }
          { \__keys_store_unused: }
      }
  }
```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set.

```
\cs_new_protected:Npn \__keys_check_groups:
  {
    \bool_set_false:N \l__keys_tmp_bool
    \seq_map_inline:Nn \l__keys_selective_seq
      {
        \clist_map_inline:Nn \l__keys_groups_clist
          {
            \str_if_eq:nnT {##1} {####1}
              {
                \bool_set_true:N \l__keys_tmp_bool
                \clist_map_break:n { \seq_map_break: }
              }
          }
      }
    \bool_if:NTF \l__keys_tmp_bool
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_store_unused: }
          { \__keys_execute: }
      }
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_execute: }
          { \__keys_store_unused: }
      }
  }
```

(*End definition for* `\__keys_set:n` *and others.*)

`\__keys_value_or_default:n`

If a value is given, return it as `#1`, otherwise send a default if available.

```
\cs_new_protected:Npn \__keys_value_or_default:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
          {
            \tl_set_eq:Nc
              \l_keys_value_tl
              { \c__keys_default_root_tl \l_keys_path_tl }
          }
          { \tl_clear:N \l_keys_value_tl }
      }
      { \tl_set:Nn \l_keys_value_tl {#1} }
  }
```

(*End definition for* `\__keys_value_or_default:n`*.*)

`\__keys_execute:`
`\__keys_execute_unknown:`
`\__keys_execute:nn`
`\__keys_store_unused:`

Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```
\cs_new_protected:Npn \__keys_execute:
  {
    \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }
      {
        \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
        \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
          \exp_after:wN { \l_keys_value_tl }
      }
      { \__keys_execute_unknown: }
  }
\cs_new_protected:Npn \__keys_execute_unknown:
  {
    \bool_if:NTF \l__keys_only_known_bool
      { \__keys_store_unused: }
      {
        \cs_if_exist:cTF
          { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
          {
            \clist_map_inline:cn
              { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
              {
                \cs_if_exist:cT
                  { \c__keys_code_root_tl ##1 / \l_keys_key_tl }
                  {
                    \cs:w \c__keys_code_root_tl ##1 / \l_keys_key_tl
                      \exp_after:wN \cs_end: \exp_after:wN
                      { \l_keys_value_tl }
                    \clist_map_break:
                  }
              }
          }
          {
            \cs_if_exist:cTF { \c__keys_code_root_tl \l__keys_module_tl / unknown }
              {
                \cs:w \c__keys_code_root_tl \l__keys_module_tl / unknown
                  \exp_after:wN \cs_end: \exp_after:wN { \l_keys_value_tl }
              }
              {
                \__msg_kernel_error:nnxx { kernel } { key-unknown }
                  { \l_keys_path_tl } { \l__keys_module_tl }
              }
          }
       }
  }
\cs_new:Npn \__keys_execute:nn #1#2
  {
    \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
      {
        \cs:w \c__keys_code_root_tl #1 \exp_after:wN \cs_end:
          \exp_after:wN { \l_keys_value_tl }
      }
      {#2}
  }
\cs_new_protected:Npn \__keys_store_unused:
  {
    \clist_put_right:Nx \l__keys_unused_clist
      {
        \exp_not:o \l_keys_key_tl
        \bool_if:NF \l__keys_no_value_bool
          { = { \exp_not:o \l_keys_value_tl } }
      }
  }
```

(*End definition for* `\__keys_execute:` *and others.*)

`\__keys_choice_find:n`
`\__keys_multichoice_find:n`

Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That always exists, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```
\cs_new:Npn \__keys_choice_find:n #1
  {
    \__keys_execute:nn { \l_keys_path_tl / \__keys_remove_spaces:n {#1} }
      { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
  }
\cs_new:Npn \__keys_multichoice_find:n #1
  { \clist_map_function:nN {#1} \__keys_choice_find:n }
```

(*End definition for* `\__keys_choice_find:n` *and* `\__keys_multichoice_find:n`*.*)

19.7 Utilities

`\__keys_parent:n`
`\__keys_parent:o`
`\__keys_parent:w`

Used to strip off the ending part of the key path after the last `/`.

```
\cs_new:Npn \__keys_parent:n #1
  { \__keys_parent:w #1 / / \q_stop { } }
\cs_generate_variant:Nn \__keys_parent:n { o }
\cs_new:Npn \__keys_parent:w #1 / #2 / #3 \q_stop #4
  {
    \tl_if_blank:nTF {#2}
      { \use_none:n #4 }
      {
        \__keys_parent:w #2 / #3 \q_stop { #4 / #1 }
      }
  }
```

(*End definition for* `\__keys_parent:n` *and* `\__keys_parent:w`*.*)

`\__keys_remove_spaces:n`
`\__keys_remove_spaces:w`

Removes all spaces from the input which is detokenized as a result. This function has the same effect as `\zap@space` in LaTeX 2$_\varepsilon$ after applying `\tl_to_str:n`. It is set up to be fast as the use case here is tightly defined. The `?` is only there to allow for a space after `\use_none:nn` responsible for ending the loop.

```
\cs_new:Npn \__keys_remove_spaces:n #1
  {
    \exp_after:wN \__keys_remove_spaces:w \tl_to_str:n {#1}
    \use_none:nn ? ~
  }
\cs_new:Npn \__keys_remove_spaces:w #1 ~
  { #1 \__keys_remove_spaces:w }
```

(*End definition for* `\__keys_remove_spaces:n` *and* `\__keys_remove_spaces:w`*.*)

`\keys_if_exist_p:nn`
`\keys_if_exist:nnTF`

A utility for others to see if a key exists.

```
\prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF
      { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_exist:nnTF`*. This function is documented on page 174.*)

`\keys_if_choice_exist_p:nnn`
`\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nnTF`.

```
\prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
  { p , T , F , TF }
  {
    \cs_if_exist:cTF
      { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 / #3 } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_choice_exist:nnnTF`*. This function is documented on page 174.*)

`\keys_show:nn`
`\__keys_show:N`

To show a key, test for its existence to issue the correct message (same message, but with a `t` or `f` argument, then build the control sequences which contain the code and other information about the key, call an intermediate auxiliary which constructs the code to be displayed to the terminal, and finally conclude with __msg_show_wrap:n.

```
\cs_new_protected:Npn \keys_show:nn #1#2
  {
    \keys_if_exist:nnTF {#1} {#2}
      {
        \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
          { \__keys_remove_spaces:n { #1 / #2 } } { t } { } { }
        \exp_args:Nc \__keys_show:N
          { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
      }
      {
        \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
          { \__keys_remove_spaces:n { #1 / #2 } } { f } { } { }
        \__msg_show_wrap:n { }
      }
  }
\cs_new_protected:Npn \__keys_show:N #1
  {
    \use:x
      {
        \__msg_show_wrap:n
          {
            \exp_not:N \__msg_show_item_unbraced:nn { code }
              { \token_get_replacement_spec:N #1 }
          }
      }
  }
```

(*End definition for* \keys_show:nn *and* __keys_show:N*. These functions are documented on page 174.*)

\keys_log:nn Redirect output of \keys_show:nn to the log.

```
\cs_new_protected:Npn \keys_log:nn
  { \__msg_log_next: \keys_show:nn }
```

(*End definition for* \keys_log:nn*. This function is documented on page 174.*)

19.8 Messages

For when there is a need to complain.

```
\__msg_kernel_new:nnnn { kernel } { boolean-values-only }
  { Key~'#1'~accepts~boolean~values~only. }
  { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
\__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
  { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
  {
    The~key~'#1'~only~accepts~predefined~values,~
    and~'#2'~is~not~one~of~these.
  }
\__msg_kernel_new:nnnn { kernel } { key-no-property }
  { No~property~given~in~definition~of~key~'#1'. }
  {
    \c__msg_coding_error_text_tl
    Inside~\keys_define:nn  each~key~name~
    needs~a~property:  \\ \\
    \iow_indent:n { #1 .<property> } \\ \\
    LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
  }
\__msg_kernel_new:nnnn { kernel } { key-unknown }
  { The~key~'#1'~is~unknown~and~is~being~ignored. }
  {
    The~module~'#2'~does~not~have~a~key~called~'#1'.\\
    Check~that~you~have~spelled~the~key~name~correctly.
  }
\__msg_kernel_new:nnnn { kernel } { nested-choice-key }
  { Attempt~to~define~'#1'~as~a~nested~choice~key. }
  {
    The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
    itself~a~choice.
  }
\__msg_kernel_new:nnnn { kernel } { property-boolean-values-only }
  { The~property~'#1'~accepts~boolean~values~only. }
  {
    \c__msg_coding_error_text_tl
    The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
  }
\__msg_kernel_new:nnnn { kernel } { property-requires-value }
  { The~property~'#1'~requires~a~value. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
    No~value~was~given~for~the~property,~and~one~is~required.
  }
\__msg_kernel_new:nnnn { kernel } { property-unknown }
  { The~key~property~'#1'~is~unknown. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
    this~property~is~not~defined.
  }
\__msg_kernel_new:nnnn { kernel } { value-forbidden }
  { The~key~'#1'~does~not~take~a~value. }
  {
    The~key~'#1'~should~be~given~without~a~value.\\
    The~value~'#2'~was~present:~the~key~will~be~ignored.
  }
\__msg_kernel_new:nnnn { kernel } { value-required }
  { The~key~'#1'~requires~a~value. }
  {
    The~key~'#1'~must~have~a~value.\\
    No~value~was~present:~the~key~will~be~ignored.
  }
\__msg_kernel_new:nnn { kernel } { show-key }
  {
    The~key~#1~
    \str_if_eq:nnTF {#2} { t }
      { has~the~properties: }
      { is~undefined. }
  }
⟨/initex | package⟩
```

20 l3fp implementation

Nothing to see here: everything is in the subfiles!

21 l3fp-aux implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

21.1 Internal representation

Internally, a floating point number ⟨*X*⟩ is a token list containing

> `\s__fp \__fp_chk:w` ⟨*case*⟩ ⟨*sign*⟩ ⟨*body*⟩ `;`

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to `f`-expansion. They must leave a recognizable mark after `f`-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under `x`-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their ⟨*case*⟩, which is a single digit:

0 zeros: `+0` and `-0`,

1 "normal" numbers (positive and negative),

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

The ⟨*sign*⟩ is `0` (positive) or `2` (negative), except in the case of `nan`, which have ⟨*sign*⟩ $= 1$. This ensures that changing the ⟨*sign*⟩ digit to $2 -$ ⟨*sign*⟩ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

> `\s__fp \__fp_chk:w` ⟨*case*⟩ ⟨*sign*⟩ `\s__fp_...` `;`

where `\s__fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers (⟨*case*⟩ $= 1$) have the form

> `\s__fp \__fp_chk:w` 1 ⟨*sign*⟩ `{`⟨*exponent*⟩`}` `{`⟨X_1⟩`}` `{`⟨X_2⟩`}` `{`⟨X_3⟩`}` `{`⟨X_4⟩`}` `;`

Table 1: Internal representation of floating point numbers.

Representation	Meaning
0 0 `\s__fp_... ;`	Positive zero.
0 2 `\s__fp_... ;`	Negative zero.
1 0 `{`$\langle exponent\rangle$`}` `{`$\langle X_1\rangle$`}` `{`$\langle X_2\rangle$`}` `{`$\langle X_3\rangle$`}` `{`$\langle X_4\rangle$`}` `;`	Positive floating point.
1 2 `{`$\langle exponent\rangle$`}` `{`$\langle X_1\rangle$`}` `{`$\langle X_2\rangle$`}` `{`$\langle X_3\rangle$`}` `{`$\langle X_4\rangle$`}` `;`	Negative floating point.
2 0 `\s__fp_... ;`	Positive infinity.
2 2 `\s__fp_... ;`	Negative infinity.
3 1 `\s__fp_... ;`	Quiet `nan`.
3 1 `\s__fp_... ;`	Signalling `nan`.

Here, the $\langle exponent\rangle$ is an integer, between -10000 and 10000. The body consists in four blocks of exactly 4 digits, $0000 \le \langle X_i\rangle \le 9999$, and the floating point is

$$(-1)^{\langle sign\rangle/2}\langle X_1\rangle\langle X_2\rangle\langle X_3\rangle\langle X_4\rangle \cdot 10^{\langle exponent\rangle-16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the $\langle exponent\rangle$ is minimal, in other words, $1000 \le \langle X_1\rangle \le 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

21.2 Using arguments and semicolons

`\__fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops `f`-type expansion.

```
\cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(*End definition for* `\__fp_use_none_stop_f:n`*.*)

`\__fp_use_s:n`
`\__fp_use_s:nn` Those functions place a semicolon after one or two arguments (typically digits).

```
\cs_new:Npn \__fp_use_s:n #1 { #1; }
\cs_new:Npn \__fp_use_s:nn #1#2 { #1#2; }
```

(*End definition for* `\__fp_use_s:n` *and* `\__fp_use_s:nn`*.*)

`\__fp_use_none_until_s:w`
`\__fp_use_i_until_s:nw`
`\__fp_use_ii_until_s:nnw` Those functions select specific arguments among a set of arguments delimited by a semicolon.

```
\cs_new:Npn \__fp_use_none_until_s:w #1; { }
\cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
\cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(*End definition for* `\__fp_use_none_until_s:w`*,* `\__fp_use_i_until_s:nw`*, and* `\__fp_use_ii_until_s:nnw`*.*)

`\__fp_reverse_args:Nww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
\cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(*End definition for* `\__fp_reverse_args:Nww`*.*)

__fp_rrot:www

Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```
\cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(*End definition for* __fp_rrot:www*.*)

__fp_use_i:ww
__fp_use_i:www

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
\cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
\cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }
```

(*End definition for* __fp_use_i:ww *and* __fp_use_i:www*.*)

21.3 Constants, and structure of floating points

\s__fp
__fp_chk:w

Floating points numbers all start with \s__fp __fp_chk:w, where \s__fp is equal to the TeX primitive \relax, and __fp_chk:w is protected. The rest of the floating point number is made of characters (or \relax). This ensures that nothing expands under f-expansion, nor under x-expansion. However, when typeset, \s__fp does nothing, and __fp_chk:w is expanded. We define __fp_chk:w to produce an error.

```
\__scan_new:N \s__fp
\cs_new_protected:Npn \__fp_chk:w #1 ;
  {
    \__msg_kernel_error:nnx { kernel } { misused-fp }
      { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
  }
```

(*End definition for* \s__fp *and* __fp_chk:w*.*)

\s__fp_mark
\s__fp_stop

Aliases of \tex_relax:D, used to terminate expressions.

```
\__scan_new:N \s__fp_mark
\__scan_new:N \s__fp_stop
```

(*End definition for* \s__fp_mark *and* \s__fp_stop*.*)

\s__fp_invalid
\s__fp_underflow
\s__fp_overflow
\s__fp_division
\s__fp_exact

A couple of scan marks used to indicate where special floating point numbers come from.

```
\__scan_new:N \s__fp_invalid
\__scan_new:N \s__fp_underflow
\__scan_new:N \s__fp_overflow
\__scan_new:N \s__fp_division
\__scan_new:N \s__fp_exact
```

(*End definition for* \s__fp_invalid *and others.*)

\c_zero_fp
\c_minus_zero_fp
\c_inf_fp
\c_minus_inf_fp
\c_nan_fp

The special floating points. We define the floating points here as “exact”.

```
\tl_const:Nn \c_zero_fp       { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
\tl_const:Nn \c_inf_fp        { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_inf_fp  { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
\tl_const:Nn \c_nan_fp        { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }
```

(*End definition for* \c_zero_fp *and others. These variables are documented on page 183.*)

\c__fp_prec_int
\c__fp_half_prec_int
\c__fp_block_int

The number of digits of floating points.

```
\int_const:Nn \c__fp_prec_int { 16 }
\int_const:Nn \c__fp_half_prec_int { 8 }
\int_const:Nn \c__fp_block_int { 4 }
```

(*End definition for* \c__fp_prec_int *,* \c__fp_half_prec_int *, and* \c__fp_block_int*.*)

\c__fp_myriad_int

Blocks have 4 digits so this integer is useful.

```
\int_const:Nn \c__fp_myriad_int { 10000 }
```

(*End definition for* \c__fp_myriad_int*.*)

\c__fp_minus_min_exponent_int
\c__fp_max_exponent_int

Normal floating point numbers have an exponent between − minus_min_exponent and max_exponent inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0. It would be more natural to define a min_exponent with the opposite sign but that would waste one TEX count.

```
\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }
\int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(*End definition for* \c__fp_minus_min_exponent_int *and* \c__fp_max_exponent_int*.*)

\c__fp_max_exp_exponent_int

If a number's exponent is larger than that, its exponential overflows/underflows.

```
\int_const:Nn \c__fp_max_exp_exponent_int { 5 }
```

(*End definition for* \c__fp_max_exp_exponent_int*.*)

\c__fp_overflowing_fp

A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```
\tl_const:Nx \c__fp_overflowing_fp
  {
    \s__fp \__fp_chk:w 1 0
      { \int_eval:n { \c__fp_max_exponent_int + 1 } }
      {1000} {0000} {0000} {0000} ;
  }
```

(*End definition for* \c__fp_overflowing_fp*.*)

__fp_zero_fp:N
__fp_inf_fp:N

In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```
\cs_new:Npn \__fp_zero_fp:N #1
  { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
\cs_new:Npn \__fp_inf_fp:N #1
  { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(*End definition for* __fp_zero_fp:N *and* __fp_inf_fp:N*.*)

__fp_exponent:w

For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.

```
\cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_use_ii_until_s:nnw
    \else:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN 0
    \fi:
  }
```

(*End definition for* `\__fp_exponent:w`.)

`\__fp_neg_sign:N`

When appearing in an integer expression or after `\__int_value:w`, this expands to the sign opposite to `#1`, namely 0 (positive) is turned to 2 (negative), 1 (`nan`) to 1, and 2 to 0.

```
\cs_new:Npn \__fp_neg_sign:N #1
  { \__int_eval:w 2 - #1 \__int_eval_end: }
```

(*End definition for* `\__fp_neg_sign:N`.)

21.4 Overflow, underflow, and exact zero

`\__fp_sanitize:Nw`
`\__fp_sanitize:wN`
`\__fp_sanitize_zero:w`

Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ±0 or overflowed to ±∞. The functions `\__fp_underflow:w` and `\__fp_overflow:w` are defined in l3fp-traps.

```
\cs_new:Npn \__fp_sanitize:Nw #1 #2;
  {
    \if_case:w
        \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
        \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
        \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
    \or: \exp_after:wN \__fp_overflow:w
    \or: \exp_after:wN \__fp_underflow:w
    \or: \exp_after:wN \__fp_sanitize_zero:w
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2}
  }
\cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
\cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3;
  { \c_zero_fp }
```

(*End definition for* `\__fp_sanitize:Nw`*,* `\__fp_sanitize:wN`*, and* `\__fp_sanitize_zero:w`.)

21.5 Expanding after a floating point number

`\__fp_exp_after_o:w`
`\__fp_exp_after_f:nw`

`\__fp_exp_after_o:w` ⟨*floating point*⟩
`\__fp_exp_after_f:nw` {⟨*tokens*⟩} ⟨*floating point*⟩

Places ⟨*tokens*⟩ (empty in the case of `\__fp_exp_after_o:w`) between the ⟨*floating point*⟩ and the ⟨*more tokens*⟩, then hits those tokens with either `o`-expansion (one `\exp_after:wN`) or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```
\cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { }
    #1
  }
```

```
\cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { \exp:w \exp_end_continue_f:w #1 }
    #2
  }
```

(*End definition for* `\__fp_exp_after_o:w` *and* `\__fp_exp_after_f:nw`.)

`\__fp_exp_after_special:nNNw`

`\__fp_exp_after_special:nNNw` {⟨*after*⟩} ⟨*case*⟩ ⟨*sign*⟩ ⟨*scan mark*⟩ ;

Special floating point numbers are easy to jump over since they contain few tokens.

```
\cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
  {
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN ;
    #1
  }
```

(*End definition for* `\__fp_exp_after_special:nNNw`.)

`\__fp_exp_after_normal:nNNw`

For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by `,`). That may be changed some day.

```
\cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
  {
    \exp_after:wN \__fp_exp_after_normal:Nwwwww
    \exp_after:wN #2
    \__int_value:w #3   \exp_after:wN ;
    \__int_value:w 1 #4 \exp_after:wN ;
    \__int_value:w 1 #5 \exp_after:wN ;
    \__int_value:w 1 #6 \exp_after:wN ;
    \__int_value:w 1 #7 \exp_after:wN ; #1
  }
\cs_new:Npn \__fp_exp_after_normal:Nwwwww
    #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
  { \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(*End definition for* `\__fp_exp_after_normal:nNNw`.)

`\__fp_exp_after_array_f:w`
`\__fp_exp_after_stop_f:nw`

`\__fp_exp_after_array_f:w`
⟨*fp*$_1$⟩ ;
...
⟨*fp*$_n$⟩ ;
`\s__fp_stop`

```
\cs_new:Npn \__fp_exp_after_array_f:w #1
  {
    \cs:w __fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
      { \__fp_exp_after_array_f:w }
    #1
  }
\cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
```

(*End definition for* `\__fp_exp_after_array_f:w` *and* `\__fp_exp_after_stop_f:nw`.)

21.6 Packing digits

When a positive integer `#1` is known to be less than 10^8, the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```
\cs_new:Npn \pack:NNNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNNw
  \__int_value:w \__int_eval:w 1 0000 0000 + #1 ;
```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by TeX's integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute $1\,2345 \times 6677\,8899$. With simplified names, we would do

```
\exp_after:wN \post_processing:w
\__int_value:w \__int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
  \__int_value:w \__int_eval:w 4 9995 0000
    + 12345 * 6677
    \exp_after:wN \pack:NNNNNw
    \__int_value:w \__int_eval:w 5 0000 0000
      + 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\__int_value:w \__int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the "leading shift"). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\__int_value:w \__int_eval:w` with starting value $4\,9995\,0000$ (the "middle shift"). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+` ⟨*5 digits*⟩ for the initial computation. The "leading shift" cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making l3fp as fast as other pure TeX floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

__fp_pack:NNNNNw
\c__fp_trailing_shift_int
\c__fp_middle_shift_int
\c__fp_leading_shift_int

This set of shifts allows for computations involving results in the range $[-4\cdot 10^8, 5\cdot 10^8-1]$. Shifted values all have exactly 9 digits.

```
\int_const:Nn \c__fp_leading_shift_int  { - 5 0000 }
\int_const:Nn \c__fp_middle_shift_int   { 5 0000 *  9999 }
\int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
\cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(*End definition for* `\__fp_pack:NNNNNw` *and others.*)

__fp_pack_big:NNNNNNw
\c__fp_big_trailing_shift_int
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int

This set of shifts allows for computations involving results in the range $[-5\cdot 10^8, 6\cdot 10^8-1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to TEX's limit of $2^{31}-1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31}, the two bounds on 10-digit integers in TEX.

```
\int_const:Nn \c__fp_big_leading_shift_int  { - 15 2374 }
\int_const:Nn \c__fp_big_middle_shift_int   { 15 2374 *  9999 }
\int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
\cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End definition for* `\__fp_pack_big:NNNNNNw` *and others.*)

__fp_pack_Bigg:NNNNNNw
\c__fp_Bigg_trailing_shift_int
\c__fp_Bigg_middle_shift_int
\c__fp_Bigg_leading_shift_int

This set of shifts allows for computations with results in the range $[-1\cdot 10^9, 147483647]$; the end-point is $2^{31}-1-2\cdot 10^9 \simeq 1.47\cdot 10^8$. Shifted values all have exactly 10 digits.

```
\int_const:Nn \c__fp_Bigg_leading_shift_int  { - 20 0000 }
\int_const:Nn \c__fp_Bigg_middle_shift_int   { 20 0000 *  9999 }
\int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
\cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End definition for* `\__fp_pack_Bigg:NNNNNNw` *and others.*)

__fp_pack_twice_four:wNNNNNNNN

`\__fp_pack_twice_four:wNNNNNNNN` ⟨*tokens*⟩ `;` ⟨≥ 8 *digits*⟩

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
\cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(*End definition for* `\__fp_pack_twice_four:wNNNNNNNN`*.*)

__fp_pack_eight:wNNNNNNNN

`\__fp_pack_eight:wNNNNNNNN` ⟨*tokens*⟩ `;` ⟨≥ 8 *digits*⟩

Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
\cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5#6#7#8#9} ; }
```

(*End definition for* `\__fp_pack_eight:wNNNNNNNN`*.*)

__fp_basics_pack_low:NNNNNw
__fp_basics_pack_high:NNNNNw
__fp_basics_pack_high_carry:w

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `\__fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

This is used in l3fp-basics and l3fp-extended.

```
\cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
  { + #1 - 1 ; {#2#3#4#5} {#6} ; }
\cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
  {
    \if_meaning:w 2 #1
      \__fp_basics_pack_high_carry:w
    \fi:
    ; {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
  { \fi: + 1 ; {1000} }
```

(*End definition for* `\__fp_basics_pack_low:NNNNNw`*,* `\__fp_basics_pack_high:NNNNNw`*, and* `\__fp_basics_pack_high_carry:w`*.*)

__fp_basics_pack_weird_low:NNNNw
__fp_basics_pack_weird_high:NNNNNNNNw

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```
\cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
  {
    \if_meaning:w 2 #1
      + 1
    \fi:
    \__int_eval_end:
    #2#3#4; {#5} ;
  }
\cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
    1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }
```

(*End definition for* `\__fp_basics_pack_weird_low:NNNNw` *and* `\__fp_basics_pack_weird_high:NNNNNNNNw`*.*)

21.7 Decimate (dividing by a power of 10)

__fp_decimate:nNnnnn

`\__fp_decimate:nNnnnn` {⟨*shift*⟩} {⟨f_1⟩}
{⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩}

Each $\langle X_i\rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1\rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1\rangle$ is called as follows:

⟨f_1⟩ ⟨*rounding*⟩ {⟨*X'*$_1$⟩} {⟨*X'*$_2$⟩} ⟨*extra-digits*⟩ ;

where $0 \leq \langle X'_i\rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^{4}\langle X_i\rangle\cdot 10^{-4i}\cdot 10^{-\langle \mathit{shift}\rangle}\right) - \bigl(\langle X'_1\rangle\cdot 10^{-8} + \langle X'_2\rangle\cdot 10^{-16}\bigr) = 0.\langle \mathit{extra\text{-}digits}\rangle\cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The ⟨*rounding*⟩ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, ⟨*rounding*⟩ is 1 (not 0), and $\langle X'_1 \rangle$ and $\langle X'_2 \rangle$ are both zero.

If the shift is 1, the ⟨*rounding*⟩ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the ⟨*rounding*⟩ digit to be placed after the $\langle X'_i \rangle$, but the choice we make involves less reshuffling.

Note that this function treats negative ⟨*shift*⟩ as 0.

```
\cs_new:Npn \__fp_decimate:nNnnnn #1
  {
    \cs:w
      __fp_decimate_
      \if_int_compare:w \__int_eval:w #1 > \c__fp_prec_int
        tiny
      \else:
        \__int_to_roman:w \__int_eval:w #1
      \fi:
      :Nnnnn
    \cs_end:
  }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(*End definition for* `\__fp_decimate:nNnnnn`*.*)

__fp_decimate_:Nnnnn
__fp_decimate_tiny:Nnnnn

If the ⟨*shift*⟩ is zero, or too big, life is very easy.

```
\cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
  { #1 0 {#2#3} {#4#5} ; }
\cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
  { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(*End definition for* `\__fp_decimate_:Nnnnn` *and* `\__fp_decimate_tiny:Nnnnn`*.*)

__fp_decimate_auxi:Nnnnn
__fp_decimate_auxii:Nnnnn
__fp_decimate_auxiii:Nnnnn
__fp_decimate_auxiv:Nnnnn
__fp_decimate_auxv:Nnnnn
__fp_decimate_auxvi:Nnnnn
__fp_decimate_auxvii:Nnnnn
__fp_decimate_auxviii:Nnnnn
__fp_decimate_auxix:Nnnnn
__fp_decimate_auxx:Nnnnn
__fp_decimate_auxxi:Nnnnn
__fp_decimate_auxxii:Nnnnn
__fp_decimate_auxxiii:Nnnnn
__fp_decimate_auxxiv:Nnnnn
__fp_decimate_auxxv:Nnnnn
__fp_decimate_auxxvi:Nnnnn

`\@_decimate_auxi:Nnnnn` $\langle f_1 \rangle$ `{`$\langle X_1 \rangle$`}` `{`$\langle X_2 \rangle$`}` `{`$\langle X_3 \rangle$`}` `{`$\langle X_4 \rangle$`}`

Shifting happens in two steps: compute the ⟨*rounding*⟩ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through `\__fp_tmp:w`. The arguments are as follows: `#1` indicates which function is being defined; after one step of expansion, `#2` yields the "extra digits" which are then converted by `\__fp_round_digit:Nw` to the ⟨*rounding*⟩ digit (note the `+` separating blocks of digits to avoid overflowing TeX's integers). This triggers the `f`-expansion of `\__fp_decimate_pack:nnnnnnnnnnw`,[10] responsible for building two blocks of 8 digits, and removing the rest. For this to work, `#3` alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```
\cs_new:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
      {
        \exp_after:wN ##1
        \__int_value:w
```

[10] No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```
          \exp_after:wN \__fp_round_digit:Nw #2 ;
        \__fp_decimate_pack:nnnnnnnnnnw #3 ;
      }
  }
\__fp_tmp:w {i}   {\use_none:nnn      #50}{    0{#2}#3{#4}#5               }
\__fp_tmp:w {ii}  {\use_none:nn       #5 }{    00{#2}#3{#4}#5              }
\__fp_tmp:w {iii} {\use_none:n        #5 }{    000{#2}#3{#4}#5             }
\__fp_tmp:w {iv}  {                   #5 }{   {0000}#2{#3}#4 #5            }
\__fp_tmp:w {v}   {\use_none:nnn    #4#5 }{   0{0000}#2{#3}#4 #5           }
\__fp_tmp:w {vi}  {\use_none:nn     #4#5 }{   00{0000}#2{#3}#4 #5          }
\__fp_tmp:w {vii} {\use_none:n      #4#5 }{   000{0000}#2{#3}#4 #5         }
\__fp_tmp:w {viii}{                 #4#5 }{  {0000}0000{#2}#3 #4 #5        }
\__fp_tmp:w {ix}  {\use_none:nnn  #3#4+#5}{  0{0000}0000{#2}#3 #4 #5       }
\__fp_tmp:w {x}   {\use_none:nn   #3#4+#5}{  00{0000}0000{#2}#3 #4 #5      }
\__fp_tmp:w {xi}  {\use_none:n    #3#4+#5}{  000{0000}0000{#2}#3 #4 #5     }
\__fp_tmp:w {xii} {               #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5    }
\__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5   }
\__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5  }
\__fp_tmp:w {xv}  {\use_none:n  #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
\__fp_tmp:w {xvi} {             #2#3+#4#5}{{0000}0000{0000}0000 #2 #3 #4 #5}
```

(*End definition for* `\__fp_decimate_auxi:Nnnnn` *and others.*)

__fp_decimate_pack:nnnnnnnnnnw

The computation of the ⟨*rounding*⟩ digit leaves an unfinished `\__int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```
\cs_new:Npn \__fp_decimate_pack:nnnnnnnnnnw #1#2#3#4#5
  { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
\cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
  { {#1} {#2#3#4#5#6} }
```

(*End definition for* `\__fp_decimate_pack:nnnnnnnnnnw`*.*)

21.8 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in l3fp must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if_case:w ⟨integer⟩ \exp_stop_f:
     \@@_case_return_o:Nw ⟨fp var⟩
\or: \@@_case_use:nw {⟨some computation⟩}
\or: \@@_case_return_same_o:w
\or: \@@_case_return:nw {⟨something⟩}
\fi:
⟨junk⟩
⟨floating point⟩
```

In this example, the case 0 returns the floating point ⟨*fp var*⟩, expanding once after that floating point. Case 1 does ⟨*some computation*⟩ using the ⟨*floating point*⟩ (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the ⟨*floating point*⟩ without modifying it, removing the ⟨*junk*⟩ and expanding once after. Case 3 closes the conditional, removes the ⟨*junk*⟩ and the ⟨*floating point*⟩, and expands ⟨*something*⟩ next. In other cases, the "⟨*junk*⟩" is expanded, performing some other operation on the ⟨*floating point*⟩. We provide similar functions with two trailing ⟨*floating points*⟩.

`\__fp_case_use:nw`

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
\cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
```

(*End definition for* `\__fp_case_use:nw`.)

`\__fp_case_return:nw`

This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the ⟨*junk*⟩ may not contain semicolons.

```
\cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(*End definition for* `\__fp_case_return:nw`.)

`\__fp_case_return_o:Nw`

This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an ⟨*fp var*⟩) then expands once after it.

```
\cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
  { \fi: \exp_after:wN #1 }
```

(*End definition for* `\__fp_case_return_o:Nw`.)

`\__fp_case_return_same_o:w`

This function ends a TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
\cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
  { \fi: \__fp_exp_after_o:w \s__fp }
```

(*End definition for* `\__fp_case_return_same_o:w`.)

`\__fp_case_return_o:Nww`

Same as `\__fp_case_return_o:Nw` but with two trailing floating points.

```
\cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
  { \fi: \exp_after:wN #1 }
```

(*End definition for* `\__fp_case_return_o:Nww`.)

`\__fp_case_return_i_o:ww`
`\__fp_case_return_ii_o:ww`

Similar to `\__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
\cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
  { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
\cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
  { \fi: \__fp_exp_after_o:w }
```

(*End definition for* `\__fp_case_return_i_o:ww` *and* `\__fp_case_return_ii_o:ww`.)

21.9 Integer floating points

__fp_int_p:w
__fp_int:w*TF*

Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from `\__fp_decimate:nNnnnn` is 0.

```
\prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
  { TF , T , F , p }
  {
    \if_case:w #1 \exp_stop_f:
           \prg_return_true:
    \or:
      \if_charcode:w 0
        \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
          \__fp_use_i_until_s:nw #4
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else: \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__fp_int:wTF`.)

21.10 Small integer floating points

__fp_small_int:wTF
__fp_small_int_true:wTF
__fp_small_int_normal:NnwTF
__fp_small_int_test:NnnwNTF

Tests if the floating point argument is an integer or ±∞. If so, it is converted to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the ⟨*true code*⟩. Otherwise, the ⟨*false code*⟩ is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the ⟨*false code*⟩. If it is, then the integer is `#2` `#3`; use `#3` if `#2` vanishes and otherwise 10^8.

```
\cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
  {
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
    \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
    \or:
      \__fp_case_return:nw
        {
          \exp_after:wN \__fp_small_int_true:wTF \__int_value:w
            \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
        }
    \else: \__fp_case_return:nw \use_ii:nn
    \fi:
    #2
  }
\cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
\cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
  {
    \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
      \__fp_small_int_test:NnnwNw
      #3 #1
  }
```

```
\cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_small_int_true:wTF
      \__int_value:w \if_meaning:w 2 #5 - \fi:
        \if_int_compare:w #2 > 0 \exp_stop_f:
          1 0000 0000
        \else:
          #3
        \fi:
      \exp_after:wN ;
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End definition for* `\__fp_small_int:wTF` *and others.*)

21.11 Length of a floating point array

`\__fp_array_count:n`
`\__fp_array_count_loop:Nw`

Count the number of items in an array of floating points. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```
\cs_new:Npn \__fp_array_count:n #1
  {
    \__int_value:w \__int_eval:w 0
      \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
      \__prg_break_point:
    \__int_eval_end:
  }
\cs_new:Npn \__fp_array_count_loop:Nw #1#2;
  { \use_none:n #1 + 1 \__fp_array_count_loop:Nw }
```

(*End definition for* `\__fp_array_count:n` *and* `\__fp_array_count_loop:Nw`*.*)

21.12 x-like expansion expandably

`\__fp_expand:n`
`\__fp_expand_loop:nwnN`

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is `f`-expanded, then the leading item (often a single character token) is moved to a storage area after `\s__fp_mark`, and `f`-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```
\cs_new:Npn \__fp_expand:n #1
  {
    \__fp_expand_loop:nwnN { }
      #1 \prg_do_nothing:
      \s__fp_mark { } \__fp_expand_loop:nwnN
      \s__fp_mark { } \__fp_use_i_until_s:nw ;
  }
\cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
  {
    \exp_after:wN #4 \exp:w \exp_end_continue_f:w
    #2
    \s__fp_mark { #3 #1 } #4
  }
```

(*End definition for* `\__fp_expand:n` *and* `\__fp_expand_loop:nwnN`*.*)

21.13 Messages

Using a floating point directly is an error.

```
\__msg_kernel_new:nnnn { kernel } { misused-fp }
  { A~floating~point~with~value~'#1'~was~misused. }
  {
    To~obtain~the~value~of~a~floating~point~variable,~use~
    '\token_to_str:N \fp_to_decimal:N',~
    '\token_to_str:N \fp_to_scientific:N',~or~other~
    conversion~functions.
  }
⟨/initex | package⟩
```

22 l3fp-traps Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

22.1 Flags

flag␣fp_invalid_operation
flag␣fp_division_by_zero
flag␣fp_overflow
flag␣fp_underflow

Flags to denote exceptions.

```
\flag_new:n { fp_invalid_operation }
\flag_new:n { fp_division_by_zero }
\flag_new:n { fp_overflow }
\flag_new:n { fp_underflow }
```

(*End definition for* `flag fp_invalid_operation` *and others. These variables are documented on page 185.*)

22.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `\__fp_invalid_operation:nnw`,
- `\__fp_invalid_operation_o:Nww`,
- `\__fp_invalid_operation_tl_o:ff`,
- `\__fp_division_by_zero_o:Nnw`,
- `\__fp_division_by_zero_o:NNww`,
- `\__fp_overflow:w`,
- `\__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` `{`⟨*exception*⟩`}` `{`⟨*way of trapping*⟩`}`, where the ⟨*way of trapping*⟩ is one of `error`, `flag`, or `none`.

We also provide `\__fp_invalid_operation_o:nw`, defined in terms of `\__fp_invalid_operation:nnw`.

`\fp_trap:nn`

```
\cs_new_protected:Npn \fp_trap:nn #1#2
  {
    \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
      {
        \clist_if_in:nnTF
          { invalid_operation , division_by_zero , overflow , underflow }
          {#1}
          {
            \__msg_kernel_error:nnxx { kernel }
              { unknown-fpu-trap-type } {#1} {#2}
          }
          {
            \__msg_kernel_error:nnx
              { kernel } { unknown-fpu-exception } {#1}
          }
      }
  }
```

(*End definition for* `\fp_trap:nn`. *This function is documented on page 185.*)

__fp_trap_invalid_operation_set_error:
__fp_trap_invalid_operation_set_flag:
__fp_trap_invalid_operation_set_none:
__fp_trap_invalid_operation_set:N

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
  { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { fp-invalid } {##2} { \fp_to_tl:n { ##3; } } { }
        \flag_raise:n { fp_invalid_operation }
        ##1
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
      {
        #1
        \__fp_error:nffn { fp-invalid-ii }
          { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
        \flag_raise:n { fp_invalid_operation }
        \exp_after:wN \c_nan_fp
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
      {
        #1
        \__fp_error:nffn { fp-invalid } {##1} {##2} { }
        \flag_raise:n { fp_invalid_operation }
        \exp_after:wN \c_nan_fp
      }
  }
```

(*End definition for* `\__fp_trap_invalid_operation_set_error:` *and others.*)

__fp_trap_division_by_zero_set_error:
__fp_trap_division_by_zero_set_flag:
__fp_trap_division_by_zero_set_none:
__fp_trap_division_by_zero_set:N

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, ±∞ or `NaN`.

```
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
  { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
        \flag_raise:n { fp_division_by_zero }
        \exp_after:wN ##1
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
      {
        #1
        \__fp_error:nffn { fp-zero-div-ii }
          { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
        \flag_raise:n { fp_division_by_zero }
        \exp_after:wN ##1
      }
  }
```

(*End definition for* `\__fp_trap_division_by_zero_set_error:` *and others.*)

- __fp_trap_overflow_set_error:
- __fp_trap_overflow_set_flag:
- __fp_trap_overflow_set_none:
- __fp_trap_overflow_set:N
- __fp_trap_underflow_set_error:
- __fp_trap_underflow_set_flag:
- __fp_trap_underflow_set_none:
- __fp_trap_underflow_set:N
- __fp_trap_overflow_set:NnNn

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\__fp_overflow:w` and `\__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for TeX, and `\__fp_overflow:w` receives $\pm\infty$ (`\__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```
\cs_new_protected:Npn \__fp_trap_overflow_set_error:
  { \__fp_trap_overflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_overflow_set_flag:
  { \__fp_trap_overflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set_none:
  { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
\cs_new_protected:Npn \__fp_trap_underflow_set_error:
  { \__fp_trap_underflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_underflow_set_flag:
  { \__fp_trap_underflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set_none:
  { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
\cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
  {
    \exp_args:Nno \use:n
      { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
      {
        #1
        \__fp_error:nffn
          { fp-flow \if_meaning:w 1 ##1 -to \fi: }
          { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
          { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
          {#2}
        \flag_raise:n { fp_#2 }
        #3 ##2
      }
  }
```

(*End definition for* `\__fp_trap_overflow_set_error:` *and others.*)

`\__fp_invalid_operation:nnw`
`\__fp_invalid_operation_o:Nww`
`\__fp_invalid_operation_tl_o:ff`
`\__fp_division_by_zero_o:Nnw`
`\__fp_division_by_zero_o:NNww`
`\__fp_overflow:w`
`\__fp_underflow:w`

Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
\cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
\cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
\cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
\cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
\cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
\cs_new:Npn \__fp_overflow:w { }
\cs_new:Npn \__fp_underflow:w { }
\fp_trap:nn { invalid_operation } { error }
\fp_trap:nn { division_by_zero } { flag }
\fp_trap:nn { overflow } { flag }
\fp_trap:nn { underflow } { flag }
```

(*End definition for* `\__fp_invalid_operation:nnw` *and others.*)

`\__fp_invalid_operation_o:nw`
`\__fp_invalid_operation_o:fw`

Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```
\cs_new:Npn \__fp_invalid_operation_o:nw
  { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
\cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
```

(*End definition for* `\__fp_invalid_operation_o:nw`*.*)

22.3 Errors

`\__fp_error:nnnn`
`\__fp_error:nnfn`
`\__fp_error:nffn`

```
\cs_new:Npn \__fp_error:nnnn
  { \__msg_kernel_expandable_error:nnnnn { kernel } }
\cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
```

(*End definition for* `\__fp_error:nnnn`*.*)

22.4 Messages

Some messages.

```
\__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
  {
    The~FPU~exception~'#1'~is~not~known:~
    that~trap~will~never~be~triggered.
  }
  {
    The~only~exceptions~to~which~traps~can~be~attached~are \\
    \iow_indent:n
      {
```

```
    * ~ invalid_operation \\
    * ~ division_by_zero \\
    * ~ overflow \\
    * ~ underflow
   }
 }
\__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
 { The~FPU~trap~type~'#2'~is~not~known. }
 {
  The~trap~type~must~be~one~of \\
  \iow_indent:n
   {
    * ~ error \\
    * ~ flag \\
    * ~ none
   }
 }
\__msg_kernel_new:nnn { kernel } { fp-flow }
 { An ~ #3 ~ occurred. }
\__msg_kernel_new:nnn { kernel } { fp-flow-to }
 { #1 ~ #3 ed ~ to ~ #2 . }
\__msg_kernel_new:nnn { kernel } { fp-zero-div }
 { Division~by~zero~in~ #1 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
 { Division~by~zero~in~ (#1) #3 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-invalid }
 { Invalid~operation~ #1 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-invalid-ii }
 { Invalid~operation~ (#1) #3 (#2) }
```

⟨/initex | package⟩

23 l3fp-round implementation

⟨*initex | package⟩

⟨@@=fp⟩

__fp_parse_word_trunc:N
__fp_parse_word_floor:N
__fp_parse_word_ceil:N

```
\cs_new:Npn \__fp_parse_word_trunc:N
 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
\cs_new:Npn \__fp_parse_word_floor:N
 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
\cs_new:Npn \__fp_parse_word_ceil:N
 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }
```

(*End definition for* __fp_parse_word_trunc:N*,* __fp_parse_word_floor:N*, and* __fp_parse_word_ceil:N*.*)

__fp_parse_word_round:N
__fp_parse_round:Nw
__fp_parse_round_no_error:Nw
__fp_parse_round_deprecation_error:Nw
round+
round0
round-

This looks for +, -, 0 after round. That syntax was deprecated in 2013 but the system to tell users about deprecated syntax was not really available then, so we did not have anything set up. When l3doc complains, remove the syntax by removing everything until the last \fi: in __fp_parse_word_round:N (and getting rid of the unused definitions of __fp_parse_round:Nw and so on, as well as the fp-deprecated error in l3fp-parse).

```
\cs_new:Npn \__fp_parse_word_round:N #1#2
  {
    \if_meaning:w + #2
      \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
    \else:
      \if_meaning:w 0 #2
        \__fp_parse_round:Nw \__fp_round_to_zero:NNN
      \else:
        \if_meaning:w - #2
          \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
        \fi:
      \fi:
    \fi:
    \__fp_parse_function:NNN
      \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
    #2
  }
\__debug:TF
  {
    \tl_gput_right:Nn \g__debug_deprecation_on_tl
      {
        \cs_set_eq:NN \__fp_parse_round:Nw
          \__fp_parse_round_deprecation_error:Nw
      }
    \tl_gput_right:Nn \g__debug_deprecation_off_tl
      {
        \cs_set_eq:NN \__fp_parse_round:Nw
          \__fp_parse_round_no_error:Nw
      }
    \cs_new:Npn \__fp_parse_round_deprecation_error:Nw
        #1 #2 \__fp_round_to_nearest:NNN #3#4
      {
        \__fp_error:nnfn { fp-deprecated } { round#4() }
          {
            \str_case:nn {#2}
              { { + } { ceil } { 0 } { trunc } { - } { floor } }
          } { }
        #2 #1 #3
      }
    \cs_new:Npn \__fp_parse_round_no_error:Nw
        #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }
    \cs_new_eq:NN \__fp_parse_round:Nw \__fp_parse_round_no_error:Nw
  }
  {
    \cs_new:Npn \__fp_parse_round:Nw
        #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }
  }
```

(*End definition for* `\__fp_parse_word_round:N` *and others.*)

23.1 Rounding tools

`\c__fp_five_int` This is used as the half-point for which numbers are rounded up/down.

```
\int_const:Nn \c__fp_five_int { 5 }
```

(*End definition for* `\c__fp_five_int`.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the "round to nearest" mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `\__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `\__fp_round:NNN` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.
- `\__fp_round_s:NNNw` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ ⟨*more digits*⟩`;` can expand to `0\exp_stop_f:;` or `1\exp_stop_f:;`.
- `\__fp_round_neg:NNN` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.

See implementation comments for details on the syntax.

`\__fp_round:NNN`
`\__fp_round_to_nearest:NNN`
`\__fp_round_to_nearest_ninf:NNN`
`\__fp_round_to_nearest_zero:NNN`
`\__fp_round_to_nearest_pinf:NNN`
`\__fp_round_to_ninf:NNN`
`\__fp_round_to_zero:NNN`
`\__fp_round_to_pinf:NNN`

`\__fp_round:NNN` ⟨*final sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩

If rounding the number ⟨*final sign*⟩⟨*digit*$_1$⟩.⟨*digit*$_2$⟩ to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:`. Typically used within the scope of an `\__int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that ⟨*final sign*⟩ be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that ⟨*final sign*⟩ is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `\__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the "round to nearest, ties to even" mode.

The "round to nearest" mode is the default. If the ⟨*digit*$_2$⟩ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that ⟨*digit*$_1$⟩ plus the result is even. In other words, round up if ⟨*digit*$_1$⟩ is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```
\cs_new:Npn \__fp_round_return_one:
  { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
\cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
  {
    \if_meaning:w 2 #1
      \if_int_compare:w #3 > 0 \exp_stop_f:
        \__fp_round_return_one:
      \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
\cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
  {
    \if_meaning:w 0 #1
      \if_int_compare:w #3 > 0 \exp_stop_f:
        \__fp_round_return_one:
      \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_int_odd:w #2 \exp_stop_f:
          \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_meaning:w 2 #1
            \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \fi:
```

```
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_meaning:w 0 #1
            \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN
```

(*End definition for* `\__fp_round:NNN` *and others.*)

`\__fp_round_s:NNNw`

`\__fp_round_s:NNNw` ⟨*final sign*⟩ ⟨*digit*⟩ ⟨*more digits*⟩ `;`

Similar to `\__fp_round:NNN`, but with an extra semicolon, this function expands to `0\exp_stop_f:;` if rounding ⟨*final sign*⟩⟨*digit*⟩.⟨*more digits*⟩ to an integer truncates, and to `1\exp_stop_f:;` otherwise. The ⟨*more digits*⟩ part must be a digit, followed by something that does not overflow a `\int_use:N \__int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```
\cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
  {
    \exp_after:wN \__fp_round:NNN
    \exp_after:wN #1
    \exp_after:wN #2
    \__int_value:w \__int_eval:w
      \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
                      \if_meaning:w 5 #3 1 \fi:
                \exp_stop_f:
        \if_int_compare:w \__int_eval:w #4 > 0 \exp_stop_f:
          1 +
        \fi:
      \fi:
      #3
    ;
  }
```

(*End definition for* `\__fp_round_s:NNNw`.)

`\__fp_round_digit:Nw`

`\__int_value:w \__fp_round_digit:Nw` ⟨*digit*⟩ ⟨*intexpr*⟩ `;`

This function should always be called within an `\__int_value:w` or `\__int_eval:w` expansion; it may add an extra `\__int_eval:w`, which means that the integer or integer expression should not be ended with a synonym of `\relax`, but with a semi-colon for instance.

```
\cs_new:Npn \__fp_round_digit:Nw #1 #2;
  {
    \if_int_odd:w \if_meaning:w 0 #1 1 \else:
                  \if_meaning:w 5 #1 1 \else:
                  0 \fi: \fi: \exp_stop_f:
```

```
     \if_int_compare:w \__int_eval:w #2 > 0 \exp_stop_f:
       \__int_eval:w 1 +
      \fi:
    \fi:
    #1
  }
```

(*End definition for* `\__fp_round_digit:Nw`.)

`\__fp_round_neg:NNN`
`\__fp_round_to_nearest_neg:NNN`
`\__fp_round_to_nearest_ninf_neg:NNN`
`\__fp_round_to_nearest_zero_neg:NNN`
`\__fp_round_to_nearest_pinf_neg:NNN`
`\__fp_round_to_ninf_neg:NNN`
`\__fp_round_to_zero_neg:NNN`
`\__fp_round_to_pinf_neg:NNN`

`\__fp_round_neg:NNN` ⟨*final sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form ⟨*final sign*⟩0.⟨*15 digits*⟩⟨*digit*$_1$⟩ with exactly 15 (non-all-zero) digits before ⟨*digit*$_1$⟩, subtract from it ⟨*final sign*⟩0.0 . . . 0⟨*digit*$_2$⟩, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

It turns out that this negative "round to nearest" is identical to the positive one. And this is the default mode.

```
\cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
\cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > 0 \exp_stop_f:
      \__fp_round_return_one:
    \fi:
    0 \exp_stop_f:
  }
\cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
\cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
\cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN \__fp_round_to_nearest_pinf:NNN
\cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
  {
    \if_int_compare:w #3 < \c__fp_five_int \else:
      \__fp_round_return_one:
    \fi:
    0 \exp_stop_f:
  }
\cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN \__fp_round_to_nearest_ninf:NNN
\cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN
```

(*End definition for* `\__fp_round_neg:NNN` *and others.*)

23.2 The round function

`\__fp_round_o:Nw`

The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `\__fp_round_to_nearest:NNN` to one of its analogues.

```
\cs_new:Npn \__fp_round_o:Nw #1#2 @
  {
    \if_case:w
      \__int_eval:w \__fp_array_count:n {#2} \__int_eval_end:
         \__fp_round_no_arg_o:Nw #1 \exp:w
    \or: \__fp_round:Nwn #1 #2 {0} \exp:w
    \or: \__fp_round:Nww #1 #2 \exp:w
    \else: \__fp_round:Nwww #1 #2 @ \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
```

(*End definition for* `\__fp_round_o:Nw`*.*)

__fp_round_no_arg_o:Nw

```
\cs_new:Npn \__fp_round_no_arg_o:Nw #1
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
      { \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
      {
        \__fp_error:nffn { fp-num-args }
          { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
      }
    \exp_after:wN \c_nan_fp
  }
```

(*End definition for* `\__fp_round_no_arg_o:Nw`*.*)

__fp_round:Nwww

Having three arguments is only allowed for `round`, not `trunc`, `ceil`, `floor`, so check for that case. If all is well, construct one of `\__fp_round_to_nearest:NNN`, `\__fp_round_to_nearest_zero:NNN`, `\__fp_round_to_nearest_ninf:NNN`, `\__fp_round_to_nearest_pinf:NNN` and act accordingly.

```
\cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
      {
        \tl_if_empty:nTF {#7}
          {
            \exp_args:Nc \__fp_round:Nww
              {
                __fp_round_to_nearest
                \if_meaning:w 0 #4 _zero \else:
                \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
                :NNN
              }
            #2 ; #3 ;
          }
          {
            \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
            \exp_after:wN \c_nan_fp
          }
      }
      {
        \__fp_error:nffn { fp-num-args }
          { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
        \exp_after:wN \c_nan_fp
      }
  }
```

(*End definition for* `\__fp_round:Nwww`*.*)

__fp_round_name_from_cs:N

```
\cs_new:Npn \__fp_round_name_from_cs:N #1
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
      {
        \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
          {
            \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
              { round }
          }
      }
  }
```

(*End definition for* __fp_round_name_from_cs:N*.*)

__fp_round:Nww
__fp_round:Nwn
__fp_round_normal:NwNNnw
__fp_round_normal:NnnwNNnn
__fp_round_pack:Nw
__fp_round_normal:NNwNnn
__fp_round_normal_end:wwNnn
__fp_round_special:NwwNnn
__fp_round_special_aux:Nw

```
\cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
  {
    \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
      {
        \__fp_invalid_operation_tl_o:ff
          { \__fp_round_name_from_cs:N #1 }
          { \__fp_array_to_clist:n { #2; #3; } }
      }
  }
\cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_round_normal:NwNNnw
      \exp_after:wN #1
      \__int_value:w #5
    \else:
      \exp_after:wN \__fp_exp_after_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
\cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
  {
    \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
      \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
  }
\cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
  {
    \exp_after:wN \__fp_round_normal:NNwNnn
    \__int_value:w \__int_eval:w
      \if_int_compare:w #2 > 0 \exp_stop_f:
        1 \__int_value:w #2
        \exp_after:wN \__fp_round_pack:Nw
        \__int_value:w \__int_eval:w 1#3 +
      \else:
        \if_int_compare:w #3 > 0 \exp_stop_f:
          1 \__int_value:w #3 +
        \fi:
      \fi:
```

```
      \exp_after:wN #5
      \exp_after:wN #6
      \use_none:nnnnnnn #3
      #1
      \__int_eval_end:
      0000 0000 0000 0000 ; #6
  }
\cs_new:Npn \__fp_round_pack:Nw #1
  { \if_meaning:w 2 #1 + 1 \fi: \__int_eval_end: }
\cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_round_special:NwwNnn
      \exp_after:wN #1
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_round_normal_end:wwNnn
    ; #2
  }
\cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
  {
    \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
    \__fp_sanitize:Nw #3 #4 ; #1 ;
  }
\cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
  {
    \if_meaning:w 0 #1
      \__fp_case_return:nw
        { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
    \else:
      \exp_after:wN \__fp_round_special_aux:Nw
      \exp_after:wN #4
      \__int_value:w \__int_eval:w 1
        \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
    \fi:
    ;
  }
\cs_new:Npn \__fp_round_special_aux:Nw #1#2;
  {
    \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
    \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
  }
```

(*End definition for* `\__fp_round:Nww` *and others.*)

```
⟨/initex | package⟩
```

24 l3fp-parse implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

24.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

__fp_parse:n

`\__fp_parse:n {⟨fpexpr⟩}`

Evaluates the ⟨*floating point expression*⟩ and leaves the result in the input stream as an internal floating point number. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully `f`-expanded.

`\__fp_parse_o:n` does the same but expands once after its result.

TEXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion lead to unrecoverable low-level TEX errors.

(*End definition for* `\__fp_parse:n`.)

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls with multiple arguments.

15 Function calls expecting exactly one argument.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

10 Binary `*`, `/`, and juxtaposition (implicit `*`).

9 Binary `+` and `-`.

7 Comparisons.

6 Logical `and`, denoted by `&&`.

5 Logical `or`, denoted by `||`.

4 Ternary operator `?:`, piece `?`.

3 Ternary operator `?:`, piece `:`.

2 Commas, and parentheses accepting commas.

1 Parentheses expecting exactly one argument.

0 Start and end of the expression.

\c__fp_prec_funcii_int
\c__fp_prec_func_int
\c__fp_prec_hatii_int
\c__fp_prec_hat_int
\c__fp_prec_not_int
\c__fp_prec_times_int
\c__fp_prec_plus_int
\c__fp_prec_comp_int
\c__fp_prec_and_int
\c__fp_prec_or_int
\c__fp_prec_quest_int
\c__fp_prec_colon_int
\c__fp_prec_comma_int
\c__fp_prec_paren_int
\c__fp_prec_end_int

```
\int_const:Nn \c__fp_prec_funcii_int { 16 }
\int_const:Nn \c__fp_prec_func_int   { 15 }
\int_const:Nn \c__fp_prec_hatii_int  { 14 }
\int_const:Nn \c__fp_prec_hat_int    { 13 }
\int_const:Nn \c__fp_prec_not_int    { 12 }
\int_const:Nn \c__fp_prec_times_int  { 10 }
\int_const:Nn \c__fp_prec_plus_int   { 9 }
\int_const:Nn \c__fp_prec_comp_int   { 7 }
\int_const:Nn \c__fp_prec_and_int    { 6 }
\int_const:Nn \c__fp_prec_or_int     { 5 }
\int_const:Nn \c__fp_prec_quest_int  { 4 }
\int_const:Nn \c__fp_prec_colon_int  { 3 }
\int_const:Nn \c__fp_prec_comma_int  { 2 }
\int_const:Nn \c__fp_prec_paren_int  { 1 }
\int_const:Nn \c__fp_prec_end_int    { 0 }
```

(*End definition for* `\c__fp_prec_funcii_int` *and others.*)

24.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to f-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be much better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \__int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w ⟨stuff⟩
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\__int_value:w`, which reads the five digits we have already found, `12345`. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \__int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\__int_value:w` has already seen `12345`, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\__int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\__int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `\__fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

24.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because × has a higher precedence than +. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation `41-2^3*4+5`. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c__fp_prec_plus_int`, …) equal to the precedence of the given operator, not directly the operator itself.

- Clean up `41` and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up `2` and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up `3` and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have `41+8*4+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?

- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `-`.
- Compare the precedences of `*` and `-`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have `41+32+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have `9+5`.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the l3fp-parse functions correspond to each step above.

First, `\__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by ⟨*precedence*⟩ the argument of `\__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `\__fp_parse_one:Nw`. A first approximation of this function is that it reads one ⟨*number*⟩, performing no computation, and finds the following binary ⟨*operator*⟩. Then it expands to

```
⟨number⟩
  \__fp_parse_infix_⟨operator⟩:N ⟨precedence⟩
```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `\__fp_parse_infix_`⟨*operator*⟩`:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the ⟨*precedence*⟩ (of the earlier operator) to the `infix` auxiliary for the following ⟨*operator*⟩, to know whether to perform the computation of the ⟨*operator*⟩. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_⟨operator⟩:N
```

and otherwise it calls `\__fp_parse_operand:Nw` with the precedence of the ⟨*operator*⟩ to find its second operand ⟨*number*$_2$⟩ and the next ⟨*operator*$_2$⟩, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
    ⟨operator⟩ ⟨number_2⟩
@ \__fp_parse_infix_⟨operator_2⟩:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand ⟨*number*⟩ is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\__fp_parse_operand:Nw` ⟨*precedence*⟩ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
  \__fp_parse_one:Nw ⟨precedence⟩
```

This expands `\__fp_parse_one:Nw` ⟨*precedence*⟩ completely, which finds a number, wraps the next ⟨*operator*⟩ into an `infix` function, feeds this function the ⟨*precedence*⟩, and expands it, yielding either

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\use_none:n \__fp_parse_infix_⟨operator⟩:N
```

or

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\__fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number_2⟩
@ \__fp_parse_infix_⟨operator_2⟩:N
```

The definition of `\__fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn \__fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }
```

In the first case, `#3` is `\use_none:n`, yielding

```
\use_none:n ⟨precedence⟩ ⟨number⟩ @
\__fp_parse_infix_⟨operator⟩:N
```

then ⟨*number*⟩ `@ \__fp_parse_infix_`⟨*operator*⟩`:N`. In the second case, `#3` is `\__fp_parse_apply_binary:NwNwN`, whose role is to compute ⟨*number*⟩ ⟨*operator*⟩ ⟨*number*$_2$⟩ and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
  ⟨precedence⟩ ⟨number⟩ @
  ⟨operator⟩ ⟨number_2⟩
@ \__fp_parse_infix_⟨operator_2⟩:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
\__fp_⟨operator⟩_o:ww ⟨number⟩ ⟨number_2⟩
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_⟨operator_2⟩:N ⟨precedence⟩
```

where __fp_⟨*operator*⟩_o:ww computes ⟨*number*⟩ ⟨*operator*⟩ $\langle\mathit{number}_2\rangle$ and expands after the result, thus triggers the comparison of the precedence of the $\langle\mathit{operator}_2\rangle$ and the ⟨*precedence*⟩, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

24.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary -, +, !) and parentheses are taken care of by the same mechanism, and functions (sin, exp, etc.) as well. Finding the argument of the unary -, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary -, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function __fp_parse_one:Nw earlier. This function is responsible for reading in the input stream the first possible ⟨*number*⟩ and the next infix ⟨*operator*⟩. If what follows __fp_parse_one:Nw ⟨*precedence*⟩ is a prefix operator, then we must find the operand of this prefix operator through a nested call to __fp_parse_operand:Nw with the appropriate precedence, then apply the operator to the operand found to yield the result of __fp_parse_one:Nw. So far, all is simple.

The unary operators +, -, ! complicate things a little bit: -3**2 should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving - a lower precedence, equal to that of the infix + and -. Unfortunately, this fails in cases such as 3**-2*4, yielding $3^{-2\times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call __fp_parse_operand:Nw with the ⟨*precedence*⟩ of the previous operator, but 0>-2+3 is then parsed as 0>-(2+3): the addition is performed because it binds more tightly than the comparision which precedes -. The correct approach is for a unary - to perform operations whose precedence is greater than both that of the previous operation, and that of the unary - itself. The unary - is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to __fp_parse_operand:Nw.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the infix functions discussed earlier, the prefix functions do perform tests on the previous ⟨*precedence*⟩ to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

24.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a "number"? A number is typically given in the form ⟨*significand*⟩`e`⟨*exponent*⟩, where the ⟨*significand*⟩ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent "`e`⟨*exponent*⟩" is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The ⟨*significand*⟩ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the ⟨*exponent*⟩ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\__fp_parse_one:Nw` is looking for a "number", here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.

- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the ⟨*significand*⟩ of a number: we look for an exponent.

- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.

- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `\__fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.

- If the next token is anything else, we check whether it is a known prefix operator, in which case `\__fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `\__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero`, with an implied multiplication.

- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.

- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\__fp_infix_`⟨*operator*⟩`:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude `0`, replace `9` by `10`. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `` `#1 `` lies in $[65, 90]$ (uppercase letters) or $[97, 112]$ (lowercase letters)

```
\if_int_compare:w \__int_eval:w
    ( `#1 \if_int_compare:w `#1 > `Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes $\{3, 6, 7, 8, 11, 12\}$ should work without trouble, but not $\{1, 2, 4, 10, 13\}$, and of course $\{0, 5, 9\}$ cannot become tokens.

Floating point expressions should behave as much as possible like ε-TEX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
  \DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
  \ExplSyntaxOff
  \test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then `f`-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the `f`-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp` ... structure. The `f`-expansion is performed by `\__fp_parse_expand:w`.

24.2 Main auxiliary functions

__fp_parse_operand:Nw

```
\exp:w \__fp_parse_operand:Nw ⟨precedence⟩ \__fp_parse_expand:w
```

Reads the “...", performing every computation with a precedence higher than ⟨*precedence*⟩, then expands to

```
⟨result⟩ @ \__fp_parse_infix_⟨operation⟩:N ...
```

where the ⟨*operation*⟩ is the first operation with a lower precedence, possibly `end`, and the "`...`" start just after the ⟨*operation*⟩.

(*End definition for* `\__fp_parse_operand:Nw`.)

`\__fp_parse_infix_+:N`

```
\__fp_parse_infix_+:N ⟨precedence⟩ ...
```

If `+` has a precedence higher than the ⟨*precedence*⟩, cleans up a second ⟨*operand*⟩ and finds the ⟨*operation*$_2$⟩ which follows, and expands to

```
@ \__fp_parse_apply_binary:NwNwN + ⟨operand⟩ @ \__fp_parse_infix_⟨operation_2⟩:N
...
```

Otherwise expands to

```
@ \use_none:n \__fp_parse_infix_+:N ...
```

A similar function exists for each infix operator.

(*End definition for* `\__fp_parse_infix_+:N`.)

`\__fp_parse_one:Nw`

```
\__fp_parse_one:Nw ⟨precedence⟩ ...
```

Cleans up one or two operands depending on how the precedence of the next operation compares to the ⟨*precedence*⟩. If the following ⟨*operation*⟩ has a precedence higher than ⟨*precedence*⟩, expands to

```
⟨operand_1⟩ @ \__fp_parse_apply_binary:NwNwN ⟨operation⟩ ⟨operand_2⟩ @
\__fp_parse_infix_⟨operation_2⟩:N ...
```

and otherwise expands to

```
⟨operand⟩ @ \use_none:n \__fp_parse_infix_⟨operation⟩:N ...
```

(*End definition for* `\__fp_parse_one:Nw`.)

24.3 Helpers

`\__fp_parse_expand:w`

```
\exp:w \__fp_parse_expand:w ⟨tokens⟩
```

This function must always come within a `\exp:w` expansion. The ⟨*tokens*⟩ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
\cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(*End definition for* `\__fp_parse_expand:w`.)

`\__fp_parse_return_semicolon:w`

This very odd function swaps its position with the following `\fi:` and removes `\__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

```
\cs_new:Npn \__fp_parse_return_semicolon:w
    #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(*End definition for* `\__fp_parse_return_semicolon:w`.)

__fp_type_from_scan:N
__fp_type_from_scan:w

`\__fp_type_from_scan:N` ⟨*token*⟩

Grabs the pieces of the stringified ⟨*token*⟩ which lies after the first `s__fp`. If the ⟨*token*⟩ does not contain that string, the result is `_?`.

```
\cs_new:Npx \__fp_type_from_scan:N #1
  {
    \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
    \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
      \tl_to_str:n { s__fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \__fp_type_from_scan:w
      ##1 \tl_to_str:n { s__fp } ##2 \exp_not:N \q_mark ##3 \exp_not:N \q_stop
      {##2}
  }
```

(*End definition for* `\__fp_type_from_scan:N` *and* `\__fp_type_from_scan:w`*.*)

__fp_parse_digits_vii:N
__fp_parse_digits_vi:N
__fp_parse_digits_v:N
__fp_parse_digits_iv:N
__fp_parse_digits_iii:N
__fp_parse_digits_ii:N
__fp_parse_digits_i:N
__fp_parse_digits_:N

These functions must be called within an `\__int_value:w` or `\__int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

⟨*digits*⟩ `;` ⟨*filling 0*⟩ `;` ⟨*length*⟩

where ⟨*filling 0*⟩ is a string of zeros such that ⟨*digits*⟩ ⟨*filling 0*⟩ has the length given by the index of the function, and ⟨*length*⟩ is the number of zeros in the ⟨*filling 0*⟩ string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
      {
        \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
          \token_to_str:N ##1 \exp_after:wN #2 \exp:w
        \else:
          \__fp_parse_return_semicolon:w #3 ##1
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w {vii}  \__fp_parse_digits_vi:N   { 0000000 ; 7 }
\__fp_tmp:w {vi}   \__fp_parse_digits_v:N    { 000000 ; 6 }
\__fp_tmp:w {v}    \__fp_parse_digits_iv:N   { 00000 ; 5 }
\__fp_tmp:w {iv}   \__fp_parse_digits_iii:N  { 0000 ; 4 }
\__fp_tmp:w {iii}  \__fp_parse_digits_ii:N   { 000 ; 3 }
\__fp_tmp:w {ii}   \__fp_parse_digits_i:N    { 00 ; 2 }
\__fp_tmp:w {i}    \__fp_parse_digits_:N     { 0 ; 1 }
\cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(*End definition for* `\__fp_parse_digits_vii:N` *and others.*)

24.4 Parsing one number

__fp_parse_one:Nw

This function finds one number, and packs the symbol which follows in an `\__fp_parse_infix_...` csname. `#1` is the previous ⟨*precedence*⟩, and `#2` the first token of the operand. We distinguish four cases: `#2` is equal to `\scan_stop:` in meaning, `#2` is a different control sequence, `#2` is a digit, and `#2` is something else (this last case is split further later). Despite the earlier `f`-expansion, `#2` may still be expandable if it was protected by `\exp_not:N`, as may happen with the LaTeX 2_ε command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `\__fp_parse_one_fp:NN` which deals with it robustly.

```
\cs_new:Npn \__fp_parse_one:Nw #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
        \exp_after:wN \reverse_if:N
      \fi:
      \if_meaning:w \scan_stop: #2
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_fp:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_register:NN
      \fi:
    \else:
      \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_digit:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_other:NN
      \fi:
    \fi:
    #1 #2
  }
```

(*End definition for* `\__fp_parse_one:Nw`.)

__fp_parse_one_fp:NN
__fp_exp_after_mark_f:nw
__fp_exp_after_?_f:nw

This function receives a ⟨*precedence*⟩ and a control sequence equal to `\scan_stop:` in meaning. There are three cases, dispatched using `\__fp_type_from_scan:N`.

- `\s__fp` starts a floating point number, and we call `\__fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_mark` is a premature end, we call `\__fp_exp_after_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s__fp`, we call `\__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_`⟨*type*⟩ and defining `\__fp_exp_after_`⟨*type*⟩`_f:nw`. In all cases, we make sure that the second argument of `\__fp_parse_infix:NN` is correctly expanded. A special case only enabled in LaTeX 2_ε is that if `\protect` is encountered then the error message mentions the control sequence which follows it rather than \protect itself. The test for LaTeX 2$_\varepsilon$ uses \@unexpandable@protect rather than \protect because \protect is often \scan_stop: hence "does not exist".

```
\cs_new:Npn \__fp_parse_one_fp:NN #1#2
  {
    \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
      {
        \exp_after:wN \__fp_parse_infix:NN
        \exp_after:wN #1 \exp:w \__fp_parse_expand:w
      }
    #2
  }
\cs_new:Npn \__fp_exp_after_mark_f:nw #1
  {
    \__msg_kernel_expandable_error:nn { kernel } { fp-early-end }
    \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
  }
\cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
  {
    \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
    \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
  }
⟨*package⟩
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_if_exist:NT #1
      {
        \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
          {
            \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
            \str_if_eq:nnTF {##2} { \protect }
              {
                \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
                { \__msg_kernel_expandable_error:nnn { kernel } { fp-robust-cmd } }
              }
              { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {##2} }
          }
      }
  }
\exp_args:Nc \__fp_tmp:w { @unexpandable@protect }
⟨/package⟩
```

(*End definition for __fp_parse_one_fp:NN, __fp_exp_after_mark_f:nw, and __fp_exp_after_?_f:nw.*)

__fp_parse_one_register:NN
__fp_parse_one_register_aux:Nw
__fp_parse_one_register_auxii:wwwNw
__fp_parse_one_register_int:www
__fp_parse_one_register_mu:www
__fp_parse_one_register_dim:ww
__fp_parse_one_register_wd:w
__fp_parse_one_register_wd:Nw

This is called whenever #2 is a control sequence other than \scan_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert ⟨*value*⟩e⟨*exponent*⟩ to a floating point number with __fp_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `\__int_value:w \__dim_eval:w` ⟨*decimal value*⟩ `pt`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16}, correctly rounded.

```
\cs_new:Npn \__fp_parse_one_register:NN #1#2
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \if_meaning:w \box_wd:N #2 \__fp_parse_one_register_wd:w \fi:
      \if_meaning:w \box_ht:N #2 \__fp_parse_one_register_wd:w \fi:
      \if_meaning:w \box_dp:N #2 \__fp_parse_one_register_wd:w \fi:
      \exp_after:wN \__fp_parse_one_register_aux:Nw
      \exp_after:wN #2
      \__int_value:w
        \exp_after:wN \__fp_parse_exponent:N
        \exp:w \__fp_parse_expand:w
  }
\cs_new:Npx \__fp_parse_one_register_aux:Nw #1
  {
    \exp_not:n
      {
        \exp_after:wN \use:nn
        \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
      }
    \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
      ; \exp_not:N \__fp_parse_one_register_dim:ww
      \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
      . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
      \exp_not:N \q_stop
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \__fp_parse_one_register_auxii:wwwNw
        ##1 . ##2 \tl_to_str:n { pt } ##3 ; ##4##5 \exp_not:N \q_stop
        { ##4 ##1.##2; }
    \cs_new:Npn \exp_not:N \__fp_parse_one_register_mu:www
      ##1 \tl_to_str:n { mu } ; ##2 ;
      { \exp_not:N \__fp_parse_one_register_dim:ww ##1 ; }
  }
\cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
  { \__fp_parse:n { #1 e #3 } }
\cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \__int_value:w #2 \exp_after:wN ,
    \__int_value:w \__dim_eval:w #1 pt ;
  }
```

The `\wd`, `\dp`, `\ht` primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker `e`. Once that "exponent" is found, use `\tex_the:D` to find the box dimension and then copy what we did for dimensions.

```
\cs_new:Npn \__fp_parse_one_register_wd:w
    #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
```

```
  {
    #1
    \exp_after:wN \__fp_parse_one_register_wd:Nw
    #4 \__fp_parse_expand:w e
  }
\cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \exp_after:wN 0 \exp_after:wN ,
    \__int_value:w \__dim_eval:w
      \exp_after:wN \use:n \exp_after:wN { \tex_the:D #1 #2 } ;
  }
```

(*End definition for* `\__fp_parse_one_register:NN` *and others.*)

`\__fp_parse_one_digit:NN`

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with `\__fp_sanitize:wN`, then `\__fp_parse_infix_after_operand:NwN` expands `\__fp_parse_infix:NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```
\cs_new:Npn \__fp_parse_one_digit:NN #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \exp_after:wN \__fp_sanitize:wN
      \__int_value:w \__int_eval:w 0 \__fp_parse_trim_zeros:N
  }
```

(*End definition for* `\__fp_parse_one_digit:NN`*.*)

`\__fp_parse_one_other:NN`

For this function, `#2` is a character token which is not a digit. If it is an ASCII letter, `\__fp_parse_letters:N` beyond this one and give the result to `\__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `\__fp_parse_prefix_`⟨*operator*⟩`:Nw`.

```
\cs_new:Npn \__fp_parse_one_other:NN #1 #2
  {
    \if_int_compare:w
        \__int_eval:w
          ( `#2 \if_int_compare:w `#2 > `Z - 32 \fi: ) / 26
        = 3 \exp_stop_f:
      \exp_after:wN \__fp_parse_word:Nw
      \exp_after:wN #1
      \exp_after:wN #2
      \exp:w \exp_after:wN \__fp_parse_letters:N
      \exp:w
    \else:
      \exp_after:wN \__fp_parse_prefix:NNN
      \exp_after:wN #1
      \exp_after:wN #2
      \cs:w
        __fp_parse_prefix_ \token_to_str:N #2 :Nw
        \exp_after:wN
      \cs_end:
```

```
      \exp:w
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_one_other:NN`*.*)

__fp_parse_word:Nw
__fp_parse_letters:N

Finding letters is a simple recursion. Once `\__fp_parse_letters:N` has done its job, we try to build a control sequence from the word `#2`. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires "inf" and "infinity" and "nan" to be recognized regardless of case, but we probably don't want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```
\cs_new:Npn \__fp_parse_word:Nw #1#2;
  {
    \cs_if_exist_use:cF { __fp_parse_word_#2:N }
      {
        \cs_if_exist_use:cF { __fp_parse_caseless_ \str_fold_case:n {#2} :N }
          {
            \__msg_kernel_expandable_error:nnn
              { kernel } { unknown-fp-word } {#2}
            \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
            \__fp_parse_infix:NN
          }
      }
      #1
  }
\cs_new:Npn \__fp_parse_letters:N #1
  {
    \exp_end_continue_f:w
    \if_int_compare:w
        \if_catcode:w \scan_stop: \exp_not:N #1
          0
        \else:
          \__int_eval:w
            ( `#1 \if_int_compare:w `#1 > `Z - 32 \fi: ) / 26
        \fi:
        = 3 \exp_stop_f:
      \exp_after:wN #1
      \exp:w \exp_after:wN \__fp_parse_letters:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_word:Nw` *and* `\__fp_parse_letters:N`*.*)

__fp_parse_prefix:NNN
__fp_parse_prefix_unknown:NNN

For this function, `#1` is the previous ⟨*precedence*⟩, `#2` is the operator just seen, and `#3` is a control sequence which implements the operator if it is a known operator. If this control sequence is `\scan_stop:`, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from `\__fp_parse_one:Nw`.

```
\cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
  {
    \if_meaning:w \scan_stop: #3
      \exp_after:wN \__fp_parse_prefix_unknown:NNN
      \exp_after:wN #2
    \fi:
    #3 #1
  }
\cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
  {
    \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { fp-missing-number } {#1}
        \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
        \__fp_parse_infix:NN #3 #1
      }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { fp-unknown-symbol } {#1}
        \__fp_parse_one:Nw #3
      }
  }
```

(*End definition for* `\__fp_parse_prefix:NNN` *and* `\__fp_parse_prefix_unknown:NNN`.)

24.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `\__fp_parse_large`...; if it is a period, the significand is < 1; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1\rangle < 0$, then read the significand with the set of functions `\__fp_parse_small`... Once the significand is read, read the exponent if `e` is present.

`\__fp_parse_trim_zeros:N`
`\__fp_parse_trim_end:w`

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `\__fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `\__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `\__fp_parse_zero:` to take care of that case.

```
\cs_new:Npn \__fp_parse_trim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      \exp_after:wN \__fp_parse_trim_zeros:N
      \exp:w
    \else:
      \if:w . \exp_not:N #1
        \exp_after:wN \__fp_parse_strim_zeros:N
        \exp:w
      \else:
        \__fp_parse_trim_end:w #1
      \fi:
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
  {
      \fi:
    \fi:
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_large:N
    \else:
      \exp_after:wN \__fp_parse_zero:
    \fi:
    #1
  }
```

(*End definition for* `\__fp_parse_trim_zeros:N` *and* `\__fp_parse_trim_end:w`.)

__fp_parse_strim_zeros:N
__fp_parse_strim_end:w

If we have removed all digits until a period (or if the body started with a period), then enter the "`small_trim`" loop which outputs -1 for each removed 0. Those -1 are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\__fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for "small trim".

```
\cs_new:Npn \__fp_parse_strim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      - 1
      \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
    \else:
      \__fp_parse_strim_end:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
  {
    \fi:
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_small:N
    \else:
      \exp_after:wN \__fp_parse_zero:
    \fi:
    #1
  }
```

(*End definition for* `\__fp_parse_strim_zeros:N` *and* `\__fp_parse_strim_end:w`.)

__fp_parse_zero:

After reading a significand of 0, find any exponent, then put a sign of `1` for `\__fp_sanitize:wN`, which removes everything and leaves an exact zero.

```
\cs_new:Npn \__fp_parse_zero:
  {
    \exp_after:wN ; \exp_after:wN 1
    \__int_value:w \__fp_parse_exponent:N
  }
```

(*End definition for `\__fp_parse_zero:`.*)

24.4.2 Number: small significand

__fp_parse_small:N

This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can't do that all at once, because `\__int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `\__fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\__int_value:w`, and grabs some more, or stops if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```
\cs_new:Npn \__fp_parse_small:N #1
  {
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \__int_value:w \__int_eval:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_small_leading:wwNN
      \__int_value:w 1
        \exp_after:wN \__fp_parse_digits_vii:N
        \exp:w \__fp_parse_expand:w
  }
```

(*End definition for `\__fp_parse_small:N`.*)

__fp_parse_small_leading:wwNN

`\__fp_parse_small_leading:wwNN 1` ⟨*digits*⟩ `;` ⟨*zeros*⟩ `;` ⟨*number of zeros*⟩

We leave ⟨*digits*⟩ ⟨*zeros*⟩ in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If `#4` is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```
\cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
    \exp_after:wN 0
    \__int_value:w \__int_eval:w 1
      \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
        \token_to_str:N #4
        \exp_after:wN \__fp_parse_small_trailing:wwNN
        \__int_value:w 1
          \exp_after:wN \__fp_parse_digits_vi:N
          \exp:w
      \else:
        0000 0000 \__fp_parse_exponent:Nw #4
      \fi:
      \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_small_leading:wwNN`*.*)

__fp_parse_small_trailing:wwNN

`\__fp_parse_small_trailing:wwNN 1` ⟨*digits*⟩ `;` ⟨*zeros*⟩ `;` ⟨*number of zeros*⟩ ⟨*next token*⟩

Leave digits 10 to 15 (arguments `#1` and `#2`) in the input stream. If the ⟨*next token*⟩ is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to `+0` or to `+1`. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```
\cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
      \token_to_str:N #4
      \exp_after:wN \__fp_parse_small_round:NN
      \exp_after:wN #4
      \exp:w
    \else:
      0 \__fp_parse_exponent:Nw #4
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_small_trailing:wwNN`*.*)

__fp_parse_pack_trailing:NNNNNNww
__fp_parse_pack_leading:NNNNNww
__fp_parse_pack_carry:w

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (`+1` in the code below). If the leading digits propagate this carry all the way up, the function `\__fp_parse_pack_carry:w` increments the exponent, and changes the significand from `0000...` to `1000...`: this is simple because such a carry can only occur to give rise to a power of 10.

```
\cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
  {
    \if_meaning:w 2 #2 + 1 \fi:
    ; #8 + #1 ; {#3#4#5#6} {#7};
  }
\cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
  {
    + #7
    \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
    ; 0 {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
  { \fi: + 1 ; 0 {1000} }
```

(*End definition for* `\__fp_parse_pack_trailing:NNNNNNww`*,* `\__fp_parse_pack_leading:NNNNNww`*, and* `\__fp_parse_pack_carry:w`*.*)

24.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

__fp_parse_large:N

This function is followed by the first non-zero digit of a "large" significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```
\cs_new:Npn \__fp_parse_large:N #1
  {
    \exp_after:wN \__fp_parse_large_leading:wwNN
    \__int_value:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_digits_vii:N
      \exp:w \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_large:N`*.*)

__fp_parse_large_leading:wwNN

`\__fp_parse_large_leading:wwNN 1 ⟨digits⟩ ; ⟨zeros⟩ ; ⟨number of zeros⟩ ⟨next token⟩`

We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the ⟨*number of zeros*⟩ (number of digits missing). Then prepare to pack the 8 first digits. If the ⟨*next token*⟩ is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the ⟨*zeros*⟩ to complete the 8 first digits, insert 8 more, and look for an exponent.

```
\cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
  {
    + \c__fp_half_prec_int - #3
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \__int_value:w \__int_eval:w 1 #1
      \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
        \exp_after:wN \__fp_parse_large_trailing:wwNN
        \__int_value:w 1 \token_to_str:N #4
          \exp_after:wN \__fp_parse_digits_vi:N
          \exp:w
      \else:
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_leading:wwNN
          \__int_value:w 1
            \cs:w
              __fp_parse_digits_
              \__int_to_roman:w #3
              :N \exp_after:wN
            \cs_end:
            \exp:w
        \else:
          #2
          \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
          \exp_after:wN 0
          \__int_value:w 1 0000 0000
          \__fp_parse_exponent:Nw #4
        \fi:
      \fi:
      \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_large_leading:wwNN`.)

__fp_parse_large_trailing:wwNN

`\__fp_parse_large_trailing:wwNN 1` ⟨*digits*⟩ `;` ⟨*zeros*⟩ `;` ⟨*number of zeros*⟩ ⟨*next token*⟩

We have just read 15 digits. If the ⟨*next token*⟩ is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the ⟨*digits*⟩ and this 16-th digit, and find how this should be rounded using `\__fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of ⟨*digits*⟩, 7 minus the ⟨*number of zeros*⟩, and we test for a decimal point. This case happens in `123451234512345.67` with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the ⟨*zeros*⟩ and providing a 16-th digit of 0.

```
\cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \exp_after:wN \c__fp_half_prec_int
      \__int_value:w \__int_eval:w 1 #1 \token_to_str:N #4
        \exp_after:wN \__fp_parse_large_round:NN
        \exp_after:wN #4
        \exp:w
    \else:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \__int_value:w \__int_eval:w 7 - #3 \exp_stop_f:
      \__int_value:w \__int_eval:w 1 #1
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_trailing:wwNN
          \__int_value:w 1
            \cs:w
              __fp_parse_digits_
              \__int_to_roman:w #3
              :N \exp_after:wN
            \cs_end:
            \exp:w
        \else:
          #2 0 \__fp_parse_exponent:Nw #4
        \fi:
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_large_trailing:wwNN`.)

24.4.4 Number: beyond 16 digits, rounding

__fp_parse_round_loop:N
__fp_parse_round_up:N

This loop is called when rounding a number (whether the mantissa is small or large). It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `;0`, otherwise by `;1`. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```
\cs_new:Npn \__fp_parse_round_loop:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      + 1
      \if:w 0 \token_to_str:N #1
        \exp_after:wN \__fp_parse_round_loop:N
        \exp:w
      \else:
        \exp_after:wN \__fp_parse_round_up:N
        \exp:w
      \fi:
    \else:
      \__fp_parse_return_semicolon:w 0 #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_round_up:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      + 1
      \exp_after:wN \__fp_parse_round_up:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w 1 #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_round_loop:N` *and* `\__fp_parse_round_up:N`.)

__fp_parse_round_after:wN

After the loop `\__fp_parse_round_loop:N`, this function fetches an exponent with `\__fp_parse_exponent:N`, and combines it with the number of digits counted by `\__fp_parse_round_loop:N`. At the same time, the result `0` or `1` is added to the surrounding integer expression.

```
\cs_new:Npn \__fp_parse_round_after:wN #1; #2
  {
    + #2 \exp_after:wN ;
    \__int_value:w \__int_eval:w #1 + \__fp_parse_exponent:N
  }
```

(*End definition for* `\__fp_parse_round_after:wN`.)

__fp_parse_small_round:NN
__fp_parse_round_after:wN

Here, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If `#2` is not a digit, then fetch an exponent and expand to `;`⟨*exponent*⟩ only. Otherwise, we expand to `+0` or `+1`, then `;`⟨*exponent*⟩. To decide which, call `\__fp_round_s:NNNw` to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```
\cs_new:Npn \__fp_parse_small_round:NN #1#2
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \__int_value:w \__int_eval:w
        \exp_after:wN \__fp_parse_round_after:wN
        \__int_value:w \__int_eval:w 0 * \__int_eval:w 0
          \exp_after:wN \__fp_parse_round_loop:N
          \exp:w
    \else:
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.*)

__fp_parse_large_round:NN
__fp_parse_large_round_test:NN
__fp_parse_large_round_aux:wNN

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```
\cs_new:Npn \__fp_parse_large_round:NN #1#2
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \__int_value:w \__int_eval:w
        \exp_after:wN \__fp_parse_large_round_aux:wNN
        \__int_value:w \__int_eval:w 1
          \exp_after:wN \__fp_parse_round_loop:N
    \else: %^^A could be dot, or e, or other
      \exp_after:wN \__fp_parse_large_round_test:NN
      \exp_after:wN #1
      \exp_after:wN #2
    \fi:
  }
```

```
\cs_new:Npn \__fp_parse_large_round_test:NN #1#2
  {
    \if:w . \exp_not:N #2
      \exp_after:wN \__fp_parse_small_round:NN
      \exp_after:wN #1
      \exp:w
    \else:
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
  {
    + #2
    \exp_after:wN \__fp_parse_round_after:wN
    \__int_value:w \__int_eval:w #1
      \if:w . \exp_not:N #3
        + 0 * \__int_eval:w 0
          \exp_after:wN \__fp_parse_round_loop:N
          \exp:w \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN ;
        \exp_after:wN 0
        \exp_after:wN #3
      \fi:
  }
```

(*End definition for* `\__fp_parse_large_round:NN`*,* `\__fp_parse_large_round_test:NN`*, and* `\__fp_parse_large_round_aux:wNN`*.*)

24.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```
\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }
```

The first case indicates that just looking one character ahead for an "`e`" is not enough, since we would mistake the function `erf` for an exponent of "`rf`". An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading 3.2, `\c_pi_fp` is expanded to `\s__fp \__fp_chk:w 1 0 {-1} {3141}` $\cdots$ `;` and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `\__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as TeX does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

__fp_parse_exponent:Nw

This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument `#2`) at a very odd place because this allows us to insert `\__int_eval:w` ... there if needed.

```
\cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
  {
    \exp_after:wN ;
    \__int_value:w #2 \__fp_parse_exponent:N #1
  }
```

(*End definition for* `\__fp_parse_exponent:Nw`*.*)

__fp_parse_exponent:N
__fp_parse_exponent_aux:N

This function should be called within an `\__int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e`, leave an exponent of 0. If there is an `e`, expand the next token to run some tests on it. The first rough test is that if the character code of `#1` is greater than that of `9` (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```
\cs_new:Npn \__fp_parse_exponent:N #1
  {
    \if:w e \exp_not:N #1
      \exp_after:wN \__fp_parse_exponent_aux:N
      \exp:w
    \else:
      0 \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_exponent_aux:N #1
  {
    \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
                0 \else: ‘#1 \fi: > ‘9 \exp_stop_f:
      0 \exp_after:wN ; \exp_after:wN e
    \else:
      \exp_after:wN \__fp_parse_exponent_sign:N
    \fi:
    #1
  }
```

(*End definition for* `\__fp_parse_exponent:N` *and* `\__fp_parse_exponent_aux:N`*.*)

__fp_parse_exponent_sign:N

Read signs one by one (if there is any).

```
\cs_new:Npn \__fp_parse_exponent_sign:N #1
  {
    \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_sign:N
      \exp:w \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN \__fp_parse_exponent_body:N
      \exp_after:wN #1
    \fi:
  }
```

(*End definition for* `\__fp_parse_exponent_sign:N`*.*)

__fp_parse_exponent_body:N

An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```
\cs_new:Npn \__fp_parse_exponent_body:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \exp:w
    \else:
      \__fp_parse_exponent_keep:NTF #1
        { \__fp_parse_return_semicolon:w #1 }
        {
          \exp_after:wN ;
          \exp:w
        }
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* __fp_parse_exponent_body:N.)

__fp_parse_exponent_digits:N

Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```
\cs_new:Npn \__fp_parse_exponent_digits:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* __fp_parse_exponent_digits:N.)

__fp_parse_exponent_keep:NTF

This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than `+`, `-` or digits, again, an error.

```
\prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
  {
    \if_catcode:w \scan_stop: \exp_not:N #1
      \if_meaning:w \scan_stop: #1
        \if_int_compare:w
      \__str_if_eq_x:nn { \s__fp } { \exp_not:N #1 }
      = 0 \exp_stop_f:
     0
     \__msg_kernel_expandable_error:nnn
      { kernel } { fp-after-e } { floating~point~ }
     \prg_return_true:
    \else:
     0
     \__msg_kernel_expandable_error:nnn
      { kernel } { bad-variable } {#1}
     \prg_return_false:
    \fi:
   \else:
    \if_int_compare:w
      \__str_if_eq_x:nn { \__int_value:w #1 } { \tex_the:D #1 }
      = 0 \exp_stop_f:
     \__int_value:w #1
    \else:
     0
     \__msg_kernel_expandable_error:nnn
      { kernel } { fp-after-e } { dimension~#1 }
    \fi:
    \prg_return_false:
   \fi:
  \else:
   0
   \__msg_kernel_expandable_error:nnn
    { kernel } { fp-missing } { exponent }
   \prg_return_true:
  \fi:
 }
```

(*End definition for* `\__fp_parse_exponent_keep:NTF`.)

24.5 Constants, functions and prefix operators

24.5.1 Prefix operators

`\__fp_parse_prefix_+:Nw`

A unary `+` does nothing: we should continue looking for a number.

```
\cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw
```

(*End definition for* `\__fp_parse_prefix_+:Nw`.)

`\__fp_parse_apply_unary:NNNwN`

Here, `#1` is a precedence, `#2` is some extra data used by some functions, `#3` is *e.g.*, `\__fp_sin_o:w`, and expands once after the calculation, `#4` is the operand, and `#5` is a `\__fp_parse_infix_...:N` function. We feed the data `#2`, and the argument `#4`, to the function `#3`, which expands `\exp:w` thus the `infix` function `#5`.

This is redefined in l3fp-extras.

```
\cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4@#5
 {
  #3 #2 #4 @
  \exp:w \exp_end_continue_f:w #5 #1
 }
```

(*End definition for* `\__fp_parse_apply_unary:NNNwN`.)

__fp_parse_prefix_-:Nw
__fp_parse_prefix_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c__fp_prec_not_int of the unary operator, then call the appropriate __fp_⟨*operation*⟩_o:w function, where the ⟨*operation*⟩ is set_sign or not.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
      {
        \exp_after:wN \__fp_parse_apply_unary:NNNwN
        \exp_after:wN ##1
        \exp_after:wN #4
        \exp_after:wN #3
        \exp:w
        \if_int_compare:w #2 < ##1
          \__fp_parse_operand:Nw ##1
        \else:
          \__fp_parse_operand:Nw #2
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
\__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?
```

(*End definition for __fp_parse_prefix_-:Nw and __fp_parse_prefix_!:Nw.*)

__fp_parse_prefix_.:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to __fp_parse_one_digit:NN but calls __fp_parse_strim_zeros:N to trim zeros after the decimal point, rather than the trim_zeros function for zeros before the decimal point.

```
\cs_new:cpn { __fp_parse_prefix_.:Nw } #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \exp_after:wN \__fp_sanitize:wN
      \__int_value:w \__int_eval:w 0 \__fp_parse_strim_zeros:N
  }
```

(*End definition for __fp_parse_prefix_.:Nw.*)

__fp_parse_prefix_(:Nw
__fp_parse_lparen_after:NwN

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. Commas are allowed if the previous precedence is 16 (function with multiple arguments). In this case, find an operand using the precedence 1; otherwise the precedence 0. Once the operand is found, the lparen_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream the array it found as an operand, fetching the following infix operator.

```
\cs_new:cpn { __fp_parse_prefix_(:Nw } #1
  {
    \exp_after:wN \__fp_parse_lparen_after:NwN
    \exp_after:wN #1
    \exp:w
    \if_int_compare:w #1 = \c__fp_prec_funcii_int
      \__fp_parse_operand:Nw \c__fp_prec_comma_int
    \else:
      \__fp_parse_operand:Nw \c__fp_prec_paren_int
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
  {
    \exp_not:N \token_if_eq_meaning:NNTF #3
      \exp_not:c { __fp_parse_infix_):N }
      {
        \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_stop
        \exp_not:N \exp_after:wN
        \exp_not:N \__fp_parse_infix:NN
        \exp_not:N \exp_after:wN #1
        \exp_not:N \exp:w
        \exp_not:N \__fp_parse_expand:w
      }
      {
        \exp_not:N \__msg_kernel_expandable_error:nnn
          { kernel } { fp-missing } { ) }
        #2 @
        \exp_not:N \use_none:n #3
      }
  }
```

(*End definition for* `\__fp_parse_prefix_(:Nw` *and* `\__fp_parse_lparen_after:NwN`.)

`\__fp_parse_prefix_):Nw`

The right parenthesis can appear as unary prefixes when arguments of a multi-argument function end with a comma, or when there is no argument, as in `max(1,2,)` or in `rand()`. In single-argument functions (precedence 0 rather than 1) forbid this.

```
\cs_new:cpn { __fp_parse_prefix_):Nw } #1
  {
    \if_int_compare:w #1 = \c__fp_prec_comma_int
    \else:
      \__msg_kernel_expandable_error:nnn
        { kernel } { fp-missing-number } { ) }
      \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
    \fi:
    \__fp_parse_infix:NN #1 )
  }
```

(*End definition for* `\__fp_parse_prefix_):Nw`.)

24.5.2 Constants

`\__fp_parse_word_inf:N`
`\__fp_parse_word_nan:N`
`\__fp_parse_word_pi:N`
`\__fp_parse_word_deg:N`
`\__fp_parse_word_true:N`
`\__fp_parse_word_false:N`

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
  }
```

```
\__fp_tmp:w { inf } \c_inf_fp
\__fp_tmp:w { nan } \c_nan_fp
\__fp_tmp:w { pi  } \c_pi_fp
\__fp_tmp:w { deg } \c_one_degree_fp
\__fp_tmp:w { true } \c_one_fp
\__fp_tmp:w { false } \c_zero_fp
```

(*End definition for* `\__fp_parse_word_inf:N` *and others.*)

`\__fp_parse_caseless_inf:N`
`\__fp_parse_caseless_infinity:N`
`\__fp_parse_caseless_nan:N`

Copies of `\__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```
\cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
\cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
\cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N
```

(*End definition for* `\__fp_parse_caseless_inf:N`*,* `\__fp_parse_caseless_infinity:N`*, and* `\__fp_parse_caseless_nan:N`*.*)

`\__fp_parse_word_pt:N`
`\__fp_parse_word_in:N`
`\__fp_parse_word_pc:N`
`\__fp_parse_word_cm:N`
`\__fp_parse_word_mm:N`
`\__fp_parse_word_dd:N`
`\__fp_parse_word_cc:N`
`\__fp_parse_word_nd:N`
`\__fp_parse_word_nc:N`
`\__fp_parse_word_bp:N`
`\__fp_parse_word_sp:N`

Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      {
        \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
        \s__fp \__fp_chk:w 10 #2 ;
      }
  }
\__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
\__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
\__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
\__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
\__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
\__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
\__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
\__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
\__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }
```

(*End definition for* `\__fp_parse_word_pt:N` *and others.*)

`\__fp_parse_word_em:N`
`\__fp_parse_word_ex:N`

The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```
\tl_map_inline:nn { {em} {ex} }
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      {
        \exp_after:wN \__fp_from_dim_test:ww
        \exp_after:wN 0 \exp_after:wN ,
        \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
        \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
      }
  }
```

(*End definition for* `\__fp_parse_word_em:N` *and* `\__fp_parse_word_ex:N`*.*)

24.5.3 Functions

__fp_parse_unary_function:NNN
__fp_parse_function:NNN

```
\cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_unary:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \exp_after:wN #1
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_function:NNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_unary:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \exp_after:wN #1
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_unary_function:NNN` *and* `\__fp_parse_function:NNN`.)

24.6 Main functions

__fp_parse:n
__fp_parse_after:ww

Start an `\exp:w` expansion so that `\__fp_parse:n` expands in two steps. The `\__fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c__fp_prec_end_int` or less, namely, the end of the expression. The marker `\s__fp_mark` indicates that the next token is an already parsed version of an infix operator, and `\__fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:`.

```
\cs_new:Npn \__fp_parse:n #1
  {
    \exp:w
      \exp_after:wN \__fp_parse_after:ww
      \exp:w
        \__fp_parse_operand:Nw \c__fp_prec_end_int
        \__fp_parse_expand:w #1
        \s__fp_mark \__fp_parse_infix_end:N
      \s__fp_stop
  }
\cs_new:Npn \__fp_parse_after:ww
    #1@ \__fp_parse_infix_end:N \s__fp_stop
  { \exp_end: #1 }
```

(*End definition for* `\__fp_parse:n` *and* `\__fp_parse_after:ww`.)

__fp_parse_o:n

```
\cs_new:Npn \__fp_parse_o:n #1
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_exp_after_o:w
      \__fp_parse:n {#1}
  }
```

(*End definition for* `\__fp_parse_o:n`*.*)

__fp_parse_operand:Nw
__fp_parse_continue:NwN

This is just a shorthand which sets up both `\__fp_parse_continue:NwN` and `\__fp_parse_one:Nw` with the same precedence. Note the trailing `\exp:w`.

```
\cs_new:Npn \__fp_parse_operand:Nw #1
  {
    \exp_end_continue_f:w
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \__fp_parse_one:Nw
    \exp_after:wN #1
    \exp:w
  }
\cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
```

(*End definition for* `\__fp_parse_operand:Nw` *and* `\__fp_parse_continue:NwN`*.*)

__fp_parse_apply_binary:NwNwN

Receives ⟨*precedence*⟩ ⟨*operand*$_1$⟩ `@` ⟨*operation*⟩ ⟨*operand*$_2$⟩ `@` ⟨*infix command*⟩. Builds the appropriate call to the ⟨*operation*⟩ `#3`.

This is redefined in l3fp-extras.

```
\cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w \cs:w __fp_#3_o:ww \cs_end: #2 #4
    \exp:w \exp_end_continue_f:w #5 #1
  }
```

(*End definition for* `\__fp_parse_apply_binary:NwNwN`*.*)

24.7 Infix operators

__fp_parse_infix_after_operand:NwN

```
\cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
  {
    \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
    #2;
  }
  \cs_new:Npn \__fp_parse_infix:NN #1 #2
    {
      \if_catcode:w \scan_stop: \exp_not:N #2
        \if_int_compare:w
            \__str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }
            = 0 \exp_stop_f:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_mark:NNN
        \else:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_juxtapose:N
        \fi:
      \else:
        \if_int_compare:w
            \__int_eval:w
              ( `#2 \if_int_compare:w `#2 > `Z - 32 \fi: ) / 26
            = 3 \exp_stop_f:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_juxtapose:N
        \else:
          \exp_after:wN \__fp_parse_infix_check:NNN
          \cs:w
            __fp_parse_infix_ \token_to_str:N #2 :N
            \exp_after:wN \exp_after:wN \exp_after:wN
          \cs_end:
        \fi:
      \fi:
      #1
      #2
    }
\cs_new:Npx \__fp_parse_infix_check:NNN #1#2#3
  {
    \exp_not:N \if_meaning:w \scan_stop: #1
      \exp_not:N \__msg_kernel_expandable_error:nnn
        { kernel } { fp-missing } { * }
      \exp_not:N \exp_after:wN
      \exp_not:c { __fp_parse_infix_*:N }
      \exp_not:N \exp_after:wN #2
      \exp_not:N \exp_after:wN #3
    \exp_not:N \else:
      \exp_not:N \exp_after:wN #1
      \exp_not:N \exp_after:wN #2
      \exp_not:N \exp:w
      \exp_not:N \exp_after:wN
      \exp_not:N \__fp_parse_expand:w
    \exp_not:N \fi:
  }
```

(*End definition for* `\__fp_parse_infix_after_operand:NwN`.)

24.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN

As an infix operator, `\s__fp_mark` means that the next token (`#3`) has already gone through `\__fp_parse_infix:NN` and should be provided the precedence `#1`. The scan mark `#2` is discarded.

```
\cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

(*End definition for* `\__fp_parse_infix_mark:NNN`.)

__fp_parse_infix_end:N

This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
\cs_new:Npn \__fp_parse_infix_end:N #1
  { @ \use_none:n \__fp_parse_infix_end:N }
```

(*End definition for* `\__fp_parse_infix_end:N`.)

__fp_parse_infix_):N

This is very similar to `\__fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression.

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 < \c__fp_prec_paren_int
          \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
          \exp_after:wN \__fp_parse_infix:NN
          \exp_after:wN ##1
          \exp:w \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_):N }
```

(*End definition for* `\__fp_parse_infix_):N`*.*)

__fp_parse_infix_,:N
__fp_parse_infix_comma:w
__fp_parse_infix_comma_error:w
__fp_,_o:ww

`\__fp_,_o:ww` is a complicated way of replacing any number of floating point arguments by `nan`.

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 > \c__fp_prec_comma_int
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \else:
          \if_int_compare:w ##1 < \c__fp_prec_comma_int
            \__fp_parse_infix_comma_error:w
          \fi:
          \exp_after:wN \__fp_parse_infix_comma:w
          \exp:w \__fp_parse_operand:Nw \c__fp_prec_comma_int
          \exp_after:wN \__fp_parse_expand:w
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_,:N }
\cs_new:Npn \__fp_parse_infix_comma:w #1 @
  { #1 @ \use_none:n }
\cs_new:Npn \__fp_parse_infix_comma_error:w #1 \exp:w
  {
    \fi:
    \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
    \exp_after:wN @
    \exp_after:wN \__fp_parse_apply_binary:NwNwN
    \exp_after:wN ,
    \exp:w
  }
```

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:Npn #1 ##1
      {
        \if_meaning:w \s__fp ##1
          \exp_after:wN \__fp_use_i_until_s:nw
          \exp_after:wN #1
        \fi:
        \exp_after:wN \c_nan_fp
        ##1
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_,_o:ww }
```

(*End definition for* `\__fp_parse_infix_,:N` *and others.*)

24.7.2 Usual infix operators

__fp_parse_infix_+:N
__fp_parse_infix_-:N
__fp_parse_infix_/:N
__fp_parse_infix_mul:N
__fp_parse_infix_and:N
__fp_parse_infix_or:N
__fp_parse_infix_^:N

As described in the "work plan", each infix operator has an associated `\..._infix_...` function, a computing function, and precedence, given as arguments to `\__fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 < #3
          \exp_after:wN @
          \exp_after:wN \__fp_parse_apply_binary:NwNwN
          \exp_after:wN #2
          \exp:w
          \__fp_parse_operand:Nw #4
          \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N }   ^
  \c__fp_prec_hatii_int \c__fp_prec_hat_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:N }   /
  \c__fp_prec_times_int \c__fp_prec_times_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
  \c__fp_prec_times_int \c__fp_prec_times_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_-:N }   -
  \c__fp_prec_plus_int  \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_+:N }   +
  \c__fp_prec_plus_int  \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
  \c__fp_prec_and_int   \c__fp_prec_and_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N }  |
  \c__fp_prec_or_int    \c__fp_prec_or_int
```

(*End definition for* `\__fp_parse_infix_+:N` *and others.*)

24.7.3 Juxtaposition

`\__fp_parse_infix_(:N`

When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using `\__fp_parse_infix_juxtapose:N`.

```
\cs_new:cpn { __fp_parse_infix_(:N } #1
  { \__fp_parse_infix_juxtapose:N #1 ( }
```

(*End definition for* `\__fp_parse_infix_(:N`.)

`\__fp_parse_infix_juxtapose:N`
`\__fp_parse_apply_juxtapose:NwwN`

Juxtaposition follows the same scheme as other binary operations, but calls `\__fp_parse_apply_juxtapose:NwwN` rather than directly calling `\__fp_parse_apply_binary:NwNwN`. This lets us catch errors such as `...(1,2,3)pt` where one operand of the juxtaposition is not a single number: both `#3` and `#5` of the `apply` auxiliary must be empty.

```
\cs_new:Npn \__fp_parse_infix_juxtapose:N #1
  {
    \if_int_compare:w #1 < \c__fp_prec_times_int
      \exp_after:wN @
      \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
      \exp:w
      \__fp_parse_operand:Nw \c__fp_prec_times_int
      \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \__fp_parse_infix_juxtapose:N
    \fi:
  }
\cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
  {
    \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
    \else:
      \__fp_error:nffn { fp-invalid-ii }
        { \__fp_array_to_clist:n { #2; #3 } }
        { \__fp_array_to_clist:n { #4; #5 } }
        { }
    \fi:
    \__fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
  }
```

(*End definition for* `\__fp_parse_infix_juxtapose:N` *and* `\__fp_parse_apply_juxtapose:NwwN`.)

24.7.4 Multi-character cases

`\__fp_parse_infix_*:N`

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
      {
        \if:w * \exp_not:N ##2
```

```
          \exp_after:wN #1
          \exp_after:wN ##1
        \else:
          \exp_after:wN \__fp_parse_infix_mul:N
          \exp_after:wN ##1
          \exp_after:wN ##2
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N }
```

(*End definition for* `\__fp_parse_infix_*:N`.)

__fp_parse_infix_|:Nw
__fp_parse_infix_&:Nw

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3
  {
    \cs_new:Npn #1 ##1##2
      {
        \if:w #2 \exp_not:N ##2
          \exp_after:wN #1
          \exp_after:wN ##1
          \exp:w \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN #3
          \exp_after:wN ##1
          \exp_after:wN ##2
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_|:N } | \__fp_parse_infix_or:N
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_&:N } & \__fp_parse_infix_and:N
```

(*End definition for* `\__fp_parse_infix_|:Nw` *and* `\__fp_parse_infix_&:Nw`.)

24.7.5 Ternary operator

__fp_parse_infix_?:N
__fp_parse_infix_::N

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 < \c__fp_prec_quest_int
          #4
          \exp_after:wN @
          \exp_after:wN #2
          \exp:w
          \__fp_parse_operand:Nw #3
          \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
  \__fp_ternary:NwwN \c__fp_prec_quest_int { }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_::N }
  \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
  {
    \__msg_kernel_expandable_error:nnnn
      { kernel } { fp-missing } { ? } { ~for~?: }
  }
```

(*End definition for* `\__fp_parse_infix_?:N` *and* `\__fp_parse_infix_::N`*.*)

24.7.6 Comparisons

__fp_parse_infix_<:N
__fp_parse_infix_=:N
__fp_parse_infix_>:N
__fp_parse_infix_!:N
__fp_parse_excl_error:
__fp_parse_compare:NNNNNNN
__fp_parse_compare_auxi:NNNNNNN
__fp_parse_compare_auxii:NNNNN
__fp_parse_compare_end:NNNNw
__fp_compare:wNNNNw

```
\cs_new:cpn { __fp_parse_infix_<:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
\cs_new:cpn { __fp_parse_infix_=:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
\cs_new:cpn { __fp_parse_infix_>:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
\cs_new:cpn { __fp_parse_infix_!:N } #1
  {
    \exp_after:wN \__fp_parse_compare:NNNNNNN
    \exp_after:wN #1
    \exp_after:wN 0
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
  }
\cs_new:Npn \__fp_parse_excl_error:
  {
    \__msg_kernel_expandable_error:nnnn
      { kernel } { fp-missing } { = } { ~after~!. }
  }
\cs_new:Npn \__fp_parse_compare:NNNNNNN #1
  {
    \if_int_compare:w #1 < \c__fp_prec_comp_int
      \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
      \exp_after:wN \__fp_parse_excl_error:
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \__fp_parse_compare:NNNNNNN
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
  {
    \if_case:w
      \__int_eval:w \exp_after:wN ` \token_to_str:N #7 - `< \__int_eval_end:
         \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
    \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
  {
    \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
    \exp_after:wN \prg_do_nothing:
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN #5
    \exp:w \exp_after:wN \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
  {
    \fi:
    \exp_after:wN @
    \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
    \exp_after:wN \c_one_fp
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
  }
\cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
    #1 #2@ #3 #4#5#6#7 #8@ #9
  {
    \if_int_odd:w
        \if_meaning:w \c_zero_fp #3
          0
        \else:
          \if_case:w \__fp_compare_back:ww #8 #2 \exp_stop_f:
            #5 \or: #6 \or: #7 \else: #4
          \fi:
        \fi:
        \exp_stop_f:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
      \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
      \exp_after:wN \c_zero_fp
    \fi:
    #1 #8 #9
  }
\cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
  {
    \if_meaning:w \__fp_parse_compare:NNNNNNN #4
      \exp_after:wN \__fp_parse_continue_compare:NNwNN
      \exp_after:wN #1
      \exp_after:wN #2
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_o:w #3;
      \exp:w \exp_end_continue_f:w
    \else:
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #2
      \exp:w \exp_end_continue_f:w
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
    \fi:
    #4 #2
  }
\cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
  { #4 #2 #3@ #1 }
```

(*End definition for `\__fp_parse_infix_<:N` and others.*)

24.8 Candidate: defining new l3fp functions

`\fp_function:Nw`

Parse the argument of the function #1 using `\__fp_parse_operand:Nw` with a precedence of 16, and pass the function and argument to `\__fp_function_apply:nw`.

```
\cs_new:Npn \fp_function:Nw #1
  {
    \exp_after:wN \__fp_function_apply:nw
    \exp_after:wN #1
    \exp:w
      \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
  }
```

(*End definition for `\fp_function:Nw`.*)

`\fp_new_function:Npn`
`\__fp_new_function:NNnnn`
`\__fp_new_function:Ncfnn`
`\__fp_function_args:Nwn`

Save the code provided by the user in the control sequence `\__fp_user_#1`. Define #1 to call `\__fp_function_apply:nw` after parsing one operand using `\__fp_parse_operand:Nw` with precedence 16. The auxiliary `\__fp_function_args:Nwn` receives the user function and the number of arguments (half of the number of tokens in the parameter text #2), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```
\cs_new_protected:Npn \fp_new_function:Npn #1#2#
  {
    \__fp_new_function:Ncfnn #1
      { __fp_user_ \cs_to_str:N #1 }
      { \int_eval:n { \tl_count:n {#2} / 2 } }
      {#2}
  }
\cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
  {
    \cs_new:Npn #1
      {
        \exp_after:wN \__fp_function_apply:nw \exp_after:wN
          {
            \exp_after:wN \__fp_function_args:Nwn
            \exp_after:wN #2
            \__int_value:w #3 \exp_after:wN ; \exp_after:wN
          }
        \exp:w
          \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
      }
    \cs_new:Npn #2 #4 {#5}
  }
\cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
\cs_new:Npn \__fp_function_args:Nwn #1#2; #3
  {
    \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
      { #1 #3 }
      {
        \__msg_kernel_expandable_error:nnnnn
          { kernel } { fp-num-args } { #1() } {#2} {#2}
        \c_nan_fp
      }
  }
```

(*End definition for* `\fp_new_function:Npn`*,* `\__fp_new_function:NNnnn`*, and* `\__fp_function_args:Nwn`*.*)

`\__fp_function_apply:nw`
`\__fp_function_store:wwNwnn`
`\__fp_function_store_end:wnnn`

The auxiliary `\__fp_function_apply:nw` is called after parsing an operand, so it receives some code `#1`, then the operand ending with `@`, then a function such as `\__fp_parse_infix_+:N` (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of `\__fp_function_store:wwNwnn` and `\__fp_function_store_end:wnnn`. Then apply `\__fp_parse:n` to the code `#1` followed by a brace group with this token list. This results in a floating point result, which is then correctly parsed as the next operand of whatever was looking for one. The trailing `\s__fp_mark` is used as a special infix operator to indicate that the next token has already gone through `\__fp_parse_infix:NN`.

```
\cs_new:Npn \__fp_function_apply:nw #1#2 @
  {
    \__fp_parse:n
      {
        \__fp_function_store:wwNwnn #2
          \s__fp_mark \__fp_function_store:wwNwnn ;
          \s__fp_mark \__fp_function_store_end:wnnn
        \s__fp_stop { } { } {#1}
      }
    \s__fp_mark
  }
\cs_new:Npn \__fp_function_store:wwNwnn
    #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
  { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
\cs_new:Npn \__fp_function_store_end:wnnn
    #1 \s__fp_stop #2#3#4
  { #4 {#2} }
```

(*End definition for* `\__fp_function_apply:nw`*,* `\__fp_function_store:wwNwnn`*, and* `\__fp_function_store_end:wnnn`*.*)

24.9 Messages

```
\__msg_kernel_new:nnn { kernel } { fp-deprecated }
  { ’#1’~deprecated;~use~’#2’ }
```

```
\__msg_kernel_new:nnn { kernel } { unknown-fp-word }
  { Unknown~fp~word~#1. }
\__msg_kernel_new:nnn { kernel } { fp-missing }
  { Missing~#1~inserted #2. }
\__msg_kernel_new:nnn { kernel } { fp-extra }
  { Extra~#1~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-early-end }
  { Premature~end~in~fp~expression. }
\__msg_kernel_new:nnn { kernel } { fp-after-e }
  { Cannot~use~#1 after~'e'. }
\__msg_kernel_new:nnn { kernel } { fp-missing-number }
  { Missing~number~before~'#1'. }
\__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
  { Unknown~symbol~#1~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-extra-comma }
  { Unexpected~comma:~extra~arguments~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-num-args }
  { #1~expects~between~#2~and~#3~arguments. }
⟨*package⟩
\cs_if_exist:cT { @unexpandable@protect }
  {
    \__msg_kernel_new:nnn { kernel } { fp-robust-cmd }
      { Robust~command~#1 invalid~in~fp~expression! }
  }
⟨/package⟩
```

```
⟨/initex | package⟩
```

25 l3fp-logic Implementation

```
⟨*initex | package⟩
```

```
⟨@@=fp⟩
```

__fp_parse_word_max:N
__fp_parse_word_min:N

Those functions may receive a variable number of arguments.

```
\cs_new:Npn \__fp_parse_word_max:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
\cs_new:Npn \__fp_parse_word_min:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
```

(*End definition for* __fp_parse_word_max:N *and* __fp_parse_word_min:N.)

25.1 Syntax of internal functions

- __fp_compare_npos:nwnw {⟨*expo*$_1$⟩} ⟨*body*$_1$⟩ ; {⟨*expo*$_2$⟩} ⟨*body*$_2$⟩ ;
- __fp_minmax_o:Nw ⟨*sign*⟩ ⟨*floating point array*⟩
- __fp_not_o:w ? ⟨*floating point array*⟩ (with one floating point number only)
- __fp_&_o:ww ⟨*floating point*⟩ ⟨*floating point*⟩
- __fp_|_o:ww ⟨*floating point*⟩ ⟨*floating point*⟩
- __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, __fp_ternary_auxii:NwwN have to be understood.

25.2 Existence test

\fp_if_exist_p:N
\fp_if_exist_p:c
\fp_if_exist:NTF
\fp_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* \fp_if_exist:NTF. *This function is documented on page 180.*)

25.3 Comparison

\fp_compare_p:n
\fp_compare:nTF
__fp_compare_return:w

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with 0.

```
\prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__fp_compare_return:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
  }
\cs_new:Npn \__fp_compare_return:w \s__fp \__fp_chk:w #1#2;
  {
    \if_meaning:w 0 #1
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* \fp_compare:nTF *and* __fp_compare_return:w. *These functions are documented on page 181.*)

\fp_compare_p:nNn
\fp_compare:nNnTF
__fp_compare_aux:wn

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to __fp_compare_back:ww, defined below. Compare the result with `#2-`=, which is −1 for <, 0 for =, 1 for > and 2 for ?.

```
\prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w
        \exp_after:wN \__fp_compare_aux:wn
          \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
        = \__int_eval:w `#2 - `= \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__fp_compare_aux:wn #1; #2
  {
    \exp_after:wN \__fp_compare_back:ww
      \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
  }
```

(*End definition for* \fp_compare:nNnTF *and* __fp_compare_aux:wn. *These functions are documented on page 181.*)

__fp_compare_back:ww
__fp_compare_nan:w

__fp_compare_back:ww ⟨*y*⟩ ; ⟨*x*⟩ ;

Expands (in the same way as \int_eval:n) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is nan, stop the comparison with __fp_compare_nan:w returning 2. If x is negative, swap the outputs 1 and -1 (*i.e.*, $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with __fp_compare_npos:nwnw, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```
\cs_new:Npn \__fp_compare_back:ww
    \s__fp \__fp_chk:w #1 #2 #3;
    \s__fp \__fp_chk:w #4 #5 #6;
  {
    \__int_value:w
      \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 2 #5 - \fi:
      \if_meaning:w #2 #5
        \if_meaning:w #1 #4
          \if_meaning:w 1 #1
            \__fp_compare_npos:nwnw #6; #3;
          \else:
            0
          \fi:
        \else:
          \if_int_compare:w #4 < #1 - \fi: 1
        \fi:
      \else:
        \if_int_compare:w #1#4 = 0 \exp_stop_f:
          0
        \else:
          1
        \fi:
      \fi:
    \exp_stop_f:
  }
\cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }
```

(*End definition for* __fp_compare_back:ww *and* __fp_compare_nan:w*.*)

__fp_compare_npos:nwnw
__fp_compare_significand:nnnnnnnn

__fp_compare_npos:nwnw {⟨*expo*$_1$⟩} ⟨*body*$_1$⟩ ; {⟨*expo*$_2$⟩} ⟨*body*$_2$⟩ ;

Within an __int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```
\cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
  {
    \if_int_compare:w #1 = #3 \exp_stop_f:
      \__fp_compare_significand:nnnnnnnn #2 #4
    \else:
      \if_int_compare:w #1 < #3 - \fi: 1
    \fi:
  }
\cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
      \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
        0
      \else:
        \if_int_compare:w #3#4 < #7#8 - \fi: 1
      \fi:
    \else:
      \if_int_compare:w #1#2 < #5#6 - \fi: 1
    \fi:
  }
```

(*End definition for* `\__fp_compare_npos:nwnw` *and* `\__fp_compare_significand:nnnnnnnn`*.*)

25.4 Floating point expression loops

\fp_do_until:nn
\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn

These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```
\cs_new:Npn \fp_do_until:nn #1#2
  {
    #2
    \fp_compare:nF {#1}
      { \fp_do_until:nn {#1} {#2} }
  }
\cs_new:Npn \fp_do_while:nn #1#2
  {
    #2
    \fp_compare:nT {#1}
      { \fp_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \fp_until_do:nn #1#2
  {
    \fp_compare:nF {#1}
      {
        #2
        \fp_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \fp_while_do:nn #1#2
  {
    \fp_compare:nT {#1}
      {
        #2
        \fp_while_do:nn {#1} {#2}
      }
  }
```

(*End definition for* `\fp_do_until:nn` *and others. These functions are documented on page 182.*)

\fp_do_until:nNnn
\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn

As above but not using the `nNn` syntax.

```
\cs_new:Npn \fp_do_until:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnF {#1} #2 {#3}
      { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_do_while:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnT {#1} #2 {#3}
      { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_until_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnF {#1} #2 {#3}
      {
        #4
        \fp_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \fp_while_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnT {#1} #2 {#3}
      {
        #4
        \fp_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
```

(*End definition for* `\fp_do_until:nNnn` *and others. These functions are documented on page 182.*)

\fp_step_function:nnnN
\fp_step_function:nnnc
__fp_step:wwwN
__fp_step:NnnnnN
__fp_step:NfnnnN

The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `\__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```
\cs_new:Npn \fp_step_function:nnnN #1#2#3
  {
    \exp_after:wN \__fp_step:wwwN
      \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
      \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
      \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
  }
\cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
%       \end{macrocode}
%   Only \enquote{normal} floating points (not $\pm 0$,
%   $\pm\texttt{inf}$, \texttt{nan}) can be used as step; if positive,
%   call \cs{__fp_step:NnnnnN} with argument |>| otherwise~|<|.  This
%   function has one more argument than its integer counterpart, namely
%   the previous value, to catch the case where the loop has made no
%   progress.  Conversion to decimal is done just before calling the
%   user's function.
%    \begin{macrocode}
\cs_new:Npn \__fp_step:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
  {
    \token_if_eq_meaning:NNTF #2 1
      {
        \token_if_eq_meaning:NNTF #3 0
          { \__fp_step:NnnnnN > }
          { \__fp_step:NnnnnN < }
      }
      {
        \token_if_eq_meaning:NNTF #2 0
          { \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#6} }
          {
            \__fp_error:nnfn { fp-bad-step } { }
              { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
          }
        \use_none:nnnnn
      }
      { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
  }
\cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
  {
    \fp_compare:nNnTF {#2} = {#3}
      {
        \__fp_error:nffn { fp-tiny-step }
          { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
      }
      {
        \fp_compare:nNnF {#2} #1 {#5}
          {
            \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
            \__fp_step:NfnnnN
              #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
          }
      }
  }
\cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }
```

(*End definition for* \fp_step_function:nnnN*,* __fp_step:wwwN*, and* __fp_step:NnnnnN*. These functions are documented on page 183.*)

\fp_step_inline:nnnn
\fp_step_variable:nnnNn
__fp_step:NNnnnn

As for \int_step_inline:nnnn, create a global function and apply it, following up with a break point.

```
\cs_new_protected:Npn \fp_step_inline:nnnn
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__fp_step:NNnnnn
      \cs_gset_protected:Npn
      { __prg_map_ \int_use:N \g__prg_map_int :w }
  }
\cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__fp_step:NNnnnn
      \cs_gset_protected:Npx
      { __prg_map_ \int_use:N \g__prg_map_int :w }
      {#1} {#2} {#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \fp_step_function:nnnN {#3} {#4} {#5} #2
    \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
  }
```

(*End definition for* `\fp_step_inline:nnnn`*,* `\fp_step_variable:nnnNn`*, and* `\__fp_step:NNnnnn`*. These functions are documented on page 183.*)

```
\__msg_kernel_new:nnn { kernel } { fp-bad-step }
  { Invalid~step~size~#2~in~step~function~#3. }
\__msg_kernel_new:nnn { kernel } { fp-tiny-step }
  { Tiny~step~size~(#1+#2=#1)~in~step~function~#3. }
```

25.5 Extrema

`\__fp_minmax_o:Nw`

The argument `#1` is 2 to find the maximum of an array `#2` of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ±0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `\__fp_minmax_loop:Nww`.

```
\cs_new:Npn \__fp_minmax_o:Nw #1#2 @
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
    \else:
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
    \fi:
    #2
    \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
    \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
  }
```

(*End definition for* `\__fp_minmax_o:Nw`*.*)

`\__fp_minmax_loop:Nww`

The first argument is $-$ or $+$ to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```
\cs_new:Npn \__fp_minmax_loop:Nww
    #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
  {
    \if_meaning:w 3 #4
      \if_meaning:w 3 #2
        \__fp_minmax_auxi:ww
      \else:
        \__fp_minmax_auxii:ww
      \fi:
    \else:
      \if_int_compare:w
          \__fp_compare_back:ww
            \s__fp \__fp_chk:w #4#5;
            \s__fp \__fp_chk:w #2#3;
          = #1 1 \exp_stop_f:
        \__fp_minmax_auxii:ww
      \else:
        \__fp_minmax_auxi:ww
      \fi:
    \fi:
    \__fp_minmax_loop:Nww #1
      \s__fp \__fp_chk:w #2#3;
      \s__fp \__fp_chk:w #4#5;
  }
```

(*End definition for* `\__fp_minmax_loop:Nww`.)

__fp_minmax_auxi:ww
__fp_minmax_auxii:ww

Keep the first/second number, and remove the other.

```
\cs_new:Npn \__fp_minmax_auxi:ww  #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
  { \fi: \fi: #2 \s__fp #3 ; }
\cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
  { \fi: \fi: #2 }
```

(*End definition for* `\__fp_minmax_auxi:ww` *and* `\__fp_minmax_auxii:ww`.)

__fp_minmax_break_o:w

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
\cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
  { \fi: \__fp_exp_after_o:w \s__fp #3; }
```

(*End definition for* `\__fp_minmax_break_o:w`.)

25.6 Boolean operations

__fp_not_o:w

Return `true` or `false`, with two expansions, one to exit the conditional, and one to please l3fp-parse. The first argument is provided by l3fp-parse and is ignored.

```
\cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_meaning:w 0 #2
      \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
    \fi:
  }
```

(*End definition for* `\__fp_not_o:w`.)

__fp_&_o:ww
__fp_|_o:ww
__fp_and_return:wNw

For and, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For or, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking __fp_&_o:ww, inserting an extra argument, \else:, before \s__fp. In all cases, expand after the floating point number.

```
\group_begin:
  \char_set_catcode_letter:N &
  \char_set_catcode_letter:N |
  \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
    {
      \if_meaning:w 0 #2 #1
        \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
      \fi:
      \__fp_exp_after_o:w
    }
  \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
\group_end:
\cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
```

(*End definition for __fp_&_o:ww, __fp_|_o:ww, and __fp_and_return:wNw.*)

25.7 Ternary operator

__fp_ternary:NwwN
__fp_ternary_auxi:NwwN
__fp_ternary_auxii:NwwN
__fp_ternary_loop_break:w
__fp_ternary_loop:Nw
__fp_ternary_map_break:
__fp_ternary_break_point:n

The first function receives the test and the true branch of the ?: ternary operator. It returns the true branch, unless the test branch is zero. In that case, the function returns a very specific nan. The second function receives the output of the first function, and the false branch. It returns the previous input, unless that is the special nan, in which case we return the false branch.

```
\cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
  {
    \if_meaning:w \__fp_parse_infix_::N #4
      \__fp_ternary_loop:Nw
        #2
        \s__fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
      \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_array_f:w #3 \s__fp_stop
      \exp_after:wN @
      \exp:w
        \__fp_parse_operand:Nw \c__fp_prec_colon_int
        \__fp_parse_expand:w
    \else:
      \__msg_kernel_expandable_error:nnnn
        { kernel } { fp-missing } { : } { ~for~?: }
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_array_f:w #3 \s__fp_stop
      \exp_after:wN #4
      \exp_after:wN #1
    \fi:
  }
```

```
\cs_new:Npn \__fp_ternary_loop_break:w
    #1 \fi: #2 \__fp_ternary_break_point:n #3
  {
    0 = 0 \exp_stop_f: \fi:
    \exp_after:wN \__fp_ternary_auxii:NwwN
  }
\cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
  {
    \if_int_compare:w #1 > 0 \exp_stop_f:
      \exp_after:wN \__fp_ternary_map_break:
    \fi:
    \__fp_ternary_loop:Nw
  }
\cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
\cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \__fp_exp_after_array_f:w #2 \s__fp_stop
    #4 #1
  }
\cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \__fp_exp_after_array_f:w #3 \s__fp_stop
    #4 #1
  }
```

(*End definition for `\__fp_ternary:NwwN` and others.*)

⟨/initex | package⟩

26 l3fp-basics Implementation

⟨*initex | package⟩

⟨@@=fp⟩

The l3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at `http://cr.yp.to/2005-590/goldberg.pdf`.

`\__fp_parse_word_abs:N`
`\__fp_parse_word_sign:N`
`\__fp_parse_word_sqrt:N`

Unary functions.

```
\cs_new:Npn \__fp_parse_word_abs:N
  { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
\cs_new:Npn \__fp_parse_word_sign:N
  { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
\cs_new:Npn \__fp_parse_word_sqrt:N
  { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(*End definition for* `\__fp_parse_word_abs:N`*,* `\__fp_parse_word_sign:N`*, and* `\__fp_parse_word_sqrt:N`*.*)

26.1 Addition and subtraction

We define here two functions, `\__fp_-_o:ww` and `\__fp_+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `\__fp_add_big_i_o:wNww`, is used in l3fp-expo.

The logic goes as follows:

- `\__fp_-_o:ww` calls `\__fp_+_o:ww` to do the work, with the sign of the second operand flipped;
- `\__fp_+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;
- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `\__fp_basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

26.1.1 Sign, exponent, and special numbers

`\__fp_-_o:ww` The `\__fp_+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `\__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `\__fp_+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```
\cs_new:cpx { __fp_-_o:ww } \s__fp
  {
    \exp_not:c { __fp_+_o:ww }
    \exp_not:n { \s__fp \__fp_neg_sign:N }
  }
```

(*End definition for* `\__fp_-_o:ww`*.*)

`\__fp_+_o:ww` This function is either called directly with an empty `#1` to compute an addition, or it is called by `\__fp_-_o:ww` with `\__fp_neg_sign:N` as `#1` to compute a subtraction, in which case the second operand's sign should be changed. If the ⟨*types*⟩ `#2` and `#4` are the same, dispatch to case `#2` (0, 1, 2, or 3), where we call specialized functions: thanks to `\__int_value:w`, those receive the tweaked ⟨$sign_2$⟩ (expansion of `#1#5`) as an argument. If the ⟨*types*⟩ are distinct, the result is simply the floating point number with the highest ⟨*type*⟩. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when ⟨*type*$_1$⟩ is greater than ⟨*type*$_2$⟩. Also note that we don't need to worry about ⟨*sign*$_2$⟩ in that case since the second operand is discarded.

```
\cs_new:cpn { __fp_+_o:ww }
    \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
  {
    \if_case:w
      \if_meaning:w #2 #4
        #2
      \else:
        \if_int_compare:w #2 > #4 \exp_stop_f:
          3
        \else:
          4
        \fi:
      \fi:
      \exp_stop_f:
           \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
    \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
    \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
    \or:   \__fp_case_return_i_o:ww
    \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
    \fi:
    #1 #5
    \s__fp \__fp_chk:w #2 #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End definition for* `\__fp_+_o:ww`*.*)

`\__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign `#1` rather than `#4`. As usual, expand after the floating point.

```
\cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
  { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```

(*End definition for* `\__fp_add_return_ii_o:Nww`*.*)

`\__fp_add_zeros_o:Nww` Adding two zeros yields `\c_zero_fp`, except if both zeros were -0.

```
\cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
  {
    \if_int_compare:w #2 #1 = 20 \exp_stop_f:
      \exp_after:wN \__fp_add_return_ii_o:Nww
    \else:
      \__fp_case_return_i_o:ww
    \fi:
    #1
    \s__fp \__fp_chk:w 0 #2
  }
```

(*End definition for* `\__fp_add_zeros_o:Nww`*.*)

`\__fp_add_inf_o:Nww` If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked ⟨*sign*$_2$⟩ (`#1`) and the ⟨*sign*$_2$⟩ (`#4`) are identical.

```
\cs_new:Npn \__fp_add_inf_o:Nww
```

```
    #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
  {
    \if_meaning:w #1 #2
      \__fp_case_return_i_o:ww
    \else:
      \__fp_case_use:nw
        {
          \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
            { \token_if_eq_meaning:NNTF #1 #4 + - }
        }
    \fi:
    \s__fp \__fp_chk:w 2 #2 #3;
    \s__fp \__fp_chk:w 2 #4
  }
```

(*End definition for `\__fp_add_inf_o:Nww`.*)

__fp_add_normal_o:Nww

`\__fp_add_normal_o:Nww` ⟨$sign_2$⟩ `\s__fp \__fp_chk:w 1` ⟨$sign_1$⟩ ⟨exp_1⟩ ⟨$body_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*initial* $sign_2$⟩ ⟨exp_2⟩ ⟨$body_2$⟩ `;`

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
\cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
  {
    \if_meaning:w #1#2
      \exp_after:wN \__fp_add_npos_o:NnwNnw
    \else:
      \exp_after:wN \__fp_sub_npos_o:NnwNnw
    \fi:
    #2
  }
```

(*End definition for `\__fp_add_normal_o:Nww`.*)

26.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

__fp_add_npos_o:NnwNnw

`\__fp_add_npos_o:NnwNnw` ⟨$sign_1$⟩ ⟨exp_1⟩ ⟨$body_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*initial* $sign_2$⟩ ⟨exp_2⟩ ⟨$body_2$⟩ `;`

Since we are doing an addition, the final sign is ⟨$sign_1$⟩. Start an `\__int_eval:w`, responsible for computing the exponent: the result, and the ⟨*final sign*⟩ are then given to `\__fp_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent `#2` or `#5`, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `\__fp_add_big_i:wNww` or `\__fp_add_big_ii:wNww`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
\cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      \if_int_compare:w #2 > #5 \exp_stop_f:
        #2
        \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
      \else:
        #5
        \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
      \fi:
      \__int_eval:w #5 - #2 ; #1 #3;
  }
```

(*End definition for* `\__fp_add_npos_o:NnwNnw`*.*)

`\__fp_add_big_i_o:wNww`
`\__fp_add_big_ii_o:wNww`

`\__fp_add_big_i_o:wNww` ⟨*shift*⟩ `;` ⟨*final sign*⟩ ⟨*body*$_1$⟩ `;` ⟨*body*$_2$⟩ `;`

Used in l3fp-expo. Shift the significand of the small number, then add with `\__fp_add_significand_o:NnnwnnnnN`.

```
\cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
  {
    \__fp_decimate:nNnnnn {#1}
      \__fp_add_significand_o:NnnwnnnnN
      #4
    #3
    #2
  }
\cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
  {
    \__fp_decimate:nNnnnn {#1}
      \__fp_add_significand_o:NnnwnnnnN
      #3
    #4
    #2
  }
```

(*End definition for* `\__fp_add_big_i_o:wNww` *and* `\__fp_add_big_ii_o:wNww`*.*)

`\__fp_add_significand_o:NnnwnnnnN`
`\__fp_add_significand_pack:NNNNNNN`
`\__fp_add_significand_test_o:N`

`\__fp_add_significand_o:NnnwnnnnN` ⟨*rounding digit*⟩ `{`⟨Y'_1⟩`}` `{`⟨Y'_2⟩`}` ⟨*extra-digits*⟩ `;` `{`⟨X_1⟩`}` `{`⟨X_2⟩`}` `{`⟨X_3⟩`}` `{`⟨X_4⟩`}` ⟨*final sign*⟩

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\cdots 95 \to 1.00\cdots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```
\cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
  {
    \exp_after:wN \__fp_add_significand_test_o:N
    \__int_value:w \__int_eval:w 1#5#6 + #2
      \exp_after:wN \__fp_add_significand_pack:NNNNNNN
      \__int_value:w \__int_eval:w 1#7#8 + #3 ; #1
  }
\cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
  {
    \if_meaning:w 2 #1
      + 1
    \fi:
    ; #2 #3 #4 #5 #6 #7 ;
  }
\cs_new:Npn \__fp_add_significand_test_o:N #1
```

```
  {
    \if_meaning:w 2 #1
      \exp_after:wN \__fp_add_significand_carry_o:wwwNN
    \else:
      \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
    \fi:
  }
```

(*End definition for* `\__fp_add_significand_o:NnnwnnnnN`*,* `\__fp_add_significand_pack:NNNNNNN`*, and* `\__fp_add_significand_test_o:N`*.*)

__fp_add_significand_no_carry_o:wwwNN

`\__fp_add_significand_no_carry_o:wwwNN` ⟨*8d*⟩ `;` ⟨*6d*⟩ `;` ⟨*2d*⟩ `;` ⟨*rounding digit*⟩ ⟨*sign*⟩

If there's no carry, grab all the digits again and round. The packing function `\__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```
\cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
    #1; #2; #3#4 ; #5#6
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1 #1
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1 #2 #3#4
        + \__fp_round:NNN #6 #4 #5
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_add_significand_no_carry_o:wwwNN`*.*)

__fp_add_significand_carry_o:wwwNN

`\__fp_add_significand_carry_o:wwwNN` ⟨*8d*⟩ `;` ⟨*6d*⟩ `;` ⟨*2d*⟩ `;` ⟨*rounding digit*⟩ ⟨*sign*⟩

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```
\cs_new:Npn \__fp_add_significand_carry_o:wwwNN
    #1; #2; #3#4; #5#6
  {
    + 1
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \__int_value:w \__int_eval:w 1 1 #1
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \__int_value:w \__int_eval:w 1 #2#3 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #6
        \exp_after:wN #3
        \__int_value:w \__fp_round_digit:Nw #4 #5 ;
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_add_significand_carry_o:wwwNN`*.*)

26.1.3 Absolute subtraction

__fp_sub_npos_o:NnwNnw
__fp_sub_eq_o:Nnwnw
__fp_sub_npos_ii_o:Nnwnw

`\__fp_sub_npos_o:NnwNnw` $\langle\mathit{sign}_1\rangle$ $\langle\mathit{exp}_1\rangle$ $\langle\mathit{body}_1\rangle$ `; \s__fp \__fp_chk:w 1` $\langle\mathit{initial\ sign}_2\rangle$ $\langle\mathit{exp}_2\rangle$ $\langle\mathit{body}_2\rangle$ `;`

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `\__fp_sub_npos_i_o:Nnwnw` with the opposite of ⟨*sign*$_1$⟩.

```
\cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
  {
    \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
      \exp_after:wN \__fp_sub_eq_o:Nnwnw
    \or:
      \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
    \else:
      \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
    \fi:
    #1 {#2} #3; {#5} #6;
  }
\cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
\cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
  {
    \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
      \__int_value:w \__fp_neg_sign:N #1
      #3; #2;
  }
```

(*End definition for* `\__fp_sub_npos_o:NnwNnw`*,* `\__fp_sub_eq_o:Nnwnw`*, and* `\__fp_sub_npos_ii_o:Nnwnw`*.*)

`\__fp_sub_npos_i_o:Nnwnw`

After the computation is done, `\__fp_sanitize:Nw` checks for overflow/underflow. It expects the ⟨*final sign*⟩ and the ⟨*exponent*⟩ (delimited by `;`). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y, then call the `far` auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```
\cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      #2
      \if_int_compare:w #2 = #4 \exp_stop_f:
        \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
      \else:
        \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
          { \__int_value:w \__int_eval:w #2 - #4 - 1 \exp_after:wN }
          \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN
      \fi:
        #5
      #3
      #1
  }
```

(*End definition for* `\__fp_sub_npos_i_o:Nnwnw`*.*)

`\__fp_sub_back_near_o:nnnnnnnnN`
`\__fp_sub_back_near_pack:NNNNNNw`
`\__fp_sub_back_near_after:wNNNNw`

`\__fp_sub_back_near_o:nnnnnnnnN` `{`⟨Y_1⟩`}` `{`⟨Y_2⟩`}` `{`⟨Y_3⟩`}` `{`⟨Y_4⟩`}` `{`⟨X_1⟩`}` `{`⟨X_2⟩`}` `{`⟨X_3⟩`}` `{`⟨X_4⟩`}` ⟨*final sign*⟩

In this case, the subtraction is exact, so we discard the ⟨*final sign*⟩ #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```
\cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
  {
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \__int_value:w \__int_eval:w 10#5#6 - #1#2 - 11
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \__int_value:w \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
  }
\cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
  { + #1#2 ; {#3#4#5#6} {#7} ; }
\cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_sub_back_shift:wnnnn
    \fi:
    ; {#1#2#3#4} {#5}
  }
```

(*End definition for* __fp_sub_back_near_o:nnnnnnnnN*,* __fp_sub_back_near_pack:NNNNNNw*, and* __fp_sub_back_near_after:wNNNNw*.*)

__fp_sub_back_shift:wnnnn
__fp_sub_back_shift_ii:ww
__fp_sub_back_shift_iii:NNNNNNNNw
__fp_sub_back_shift_iv:nnnnw

__fp_sub_back_shift:wnnnn ; {⟨Z_1⟩} {⟨Z_2⟩} {⟨Z_3⟩} {⟨Z_4⟩} ;

This function is called with ⟨Z_1⟩ ≤ 999. Act with \number to trim leading zeros from ⟨Z_1⟩ ⟨Z_2⟩ (we don't do all four blocks at once, since non-zero blocks would then overflow TeX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```
\cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
  {
    \exp_after:wN \__fp_sub_back_shift_ii:ww
    \__int_value:w #1 #2 0 ;
  }
\cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
  {
    \if_meaning:w @ #1 @
      - 7
      - \exp_after:wN \use_i:nnn
        \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
        \__int_value:w #2#3 0 ~ 123456789;
    \else:
      - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
    \fi:
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
    \exp_after:wN ;
    \__int_value:w
    #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
  }
\cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
\cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
```

(*End definition for* `\__fp_sub_back_shift:wnnnn` *and others.*)

__fp_sub_back_far_o:NnnwnnnnN

`\__fp_sub_back_far_o:NnnwnnnnN` $\langle$*rounding*$\rangle$ `{`$\langle Y'_1\rangle$`}` `{`$\langle Y'_2\rangle$`}` $\langle$*extra-digits*$\rangle$ `;` `{`$\langle X_1\rangle$`}` `{`$\langle X_2\rangle$`}` `{`$\langle X_3\rangle$`}` `{`$\langle X_4\rangle$`}` $\langle$*final sign*$\rangle$

If the difference is greater than $10^{\langle expo_x\rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x\rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1\rangle\langle Y'_2\rangle = \langle X_1\rangle\langle X_2\rangle\langle X_3\rangle\langle X_4\rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `\__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `;` delimiter).

```
\cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
  {
    \if_case:w
      \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
        \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
          0
        \else:
          \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
        \fi:
      \else:
        \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
      \fi:
      \exp_stop_f:
           \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
    \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwwNN
    \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwwNN
    \fi:
    #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
  }
```

(*End definition for* `\__fp_sub_back_far_o:NnnwnnnnN`*.*)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y. Then the $\langle$*rounding*$\rangle$ `#3` and the $\langle$*final sign*$\rangle$ `#4` control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the $\langle$*rounding*$\rangle$ digit is less than or equal to 5 (remember that the $\langle$*rounding*$\rangle$ digit is only equal to 5 if there was no further non-zero digit).

```
\cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
  {
    \exp_after:wN \__fp_sub_back_quite_far_ii:NN
    \exp_after:wN #3
    \exp_after:wN #4
  }
\cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
  {
    \if_case:w \__fp_round_neg:NNN #2 0 #1
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      { ; {1000} {0000} {0000} {0000} ; }
      { - 1 ; {9999} {9999} {9999} {9999} ; }
  }
```

(*End definition for* `\__fp_sub_back_quite_far_o:wwNN` *and* `\__fp_sub_back_quite_far_ii:NN`*.*)

`\__fp_sub_back_not_far_o:wwwwNN`

In the present case, x and y have different exponents, but y is large enough that $x-y$ has a smaller exponent than x. Decrement the exponent (with `-1`). Then proceed in a way similar to the `near` auxiliaries seen earlier, but multiplying x by 10 (`#30` and `#40` below), and with the added quirk that the ⟨*rounding*⟩ digit has to be taken into account. Namely, we may have to decrease the result by one unit if `\__fp_round_neg:NNN` returns 1. This function expects the ⟨*final sign*⟩ `#6`, the last digit of `1100000000+#40-#2`, and the ⟨*rounding*⟩ digit. Instead of redoing the computation for the second argument, we note that `\__fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of `#2`.

```
\cs_new:Npn \__fp_sub_back_not_far_o:wwwwNN #1 ~ #2; #3 ~ #4; #5#6
  {
    - 1
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \__int_value:w \__int_eval:w 1#30 - #1 - 11
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \__int_value:w \__int_eval:w 11 0000 0000 + #40 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #6
          \use_none:nnnnnnn #2 #5
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_sub_back_not_far_o:wwwwNN`*.*)

`\__fp_sub_back_very_far_o:wwwwNN`
`\__fp_sub_back_very_far_ii_o:nnNwwNN`

The case where $x-y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two ⟨*rounding*⟩ digits `#3` and `#6` (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\__int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```
\cs_new:Npn \__fp_sub_back_very_far_o:wwwwNN #1#2#3#4#5#6#7
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sub_back_very_far_ii_o:nnNwwNN
    { 0 #1#2#3 #4#5#6#7 }
    ;
  }
\cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1#4 - #1 - 1
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 2#5 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #7
```

```
     \__int_value:w
      \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
       1 \else: 2 \fi:
     \__int_value:w \__fp_round_digit:Nw #3 #6 ;
   \exp_after:wN ;
 }
```

(*End definition for* `\__fp_sub_back_very_far_o:wwwwNN` *and* `\__fp_sub_back_very_far_ii_o:nnNwwNN`.)

26.2 Multiplication

26.2.1 Signs, and special numbers

`\__fp_*_o:ww` We go through an auxiliary, which is common with `\__fp_/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `\__fp_/_o:ww`.

```
\cs_new:cpn { __fp_*_o:ww }
 {
  \__fp_mul_cases_o:NnNnww
   *
   { - 2 + }
   \__fp_mul_npos_o:Nww
   { }
 }
```

(*End definition for* `\__fp_*_o:ww`.)

`\__fp_mul_cases_o:nNnnww` Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `\__fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0\times 0 = 0\times 1 = 1\times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0\times\infty$ and $\infty\times 0$ to case 6 or 7 (invalid operation), and the products $1\times\infty = \infty\times 1 = \infty\times\infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument `#4`, because it differs in the case of divisions.

```
\cs_new:Npn \__fp_mul_cases_o:NnNnww
  #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
 {
  \if_case:w \__int_eval:w
         \if_int_compare:w #5 #8 = 11 ~
          1
         \else:
          \if_meaning:w 3 #8
           3
          \else:
           \if_meaning:w 3 #5
            2
           \else:
```

```
                       \if_int_compare:w #5 #8 = 10 ~
                         9 #2 - 2
                       \else:
                         (#5 #2 #8) / 2 * 2 + 7
                       \fi:
                     \fi:
                   \fi:
                 \fi:
                 \if_meaning:w #6 #9 - 1 \fi:
               \__int_eval_end:
         \__fp_case_use:nw { #3 0 }
    \or: \__fp_case_use:nw { #3 2 }
    \or: \__fp_case_return_i_o:ww
    \or: \__fp_case_return_ii_o:ww
    \or: \__fp_case_return_o:Nww \c_zero_fp
    \or: \__fp_case_return_o:Nww \c_minus_zero_fp
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_return_o:Nww \c_inf_fp
    \or: \__fp_case_return_o:Nww \c_minus_inf_fp
    #4
    \fi:
    \s__fp \__fp_chk:w #5 #6 #7;
    \s__fp \__fp_chk:w #8 #9
  }
```

(*End definition for* `\__fp_mul_cases_o:nNnnww`*.*)

26.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

`\__fp_mul_npos_o:Nww`

`\__fp_mul_npos_o:Nww` ⟨*final sign*⟩ `\s__fp \__fp_chk:w 1` ⟨*sign*$_1$⟩ `{`⟨*exp*$_1$⟩`}` ⟨*body*$_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*sign*$_2$⟩ `{`⟨*exp*$_2$⟩`}` ⟨*body*$_2$⟩ `;`

After the computation, `\__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `\__int_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The ⟨*final sign*⟩ is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `\__fp_mul_significand_o:nnnnNnnnn`.

This is also used in l3fp-convert.

```
\cs_new:Npn \__fp_mul_npos_o:Nww
    #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      #4 + #8
      \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
  }
```

(*End definition for* `\__fp_mul_npos_o:Nww`*.*)

`\__fp_mul_significand_o:nnnnNnnnn`
`\__fp_mul_significand_drop:NNNNNw`
`\__fp_mul_significand_keep:NNNNNw`

`\__fp_mul_significand_o:nnnnNnnnn` `{`⟨X_1⟩`}` `{`⟨X_2⟩`}` `{`⟨X_3⟩`}` `{`⟨X_4⟩`}` ⟨*sign*⟩ `{`⟨Y_1⟩`}` `{`⟨Y_2⟩`}` `{`⟨Y_3⟩`}` `{`⟨Y_4⟩`}`

Note the three semicolons at the end of the definition. One is for the last __fp_mul_significand_drop:NNNNNw; one is for __fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an __int_eval:w), is used by __fp_basics_pack_low:NNNNNw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of __int_eval:w.

```
\cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
  {
    \exp_after:wN \__fp_mul_significand_test_f:NNN
    \exp_after:wN #5
    \__int_value:w \__int_eval:w 99990000 + #1*#6 +
      \exp_after:wN \__fp_mul_significand_keep:NNNNNw
      \__int_value:w \__int_eval:w 99990000 + #1*#7 + #2*#6 +
        \exp_after:wN \__fp_mul_significand_keep:NNNNNw
        \__int_value:w \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
          \exp_after:wN \__fp_mul_significand_drop:NNNNNw
          \__int_value:w \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
            \exp_after:wN \__fp_mul_significand_drop:NNNNNw
            \__int_value:w \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
              \exp_after:wN \__fp_mul_significand_drop:NNNNNw
              \__int_value:w \__int_eval:w 99990000 + #3*#9 + #4*#8 +
                \exp_after:wN \__fp_mul_significand_drop:NNNNNw
                \__int_value:w \__int_eval:w 100000000 + #4*#9 ;
    ; \exp_after:wN ;
  }
\cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; + #6 }
\cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; #6 ; }
```

(*End definition for __fp_mul_significand_o:nnnnNnnnn, __fp_mul_significand_drop:NNNNNw, and __fp_mul_significand_keep:NNNNNw.*)

__fp_mul_significand_test_f:NNN

__fp_mul_significand_test_f:NNN ⟨*sign*⟩ 1 ⟨*digits 1–8*⟩ ; ⟨*digits 9–12*⟩ ; ⟨*digits 13–16*⟩ ; + ⟨*digits 17–20*⟩ + ⟨*digits 21–24*⟩ + ⟨*digits 25–28*⟩ + ⟨*digits 29–32*⟩ ; \exp_after:wN ;

If the ⟨*digit 1*⟩ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if ⟨*digit 1*⟩ is zero, we care about digits 17 and 18, and whether further digits are zero.

```
\cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
  {
    \if_meaning:w 0 #3
      \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
    \else:
      \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
    \fi:
    #1 #3
  }
```

(*End definition for* `\__fp_mul_significand_test_f:NNN`.)

__fp_mul_significand_large_f:NwwNNNN

In this branch, ⟨*digit 1*⟩ is non-zero. The result is thus ⟨*digits 1–16*⟩, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `\__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a ⟨*rounding digit*⟩, suitable for `\__fp_round:NNN`.

```
\cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1#3#4#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
          \exp_after:wN #7
          \__int_value:w \__fp_round_digit:Nw
  }
```

(*End definition for* `\__fp_mul_significand_large_f:NwwNNNN`.)

__fp_mul_significand_small_f:NNwwwN

In this branch, ⟨*digit 1*⟩ is zero. Our result is thus ⟨*digits 2–17*⟩, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits `1#3` are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```
\cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
  {
    - 1
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1#3#4
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
          \exp_after:wN #7
          \__int_value:w \__fp_round_digit:Nw
  }
```

(*End definition for* `\__fp_mul_significand_small_f:NNwwwN`.)

26.3 Division

26.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

__fp_/_o:ww

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `\__fp_div_npos_o:Nww` rather than `\__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `\__fp_mul_cases_o:NnNnww` are provided as the fourth argument here.

```
\cs_new:cpn { __fp_/_o:ww }
```

```
  {
    \__fp_mul_cases_o:NnNnww
      /
      { - }
      \__fp_div_npos_o:Nww
      {
        \or:
          \__fp_case_use:nw
            { \__fp_division_by_zero_o:NNww \c_inf_fp / }
        \or:
          \__fp_case_use:nw
            { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
      }
  }
```

(*End definition for* `\__fp_/_o:ww`.)

__fp_div_npos_o:Nww

`\__fp_div_npos_o:Nww` ⟨*final sign*⟩ `\s__fp` `\__fp_chk:w` 1 ⟨$sign_A$⟩ {⟨*exp* A⟩} {⟨A_1⟩} {⟨A_2⟩} {⟨A_3⟩} {⟨A_4⟩} ; `\s__fp` `\__fp_chk:w` 1 ⟨$sign_Z$⟩ {⟨*exp* Z⟩} {⟨Z_1⟩} {⟨Z_2⟩} {⟨Z_3⟩} {⟨Z_4⟩} ;

We want to compute A/Z. As for multiplication, `\__fp_sanitize:Nw` checks for overflow or underflow; we provide it with the ⟨*final sign*⟩, and an integer expression in which we compute the exponent. We set up the arguments of `\__fp_div_significand_i_o:wnnw`, namely an integer ⟨y⟩ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four {⟨A_i⟩}, then the four {⟨Z_i⟩}, a semi-colon, and the ⟨*final sign*⟩, used for rounding at the end.

```
\cs_new:Npn \__fp_div_npos_o:Nww
    #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      #3 - #6
      \exp_after:wN \__fp_div_significand_i_o:wnnw
        \__int_value:w \__int_eval:w #7 \use_i:nnnn #8 + 1 ;
        #4
        {#7}{#8}#9 ;
        #1
  }
```

(*End definition for* `\__fp_div_npos_o:Nww`.)

26.3.2 Work plan

In this subsection, we explain how to avoid overflowing TeX's integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z, we proceed as follows.

- Find an integer $Q_A \simeq 10^4A/Z$.
- Replace A by $B = 10^4A - Q_AZ$.

- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4} Q_A + 10^{-8} Q_B + 10^{-12} Q_C + 10^{-16} Q_D + \text{rounding}$. Since the Q_i are integers, B, C, D, and E are all exact multiples of 10^{-16}, in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B, C, D, and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ *etc.* A reasonable attempt would be to define Q_A as

$$\texttt{\textbackslash int\_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that ε-TEX's `\__int_eval:w` rounds divisions instead of truncating (really, 1/2 would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TEX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \texttt{\textbackslash int\_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A, using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that ε-TEX rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147\cdots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B:

$$\begin{aligned}
10^5 B &= A_1A_20 + 10 \cdot 0.A_3A_4 - 10 \cdot Z_1.Z_2Z_3Z_4 \cdot Q_A \\
&< A_1A_20 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2Z_3Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2Z_3Z_4 + 10 \\
&\le \frac{A_1A_20 \cdot (y - 10 \cdot Z_1.Z_2Z_3Z_4)}{y} + \frac{3}{2}y + 10 \\
&\le \frac{A_1A_20 \cdot 1}{y} + \frac{3}{2}y + 10 \le \frac{10^9 A}{y} + 1.6 \cdot y.
\end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned}
10^5 B &< 10^9 A/y + 1.6y, \\
10^5 C &< 10^9 B/y + 1.6y, \\
10^5 D &< 10^9 C/y + 1.6y, \\
10^5 E &< 10^9 D/y + 1.6y.
\end{aligned}$$

The goal is now to prove that none of B, C, D, and E can go beyond $(2^{31} - 1)/10^9 = 2.147\cdots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned}
10^5 B &< 10^9/y + 1.6y, \\
10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\
10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\
10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2).
\end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \le 10^5$. Thus,

$$\begin{aligned}
10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\
10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\
10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\
10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5).
\end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within TeX's bounds in all cases!

We later need to have a bound on the Q_i. Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i. Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^{4} \left(10^{-4i} Q_i\right) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15}, so we only need to know the integer part of E/Z, and a "rounding" digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2\frac{10^5 E}{10^5 Z} \leq 2\frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε-TeX round

$$P = \texttt{\textbackslash int\_eval:n}\left\{\frac{2 \cdot E_1 E_2}{Z_1 Z_2}\right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2\left|\frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2}\right| + 2\left|\frac{10^8 E - E_1 E_2}{Z_1 Z_2}\right| < 1,$$

($1/2$ comes from ε-TeX's rounding) because each absolute value is less than 10^{-7}. Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in \big((P-1)/2, P/2\big)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in \big(P/2, (P-1)/2\big)$. In each case, we know how to round to an integer, depending on the parity of P, and the rounding mode.

26.3.3 Implementing the significand division

__fp_div_significand_i_o:wnnw

`\__fp_div_significand_i_o:wnnw` ⟨*y*⟩ `;` `{`⟨A_1⟩`}` `{`⟨A_2⟩`}` `{`⟨A_3⟩`}` `{`⟨A_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` `;` ⟨*sign*⟩

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace ⟨A_1⟩ and ⟨A_2⟩, and prepare the ⟨*continuation*⟩ arguments for 4 consecutive calls to `\__fp_div_significand_calc:wwnnnnnnn`. Each of these calls needs ⟨*y*⟩ (`#1`), and it turns out that we need post-expansion there, hence the `\__int_value:w`. Here, `#4` is six brace groups, which give the six first `n`-type arguments of the `calc` function.

```
\cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
  {
    \exp_after:wN \__fp_div_significand_test_o:w
    \__int_value:w \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ;
        #2 #3 ;
        #4
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_iii:wwnnnnn \__int_value:w #1 }
  }
```

(*End definition for* `\__fp_div_significand_i_o:wnnw`.)

__fp_div_significand_calc:wwnnnnnnn
__fp_div_significand_calc_i:wwnnnnnnn
__fp_div_significand_calc_ii:wwnnnnnnn

`\__fp_div_significand_calc:wwnnnnnnn` ⟨$10^6 + Q_A$⟩ `;` ⟨A_1⟩ ⟨A_2⟩ `;` `{`⟨A_3⟩`}` `{`⟨A_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` `{`⟨*continuation*⟩`}`

expands to

$\langle 10^6 + Q_A\rangle$ $\langle$*continuation*$\rangle$; $\langle B_1\rangle$ $\langle B_2\rangle$; {$\langle B_3\rangle$} {$\langle B_4\rangle$} {$\langle Z_1\rangle$} {$\langle Z_2\rangle$} {$\langle Z_3\rangle$} {$\langle Z_4\rangle$}

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C, D, E (with the input shifted accordingly), and is used in l3fp-expo.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within TEX's bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A, implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a $\langle$*continuation*$\rangle$, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i\rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i\rangle \cdot \#7)$$
$$+ 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i\rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i\rangle$ stands for the 10^5 digit of Q_A, which is 0 or 1, and #1, #2, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i\rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9. Indeed, for the auxiliary with $\langle i\rangle = 1$, #1 is at most 80000, leading to contributions of at worse $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i\rangle = 0$, #1 can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with TEX's limits once more.

```
\cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
    \else:
      \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
    \fi:
  }
\cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
  {
    1 1 #1
    #9 \exp_after:wN ;
    \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5 - #5#60
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6 - #70
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7 - #80
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
\cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
  {
    1 0 #1
    #9 \exp_after:wN ;
    \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
```

(*End definition for* __fp_div_significand_calc:wwnnnnnnn *,* __fp_div_significand_calc_i:wwnnnnnnn *, and* __fp_div_significand_calc_ii:wwnnnnnnn*.*)

__fp_div_significand_ii:wwn

__fp_div_significand_ii:wwn $\langle y\rangle$; $\langle B_1\rangle$; {$\langle B_2\rangle$} {$\langle B_3\rangle$} {$\langle B_4\rangle$} {$\langle Z_1\rangle$} {$\langle Z_2\rangle$} {$\langle Z_3\rangle$} {$\langle Z_4\rangle$} $\langle continuations\rangle$ $\langle sign\rangle$

Compute Q_B by evaluating $\langle B_1\rangle\langle B_2\rangle 0/y-1$. The result is output to the left, in an __int_eval:w which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q. This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B.

```
\cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \__int_value:w \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
  }
```

(*End definition for* __fp_div_significand_ii:wwn*.*)

__fp_div_significand_iii:wwnnnnn

__fp_div_significand_iii:wwnnnnn $\langle y\rangle$; $\langle E_1\rangle$; {$\langle E_2\rangle$} {$\langle E_3\rangle$} {$\langle E_4\rangle$} {$\langle Z_1\rangle$} {$\langle Z_2\rangle$} {$\langle Z_3\rangle$} {$\langle Z_4\rangle$} $\langle sign\rangle$

We compute $P\simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5\cdot P$, which amounts to adding $P/2\simeq E/Z$ to Q_D, the appropriate correction from a hypothetical Q_E.

```
\cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
  {
    0
    \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
    \__int_value:w \__int_eval:w ( 2 * #2 #3) / #6 #7 ; % <- P
      #2 ; {#3} {#4} {#5}
      {#6} {#7}
  }
```

(*End definition for* `\__fp_div_significand_iii:wwnnnnn`*.*)

__fp_div_significand_iv:wwnnnnnnn
__fp_div_significand_v:NNw
__fp_div_significand_vi:Nw

`\__fp_div_significand_iv:wwnnnnnnn` $\langle P\rangle$ `;` $\langle E_1\rangle$ `;` `{`$\langle E_2\rangle$`}` `{`$\langle E_3\rangle$`}` `{`$\langle E_4\rangle$`}` `{`$\langle Z_1\rangle$`}` `{`$\langle Z_2\rangle$`}` `{`$\langle Z_3\rangle$`}` `{`$\langle Z_4\rangle$`}` $\langle sign\rangle$

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \operatorname{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D, with an extra ⟨*rounding*⟩ digit. This ⟨*rounding*⟩ digit is 0 or 5 if T does not contribute, *i.e.,* if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8, but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P, which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use `+` as a separator (ending integer expressions explicitly). T is negative if the first character is `-`, it is positive if the first character is neither `0` nor `-`. It is also positive if the first character is `0` and second argument of `\__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```
\cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
  {
    + 5 * #1
    \exp_after:wN \__fp_div_significand_vi:Nw
    \__int_value:w \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
      \exp_after:wN \__fp_div_significand_v:NN
      \__int_value:w \__int_eval:w 199980 + 2*#4 - #1*#8 +
        \exp_after:wN \__fp_div_significand_v:NN
        \__int_value:w \__int_eval:w 200000 + 2*#5 - #1*#9 ;
  }
\cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
\cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
  {
    \if_meaning:w 0 #1
      \if_int_compare:w \__int_eval:w #2 > 0 + 1 \fi:
    \else:
      \if_meaning:w - #1 - \else: + \fi: 1
    \fi:
    ;
  }
```

(*End definition for* `\__fp_div_significand_iv:wwnnnnnnn`*,* `\__fp_div_significand_v:NNw`*, and* `\__fp_div_significand_vi:Nw`*.*)

__fp_div_significand_pack:NNN

At this stage, we are in the following situation: TEX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

> __fp_div_significand_test_o:w $10^6 + Q_A$ __fp_div_significand_pack:NNN $10^6 + Q_B$ __fp_div_significand_pack:NNN $10^6 + Q_C$ __fp_div_significand_pack:NNN $10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon$; ⟨*sign*⟩

Here, $\varepsilon = \operatorname{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5's and the 10^4's).

```
\cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(*End definition for __fp_div_significand_pack:NNN.*)

__fp_div_significand_test_o:w

> __fp_div_significand_test_o:w 1 0 ⟨*5d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*sign*⟩

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence $\widetilde{Q_A}$ (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q_A} = 10 \cdots$.

It is now time to round. This depends on how many digits the final result will have.

```
\cs_new:Npn \__fp_div_significand_test_o:w 10 #1
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
    \else:
      \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
    \fi:
    #1
  }
```

(*End definition for __fp_div_significand_test_o:w.*)

__fp_div_significand_small_o:wwwNNNNwN

> __fp_div_significand_small_o:wwwNNNNwN 0 ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*final sign*⟩

Standard use of the functions __fp_basics_pack_low:NNNNNw and __fp_basics_pack_high:NNNNNw. We finally get to use the ⟨*final sign*⟩ which has been sitting there for a while.

```
\cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
    0 #1; #2; #3; #4#5#6#7#8; #9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1 #3#4#5#6#7
        + \__fp_round:NNN #9 #7 #8
        \exp_after:wN ;
  }
```

(*End definition for __fp_div_significand_small_o:wwwNNNNwN.*)

__fp_div_significand_large_o:wwwNNNNwN

`\__fp_div_significand_large_o:wwwNNNNwN` ⟨*5d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*sign*⟩

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the ⟨*rounding digit*⟩ from the last two of our 18 digits.

```
\cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
    #1; #2; #3; #4#5#6#7#8; #9
  {
    + 1
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \__int_value:w \__int_eval:w 1 #1 #2
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \__int_value:w \__int_eval:w 1 #3 #4 #5 #6 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #9
        \exp_after:wN #6
        \__int_value:w \__fp_round_digit:Nw #7 #8 ;
      \exp_after:wN ;
  }
```

(*End definition for* `\__fp_div_significand_large_o:wwwNNNNwN`*.*)

26.4 Square root

__fp_sqrt_o:w

Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```
\cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
    \fi:
    \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
    \__fp_sqrt_npos_o:w
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_sqrt_o:w`*.*)

__fp_sqrt_npos_o:w
__fp_sqrt_npos_auxi_o:wwnnN
__fp_sqrt_npos_auxii_o:wNNNNNNNN

Prepare `\__fp_sanitize:Nw` to receive the final sign `0` (the result is always positive) and the exponent, equal to half of the exponent `#1` of the argument. If the exponent `#1` is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8$`#2#3`+`#4#5` through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of of the argument by one digit, getting $a_1' \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```
\cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \__int_value:w \__int_eval:w
      \if_int_odd:w #1 \exp_stop_f:
        \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN
      \fi:
      #1 / 2
      \__fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
  }
\cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
  {
    ( #1 + 1 ) / 2
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sqrt_npos_auxii_o:wNNNNNNNN
    ;
    0 #3 #4
  }
\cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  { \__fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }
```

(*End definition for* `\__fp_sqrt_npos_o:w`*,* `\__fp_sqrt_npos_auxi_o:wwnnN`*, and* `\__fp_sqrt_npos_auxii_o:wNNNNNNNN`*.*)

`\__fp_sqrt_Newton_o:wwn`

Newton's method maps $x \mapsto \bigl[(x + [10^8 a_1/x])/2\bigr]$ in each iteration, where $[b/c]$ denotes ε-TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε-TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x+t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left[\frac{x + [10^8 a_1/x]}{2}\right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}\,.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2}\frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}\,.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4}\frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x-1)] \geq 2x - 1$$

hence $10^8 a_1/(x-1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x. To stop the iteration when two consecutive results are equal, the function `\__fp_sqrt_Newton_o:wwn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε-TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```
\cs_new:Npn \__fp_sqrt_Newton_o:wwn #1; #2; #3
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
      \__int_value:w \__int_eval:w 9999 9999 +
        \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \exp_after:wN \__fp_sqrt_Newton_o:wwn
    \__int_value:w \__int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
    #1; {#3}
  }
```

(*End definition for* `\__fp_sqrt_Newton_o:wwn`.)

`\__fp_sqrt_auxi_o:NNNNwnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then `;` {$\langle a_1\rangle$} {$\langle a_2\rangle$} $\langle a'\rangle$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x-1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}\,.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8}\,,$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `\__fp_sqrt_auxii_o:NnnnnnnnN` is called several times to get closer and closer underestimates of $\sqrt{a}$. By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```
\cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn
      {#1#2#3#4} {#5} {2499} {9988} {7500}
  }
```

(*End definition for* `\__fp_sqrt_auxi_o:NNNNwnnN`.)

`\__fp_sqrt_auxii_o:NnnnnnnnN` This receives a continuation function `#1`, then five blocks of 4 digits for y, then two 8-digit blocks and a single digit for a. A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε-TeX's rounding division)

$$10^{4j}z = \Bigl[\bigl(\lfloor 10^{4j}(a - y^2)\rfloor - 257\bigr) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8 y + 1\rfloor\Bigr]\,.$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8/10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a}-y$. On the one hand, $\sqrt{a}-y \leq 16\cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a}-y \leq 5(\sqrt{a}+y)(\sqrt{a}-y) = 5(a-y^2) < 10^{9-4j}$. For $j=3$, the first bound is better, while for larger j, the second bound is better. For all $j \in [3,6]$, we find $\sqrt{a}-y < 16\cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a}-y) = \frac{10^{4j}\bigl(a-y^2-(\sqrt{a}-y)^2\bigr)}{2y} \geq \frac{\bigl\lfloor 10^{4j}(a-y^2)\bigr\rfloor - 257}{2\cdot 10^{-8}\lfloor 10^8 y+1\rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16\cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/\bigl(2\cdot 10^{-8}\lfloor 10^8 y+1\rfloor\bigr) \geq 1/2$. Given that ε-TEX's integer division obeys $[b/c] \leq b/c+1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a}-y)$, hence $y+z \leq \sqrt{a}$ is an underestimate of $\sqrt{a}$, as claimed. One implementation detail: because the computation involves `-#4*#4 - 2*#3*#5 - 2*#2*#6` which may be as low as $-5\cdot 10^8$, we need to use the `pack_big` functions, and the `big` shifts.

```
\cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN #1
    \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
      + #7 - #2 * #2
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
        - 2 * #2 * #3
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
          + #8 - #3 * #3 - 2 * #2 * #4
          \exp_after:wN \__fp_pack_big:NNNNNNw
          \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
            - 2 * #3 * #4 - 2 * #2 * #5
            \exp_after:wN \__fp_pack_big:NNNNNNw
            \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
              + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
              \exp_after:wN \__fp_pack_big:NNNNNNw
              \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
                - 2 * #4 * #5 - 2 * #3 * #6
                \exp_after:wN \__fp_pack_big:NNNNNNw
                \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
                  - #5 * #5 - 2 * #4 * #6
                  \exp_after:wN \__fp_pack_big:NNNNNNw
                  \__int_value:w \__int_eval:w
                    \c__fp_big_middle_shift_int
                    - 2 * #5 * #6
                    \exp_after:wN \__fp_pack_big:NNNNNNw
                    \__int_value:w \__int_eval:w
                      \c__fp_big_trailing_shift_int
                      - #6 * #6 ;
    % (
    - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
    {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
  }
```

(*End definition for* `\__fp_sqrt_auxii_o:NnnnnnnnN`.)

__fp_sqrt_auxiii_o:wnnnnnnnn
__fp_sqrt_auxiv_o:NNNNNw
__fp_sqrt_auxv_o:NNNNNw
__fp_sqrt_auxvi_o:NNNNNw
__fp_sqrt_auxvii_o:NNNNNw

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2\rangle$ `;` `{`$\langle d_3\rangle$`}` ... `{`$\langle d_{10}\rangle$`}`, where each block has 4 digits, except $\langle d_2\rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a-y^2) < 2\cdot 10^8$, then leaves an open parenthesis and the integer $\bigl\lfloor 10^{4j}(a-y^2)\bigr\rfloor$ in an integer expression. The closing parenthesis is provided by the caller `\__fp_sqrt_auxii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \Bigl[\bigl(\lfloor 10^{4j}(a-y^2)\rfloor - 257\bigr)\cdot(0.5\cdot 10^8) \Big/ \lfloor 10^8 y + 1\rfloor\Bigr]$$

for an estimate of $10^{4j}(\sqrt{a}-y)$. If $d_2 \geq 2$, $j=3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j=4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y, then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a}-y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded $\sqrt{a}$ with only one more call to `\__fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching $j=6$, because for $j<6$, one has $2\cdot 10^8 \leq 10^{4(j+1)}(a-y^2)$, hence

$$10^{4j}z \geq \frac{(20000-257)(0.5\cdot 10^8)}{\lfloor 10^8 y + 1\rfloor} \geq (20000-257)\cdot 0.5 > 0\,.$$

```
\cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
  #1; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 > 1 \exp_stop_f:
      \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
      \__int_value:w \__int_eval:w (#1#2 %)
    \else:
      \if_int_compare:w #1#2 > 1 \exp_stop_f:
        \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
        \__int_value:w \__int_eval:w (#1#2#3 %)
      \else:
        \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
          \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
          \__int_value:w \__int_eval:w (#1#2#3#4 %)
        \else:
          \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
          \__int_value:w \__int_eval:w (#1#2#3#4#5 %)
        \fi:
      \fi:
    \fi:
  }
\cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
\cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
\cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
\cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
  {
    \if_int_compare:w #1#2 = 0 \exp_stop_f:
      \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
    \fi:
```

```
    \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
  }
```

(*End definition for* `\__fp_sqrt_auxiii_o:wnnnnnnnn` *and others.*)

`\__fp_sqrt_auxviii_o:nnnnnnn`
`\__fp_sqrt_auxix_o:wnwnw`

Simply add the two 8-digit blocks of z, aligned to the last four of the five 4-digit blocks of y, then call the `auxii` auxiliary to evaluate $y'^2 = (y+z)^2$.

```
\cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
  {
    \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
    \__int_value:w \__int_eval:w #3
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w #1 + 1#4#5
        \exp_after:wN \__fp_basics_pack_low:NNNNNw
        \__int_value:w \__int_eval:w #2 + 1#6#7 ;
  }
\cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
  }
```

(*End definition for* `\__fp_sqrt_auxviii_o:nnnnnnn` *and* `\__fp_sqrt_auxix_o:wnwnw`*.*)

`\__fp_sqrt_auxx_o:Nnnnnnnn`
`\__fp_sqrt_auxxi_o:wwnnN`

At this stage, $j = 6$ and $10^{24}z < 10^7$, hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \ge \bigl(10^{24}(a-y^2) - 258\bigr) \cdot (0.5 \cdot 10^8) \Bigm/ (10^8 y + 1)\,,$$

then $10^{24}(a-y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a-y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a})\,,$$

which finally implies $0 \le \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of $\sqrt{a}$ and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m, and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments `#1`, `#2`, `#3`, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits `#8` of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \le 1 - 10^{-16}$.

```
\cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
  {
    \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
    \__int_value:w \__int_eval:w
      (#8 + 2499) / 5000 * 5000 ;
      {#4} {#5} {#6} {#7} ;
  }
\cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
```

```
      \__fp_sqrt_auxxii_o:nnnnnnnnw
      #2 {#1}
      {#3} { #4 + 1 } #5
  }
```

(*End definition for* `\__fp_sqrt_auxx_o:Nnnnnnnn` *and* `\__fp_sqrt_auxxi_o:wwnnN`.)

__fp_sqrt_auxxii_o:nnnnnnnnw
__fp_sqrt_auxxiii_o:w

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```
\cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
  {
    \if_int_compare:w #1#2 > 0 \exp_stop_f:
      \if_int_compare:w #1#2 = 1 \exp_stop_f:
        \if_int_compare:w #3#4 = 0 \exp_stop_f:
          \if_int_compare:w #5#6 = 0 \exp_stop_f:
            \if_int_compare:w #7#8 = 0 \exp_stop_f:
              \__fp_sqrt_auxxiii_o:w
            \fi:
          \fi:
        \fi:
      \fi:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \__int_value:w 9998
    \else:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \__int_value:w 10000
    \fi:
    ;
  }
\cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
  {
    \fi: \fi: \fi: \fi: \fi:
    \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
  }
```

(*End definition for* `\__fp_sqrt_auxxii_o:nnnnnnnnw` *and* `\__fp_sqrt_auxxiii_o:w`.)

__fp_sqrt_auxxiv_o:wnnnnnnnN

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by `\__fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are "rounded to even", and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `\__fp_round_digit:Nw`, which receives (after removal of the 10000's digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```
\cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1 0000 0000 + #2#3
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1 0000 0000
        + #4#5
        \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN 0
          \exp_after:wN 0
          \__int_value:w
            \exp_after:wN \use_i:nn
            \exp_after:wN \__fp_round_digit:Nw
            \__int_value:w \__int_eval:w #6 + 19999 - #1 ;
    \exp_after:wN ;
  }
```

(*End definition for `\__fp_sqrt_auxxiv_o:wnnnnnnnN`.*)

26.5 About the sign

`\__fp_sign_o:w`
`\__fp_sign_aux_o:w`

Find the sign of the floating point: `nan`, `+0`, `-0`, `+1` or `-1`.

```
\cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
  {
    \if_case:w #1 \exp_stop_f:
          \__fp_case_return_same_o:w
    \or:   \exp_after:wN \__fp_sign_aux_o:w
    \or:   \exp_after:wN \__fp_sign_aux_o:w
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #1 #2;
  }
\cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
  { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }
```

(*End definition for `\__fp_sign_o:w` and `\__fp_sign_aux_o:w`.*)

`\__fp_set_sign_o:w`

This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on `#1`. It also expands after itself in the input stream, just like `\__fp_+_o:ww`.

```
\cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \__int_value:w
      \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
    #4;
  }
```

(*End definition for `\__fp_set_sign_o:w`.*)

```
⟨/initex | package⟩
```

27 l3fp-extended implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

27.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{\langle a_1\rangle\}\ \{\langle a_2\rangle\}\ \{\langle a_3\rangle\}\ \{\langle a_4\rangle\}\ \{\langle a_5\rangle\}\ \{\langle a_6\rangle\}\ ;$$

where each $\langle a_i\rangle$ is exactly 4 digits (ranging from `0000` to `9999`), except $\langle a_1\rangle$, which may be any "not-too-large" non-negative integer, with or without leading zeros. Here, "not-too-large" depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^{6}\langle a_i\rangle \cdot 10^{-4i}$.

Most functions we define here have the form

```
\__fp_fixed_⟨calculation⟩:wwn ⟨operand$_1$⟩ ; ⟨operand$_2$⟩ ; {⟨continuation⟩}
```

They perform the ⟨*calculation*⟩ on the two ⟨*operands*⟩, then feed the result (6 brace groups followed by a semicolon) to the ⟨*continuation*⟩, responsible for the next step of the calculation. Some functions only accept an `N`-type ⟨*continuation*⟩. This allows constructions such as

```
\__fp_fixed_add:wwn ⟨X$_1$⟩ ; ⟨X$_2$⟩ ;
\__fp_fixed_mul:wwn ⟨X$_3$⟩ ;
\__fp_fixed_add:wwn ⟨X$_4$⟩ ;
```

to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `\__fp_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

27.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in l3fp-expo.

```
\tl_const:Nn \c__fp_one_fixed_tl
  { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(*End definition for* `\c__fp_one_fixed_tl`.)

`\__fp_fixed_continue:wn` This function simply calls the next function.

```
\cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(*End definition for* `\__fp_fixed_continue:wn`.)

`\__fp_fixed_add_one:wN`

`\__fp_fixed_add_one:wN` ⟨*a*⟩ `;` ⟨*continuation*⟩

This function adds 1 to the fixed point $\langle a\rangle$, by changing a_1 to $10000 + a_1$, then calls the ⟨*continuation*⟩. This requires $a_1 + 10000 < 2^{31}$.

```
\cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
  {
    \exp_after:wN #3 \exp_after:wN
      { \__int_value:w \__int_eval:w \c__fp_myriad_int + #1 } #2 ;
  }
```

(*End definition for* `\__fp_fixed_add_one:wN`.)

`\__fp_fixed_div_myriad:wn`

Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group `#1` may have any number of digits, and we must split `#1` into the new first group and a second group of exactly 4 digits. The choice of shifts allows `#1` to be in the range $[0, 5\cdot 10^8 - 1]$.

```
\cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
        + #1 ; {#2}{#3}{#4}{#5};
  }
```

(*End definition for* `\__fp_fixed_div_myriad:wn`.)

`\__fp_fixed_mul_after:wwn`

The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the ⟨*continuation*⟩ `#3` in front.

```
\cs_new:Npn \__fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(*End definition for* `\__fp_fixed_mul_after:wwn`.)

27.3 Multiplying a fixed point number by a short one

`\__fp_fixed_mul_short:wwn`

```
\__fp_fixed_mul_short:wwn
  {⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
  {⟨b0⟩} {⟨b1⟩} {⟨b2⟩} ; {⟨continuation⟩}
```

Computes the product $c = ab$ of $a = \sum_i \langle a_i\rangle 10^{-4i}$ and $b = \sum_i \langle b_i\rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24}, and leaves ⟨*continuation*⟩ `{`$\langle c_1\rangle$`}` … `{`$\langle c_6\rangle$`} ;` in the input stream, where each of the $\langle c_i\rangle$ are blocks of 4 digits, except $\langle c_1\rangle$, which is any TeX integer. Note that indices for $\langle b\rangle$ start at 0: for instance a second operand of `{0001}{0000}{0000}` leaves the first operand unchanged (rather than dividing it by 10^4, as `\__fp_fixed_mul:wwn` would).

```
\cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
      + #1*#7
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_middle_shift_int
        + #1*#8 + #2*#7
    \exp_after:wN \__fp_pack:NNNNNw
    \__int_value:w \__int_eval:w \c__fp_middle_shift_int
     + #1*#9 + #2*#8 + #3*#7
     \exp_after:wN \__fp_pack:NNNNNw
     \__int_value:w \__int_eval:w \c__fp_middle_shift_int
      + #2*#9 + #3*#8 + #4*#7
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_middle_shift_int
       + #3*#9 + #4*#8 + #5*#7
       \exp_after:wN \__fp_pack:NNNNNw
       \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
        + #4*#9 + #5*#8 + #6*#7
        + ( #5*#9 + #6*#8 + #6*#9 / \c__fp_myriad_int )
        / \c__fp_myriad_int ; ;
 }
```

(*End definition for* `\__fp_fixed_mul_short:wwn`.)

27.4 Dividing a fixed point number by a small integer

`\__fp_fixed_div_int:wwN`
`\__fp_fixed_div_int:wnN`
`\__fp_fixed_div_int_auxi:wnn`
`\__fp_fixed_div_int_auxii:wnn`
`\__fp_fixed_div_int_pack:Nw`
`\__fp_fixed_div_int_after:Nw`

`\__fp_fixed_div_int:wwN` ⟨a⟩ ; ⟨n⟩ ; ⟨*continuation*⟩

Divides the fixed point number ⟨*a*⟩ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the ⟨*continuation*⟩. There is no bound on a_1.

The arguments of the `i` auxiliary are 1: one of the a_i, 2: n, 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n.

The `ii` auxiliary receives Q_i, n, and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1}. The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

`\__fp_fixed_div_int_after:Nw` ⟨*continuation*⟩
$-1 + Q_1$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_2$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_3$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_4$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_5$
`\__fp_fixed_div_int_pack:Nw` 9999
`\__fp_fixed_div_int_auxii:wnn` Q_6 `;` `{`⟨*n*⟩`}` `{`⟨a_6⟩`}`

where expansion is happening from the last line up. The `iii` auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each `pack` auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the ⟨*continuation*⟩ as appropriate.

```
\cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
 {
  \exp_after:wN \__fp_fixed_div_int_after:Nw
  \exp_after:wN #8
  \__int_value:w \__int_eval:w - 1
```

```
      \__fp_fixed_div_int:wnN
      #1; {#7} \__fp_fixed_div_int_auxi:wnn
      #2; {#7} \__fp_fixed_div_int_auxi:wnn
      #3; {#7} \__fp_fixed_div_int_auxi:wnn
      #4; {#7} \__fp_fixed_div_int_auxi:wnn
      #5; {#7} \__fp_fixed_div_int_auxi:wnn
      #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
  }
\cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
  {
    \exp_after:wN #3
    \__int_value:w \__int_eval:w #1 / #2 - 1 ;
    {#2}
    {#1}
  }
\cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
  {
    + #1
    \exp_after:wN \__fp_fixed_div_int_pack:Nw
    \__int_value:w \__int_eval:w 9999
      \exp_after:wN \__fp_fixed_div_int:wnN
      \__int_value:w \__int_eval:w #3 - #1*#2 \__int_eval_end:
  }
\cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
\cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
\cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
```

(*End definition for* `\__fp_fixed_div_int:wwN` *and others.*)

27.5 Adding and subtracting fixed points

__fp_fixed_add:wwn
__fp_fixed_sub:wwn
__fp_fixed_add:Nnnnnwnn
__fp_fixed_add:nnNnnnwn
__fp_fixed_add_pack:NNNNNwn
__fp_fixed_add_after:NNNNNwn

`\__fp_fixed_add:wwn ⟨a⟩ ; ⟨b⟩ ; {⟨continuation⟩}`

Computes $a+b$ (resp. $a-b$) and feeds the result to the ⟨*continuation*⟩. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, $a_1, \ldots, a_4$, the rest of a, and b_1 and b_2. The second auxiliary receives the rest of a, the sign multiplying b, the rest of b, and the ⟨*continuation*⟩ as arguments. After going down through the various level, we go back up, packing digits and bringing the ⟨*continuation*⟩ (`#8`, then `#7`) from the end of the argument list to its start.

```
\cs_new:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
\cs_new:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwnn - }
\cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
  {
    \exp_after:wN \__fp_fixed_add_after:NNNNNwn
    \__int_value:w \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
      \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
      \__int_value:w \__int_eval:w 1 9999 9998 + #4#5
        \__fp_fixed_add:nnNnnnwn #6 #1
  }
\cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
  {
    #3 #4#5
    \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
    \__int_value:w \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
  }
\cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
  { + #1 ; {#7} {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
  { #7 {#1#2#3#4#5} {#6} }
```

(*End definition for* `\__fp_fixed_add:wwn` *and others.*)

27.6 Multiplying fixed points

`\__fp_fixed_mul:wwn`
`\__fp_fixed_mul:nnnnnnnw`

`\__fp_fixed_mul:wwn` ⟨a⟩ ; ⟨b⟩ ; {⟨*continuation*⟩}

Computes $a \times b$ and feeds the result to ⟨*continuation*⟩. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for TEX macros. Note that we don't need to obtain an exact rounding, contrarily to the `*` operator, so things could be harder. We wish to perform carries in

$$\begin{aligned} a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\ & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\ & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ & + \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\ & \qquad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\ & \qquad + a_1 \cdot b_5 + a_5 \cdot b_1 \Big) \cdot 10^{-24} + O(10^{-24}), \end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 `ulp`. Note how the first 15 terms only depend on $a_1, \ldots, a_4$ and $b_1, \ldots, b_4$, while the last 6 terms only depend on a_1, a_2, a_5, a_6, and the corresponding parts of b. Hence, the first function grabs $a_1, \ldots, a_4$, the rest of a, and $b_1, \ldots, b_4$, and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6 and finally the ⟨*continuation*⟩ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The ⟨*continuation*⟩ is finally placed in front of the 6 brace groups by `\__fp_fixed_mul_after:wwn`.

```
\cs_new:Npn \__fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_middle_shift_int
        + #1*#6
        \exp_after:wN \__fp_pack:NNNNNw
        \__int_value:w \__int_eval:w \c__fp_middle_shift_int
          + #1*#7 + #2*#6
          \exp_after:wN \__fp_pack:NNNNNw
          \__int_value:w \__int_eval:w \c__fp_middle_shift_int
      + #1*#8 + #2*#7 + #3*#6
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_middle_shift_int
       + #1*#9 + #2*#8 + #3*#7 + #4*#6
       \exp_after:wN \__fp_pack:NNNNNw
       \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
        + #2*#9 + #3*#8 + #4*#7
        + ( #3*#9 + #4*#8
         + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7}  {#1}{#2}
 }
\cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
 {
  #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
  + #1*#3 + #5*#7 ; ;
 }
```

(*End definition for* `\__fp_fixed_mul:wwn` *and* `\__fp_fixed_mul:nnnnnnnw`.)

27.7 Combining product and sum of fixed points

`\__fp_fixed_mul_add:wwwn`
`\__fp_fixed_mul_sub_back:wwwn`
`\__fp_fixed_mul_one_minus_mul:wwn`

`\__fp_fixed_mul_add:wwwn` ⟨*a*⟩ ; ⟨*b*⟩ ; ⟨*c*⟩ ; {⟨*continuation*⟩}
`\__fp_fixed_mul_sub_back:wwwn` ⟨*a*⟩ ; ⟨*b*⟩ ; ⟨*c*⟩ ; {⟨*continuation*⟩}
`\__fp_fixed_one_minus_mul:wwn` ⟨*a*⟩ ; ⟨*b*⟩ ; {⟨*continuation*⟩}

Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the ⟨*continuation*⟩. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned} a \times b + c =&(a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\ &+ (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ &+ (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\ &+ (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ &+ \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\ &\qquad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\ &\qquad + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \Big) \cdot 10^{-24} + O(10^{-24}), \end{aligned}$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c, and $\cdot$ denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4}, is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+ c_5 c_6$ `; {`⟨*continuation*⟩`} ;`. The $+ c_5 c_6$ piece, which is omitted for `\__fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the 10^{-24} level.

```
\cs_new:Npn \__fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
 {
  \exp_after:wN \__fp_fixed_mul_after:wwn
  \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
```

```
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn +
          + #5 #6 ; #2 ; #1 ; #2 ; +
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn -
          + #5 #6 ; #2 ; #1 ; #2 ; -
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
        \__fp_fixed_mul_add:Nwnnnwnnn -
          ; #2 ; #1 ; #2 ; -
          ; ;
  }
```

(*End definition for* `\__fp_fixed_mul_add:wwwn`*,* `\__fp_fixed_mul_sub_back:wwwn`*, and* `\__fp_fixed_-mul_one_minus_mul:wwn`*.*)

__fp_fixed_mul_add:Nwnnnwnnn

`\__fp_fixed_mul_add:Nwnnnwnnn` $\langle op\rangle$ `+` $\langle c_3\rangle$ $\langle c_4\rangle$ `;`
$\langle b\rangle$ `;` $\langle a\rangle$ `;` $\langle b\rangle$ `;` $\langle op\rangle$
`+` $\langle c_5\rangle$ $\langle c_6\rangle$ `;`

Here, $\langle op\rangle$ is either `+` or `-`. Arguments `#3`, `#4`, `#5` are $\langle b_1\rangle$, $\langle b_2\rangle$, $\langle b_3\rangle$; arguments `#7`, `#8`, `#9` are $\langle a_1\rangle$, $\langle a_2\rangle$, $\langle a_3\rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8}, $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12}, and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16}. The a–b products use the sign `#1`. Note that `#2` is empty for `\__fp_fixed_one_minus_mul:wwn`. We call the `ii` auxiliary for levels 10^{-20} and 10^{-24}, keeping the pieces of $\langle a\rangle$ we've read, but not $\langle b\rangle$, since there is another copy later in the input stream.

```
\cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
  {
    #1 #7*#3
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
      #1 #7*#4 #1 #8*#3
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
        #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
          #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
  }
```

(*End definition for* `\__fp_fixed_mul_add:Nwnnnwnnn`*.*)

__fp_fixed_mul_add:nnnnwnnnn

`\__fp_fixed_mul_add:nnnnwnnnn` ⟨*a*⟩ ; ⟨*b*⟩ ; ⟨*op*⟩
  + ⟨c_5⟩ ⟨c_6⟩ ;

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the `i` auxiliary. Then we prepare level 10^{-24}. We don't have access to all parts of ⟨*a*⟩ and ⟨*b*⟩ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$
$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1, a_5, a_6, and the corresponding pieces of ⟨*b*⟩.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
  {
    ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \__int_value:w \__int_eval:w \c__fp_big_trailing_shift_int
      \__fp_fixed_mul_add:nnnnwnnwN
        { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
        { #7 + #4*#8 + #3*#9 + #2 }
        {#1} #5;
        {#6}
  }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnnn`.)

__fp_fixed_mul_add:nnnnwnnwN

`\__fp_fixed_mul_add:nnnnwnnwN` {⟨*partial*$_1$⟩} {⟨*partial*$_2$⟩}
  {⟨a_1⟩} {⟨a_5⟩} {⟨a_6⟩} ; {⟨b_1⟩} {⟨b_5⟩} {⟨b_6⟩} ;
  ⟨*op*⟩ + ⟨c_5⟩ ⟨c_6⟩ ;

Complete the ⟨*partial*$_1$⟩ and ⟨*partial*$_2$⟩ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28}. The trailing $+ c_5 c_6$ is taken into the expression for level 10^{-24}. Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See l3fp-aux for the definition of the shifts and packing auxiliaries.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
  {
    #9 (#4* #1 *#7)
    #9 (#5*#6+#4* #2 *#7+#3*#8) / \c__fp_myriad_int
  }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnwN`.)

27.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures ("extended-precision" numbers, in short, "ep"), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits\rangle \cdot 10^{\langle exponent\rangle}$. This convention differs from floating points.

__fp_ep_to_fixed:wwn
__fp_ep_to_fixed_auxi:www
__fp_ep_to_fixed_auxii:nnnnnnnwn

Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```
\cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
  {
    \exp_after:wN \__fp_ep_to_fixed_auxi:www
    \__int_value:w \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
    \exp:w \exp_end_continue_f:w
    \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
  }
\cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
    #2 #1#3#4#5#6#7 0000 !
  }
\cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
  { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(*End definition for* __fp_ep_to_fixed:wwn *,* __fp_ep_to_fixed_auxi:www *, and* __fp_ep_to_fixed_auxii:nnnnnnnwn*.*)

__fp_ep_to_ep:wwN
__fp_ep_to_ep_loop:N
__fp_ep_to_ep_end:www
__fp_ep_to_ep_zero:ww

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The loop auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the end auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with __fp_use_i:ww any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```
\cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
  {
    \exp_after:wN #8
    \__int_value:w \__int_eval:w #1 + 4
      \exp_after:wN \use_i:nn
      \exp_after:wN \__fp_ep_to_ep_loop:N
      \__int_value:w \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
      #3#4#5#6#7 ; ; !
  }
\cs_new:Npn \__fp_ep_to_ep_loop:N #1
  {
    \if_meaning:w 0 #1
      - 1
    \else:
      \__fp_ep_to_ep_end:www #1
    \fi:
    \__fp_ep_to_ep_loop:N
  }
\cs_new:Npn \__fp_ep_to_ep_end:www
    #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
  {
    \fi:
    \if_meaning:w ; #1
      - 2 * \c__fp_max_exponent_int
      \__fp_ep_to_ep_zero:ww
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_use_i:ww , ;
    #1 #2 0000 0000 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
  { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(*End definition for* `\__fp_ep_to_ep:wwN` *and others.*)

`\__fp_ep_compare:wwww`
`\__fp_ep_compare_aux:wwww`

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in $[1000, 9999]$.

```
\cs_new:Npn \__fp_ep_compare:wwww #1,#2#3#4#5#6#7;
  { \__fp_ep_compare_aux:wwww {#1}{#2}{#3}{#4}{#5}; #6#7; }
\cs_new:Npn \__fp_ep_compare_aux:wwww #1;#2;#3,#4#5#6#7#8#9;
  {
    \if_case:w
      \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
            \if_int_compare:w #2 = #8#9 \exp_stop_f:
              0
            \else:
              \if_int_compare:w #2 < #8#9 - \fi: 1
            \fi:
    \or:    1
    \else: -1
    \fi:
  }
```

(*End definition for* `\__fp_ep_compare:wwww` *and* `\__fp_ep_compare_aux:wwww`*.*)

`\__fp_ep_mul:wwwwn`
`\__fp_ep_mul_raw:wwwwN`

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in $[100, 9999]$.

```
\cs_new:Npn \__fp_ep_mul:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #3,#4;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #1,#2;
      \__fp_ep_mul_raw:wwwwN
    }
    \__fp_fixed_continue:wn
  }
\cs_new:Npn \__fp_ep_mul_raw:wwwwN #1,#2; #3,#4; #5
  {
    \__fp_fixed_mul:wwn #2; #4;
    { \exp_after:wN #5 \__int_value:w \__int_eval:w #1 + #3 , }
  }
```

(*End definition for* `\__fp_ep_mul:wwwwn` *and* `\__fp_ep_mul_raw:wwwwN`*.*)

27.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n\rangle$ the numerator and $\langle d\rangle$ the denominator. After a simple normalization step, we can assume that $\langle n\rangle \in [0.1, 1)$ and $\langle d\rangle \in [0.1, 1)$, and compute $\langle n\rangle/(10\langle d\rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1\rangle \cdots \langle n_6\rangle$ and the 6 blocks $\langle d_1\rangle \cdots \langle d_6\rangle$, the condition translates to $\langle n_1\rangle, \langle d_1\rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8/\langle d\rangle$ by computing

$$\begin{aligned}
\alpha &= \left[\frac{10^9}{\langle d_1\rangle + 1}\right] \\
\beta &= \left[\frac{10^9}{\langle d_1\rangle}\right] \\
a &= 10^3\alpha + (\beta - \alpha)\cdot\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right) - 1250,
\end{aligned}$$

where $\left[\frac{\bullet}{\bullet}\right]$ denotes ε-TeX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2\rangle/10^4$, so that when $\langle d_2\rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1\rangle \simeq 10^8/\langle d\rangle$, while when $\langle d_2\rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1\rangle + 1) \simeq 10^8/\langle d\rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d\rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d\rangle a/10^8 = 1 - \epsilon$ using the relation $1/(1-\epsilon) \simeq (1+\epsilon)(1+\epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n\rangle}{\langle d\rangle} = \frac{\langle n\rangle a}{10^8}\big((1+\epsilon)(1+\epsilon^2) + \epsilon^4\big).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d\rangle < 10^3\langle d_1\rangle + 10^{-1}(\langle d_2\rangle + 1)$, and that ε-TeX's division $\left[\frac{\langle d_2\rangle}{10}\right]$ underestimates $10^{-1}(\langle d_2\rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2\rangle$. Then,

$$\begin{aligned}
10^7\langle d\rangle a &< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\beta + \left[\frac{\langle d_2\rangle}{10}\right]\alpha - 1250\right) \tag{1}\\
&< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right) \tag{2}\\
&\quad \left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\left(\frac{10^9}{\langle d_1\rangle} + \frac{1}{2}\right) + \left[\frac{\langle d_2\rangle}{10}\right]\left(\frac{10^9}{\langle d_1\rangle + 1} + \frac{1}{2}\right) - 1250\right) \tag{3}\\
&< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\frac{10^{12}}{\langle d_1\rangle} - \left[\frac{\langle d_2\rangle}{10}\right]\frac{10^9}{\langle d_1\rangle(\langle d_1\rangle + 1)} - 750\right) \tag{4}
\end{aligned}$$

We recognize a quadratic polynomial in $[\langle d_2\rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2\rangle/10]+a)(b-c[\langle d_2\rangle/10]) \leq (b+ca)^2/(4c)$. Hence,

$$10^7\langle d\rangle a < \frac{10^{15}}{\langle d_1\rangle(\langle d_1\rangle + 1)}\left(\langle d_1\rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8}\cdot 10^{-9}\langle d_1\rangle(\langle d_1\rangle + 1)\right)^2$$

Since $\langle d_1\rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1\rangle(\langle d_1\rangle+1)$, hence $10^7\langle d\rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\ldots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7\langle d\rangle a > \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] - \frac{1}{2}\right)\left(\frac{10^{12}}{\langle d_1\rangle} - \left[\frac{\langle d_2\rangle}{10}\right]\frac{10^9}{\langle d_1\rangle(\langle d_1\rangle + 1)} - 1750\right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8/\langle d\rangle \leq 10^9$, hence we can compute a safely as a TeX integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`\__fp_ep_div:wwwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle$*continuation*$\rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `\__fp_ep_div_esti:wwwwn` $\langle$*denominator*$\rangle$ $\langle$*numerator*$\rangle$, responsible for estimating the inverse of the denominator.

```
\cs_new:Npn \__fp_ep_div:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #3,#4;
      \__fp_ep_div_esti:wwwwn
    }
  }
```

(*End definition for* __fp_ep_div:wwwwn.)

__fp_ep_div_esti:wwwwn
__fp_ep_div_estii:wwnnwwn
__fp_ep_div_estiii:NNNNNwwwn

The esti function evaluates $\alpha = 10^9/(\langle d_1\rangle + 1)$, which is used twice in the expression for a, and combines the exponents #1 and #4 (with a shift by 1 because we later compute $\langle n\rangle/(10\langle d\rangle)$). Then the estii function evaluates $10^9 + a$, and puts the exponent #2 after the continuation #7: from there on we can forget exponents and focus on the mantissa. The estiii function multiplies the denominator #7 by $10^{-8}a$ (obtained as a split into the single digit #1 and two blocks of 4 digits, #2#3#4#5 and #6). The result $10^{-8}a\langle d\rangle = (1-\epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to __fp_ep_div_epsi:wnNNNNn, which computes $10^{-9}a/(1-\epsilon)$, that is, $1/(10\langle d\rangle)$ and we finally multiply this by the numerator #8.

```
\cs_new:Npn \__fp_ep_div_esti:wwwwn #1,#2#3; #4,
  {
    \exp_after:wN \__fp_ep_div_estii:wwnnwwn
    \__int_value:w \__int_eval:w 10 0000 0000 / ( #2 + 1 )
      \exp_after:wN ;
    \__int_value:w \__int_eval:w #4 - #1 + 1 ,
    {#2} #3;
  }
\cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
  {
    \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
    \__int_value:w \__int_eval:w 10 0000 0000 - 1750
      + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
    {#3}{#4}#5; #6; { #7 #2, }
  }
\cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
  {
    \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
    \__fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
    \__fp_fixed_mul:wwn
  }
```

(*End definition for* __fp_ep_div_esti:wwwwn*,* __fp_ep_div_estii:wwnnwwn*, and* __fp_ep_div_estiii:NNNNNwwwn*.*)

__fp_ep_div_epsi:wnNNNNNn
__fp_ep_div_eps_pack:NNNNNw
__fp_ep_div_epsii:wwnNNNNNn

The bounds shown above imply that the epsi function's first operand is $(1-\epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The epsi function computes ϵ as $1-(1-\epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use #1 (which is 9999). Then epsii evaluates $10^{-9}a/(1-\epsilon)$ as $(1+\epsilon^2)(1+\epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of short_mul and div_myriad is both faster and more precise than a simple mul.

```
\cs_new:Npn \__fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_ep_div_epsii:wwnNNNNNn
    \__int_value:w \__int_eval:w 1 9998 - #2
      \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
      \__int_value:w \__int_eval:w 1 9999 9998 - #3#4
        \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
        \__int_value:w \__int_eval:w 2 0000 0000 - #5#6 ; ;
  }
```

```
\cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
  { + #1 ; {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
  {
    \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn {10000} {#1} #2 ;
    {
      \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
      \__fp_fixed_div_myriad:wn
      \__fp_fixed_mul:wwn
    }
    \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
  }
```

(*End definition for `\__fp_ep_div_epsi:wnNNNNNn`, `\__fp_ep_div_eps_pack:NNNNNw`, and `\__fp_ep_div_epsii:wwnNNNNNn`.*)

27.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r, we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

`\__fp_ep_isqrt:wwn`
`\__fp_ep_isqrt_aux:wwn`
`\__fp_ep_isqrt_auxii:wwnnnwn`

First normalize the input, then check the parity of the exponent `#1`. If it is even, the result's exponent will be $-$`#1`$/2$, otherwise it will be $($`#1`$ - 1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as `#1` the result's exponent just computed, as `#2` the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as `#3` and `#4` one empty argument and one `0`, depending on the parity of the original exponent, as `#5` and `#6` the normalized mantissa (`#5` $\in [1000, 9999]$), and as `#7` the continuation. It sets up the iteration giving r: the `esti` function thus receives the initial two guesses `#2` and 0, an approximation `#5` of $10^4 x$ (its first block of digits), and the empty/zero arguments `#3` and `#4`, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```
\cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_ep_isqrt_auxi:wwn
  }
\cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
  {
    \exp_after:wN \__fp_ep_isqrt_auxii:wwnnnwn
    \__int_value:w \__int_eval:w
      \int_if_odd:nTF {#1}
        { (1 - #1) / 2 , 535 , { 0 } { } }
        { 1 - #1 / 2 , 168 , { } { 0 } }
  }
\cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
  {
    \__fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
      {#5} #6 ; { #7 #1 , }
  }
```

(*End definition for* `\__fp_ep_isqrt:wwn`*,* `\__fp_ep_isqrt_aux:wwn`*, and* `\__fp_ep_isqrt_auxii:wwnnnwn`*.*)

__fp_ep_isqrt_esti:wwwnnwn
__fp_ep_isqrt_estii:wwwnnwn
__fp_ep_isqrt_estiii:NNNNNwwwn

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev\rangle + 1.005 \cdot 10^{8 \text{ or } 9}/(\langle prev\rangle \cdot x))/2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if `#4` is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if `#4` is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `\__fp_ep_isqrt_epsi:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r.

```
\cs_new:Npn \__fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
    \fi:
    \exp_after:wN \__fp_ep_isqrt_esti:wwwnnwn
    \__int_value:w \__int_eval:w
      (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
    #1, #3, {#4}
  }
\cs_new:Npn \__fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
  {
    \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwwn
    \__int_value:w \__int_eval:w 1000 0000 + #2 * #2 #5 * 5
      \exp_after:wN , \__int_value:w \__int_eval:w 10000 + #2 ;
  }
\cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
  {
    \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
    \__fp_ep_isqrt_epsi:wN
    \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
  }
```

(*End definition for* `\__fp_ep_isqrt_esti:wwwnnwn`*,* `\__fp_ep_isqrt_estii:wwwnnwn`*, and* `\__fp_ep_isqrt_estiii:NNNNNwwwn`*.*)

__fp_ep_isqrt_epsi:wN
__fp_ep_isqrt_epsii:wwN

Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as `#1` and y as `#2`.

```
\cs_new:Npn \__fp_ep_isqrt_epsi:wN #1;
  {
    \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
  }
\cs_new:Npn \__fp_ep_isqrt_epsii:wwN #1; #2;
  {
    \__fp_fixed_mul:wwn #1; #1;
    \__fp_fixed_mul_sub_back:wwwn #2;
      {15000}{0000}{0000}{0000}{0000}{0000};
    \__fp_fixed_mul:wwn #1;
  }
```

(*End definition for* `\__fp_ep_isqrt_epsi:wN` *and* `\__fp_ep_isqrt_epsii:wwN`.)

27.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`\__fp_ep_to_float_o:wwN`
`\__fp_ep_inv_to_float_o:wwN`

An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
\cs_new:Npn \__fp_ep_to_float_o:wwN #1,
  { + \__int_eval:w #1 \__fp_fixed_to_float_o:wN }
\cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2;
  {
    \__fp_ep_div:wwwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
    \__fp_ep_to_float_o:wwN
  }
```

(*End definition for* `\__fp_ep_to_float_o:wwN` *and* `\__fp_ep_inv_to_float_o:wwN`.)

`\__fp_fixed_inv_to_float_o:wN`

Another function which reduces to converting an extended precision number to a float.

```
\cs_new:Npn \__fp_fixed_inv_to_float_o:wN
  { \__fp_ep_inv_to_float_o:wwN 0, }
```

(*End definition for* `\__fp_fixed_inv_to_float_o:wN`.)

`\__fp_fixed_to_float_rad_o:wN`

Converts the fixed point number `#1` from degrees to radians then to a floating point number. This could perhaps remain in l3fp-trig.

```
\cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1;
  {
    \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
    { \__fp_ep_to_float_o:wwN 2, }
  }
```

(*End definition for* `\__fp_fixed_to_float_rad_o:wN`.)

__fp_fixed_to_float_o:wN
__fp_fixed_to_float_o:Nw

... __int_eval:w $\langle exponent\rangle$ __fp_fixed_to_float_o:wN {$\langle a_1\rangle$} {$\langle a_2\rangle$} {$\langle a_3\rangle$} {$\langle a_4\rangle$} {$\langle a_5\rangle$} {$\langle a_6\rangle$} ; $\langle sign\rangle$

yields

$\langle exponent'\rangle$; {$\langle a'_1\rangle$} {$\langle a'_2\rangle$} {$\langle a'_3\rangle$} {$\langle a'_4\rangle$} ;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \le \langle a'_1\rangle \le 9999$. At this stage, we know that $\langle a_1\rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8.[11]

```
\cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2; { \__fp_fixed_to_float_o:wN #2; #1 }
\cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
  {
    + \__int_eval:w \c__fp_block_int % for the 8-digit-at-the-start thing.
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_fixed_to_loop:N
    \exp_after:wN \use_none:n
    \__int_value:w \__int_eval:w
      1 0000 0000 + #1   \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w   1#2 \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w 1#5#6
    \exp_after:wN ;
    \exp_after:wN ;
  }
\cs_new:Npn \__fp_fixed_to_loop:N #1
  {
    \if_meaning:w 0 #1
      - 1
      \exp_after:wN \__fp_fixed_to_loop:N
    \else:
      \exp_after:wN \__fp_fixed_to_loop_end:w
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
  {
    \if_meaning:w ; #1
      \exp_after:wN \__fp_fixed_to_float_zero:w
    \else:
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_fixed_to_float_pack:ww
      \exp_after:wN ;
    \fi:
    #1 #2 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
  {
    - 2 * \c__fp_max_exponent_int ;
    {0000} {0000} {0000} {0000} ;
  }
\cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
```

[11] Bruno: I must double check this assumption.

```
  {
    \if_int_compare:w #2 > 4 \exp_stop_f:
      \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
    \fi:
    ; #1 ;
  }
\cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \__int_value:w \__int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \__int_value:w \__int_eval:w 1 #3#4 + 1 ;
  }
```

(*End definition for* `\__fp_fixed_to_float_o:wN` *and* `\__fp_fixed_to_float_o:Nw`.)

⟨/initex | package⟩

28 l3fp-expo implementation

⟨*initex | package⟩

⟨@@=fp⟩

`\__fp_parse_word_exp:N`
`\__fp_parse_word_ln:N`

Unary functions.

```
\cs_new:Npn \__fp_parse_word_exp:N
  { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
\cs_new:Npn \__fp_parse_word_ln:N
  { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
```

(*End definition for* `\__fp_parse_word_exp:N` *and* `\__fp_parse_word_ln:N`.)

28.1 Logarithm

28.1.1 Work plan

As for many other functions, we filter out special cases in `\__fp_ln_o:w`. Then `\__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ *is such that* $0.7 \le ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \le c < 50$ such that $0.91 \le ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t\left(1 + t^2\left(\frac{1}{3} + t^2\left(\frac{1}{5} + t^2\left(\frac{1}{7} + t^2\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4)\ldots,$$

is not too difficult to compute.

28.1.2 Some constants

\c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl
\c__fp_ln_iii_fixed_tl
\c__fp_ln_iv_fixed_tl
\c__fp_ln_vi_fixed_tl
\c__fp_ln_vii_fixed_tl
\c__fp_ln_viii_fixed_tl
\c__fp_ln_ix_fixed_tl
\c__fp_ln_x_fixed_tl

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of ln(5).

```
\tl_const:Nn \c__fp_ln_i_fixed_tl    { {0000}{0000}{0000}{0000}{0000}{0000};}
\tl_const:Nn \c__fp_ln_ii_fixed_tl   { {6931}{4718}{0559}{9453}{0941}{7232};}
\tl_const:Nn \c__fp_ln_iii_fixed_tl {{10986}{1228}{8668}{1096}{9139}{5245};}
\tl_const:Nn \c__fp_ln_iv_fixed_tl  {{13862}{9436}{1119}{8906}{1883}{4464};}
\tl_const:Nn \c__fp_ln_vi_fixed_tl  {{17917}{5946}{9228}{0550}{0081}{2477};}
\tl_const:Nn \c__fp_ln_vii_fixed_tl {{19459}{1014}{9055}{3133}{0510}{5353};}
\tl_const:Nn \c__fp_ln_viii_fixed_tl{{20794}{4154}{1679}{8359}{2825}{1696};}
\tl_const:Nn \c__fp_ln_ix_fixed_tl  {{21972}{2457}{7336}{2193}{8279}{0490};}
\tl_const:Nn \c__fp_ln_x_fixed_tl   {{23025}{8509}{2994}{0456}{8401}{7991};}
```

(*End definition for* `\c__fp_ln_i_fixed_tl` *and others.*)

28.1.3 Sign, exponent, and special numbers

__fp_ln_o:w

The logarithm of negative numbers (including $-\infty$ and -0) raises the "invalid" exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a nan is itself. Positive normal numbers call `\__fp_ln_npos_o:w`.

```
\cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
    \fi:
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
    \or:
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End definition for* `\__fp_ln_o:w`.)

28.1.4 Absolute ln

__fp_ln_npos_o:w

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4}, and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```
\cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
  { %^^A todo: ln(1) should be "exact zero", not "underflow"
    \exp_after:wN \__fp_sanitize:Nw
    \__int_value:w % for the overall sign
```

```
      \if_int_compare:w #1 < 1 \exp_stop_f:
        2
      \else:
        0
      \fi:
      \exp_after:wN \exp_stop_f:
      \__int_value:w \__int_eval:w % for the exponent
        \__fp_ln_significand:NNNNnnnN #2#3
        \__fp_ln_exponent:wn {#1}
  }
```

(*End definition for* `\__fp_ln_npos_o:w`.)

`\__fp_ln_significand:NNNNnnnN`

`\__fp_ln_significand:NNNNnnnN` ⟨X_1⟩ {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*continuation*⟩

This function expands to

⟨*continuation*⟩ {⟨Y_1⟩} {⟨Y_2⟩} {⟨Y_3⟩} {⟨Y_4⟩} {⟨Y_5⟩} {⟨Y_6⟩} ;

where $Y = -\ln(X)$ as an extended fixed point.

```
\cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
  {
    \exp_after:wN \__fp_ln_x_ii:wnnnn
    \__int_value:w
      \if_case:w #1 \exp_stop_f:
      \or:
        \if_int_compare:w #2 < 4 \exp_stop_f:
          \__int_eval:w 10 - #2
        \else:
          6
        \fi:
      \or: 4
      \or: 3
      \or: 2
      \or: 2
      \or: 2
      \else: 1
      \fi:
    ; { #1 #2 #3 #4 }
  }
```

(*End definition for* `\__fp_ln_significand:NNNNnnnN`.)

`\__fp_ln_x_ii:wnnnn`

We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```
\cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
  {
    \exp_after:wN \__fp_ln_div_after:Nw
    \cs:w c__fp_ln_ \__int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
    \__int_value:w
      \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
      \__int_value:w \__int_eval:w
        \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
        \__int_value:w \__int_eval:w 9999 9990 + #1*#2#3 +
          \exp_after:wN \__fp_ln_x_iii:NNNNNNw
          \__int_value:w \__int_eval:w 10 0000 0000 + #1*#4#5 ;
    {20000} {0000} {0000} {0000}
  } %^^A todo: reoptimize (a generalization attempt failed).
\cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
  { #1#2; {#3#4#5#6} {#7} }
\cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
  {
    #1#2#3#4#5 + 1 ;
    {#1#2#3#4#5} {#6}
  }
```

The Taylor series to be used is expressed in terms of $t = (x-1)/(x+1) = 1-2/(x+1)$. We now compute the quotient with extended precision, reusing some code from `\__fp_-/_o:ww`. Note that $1+x$ is known exactly.

To reuse notations from l3fp-basics, we want to compute A/Z with $A = 2$ and $Z = x+1$. In l3fp-basics, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A, B, C, etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147\cdots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4, and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor .$$

(The $1/2$ comes from how eTeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned} 10^4 B &\leq A_1 A_2 . A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\ &\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\ &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y \end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
\end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

```
\__fp_ln_x_iv:wnnnnnnnn ⟨1 or 2⟩ ⟨8d⟩ ; {⟨4d⟩} {⟨4d⟩} ⟨fixed-tl⟩
```

The number is x. Compute y by adding 1 to the five first digits.

```
\cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \__int_value:w \__int_eval:w
    \__fp_ln_div_i:w #1 ;
      #6 #7 ; {#8} {#9}
      {#2} {#3} {#4} {#5}
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_vi:wwn \__int_value:w #1 }
  }
\cs_new:Npn \__fp_ln_div_i:w #1;
  {
    \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
    \__int_value:w \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
  }
\cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \__int_value:w \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \__int_value:w \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
      #2 #3 ;
  }
\cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \__int_value:w \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
  }
```

We now have essentially

> `\__fp_ln_div_after:Nw` ⟨*fixed tl*⟩
> `\__fp_div_significand_pack:NNN` $10^6 + Q_1$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_2$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_3$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_4$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_5$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_6$ `;`
> ⟨*exponent*⟩ `;` ⟨*continuation*⟩

where ⟨*fixed tl*⟩ holds the logarithm of a number in $[1, 10]$, and ⟨*exponent*⟩ is the exponent. Also, the expansion is done backwards. Then `\__fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

> `\__fp_ln_div_after:Nw` ⟨*fixed-tl*⟩ ⟨*1d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;`
> ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*exponent*⟩ `;`

Just as with division, we know that the first two digits are `1` and `0` because of bounds on the final result of the division $2/(x + 1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x + 1)$, after testing whether $2/(x + 1)$ is greater than or smaller than 1.

```
\cs_new:Npn \__fp_ln_div_after:Nw #1#2;
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_ln_t_small:Nw
    \else:
      \exp_after:wN \__fp_ln_t_large:NNw
      \exp_after:wN -
    \fi:
    #1
  }
\cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
  {
    \exp_after:wN \__fp_ln_t_large:NNw
    \exp_after:wN + % <sign>
    \exp_after:wN #1
    \__int_value:w \__int_eval:w 9999 - #2 \exp_after:wN ;
    \__int_value:w \__int_eval:w 9999 - #3 \exp_after:wN ;
    \__int_value:w \__int_eval:w 9999 - #4 \exp_after:wN ;
    \__int_value:w \__int_eval:w 9999 - #5 \exp_after:wN ;
    \__int_value:w \__int_eval:w 9999 - #6 \exp_after:wN ;
    \__int_value:w \__int_eval:w 1 0000 - #7 ;
  }
```

> `\__fp_ln_t_large:NNw` ⟨*sign*⟩ ⟨*fixed tl*⟩
> ⟨t_1⟩`;` ⟨t_2⟩ `;` ⟨t_3⟩`;` ⟨t_4⟩`;` ⟨t_5⟩ `;` ⟨t_6⟩`;`
> ⟨*exponent*⟩ `;` ⟨*continuation*⟩

Compute the square $\mathtt{t}^2$, and keep `t` at the end with its sign. We know that $\mathtt{t} < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `\__fp_ln_t_small:w`, they can have less than 4 digits.

```
\cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
  {
    \exp_after:wN \__fp_ln_square_t_after:w
    \__int_value:w \__int_eval:w 9999 0000 + #3*#3
      \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
      \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#4
        \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
        \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
          \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
          \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
            \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
            \__int_value:w \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
              + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
              % ; ; ;
    \exp_after:wN \__fp_ln_twice_t_after:w
    \__int_value:w \__int_eval:w -1 + 2*#3
      \exp_after:wN \__fp_ln_twice_t_pack:Nw
      \__int_value:w \__int_eval:w 9999 + 2*#4
        \exp_after:wN \__fp_ln_twice_t_pack:Nw
        \__int_value:w \__int_eval:w 9999 + 2*#5
          \exp_after:wN \__fp_ln_twice_t_pack:Nw
          \__int_value:w \__int_eval:w 9999 + 2*#6
            \exp_after:wN \__fp_ln_twice_t_pack:Nw
            \__int_value:w \__int_eval:w 9999 + 2*#7
              \exp_after:wN \__fp_ln_twice_t_pack:Nw
              \__int_value:w \__int_eval:w 10000 + 2*#8 ; ;
    { \__fp_ln_c:NwNw #1 }
    #2
  }
\cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
\cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;;; {#1} }
\cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
  { + #1#2#3#4#5 ; {#6} }
\cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
  { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
```

(*End definition for* `\__fp_ln_x_ii:wnnnn`.)

`\__fp_ln_Taylor:wwNw` Denoting $T = t^2$, we get

```
\__fp_ln_Taylor:wwNw
  {⟨T_1⟩} {⟨T_2⟩} {⟨T_3⟩} {⟨T_4⟩} {⟨T_5⟩} {⟨T_6⟩} ; ;
  {⟨(2t)_1⟩} {⟨(2t)_2⟩} {⟨(2t)_3⟩} {⟨(2t)_4⟩} {⟨(2t)_5⟩} {⟨(2t)_6⟩} ;
  { \__fp_ln_c:NwNw ⟨sign⟩ }
  ⟨fixed tl⟩ ⟨exponent⟩ ; ⟨continuation⟩
```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t\left(1 + T\left(\frac{1}{3} + T\left(\frac{1}{5} + T\left(\frac{1}{7} + T\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows

```
\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
```

```
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;
```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
\cs_new:Npn \__fp_ln_Taylor:wwNw
  { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
\cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = 1 \exp_stop_f:
      \__fp_ln_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
    \__fp_fixed_add:wwn #2;
    \__fp_fixed_mul:wwn #3;
    {
      \exp_after:wN \__fp_ln_Taylor_loop:www
      \__int_value:w \__int_eval:w #1 - 2 ;
    }
    #3;
  }
\cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
  {
    \fi:
    \exp_after:wN \__fp_fixed_mul:wwn
    \exp_after:wN { \__int_value:w \__int_eval:w 10000 + #2 } #3;
  }
```

(*End definition for* `\__fp_ln_Taylor:wwNw`.)

`\__fp_ln_c:NwNw`

```
\__fp_ln_c:NwNw ⟨sign⟩
  {⟨r1⟩} {⟨r2⟩} {⟨r3⟩} {⟨r4⟩} {⟨r5⟩} {⟨r6⟩} ;
  ⟨fixed tl⟩ ⟨exponent⟩ ; ⟨continuation⟩
```

We are now reduced to finding $\ln(c)$ and ⟨*exponent*⟩ $\ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get `b` $\ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```
\cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
  {
    \if_meaning:w + #1
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
    \fi:
    #3 #2 ;
  }
```

(*End definition for* `\__fp_ln_c:NwNw`.)

`\__fp_ln_exponent:wn`

```
\__fp_ln_exponent:wn
  {⟨s1⟩} {⟨s2⟩} {⟨s3⟩} {⟨s4⟩} {⟨s5⟩} {⟨s6⟩} ;
  {⟨exponent⟩}
```

Compute ⟨*exponent*⟩ times ln(10). Apart from the cases where ⟨*exponent*⟩ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4. Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```
\cs_new:Npn \__fp_ln_exponent:wn #1; #2
  {
    \if_case:w #2 \exp_stop_f:
      0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
    \or:
      \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
    \else:
      \if_int_compare:w #2 > 0 \exp_stop_f:
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 0
        \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
      \else:
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 2
        \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
      \fi:
    \fi:
    #2; #1;
  }
```

Now we painfully write all the cases.[12] No overflow nor underflow can happen, except when computing `ln(1)`.

```
\cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
  {
    0
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
    \__fp_fixed_to_float_o:wN 0
  }
```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```
\cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
  {
    4
    \exp_after:wN \__fp_fixed_mul:wwn
      \c__fp_ln_x_fixed_tl
      {#3}{0000}{0000}{0000}{0000}{0000} ;
    #2
      {0000}{#4}{#5}{#6}{#7}{#8};
    \__fp_fixed_to_float_o:wN #1
  }
```

(*End definition for* `\__fp_ln_exponent:wn`.)

[12] Bruno: do rounding.

28.2 Exponential

28.2.1 Sign, exponent, and special numbers

__fp_exp_o:w

```
\cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_o:Nw \c_one_fp
    \or:
      \exp_after:wN \__fp_exp_normal_o:w
    \or:
      \if_meaning:w 0 #3
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End definition for __fp_exp_o:w.*)

__fp_exp_normal_o:w
__fp_exp_pos_o:Nnwnw
__fp_exp_overflow:NN

```
\cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
  {
    \if_meaning:w 0 #1
      \__fp_exp_pos_o:NNwnw + \__fp_fixed_to_float_o:wN
    \else:
      \__fp_exp_pos_o:NNwnw - \__fp_fixed_inv_to_float_o:wN
    \fi:
  }
\cs_new:Npn \__fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
  {
    \fi:
    \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
      \token_if_eq_charcode:NNTF + #1
        { \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp }
        { \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp }
      \exp:w
    \else:
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN 0
      \__int_value:w #1 \__int_eval:w
        \if_int_compare:w #4 < 0 \exp_stop_f:
          \exp_after:wN \use_i:nn
        \else:
          \exp_after:wN \use_ii:nn
        \fi:
        {
          0
          \__fp_decimate:nNnnnn { - #4 }
            \__fp_exp_Taylor:Nnnwn
        }
        {
          \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
            \__fp_exp_pos_large:NnnNwn
        }
        #5
        {#4}
        #1 #2 0
        \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
\cs_new:Npn \__fp_exp_overflow:NN #1#2
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN #1
    \exp_after:wN #2
  }
```

(*End definition for* `\__fp_exp_normal_o:w`*,* `\__fp_exp_pos_o:Nnwnw`*, and* `\__fp_exp_overflow:NN`*.*)

`\__fp_exp_Taylor:Nnnwn`
`\__fp_exp_Taylor_loop:www`
`\__fp_exp_Taylor_break:Nww`

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```
\cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
  {
    #6
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_exp_Taylor_ii:ww
    ; #2#3#4 0000 0000 ;
  }
\cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
  { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__stop }
\cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = 1 \exp_stop_f:
      \exp_after:wN \__fp_exp_Taylor_break:Nww
    \fi:
    \__fp_fixed_div_int:wwN #3 ; #1 ;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn #2 ;
    {
      \exp_after:wN \__fp_exp_Taylor_loop:www
      \__int_value:w \__int_eval:w #1 - 1 ;
      #2 ;
    }
  }
\cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
  { \__fp_fixed_add_one:wN #2 ; }
```

(*End definition for* `\__fp_exp_Taylor:Nnnwn`*,* `\__fp_exp_Taylor_loop:www`*, and* `\__fp_exp_Taylor_break:Nww`*.*)

__fp_exp_pos_large:NnnNwn
__fp_exp_large_after:wwn
__fp_exp_large:w
__fp_exp_large_v:wN
__fp_exp_large_iv:wN
__fp_exp_large_iii:wN
__fp_exp_large_ii:wN
__fp_exp_large_i:wN
__fp_exp_large_:wN

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range $[0,5]$. Remove leading zeros from the integer part: putting `#4` in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle digit\rangle\cdot 10^{\langle exponent\rangle})$ in a table, and multiplying that to the current total. The loop is done by having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an `\__int_eval:w`), and keeping track of a fixed point number, `#1` for the numbered auxiliaries. Our usage of `\if_case:w` is somewhat dirty for optimization: TEX jumps to the appropriate case, but we then close the `\if_case:w` "by hand", using `\or:` and `\fi:` as delimiters.

```
\cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
  {
    \exp_after:wN \exp_after:wN
    \cs:w __fp_exp_large_ \__int_to_roman:w #6 :wN \exp_after:wN \cs_end:
    \exp_after:wN \c__fp_one_fixed_tl
    \__int_value:w #3 #4 \exp_stop_f:
    #5 00000 ;
  }
\cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
  { \fi: \__fp_fixed_mul:wwn #1; }
\cs_new:Npn \__fp_exp_large_v:wN #1; #2
  {
    \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
      +  8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
      + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
      + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
      + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
      + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
      + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
      + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
      + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
    \fi:
    #1;
    \__fp_exp_large_iv:wN
  }
\cs_new:Npn \__fp_exp_large_iv:wN #1; #2
  {
    \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
      +  869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
      + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
      + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
      + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
      + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
      + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
      + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
      + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
    \fi:
```

```
    #1;
    \__fp_exp_large_iii:wN
  }
\cs_new:Npn \__fp_exp_large_iii:wN #1; #2
  {
    \if_case:w #2 ~         \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
      +  87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
      + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
      + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
      + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
      + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
      + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
      + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
      + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
    \fi:
    #1;
    \__fp_exp_large_ii:wN
  }
\cs_new:Npn \__fp_exp_large_ii:wN #1; #2
  {
    \if_case:w #2 ~        \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
      +  9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
      + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
      + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
      + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
      + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
      + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
      + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
      + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
    \fi:
    #1;
    \__fp_exp_large_i:wN
  }
\cs_new:Npn \__fp_exp_large_i:wN #1; #2
  {
    \if_case:w #2 ~       \exp_after:wN \__fp_fixed_continue:wn  \or:
      + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
      + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
      + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
      + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
      + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
      + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
      + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
      + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
      + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
    \fi:
    #1;
    \__fp_exp_large_:wN
  }
\cs_new:Npn \__fp_exp_large_:wN #1; #2
  {
    \if_case:w #2 ~       \exp_after:wN \__fp_fixed_continue:wn  \or:
      + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
      + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
      + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
      + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
      + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
      + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
      + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
      + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
      + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
    \fi:
    #1;
    \__fp_exp_large_after:wwn
  }
\cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
  {
    \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
    \__fp_fixed_mul:wwn #1;
  }
```

(*End definition for* `\__fp_exp_pos_large:NnnNwn` *and others.*)

28.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty,-0)$	$-$integer	± 0	$+$integer	$(0,\infty)$	$+\infty$	NaN
$+\infty$	$+0$	$+0$		$+1$	$+\infty$		$+\infty$	NaN
$(1,\infty)$	$+0$	$+\lvert a\rvert^b$		$+1$	$+\lvert a\rvert^b$		$+\infty$	NaN
$+1$	$+1$	$+1$		$+1$	$+1$		$+1$	$+1$
$(0,1)$	$+\infty$	$+\lvert a\rvert^b$		$+1$	$+\lvert a\rvert^b$		$+0$	NaN
$+0$	$+\infty$	$+\infty$		$+1$	$+0$		$+0$	NaN
-0	$+\infty$	NaN	$(-1)^b\infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	NaN
$(-1,0)$	$+\infty$	NaN	$(-1)^b\lvert a\rvert^b$	$+1$	$(-1)^b\lvert a\rvert^b$	NaN	$+0$	NaN
-1	$+1$	NaN	$(-1)^b$	$+1$	$(-1)^b$	NaN	$+1$	NaN
$(-\infty,-1)$	$+0$	NaN	$(-1)^b\lvert a\rvert^b$	$+1$	$(-1)^b\lvert a\rvert^b$	NaN	$+\infty$	NaN
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b\infty$	NaN	$+\infty$	NaN
NaN	NaN	NaN	NaN	$+1$	NaN	NaN	NaN	NaN

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\mathtt{NaN}^0 = 1^{\mathtt{NaN}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`\__fp_^_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a, even `nan`. Then test the sign of a.

- If it is positive, and a is a normal number, call `\__fp_pow_normal_o:ww` followed by the two `fp` a and b. For $a = +0$ or $+\inf$, call `\__fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of b) and return `nan`.

- Finally, if a is negative, compute $\mathtt{a}^b$ (`\__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing { b a }, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```
\cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
    \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
  {
    \if_meaning:w 0 #4
      \__fp_case_return_o:Nw \c_one_fp
    \fi:
    \if_case:w #2 \exp_stop_f:
      \exp_after:wN \use_i:nn
    \or:
      \__fp_case_return_o:Nw \c_nan_fp
    \else:
      \exp_after:wN \__fp_pow_neg:www
      \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
    \fi:
    {
      \if_meaning:w 1 #1
        \exp_after:wN \__fp_pow_normal_o:ww
      \else:
        \exp_after:wN \__fp_pow_zero_or_inf:ww
      \fi:
      \s__fp \__fp_chk:w #1#2#3;
    }
    { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
    \s__fp \__fp_chk:w #4#5#6;
  }
```

(*End definition for `\__fp_^_o:ww`.*)

`\__fp_pow_zero_or_inf:ww` Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0, since those conveniently take the same possible values, 0 and 2. Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```
\cs_new:Npn \__fp_pow_zero_or_inf:ww
    \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
  {
    \if_meaning:w 1 #4
      \__fp_case_return_same_o:w
    \fi:
    \if_meaning:w #1 #4
      \__fp_case_return_o:Nw \c_zero_fp
    \fi:
    \if_meaning:w 2 #1
      \__fp_case_return_o:Nw \c_inf_fp
    \fi:
    \if_meaning:w 2 #3
      \__fp_case_return_o:Nw \c_inf_fp
    \else:
      \__fp_case_use:nw
```

```
        {
          \__fp_division_by_zero_o:NNww \c_inf_fp ^
            \s__fp \__fp_chk:w #1 #2 ;
        }
    \fi:
    \s__fp \__fp_chk:w #3#4
  }
```

(*End definition for* `\__fp_pow_zero_or_inf:ww`.)

`\__fp_pow_normal_o:ww`

We have in front of us a, and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1, unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise "invalid" when the sign is considered. If $|a| \neq 1$, test the type of b:

0 Impossible, we already filtered $b = \pm 0$.

1 Call `\__fp_pow_npos_o:Nww`.

2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.

3 Return b.

```
\cs_new:Npn \__fp_pow_normal_o:ww
    \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
  {
    \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
              { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
      \if_int_compare:w #4 #1 = 32 \exp_stop_f:
        \exp_after:wN \__fp_case_return_ii_o:ww
      \fi:
      \__fp_case_return_o:Nww \c_one_fp
    \fi:
    \if_case:w #4 \exp_stop_f:
    \or:
      \exp_after:wN \__fp_pow_npos_o:Nww
      \exp_after:wN #5
    \or:
      \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
      \if_int_compare:w #2 > 0 \exp_stop_f:
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_ii_o:ww
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End definition for* `\__fp_pow_normal_o:ww`.)

__fp_pow_npos_o:Nww We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of `\__fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```
\cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \__int_value:w
      \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN +
        \exp_after:wN \__fp_fixed_to_float_o:wN
      \else:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN -
        \exp_after:wN \__fp_fixed_inv_to_float_o:wN
      \fi:
      {#3}
  }
```

(*End definition for* `\__fp_pow_npos_o:Nww`*.*)

__fp_pow_npos_aux:NNnww The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x, followed by b. Compute $-\ln(x)$.

```
\cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
  {
    #1
    \__int_eval:w
      \__fp_ln_significand:NNNNnnnN #4#5
      \__fp_pow_exponent:wnN {#3}
      \__fp_fixed_mul:wwn #8 {0000}{0000} ;
      \__fp_pow_B:wwN #7;
      #1 #2 0 % fixed_to_float_o:wN
  }
\cs_new:Npn \__fp_pow_exponent:wnN #1; #2
  {
    \if_int_compare:w #2 > 0 \exp_stop_f:
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
      \exp_after:wN +
    \else:
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
      \exp_after:wN -
    \fi:
    #2; #1;
  }
\cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
  { %^^A todo: use that in ln.
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
```

```
      \exp_after:wN \__fp_pack:NNNNNw
      \__int_value:w \__int_eval:w \c__fp_middle_shift_int
        #1#2*23025 - #1 #3
        \exp_after:wN \__fp_pack:NNNNNw
        \__int_value:w \__int_eval:w \c__fp_middle_shift_int
          #1 #2*8509 - #1 #4
          \exp_after:wN \__fp_pack:NNNNNw
          \__int_value:w \__int_eval:w \c__fp_middle_shift_int
            #1 #2*2994 - #1 #5
            \exp_after:wN \__fp_pack:NNNNNw
            \__int_value:w \__int_eval:w \c__fp_middle_shift_int
              #1 #2*0456 - #1 #6
              \exp_after:wN \__fp_pack:NNNNNw
              \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
                #1 #2*8401 - #1 #7
                #1 ( #2*7991 - #8 ) / 1 0000 ; ;
  }
\cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
  {
    \if_int_compare:w #7 < 0 \exp_stop_f:
      \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
    \else:
      \if_int_compare:w #7 < 22 \exp_stop_f:
        \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
      \fi:
    \fi:
    #7 \exp_after:wN ;
    \__int_value:w \__int_eval:w 10 0000 + #1 \__int_eval_end:
    #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^^A todo: how many 0?
  }
\cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
  {
    + 2 * \c__fp_max_exponent_int
    \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
  }
\cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
    \prg_replicate:nn {#1} {0}
  }
\cs_new:Npn \__fp_pow_C_pos:w #1; 1
  { \__fp_pow_C_pos_loop:wN #1; }
\cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_pow_C_pack:w
      \exp_after:wN #2
    \else:
      \if_meaning:w 0 #2
        \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
      \fi:
      \__int_eval:w #1 - 1 \exp_after:wN ;
    \fi:
  }
\cs_new:Npn \__fp_pow_C_pack:w
  { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_tl }
```

(*End definition for* `\__fp_pow_npos_aux:NNnww`.)

`\__fp_pow_neg:www`
`\__fp_pow_neg_aux:wNN`

This function is followed by three floating point numbers: a^b, $a \in [-\infty, -0]$, and b. If b is an even integer (case -1), $a^b = \mathsf{a}^b$. If b is an odd integer (case 0), $a^b = -\mathsf{a}^b$, obtained by a call to `\__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or `nan`, in which case we return that as a^b. In particular, since the underflow detection occurs before `\__fp_pow_neg:www` is called, `(-0.1)**(12345.67)` gives $+0$ rather than complaining that the sign is not defined.

```
\cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
  {
    \if_case:w \__fp_pow_neg_case:w #4 ;
      \exp_after:wN \__fp_pow_neg_aux:wNN
    \or:
      \if_int_compare:w \__int_eval:w #1 / 2 = 1 \exp_stop_f:
        \__fp_invalid_operation_o:Nww ^ #3; #4;
        \exp:w \exp_end_continue_f:w
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_use_none_until_s:w
      \fi:
    \fi:
    \__fp_exp_after_o:w
    \s__fp \__fp_chk:w #1#2;
  }
\cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \__int_value:w \__int_eval:w 2 - #3 \__int_eval_end:
  }
```

(*End definition for* `\__fp_pow_neg:www` *and* `\__fp_pow_neg_aux:wNN`.)

`\__fp_pow_neg_case:w`
`\__fp_pow_neg_case_aux:nnnnn`
`\__fp_pow_neg_case_aux:Nnnw`

This function expects a floating point number, and determines its “parity”. It should be used after `\if_case:w` or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `\__fp_decimate:nNnnnn` the argument `#1` of `\__fp_pow_neg_case_aux:Nnnw` is a rounding digit, `0` if and only if the number was an integer, and `#3` is the 8 least significant digits of that integer.

```
\cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
  {
    \if_case:w #1 \exp_stop_f:
           -1
    \or:   \__fp_pow_neg_case_aux:nnnnn #3
    \or:   -1
    \else: 1
    \fi:
    \exp_stop_f:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
  {
    \if_int_compare:w #1 > \c__fp_prec_int
      -1
    \else:
      \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
        \__fp_pow_neg_case_aux:Nnnw
        {#2} {#3} {#4} {#5}
    \fi:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
  {
    \if_meaning:w 0 #1
      \if_int_odd:w #3 \exp_stop_f:
        0
      \else:
        -1
      \fi:
    \else:
      1
    \fi:
  }
```

(*End definition for* `\__fp_pow_neg_case:w`, `\__fp_pow_neg_case_aux:nnnnn`, *and* `\__fp_pow_neg_case_aux:Nnnw`.)

⟨/initex | package⟩

29 **l3fp-trig** Implementation

⟨*initex | package⟩

⟨@@=fp⟩

__fp_parse_word_acos:N
__fp_parse_word_acosd:N
__fp_parse_word_acsc:N
__fp_parse_word_acscd:N
__fp_parse_word_asec:N
__fp_parse_word_asecd:N
__fp_parse_word_asin:N
__fp_parse_word_asind:N
__fp_parse_word_cos:N
__fp_parse_word_cosd:N
__fp_parse_word_cot:N
__fp_parse_word_cotd:N
__fp_parse_word_csc:N
__fp_parse_word_cscd:N
__fp_parse_word_sec:N
__fp_parse_word_secd:N
__fp_parse_word_sin:N
__fp_parse_word_sind:N
__fp_parse_word_tan:N
__fp_parse_word_tand:N

Unary functions.

```
\tl_map_inline:nn
  {
    {acos} {acsc} {asec} {asin}
    {cos} {cot} {csc} {sec} {sin} {tan}
  }
  {
    \cs_new:cpx { __fp_parse_word_#1:N }
      {
        \exp_not:N \__fp_parse_unary_function:NNN
        \exp_not:c { __fp_#1_o:w }
        \exp_not:N \use_i:nn
      }
    \cs_new:cpx { __fp_parse_word_#1d:N }
      {
        \exp_not:N \__fp_parse_unary_function:NNN
        \exp_not:c { __fp_#1_o:w }
        \exp_not:N \use_ii:nn
      }
  }
```

(*End definition for* `\__fp_parse_word_acos:N` *and others.*)

__fp_parse_word_acot:N
__fp_parse_word_acotd:N
__fp_parse_word_atan:N
__fp_parse_word_atand:N

Those functions may receive a variable number of arguments.

```
\cs_new:Npn \__fp_parse_word_acot:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
\cs_new:Npn \__fp_parse_word_acotd:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
\cs_new:Npn \__fp_parse_word_atan:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
\cs_new:Npn \__fp_parse_word_atand:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
```

(*End definition for* `\__fp_parse_word_acot:N` *and others.*)

29.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0, $\pm\inf$ and `NaN`).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{\mathtt{x}}{\pi/4} \rfloor \mod 8$ (in degrees, the same formula with $\pi/4 \to 45$), the sign, and the function to compute.

29.1.1 Filtering special cases

__fp_sin_o:w

This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or `NaN` is the same float. The sine of $\pm\infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `\__fp_sin_series_o:NNwwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `\__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```
\cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_to_float_o:wwN #3 0
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_sin_o:w`.)

`\__fp_cos_o:w` The cosine of ±0 is 1. The cosine of ±∞ raises an invalid operation exception. The cosine of `NaN` is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign `0` regardless of the sign of x, and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_to_float_o:wwN 0 2
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_cos_o:w`.)

`\__fp_csc_o:w` The cosecant of ±0 is ±∞ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of ±∞ raises an invalid operation exception. The cosecant of `NaN` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign `#3`, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3\bigl(\sin|x|\bigr)^{-1}$.

```
\cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_inv_to_float_o:wwN #3 0
             }
    \or:   \__fp_case_use:nw
            { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_csc_o:w`.)

`\__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `NaN` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
    \or:   \__fp_case_use:nw
            {
              \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                \__fp_ep_inv_to_float_o:wwN 0 2
            }
    \or:   \__fp_case_use:nw
            { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_sec_o:w`.)

`\__fp_tan_o:w` The tangent of ± 0 or `NaN` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `\__fp_tan_series_o:NNwwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `\__fp_cot_o:w` for an explanation of the 0 argument.

```
\cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
            {
              \__fp_trig:NNNNNwn #1
                \__fp_tan_series_o:NNwwww 0 #3 1
            }
    \or:   \__fp_case_use:nw
            { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_tan_o:w`.)

__fp_cot_o:w
__fp_cot_zero_o:Nfw

The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw`. The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `NaN` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `\__fp_tan_series_o:NNwwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```
\cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1
                 \__fp_tan_series_o:NNwwww 2 #3 3
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
\cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
  {
    \fi:
    \token_if_eq_meaning:NNTF 0 #1
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
    {#2}
  }
```

(*End definition for* `\__fp_cot_o:w` *and* `\__fp_cot_zero_o:Nfw`.)

29.1.2 Distinguishing small and large arguments

__fp_trig:NNNNNwn

The first argument is `\use_i:nn` if the operand is in radians and `\use_ii:nn` if it is in degrees. Arguments `#2` to `#5` control what trigonometric function we compute, and `#6` to `#8` are pieces of a normal floating point number. Call the `_series` function `#2`, with arguments `#3`, either a conversion function (`\__fp_ep_to_float_o:wN` or `\__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; `#4`, a sign 0 or 2; the octant, computed in an integer expression starting with `#5` and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits `#1`, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig`/`trigd` auxiliaries receive the operand as an extended-precision number.

```
\cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
```

```
  {
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \__int_value:w \__int_eval:w #5
      \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
      \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
        #1 \__fp_trig_large:ww \__fp_trigd_large:ww
      \else:
        #1 \__fp_trig_small:ww \__fp_trigd_small:ww
      \fi:
    #7,#8{0000}{0000};
  }
```

(*End definition for* `\__fp_trig:NNNNNwn`.)

29.1.3 Small arguments

`\__fp_trig_small:ww`

This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
\cs_new:Npn \__fp_trig_small:ww #1,#2;
  { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(*End definition for* `\__fp_trig_small:ww`.)

`\__fp_trigd_small:ww`

Convert the extended-precision number to radians, then call `\__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
\cs_new:Npn \__fp_trigd_small:ww #1,#2;
  {
    \__fp_ep_mul_raw:wwwwN
      -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
    \__fp_trig_small:ww
  }
```

(*End definition for* `\__fp_trigd_small:ww`.)

29.1.4 Argument reduction in degrees

`\__fp_trigd_large:ww`
`\__fp_trigd_large_auxi:nnnnwNNNN`
`\__fp_trigd_large_auxii:wNw`
`\__fp_trigd_large_auxiii:www`

Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle \mathit{block}_1 \rangle \cdots \langle \mathit{block}_3 \rangle . \langle \mathit{block}_4 \rangle \cdots \langle \mathit{block}_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle \mathit{block}_1 \rangle + \langle \mathit{block}_2 \rangle \pmod 9$, a single digit, and prepends it to the 4 digits of $\langle \mathit{block}_3 \rangle$. It also unpacks $\langle \mathit{block}_4 \rangle$ and grabs the 4 digits of $\langle \mathit{block}_7 \rangle$. The second auxiliary grabs the $\langle \mathit{block}_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle \mathit{block}_4 \rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new ⟨*block*$_4$⟩. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `\__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various ⟨*blocks*⟩. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
\cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
  {
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
    \exp_after:wN ;
    \exp:w \exp_end_continue_f:w
    \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
    #2#3#4#5#6#7 0000 0000 0000 !
  }
\cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
  {
    \exp_after:wN \__fp_trigd_large_auxii:wNw
    \__int_value:w \__int_eval:w #1 + #2
      - (#1 + #2 - 4) / 9 * 9 \__int_eval_end:
    #3;
    #4; #5{#6#7#8#9};
  }
\cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
  {
    + (#1#2 - 4) / 9 * 2
    \exp_after:wN \__fp_trigd_large_auxiii:www
    \__int_value:w \__int_eval:w #1#2
      - (#1#2 - 4) / 9 * 9 \__int_eval_end: #3 ;
  }
\cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
  {
    \if_int_compare:w #1 < 4500 \exp_stop_f:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN \__fp_fixed_continue:wn
    \else:
      + 1
    \fi:
    \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
      {#1}#2{0000}{0000};
    { \__fp_trigd_small:ww 2, }
  }
```

(*End definition for* `\__fp_trigd_large:ww` *and others.*)

29.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π, then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \to \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100}, an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π.

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

__fp_trig_inverse_two_pi: This macro expands to `,,!` or `,!` followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 ($4 - 1$ groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```
\cs_new:Npx \__fp_trig_inverse_two_pi:
  {
    \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
    \cs:w , , !
    0000000000000000159154943091895335768883763372514362034459645740 ~
    4564487476673440588967976342265350901138027662530859560728427267 ~
    5795803689291184611457865287796741073169983922923996693740907757 ~
    3077746396925307688717392896217397661693362390241723629011832380 ~
    1142226997557159404618900869026739561204894109369378440855287230 ~
    9994644340024867234773945961089832309678307490616698646280469944 ~
    8652187881574786566964241038995874139348609983868099199962442875 ~
    5851711788584311175187671605465475369880097394603647593337680593 ~
    0249449663530532715677550322032477781639716602294674811959816584 ~
    0606016803035998133911987498832786654435279755070016240677564388 ~
    8495713108801221993761476813777647378906330680464579784817613124 ~
    2731406996077502450029775985708905690279678513152521001631774602 ~
    0924811606240561456203146484089248459191435211575407556200871526 ~
    6068022171591407574745827225977462853998751553293908139817724093 ~
    5825479707332871904069997590765770784934703935898280871734256403 ~
    6689511662545705943327631268650026122717971153211259950438667945 ~
    0376255608363171169525975812822494162333431451061235368785631136 ~
    3669216714206974696012925057833605311960859450983955671870995474 ~
    6510431623815517580839442979970999505254387566129445883306846050 ~
    7852915151410404892988506388160776196993073410389995786918905980 ~
    9373777206187543222718930136625526123878038753888110681406765434 ~
    0828278526933426799556070790386060352738996245125995749276297023 ~
    5940955843011648296411855777124057544494570217897697924094903272 ~
    9477021664960356531815354400384068987471769158876319096650696440 ~
    4776970687683656778104779795450353395758301881838687937766124814 ~
    9530599655802190835987510351271290432315804987196868777594656634 ~
    6221034204440855497850379273869429353661937782928735937843470323 ~
```

```
0237145837923557118636341929460183182291964165008783079331353497 ~
7909974586492902674506098936890945883050337030538054731232158094 ~
3197676032283131418980974982243833517435698984750103950068388003 ~
9786723599608024002739010874954854787923568261139948903268997427 ~
0834961149208289037767847430355045684560836714793084567233270354 ~
8539255620208683932409956221175331839402097079357077496549880868 ~
6066360968661967037474542102831219251846224834991161149566556037 ~
9696761399312829960776082779901007830360023382729879085402387615 ~
5744543092601191005433799838904654921248295160707285300522721023 ~
6017523313173179759311050328155109373913639645305792607180083617 ~
9548767246459804739772924481092009371257869183328958862839904358 ~
6866663975673445140950363732719174311388066383072592302759734506 ~
0548212778037065337783032170987734966568490800326988506741791464 ~
6835082816168533143361607309951498531198197337584442098416559541 ~
5225064339431286444038388356150879771645017064706751877456059160 ~
8716857857939226234756331711132998655941596890719850688744230057 ~
5191977056900382183925622033874235362568083541565172971088117217 ~
9593683256488518749974870855311659830610139214454460161488452770 ~
2511411070248521739745103866736403872860099674893173561812071174 ~
0478899368886556923078485023057057144063638632023685201074100574 ~
8592281115721968003978247595300166958522123034641877365043546764 ~
6456565971901123084767099309708591283646669191776938791433315566 ~
5066981321641521008957117286238426070678451760111345080069947684 ~
2235698962488051577598095339708085475059753626564903439445420581 ~
7886435683042000315095594743439252544850674914290864751442303321 ~
3324569511634945677539394240360905438335528292434220349484366151 ~
4663228602477666660495314065734357553014090827988091478669343492 ~
2737602634997829957018161964321233140475762897484082891174097478 ~
2637899181699939487497715198981872666294601830539583275209236350 ~
6853889228468247259972528300766856937583659722919824429747406163 ~
8183113958306744348516928597383237392662402434501997809940402189 ~
6134834273613676449913827154166063424829363741850612261086132119 ~
9863346284709941839942742955915628333990480382117501161211667205 ~
1912579303552929241134403116134112495318385926958490443846807849 ~
0973982808855297045153053991400988698840883654836652224668624087 ~
2540140400911787421220452307533473972538149403884190586842311594 ~
6322744339066125162393106283195323883392131534556381511752035108 ~
7459558201123754359768155340187407394340363397803881721004531691 ~
8295194879591767395417787924352761740724605939160273228287946819 ~
3649128949714953432552723591659298072479985806126900733218844526 ~
7943350455801952492566306204876616134365339920287545208555344144 ~
0990512982727454659118132223284051166615650709837557433729548631 ~
2041121716380915606161165732000083306114606181280326258695951602 ~
4632166138576614804719932707771316441201594960110632830520759583 ~
4850305079095584982982186740289838551383239570208076397550429225 ~
9847647071016426974384504309165864528360324933604354657237557916 ~
1366324120457809969715663402215880545794313282780055246132088901 ~
8742121092448910410052154968097113720754005710963406643135745439 ~
9159769435788920793425617783022237011486424925239248728713132021 ~
7667360756645598272609574156602343787436291321097485897150713073 ~
9104072643541417970572226547980381512759579124002534468048220261 ~
7342299001020483062463033796474678190501811830375153802879523433 ~
4195502135689770912905614317878792086205744999257897569018492103 ~
2420647138519113881475640209760554895793785141404145305151583964 ~
```

```
2823265406020603311891586570272086250269916393751527887360608114 ~
5569484210322407772727421651364234366992716340309405307480652685 ~
0930165892136921414312937134106157153714062039784761842650297807 ~
8606266969960809184223476335047746719017450451446166382846208240 ~
8673595102371302904443779408535034454426334130626307459513830310 ~
2293146934466832851766328241515210179422644395718121717021756492 ~
1964449396532222187658488244511909401340504432139858628621083179 ~
3939608443898019147873897723310286310131486955212620518278063494 ~
5711866277825659883100535155231665984394090221806314454521212978 ~
9734471488741258268223860236027109981191520568823472398358013366 ~
0683786328867928619732367253606685216856320119489780733958419190 ~
6659583867852941241871821727987506103946064819585745620060892122 ~
8416394373846549589932028481236433466119707324309545859073361878 ~
6290631850165106267576851216357588696307451999220010776676830946 ~
9814975622682434793671310841210219520899481912444048751171059184 ~
4139907889455775184621619041530934543802808938628073237578615267 ~
7971143323241969857805637630180884386640607175368321362629671224 ~
2609428540110963218262765120117022552929289655594608204938409069 ~
0760692003954646191640021567336017909631872891998634341086903200 ~
5796637103128612356988817640364252540837098108148351903121318624 ~
7228181050845123690190646632235938872454630737272808789830041018 ~
9485913673742589418124056729191238003306344998219631580386381054 ~
2457893450084553280313511884341007373060595654437362488771292628 ~
9807423539074061786905784443105274262641767830058221486462289361 ~
9296692992033046693328438158053564864073184440599549689353773183 ~
6726613130108623588021288043289344562140479789454233736058506327 ~
0439981932635916687341943656783901281912202816229500333012236091 ~
8587559201959081224153679499095448881099758919890811581163538891 ~
6339402923722049848375224236209100834097566791710084167957022331 ~
7897107102928884897013099533995424415335060625843921452433864640 ~
3432440657317477553405404481006177612569084746461432976543900008 ~
3826521145210162366431119798731902751191441213616962045693602633 ~
6102355962140467029012156796418735746835873172331004745963339773 ~
2477044918885134415363760091537564267438450166221393719306748706 ~
2881595464819775192207710236743289062690709117919412776212245117 ~
2354677115640433357720616661564674474627305622913332030953340551 ~
3841718194605321501426328000879551813296754972846701883657425342 ~
5016994231069156343106626043412205213831587971115075454063290657 ~
0248488648697402872037259869281149360627403842332874942332178578 ~
7750735571857043787379693402336902911446961448649769719434527467 ~
4429603089437192540526658890710662062575509930379976658367936112 ~
8137451104971506153783743579555867972129358764463093757203221320 ~
2460565661129971310275869112846043251843432691552928458573495971 ~
5042565399302112184947232132380516549802909919676815118022483192 ~
5127372199792134331067642187484426215985121676396779352982985195 ~
8545392106957880586853123277545433229161989053189053725391582222 ~
9232597278133427818256064882333760719681014481453198336237910767 ~
1255017528826351836492103572587410356573894694875444694018175923 ~
0609370828146501857425324969212764624247832210765473750568198834 ~
5641035458027261252285503154325039591848918982630498759115406321 ~
0354263890012837426155187877318375862355175378506956599570028011 ~
5841258870150030170259167463020842412449128392380525772514737141 ~
2310230172563968305553583262840383638157686828464330456805994018 ~
7001071952092970177990583216417579868116586547147748964716547948 ~
```

```
    8312140431836079844314055731179349677763739898930227765607058530 ~
    4083747752640947435070395214524701683884070908706147194437225650 ~
    2823145872995869738316897126851939042297110721350756978037262545 ~
    8141095038270388987364516284820180468288205829135339013835649144 ~
    3004015706509887926715417450706686888783438055583501196745862340 ~
    8059532724727843829259395771584036885940989939255241688378793572 ~
    7967951654076673927031256418760962190243046993485989199060012977 ~
    7469214532970421677817261517850653008552559997940209969455431545 ~
    2745856704403686680428648404512881182309793496962721836492935516 ~
    2029872469583299481932978335803459023227052612542114437084359584 ~
    9443383638388317751841160881711251279233374577219339820819005406 ~
    3292937775306906607415304997682647124407768817248673421685881509 ~
    9133422075930947173855159340808957124410634720893194912880783576 ~
    3115829400549708918023366596077070927599010527028150868897828549 ~
    4340372642729262103487013992868853550062061514343078665396085995 ~
    0058714939141652065302070085265624074703660736605333805263766757 ~
    2018839497277047222153633851135483463624619855425993871933367482 ~
    0422097449956672702505446423243957506869591330193746919142980999 ~
    3424230550172665212092414559625960554427590951996824313084279693 ~
    7113207021049823238195747175985519501864630940297594363194450091 ~
    9150616049228764323192129703446093584259267276386814363309856853 ~
    2786024332141052330760658841495858718197071242995959226781172796 ~
    4438853796763139274314227953114500064922126500133268623021550837
    \cs_end:
  }
```

(*End definition for* `\__fp_trig_inverse_two_pi:`.)

`\__fp_trig_large:ww`
`\__fp_trig_large_auxi:wwwwww`
`\__fp_trig_large_auxii:ww`
`\__fp_trig_large_auxiii:wNNNNNNNN`
`\__fp_trig_large_auxiv:wN`

The exponent `#1` is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$, that is, the first `#1` digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to $x/(2\pi)$. The `auxii` auxiliary discards 64 digits at a time thanks to spaces inserted in the result of `\__fp_trig_inverse_two_pi:`, while `auxiii` discards 8 digits at a time, and `auxiv` discards digits one at a time. Then 64 digits are packed into groups of 4 and the `auxv` auxiliary is called.

```
\cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
  {
    \exp_after:wN \__fp_trig_large_auxi:wwwwww
    \__int_value:w \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
    \__int_value:w \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
    \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
    {#2}{#3}{#4}{#5} ;
  }
\cs_new:Npn \__fp_trig_large_auxi:wwwwww #1, #2, #3, #4!
  {
    \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
    \prg_replicate:nn { #2 - #1 * 8 }
      { \__fp_trig_large_auxiii:wNNNNNNNN }
    \prg_replicate:nn { #3 - #2 * 8 }
      { \__fp_trig_large_auxiv:wN }
    \prg_replicate:nn { 8 } { \__fp_pack_twice_four:wNNNNNNNN }
    \__fp_trig_large_auxv:www
    ;
  }
\cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
```

```
\cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
  #1; #2#3#4#5#6#7#8#9 { #1; }
\cs_new:Npn \__fp_trig_large_auxiv:wN #1; #2 { #1; }
```

(*End definition for* `\__fp_trig_large:ww` *and others.*)

__fp_trig_large_auxv:www
__fp_trig_large_auxvi:wnnnnnnnn
__fp_trig_large_pack:NNNNNw

First come the first 64 digits of the fractional part of $10^{\#1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then some more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.,* `\__fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```
\cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxvii:w
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { 13 }
        { \__fp_trig_large_auxvi:wnnnnnnnn }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      \__fp_use_i_until_s:nw
      ; #3 #1 ; ;
  }
\cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \__int_value:w \__int_eval:w \c__fp_middle_shift_int
      + #2*#9 + #3*#8 + #4*#7 + #5*#6
      #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
  }
\cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
  { + #1#2#3#4#5 ; #6 }
```

(*End definition for* `\__fp_trig_large_auxv:www`*,* `\__fp_trig_large_auxvi:wnnnnnnnn`*, and* `\__fp_trig_large_pack:NNNNNw`*.*)

__fp_trig_large_auxvii:w
__fp_trig_large_auxviii:w
__fp_trig_large_auxix:Nw
__fp_trig_large_auxx:wNNNNN
__fp_trig_large_auxxi:w

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3`/125, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `\__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```
\cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
```

```
  {
    \exp_after:wN \__fp_trig_large_auxviii:ww
    \__int_value:w \__int_eval:w (#1#2#3 - 62) / 125 ;
    #1#2#3
  }
\cs_new:Npn \__fp_trig_large_auxviii:ww #1;
  {
    + #1
    \if_int_odd:w #1 \exp_stop_f:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN -
    \else:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN +
    \fi:
  }
\cs_new:Npn \__fp_trig_large_auxix:Nw
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxxi:w
    \__int_value:w \__int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { 13 }
        { \__fp_trig_large_auxx:wNNNNN }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      ;
  }
\cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \__int_value:w \__int_eval:w \c__fp_middle_shift_int
      #2 8 * #3#4#5#6
      #1; #2
  }
\cs_new:Npn \__fp_trig_large_auxxi:w #1;
  {
    \exp_after:wN \__fp_ep_mul_raw:wwwwN
    \__int_value:w \__int_eval:w 0 \__fp_ep_to_ep_loop:N #1 ; ; !
    0,{7853}{9816}{3397}{4483}{0961}{5661};
    \__fp_trig_small:ww
  }
```

(*End definition for* `\__fp_trig_large_auxvii:w` *and others.*)

29.1.6 Computing the power series

__fp_sin_series_o:NNwwww
__fp_sin_series_aux_o:NNnwww

Here we receive a conversion function `\__fp_ep_to_float_o:wwN` or `\__fp_ep_inv_to_float_o:wwN`, a ⟨*sign*⟩ (0 or 2), a (non-negative) ⟨*octant*⟩ delimited by a dot, a ⟨*fixed point*⟩ number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function `#1`;
- the final sign, which depends on the octant `#3` and the sign `#2`;
- the octant `#3`, which controls the series we use;

- the square `#4 * #4` of the argument as a fixed point number, computed with `\__fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in $\{1, 2, 5, 6, \dots\}$, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2\left(\frac{1}{2!} - x^2\left(\frac{1}{4!} - x^2\left(\cdots\right)\right)\right).$$

Otherwise, the series

$$\sin(x) = x\left(1 - x^2\left(\frac{1}{3!} - x^2\left(\frac{1}{5!} - x^2\left(\cdots\right)\right)\right)\right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `\__fp_sanitize:Nw` checks for overflow and underflow.

```
\cs_new:Npn \__fp_sin_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_sin_series_aux_o:NNnwww
      \exp_after:wN #1
      \__int_value:w
        \if_int_odd:w \__int_eval:w (#3 + 2) / 4 \__int_eval_end:
          #2
        \else:
          \if_meaning:w #2 0 2 \else: 0 \fi:
        \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
  {
    \if_int_odd:w \__int_eval:w #3 / 2 \__int_eval_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { % 1/18!
      \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0001}{5619}{2070};
                                    #4;{0000}{0000}{0000}{0477}{9477}{3324};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
      \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
      \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
      \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_fixed_continue:wn 0, }
    }
    { % 1/17!
      \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0028}{1145}{7254};
                                    #4;{0000}{0000}{0000}{7647}{1637}{3182};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
      \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
      \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
      \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_ep_mul:wwwwn 0, } #5,#6;
    }
    {
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN #2
      \__int_value:w \__int_eval:w #1
    }
    #2
  }
```

(*End definition for* `\__fp_sin_series_o:NNwwww` *and* `\__fp_sin_series_aux_o:NNnwww`.)

`\__fp_tan_series_o:NNwwww`
`\__fp_tan_series_aux_o:Nnwww`

Contrarily to `\__fp_sin_series_o:NNwwww` which received a conversion auxiliary as `#1`, here, `#1` is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant `#3` starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor(\texttt{\#3}+1)/2\rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first `\__int_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1-x^2(a_1-x^2(a_2-x^2(a_3-x^2(a_4-x^2a_5)))))}{1-x^2(b_1-x^2(b_2-x^2(b_3-x^2(b_4-x^2b_5))))}.$$

The ratio is computed by `\__fp_ep_div:wwwwn`, then converted to a floating point number. For octants `#3` (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```
\cs_new:Npn \__fp_tan_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_tan_series_aux_o:Nnwww
      \__int_value:w
        \if_int_odd:w \__int_eval:w #3 / 2 \__int_eval_end:
          \exp_after:wN \reverse_if:N
        \fi:
        \if_meaning:w #1#2 2 \else: 0 \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
  {
```

```
  \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{1527}{3493}{0856}{7059};
                              #3; {0000}{0159}{6080}{0274}{5257}{6472};
  \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
  \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
  \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
  \__fp_fixed_mul_sub_back:wwwn #3;{10000}{0000}{0000}{0000}{0000}{0000};
  { \__fp_ep_mul:wwwwn 0, } #4,#5;
  {
    \__fp_fixed_mul_sub_back:wwwn    {0000}{0007}{0258}{0681}{9408}{4706};
                                #3;{0000}{2343}{7175}{1399}{6151}{7670};
    \__fp_fixed_mul_sub_back:wwwn #3;{0019}{2638}{4588}{9232}{8861}{3691};
    \__fp_fixed_mul_sub_back:wwwn #3;{0536}{6357}{0691}{4344}{6852}{4252};
    \__fp_fixed_mul_sub_back:wwwn #3;{5263}{1578}{9473}{6842}{1052}{6315};
    \__fp_fixed_mul_sub_back:wwwn#3;{10000}{0000}{0000}{0000}{0000}{0000};
    {
      \reverse_if:N \if_int_odd:w
          \__int_eval:w (#2 - 1) / 2 \__int_eval_end:
        \exp_after:wN \__fp_reverse_args:Nww
      \fi:
      \__fp_ep_div:wwwwn 0,
    }
  }
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w \__fp_ep_to_float_o:wwN
  }
  #1
}
```

(*End definition for* `\__fp_tan_series_o:NNwwww` *and* `\__fp_tan_series_aux_o:Nnwww`*.*)

29.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arccosecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\operatorname{atan}(y, x) = \operatorname{atan}(y/x)$ for generic y and x. Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of atan as

$$
\begin{aligned}
\operatorname{acos} x &= \operatorname{atan}(\sqrt{1-x^2}, x) && (5)\\
\operatorname{asin} x &= \operatorname{atan}(x, \sqrt{1-x^2}) && (6)\\
\operatorname{asec} x &= \operatorname{atan}(\sqrt{x^2-1}, 1) && (7)\\
\operatorname{acsc} x &= \operatorname{atan}(1, \sqrt{x^2-1}) && (8)\\
\operatorname{atan} x &= \operatorname{atan}(x, 1) && (9)\\
\operatorname{acot} x &= \operatorname{atan}(1, x). && (10)
\end{aligned}
$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x, because $\operatorname{atan}(y,x) = \operatorname{atan}(y/x)$ is the angular coordinate of the point (x,y).

As for direct trigonometric functions, the first step in computing $\operatorname{atan}(y,x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x,|y|)$ can lie, of angular size roughly $\pi/8$, characterized by their "octant", between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y>0$: otherwise replace y by $-y$ below):

0 $0<|y|<0.41421x$, then $\operatorname{atan}\frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0<0.41421x<|y|<x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan}\frac{x-|y|}{x+|y|}$;

2 $0<0.41421|y|<x<|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan}\frac{-x+|y|}{x+|y|}$;

3 $0<x<0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan}\frac{x}{|y|}$;

4 $0<-x<0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan}\frac{-x}{|y|}$;

5 $0<0.41421|y|<-x<|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan}\frac{x+|y|}{-x+|y|}$;

6 $0<-0.41421x<|y|<-x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan}\frac{-x-|y|}{-x+|y|}$;

7 $0<|y|<-0.41421x$, then $\operatorname{atan}\frac{|y|}{x} = \pi - \operatorname{atan}\frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

29.2.1 Arctangent and arccotangent

__fp_atan_o:Nw
__fp_acot_o:Nw
__fp_atan_dispatch_o:NNnNw

The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result should be given in radians or in degrees. Here, we dispatch according to the number of arguments. The one-argument versions of arctangent and arccotangent are special cases of the two-argument ones: $\operatorname{atan}(y) = \operatorname{atan}(y,1) = \operatorname{acot}(1,y)$ and $\operatorname{acot}(x) = \operatorname{atan}(1,x) = \operatorname{acot}(x,1)$.

```
\cs_new:Npn \__fp_atan_o:Nw
  {
    \__fp_atan_dispatch_o:NNnNw
      \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
  }
\cs_new:Npn \__fp_acot_o:Nw
  {
    \__fp_atan_dispatch_o:NNnNw
      \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
  }
\cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
  {
    \if_case:w
      \__int_eval:w \__fp_array_count:n {#5} - 1 \__int_eval_end:
         \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
         \exp:w
    \or: #2 #4 #5 \exp:w
    \else:
      \__msg_kernel_expandable_error:nnnnn
        { kernel } { fp-num-args } { #3() } { 1 } { 2 }
      \exp_after:wN \c_nan_fp \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
```

(*End definition for* `\__fp_atan_o:Nw`*,* `\__fp_acot_o:Nw`*, and* `\__fp_atan_dispatch_o:NNnNw`*.*)

`\__fp_atanii_o:Nww`
`\__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `\__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `\__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm 3\pi/4\}$ (in degrees, $\{\pm 45, \pm 135\}$). Otherwise, one is much bigger than the other, and we call `\__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ± 90), or 0, leading to $\{\pm 0, \pm\pi\}$ (in degrees, $\{\pm 0, \pm 180\}$). Since $\operatorname{acot}(x, y) = \operatorname{atan}(y, x)$, `\__fp_acotii_o:ww` simply reverses its two arguments.

```
\cs_new:Npn \__fp_atanii_o:Nww
    #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5
  {
    \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
    \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
    \if_case:w
      \if_meaning:w #2 #5
        \if_meaning:w 1 #2 10 \else: 0 \fi:
      \else:
        \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
      \fi:
      \exp_stop_f:
         \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
    \fi:
    \__fp_atan_normal_o:NNnwNnw #1
    \s__fp \__fp_chk:w #2#3#4;
    \s__fp \__fp_chk:w #5
  }
\cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
  { \__fp_atanii_o:Nww #1#3; #2; }
```

(*End definition for* `\__fp_atanii_o:Nww` *and* `\__fp_acotii_o:Nww`*.*)

`\__fp_atan_inf_o:NNNw`

This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is `NaN`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `\__fp_atan_combine_o:NwwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ is computed to be 0, and the result is $[\texttt{#3}/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \texttt{#3})/2] \cdot \pi/4$ for negative x, where the divisions are rounded up.

```
\cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
```

```
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #2
    \__int_value:w \__int_eval:w
      \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
    \c__fp_one_fixed_tl
    {0000}{0000}{0000}{0000}{0000}{0000};
    0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
  }
```

(*End definition for* `\__fp_atan_inf_o:NNNw`*.*)

`\__fp_atan_normal_o:NNnwNnw`

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```
\cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
    #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
  {
    \__fp_atan_test_o:NwwNwwN
      #2 #3, #4{0000}{0000};
      #5 #6, #7{0000}{0000}; #1
  }
```

(*End definition for* `\__fp_atan_normal_o:NNnwNnw`*.*)

`\__fp_atan_test_o:NwwNwwN`

This receives: the sign `#1` of y, its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x. We prepare to call `\__fp_atan_combine_o:NwwwwwN` which expects the sign `#1`, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \cdots$, and the value of z, both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place `#1` as a first argument, and start an integer expression for the octant. The sign of x does not affect z, so we simply leave a contribution to the octant: $\langle octant\rangle \to 7 - \langle octant\rangle$ for negative x. Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert $3-$ in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `\__fp_atan_div:wnwwnw` after the operands have been ordered.

```
\cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      \if_meaning:w 2 #4
        7 - \__int_eval:w
      \fi:
      \if_int_compare:w
          \__fp_ep_compare:wwww #2,#3; #5,#6; > 0 \exp_stop_f:
        3 -
        \exp_after:wN \__fp_reverse_args:Nww
      \fi:
      \__fp_atan_div:wnwwnw #2,#3; #5,#6;
  }
```

(*End definition for* `\__fp_atan_test_o:NwwNwwN`*.*)

__fp_atan_div:wnwwnw
__fp_atan_near:wwwn
__fp_atan_near_aux:wwn

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are "near", hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7− and 3− inserted earlier) and we wish to compute atan $\frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute atan $\frac{a}{b}$. In any case, call `\__fp_atan_auxi:ww` followed by z, as a comma-delimited exponent and a fixed point number.

```
\cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
  {
    \if_int_compare:w
      \__int_eval:w 41421 * #5 < #2 000
        \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
      \exp_stop_f:
      \exp_after:wN \__fp_atan_near:wwwn
    \fi:
    0
    \__fp_ep_div:wwwwn #1,{#2}#3; #4,{#5}#6;
    \__fp_atan_auxi:ww
  }
\cs_new:Npn \__fp_atan_near:wwwn
    0 \__fp_ep_div:wwwwn #1,#2; #3,
  {
    1
    \__fp_ep_to_fixed:wwn #1 - #3, #2;
    \__fp_atan_near_aux:wwn
  }
\cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
  {
    \__fp_fixed_add:wwn #1; #2;
    { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwwn 0, } 0, }
  }
```

(*End definition for* `\__fp_atan_div:wnwwnw`*,* `\__fp_atan_near:wwwn`*, and* `\__fp_atan_near_aux:wwn`*.*)

__fp_atan_auxi:ww
__fp_atan_auxii:w

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to `\__fp_atan_Taylor_loop:www`, followed by the fixed point representation of z and the old representation.

```
\cs_new:Npn \__fp_atan_auxi:ww #1,#2;
  { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
\cs_new:Npn \__fp_atan_auxii:w #1;
  {
    \__fp_fixed_mul:wwn #1; #1;
    {
      \__fp_atan_Taylor_loop:www 39 ;
        {0000}{0000}{0000}{0000}{0000}{0000} ;
    }
    ! #1;
  }
```

(*End definition for* `\__fp_atan_auxi:ww` *and* `\__fp_atan_auxii:w`*.*)

__fp_atan_Taylor_loop:www
__fp_atan_Taylor_break:w

We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has `#1` $= 2k - 1$, `#2` $= \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\cdots - z^2\frac{1}{39}))$, and `#3` $= z^2$. To go to the next step $k \to k - 1$, we compute $\frac{1}{2k-1}$, then subtract from it z^2 times `#2`. The loop stops when $k = 0$: then `#2` is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result `#2` afterwards.

```
\cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = -1 \exp_stop_f:
      \__fp_atan_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
    \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
    {
      \exp_after:wN \__fp_atan_Taylor_loop:www
      \__int_value:w \__int_eval:w #1 - 2 ;
    }
    #3;
  }
\cs_new:Npn \__fp_atan_Taylor_break:w
    \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
  { \fi: ; #2 ; }
```

(*End definition for* `\__fp_atan_Taylor_loop:www` *and* `\__fp_atan_Taylor_break:w`.)

`\__fp_atan_combine_o:NwwwwwN`
`\__fp_atan_combine_aux:ww`

This receives a ⟨*sign*⟩, an ⟨*octant*⟩, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z, and another representation of z, as an ⟨*exponent*⟩ and the fixed point number $10^{-\langle exponent\rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign\rangle \left(\left\lceil \frac{\langle octant\rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant\rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \tag{11}$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z. The floating point result is passed to `\__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent `#5` for `\__fp_sanitize:Nw`, and multiply `#3` $= \frac{\operatorname{atan} z}{z}$ with `#6`, the adjusted z. Otherwise, multiply `#3` $= \frac{\operatorname{atan} z}{z}$ with `#4` $= z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product `#3` $\cdot$ `#4`. In both cases, convert to a floating point with `\__fp_fixed_to_float_o:wN`.

```
\cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \__int_value:w \__int_eval:w
      \if_meaning:w 0 #2
        \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \use_ii:nn
      \fi:
      { #5 \__fp_fixed_mul:wwn #3; #6; }
      {
        \__fp_fixed_mul:wwn #3; #4;
        {
          \exp_after:wN \__fp_atan_combine_aux:ww
          \__int_value:w \__int_eval:w #2 / 2 ; #2;
        }
      }
      { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
      #1
  }
\cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
  {
    \__fp_fixed_mul_short:wwn
      {7853}{9816}{3397}{4483}{0961}{5661};
      {#1}{0000}{0000};
    {
      \if_int_odd:w #2 \exp_stop_f:
        \exp_after:wN \__fp_fixed_sub:wwn
      \else:
        \exp_after:wN \__fp_fixed_add:wwn
      \fi:
    }
  }
```

(*End definition for* `\__fp_atan_combine_o:NwwwwwN` *and* `\__fp_atan_combine_aux:ww`.)

29.2.2 Arcsine and arccosine

`\__fp_asin_o:w` Again, the first argument provided by l3fp-parse is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ±0 or `NaN` is the same floating point number. The arcsine of ±∞ raises an invalid operation exception. Otherwise, call an auxiliary common with `\__fp_acos_o:w`, feeding it information about what function is being performed (for "invalid operation" exceptions).

```
\cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_same_o:w
    \or:
      \__fp_case_use:nw
        { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_asin_o:w`.)

`\__fp_acos_o:w` The arccosine of ±0 is $\pi/2$ (in degrees, 90). The arccosine of ±∞ raises an invalid operation exception. The arccosine of `NaN` is itself. Otherwise, call an auxiliary common with `\__fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```
\cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
    \or:
      \__fp_case_use:nw
        {
          \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
            \__fp_reverse_args:Nww
        }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_acos_o:w`.)

__fp_asin_normal_o:NfwNnnnnw

If the exponent `#5` is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call `\__fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that `#5` ≥ 1, `#6#7` ≥ 10000000, `#8#9` ≥ 0, with equality only for ± 1), we also call `\__fp_asin_auxi_o:NnNww`. Otherwise, `\__fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```
\cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
     #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
  {
    \if_int_compare:w #5 < 1 \exp_stop_f:
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \__fp_use_i:ww
    \__fp_invalid_operation_o:fw {#2}
      \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
    \__fp_asin_auxi_o:NnNww
      #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
  }
```

(*End definition for* `\__fp_asin_normal_o:NfwNnnnnw`.)

__fp_asin_auxi_o:NnNww
__fp_asin_isqrt:wn

We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x=1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`\__fp_reverse_args:Nww` in that case) before `\__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```
\cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
```

```
  {
    \__fp_ep_to_fixed:wwn #4,#5;
    \__fp_asin_isqrt:wn
    \__fp_ep_mul:wwwwn #4,#5;
    \__fp_ep_to_ep:wwN
    \__fp_fixed_continue:wn
    { #2 \__fp_atan_test_o:NwwNwwN #3 }
    0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
  }
\cs_new:Npn \__fp_asin_isqrt:wn #1;
  {
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl #1;
    {
      \__fp_fixed_add_one:wN #1;
      \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
    }
    \__fp_ep_isqrt:wwn
  }
```

(*End definition for* `\__fp_asin_auxi_o:NnNww` *and* `\__fp_asin_isqrt:wn`*.*)

29.2.3 Arccosecant and arcsecant

`\__fp_acsc_o:w` Cases are mostly labelled by `#2`, except when `#2` is 2: then we use `#3#2`, which is $02 = 2$ when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of `NaN` is itself. Otherwise, `\__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```
\cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
           \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
    \or:   \__fp_case_use:nw
             { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
    \or:   \__fp_case_return_o:Nw \c_zero_fp
    \or:   \__fp_case_return_same_o:w
    \else: \__fp_case_return_o:Nw \c_minus_zero_fp
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_acsc_o:w`*.*)

`\__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of `NaN` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `\__fp_reverse_args:Nww` following precisely that appearing in `\__fp_acos_o:w`.

```
\cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
    \or:
      \__fp_case_use:nw
        {
          \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
            \__fp_reverse_args:Nww
        }
    \or:   \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_asec_o:w`.)

__fp_acsc_normal_o:NfwNnw

If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `\__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\operatorname{acsc}(x) = \operatorname{asin}(1/x)$ and $\operatorname{asec}(x) = \operatorname{acos}(1/x)$.

```
\cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
  {
    \int_compare:nNnTF {#5} < 1
      {
        \__fp_invalid_operation_o:fw {#2}
          \s__fp \__fp_chk:w 1#4{#5}#6;
      }
      {
        \__fp_ep_div:wwwwn
          1,{1000}{0000}{0000}{0000}{0000}{0000};
          #5,#6{0000}{0000};
        { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
      }
  }
```

(*End definition for* `\__fp_acsc_normal_o:NfwNnw`.)

```
⟨/initex | package⟩
```

30 l3fp-convert implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

30.1 Trimming trailing zeros

__fp_trim_zeros:w
__fp_trim_zeros_loop:w
__fp_trim_zeros_dot:w
__fp_trim_zeros_end:w

If `#1` ends with a 0, the `loop` auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same `loop` auxiliary. Once the last trailing zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```
\cs_new:Npn \__fp_trim_zeros:w #1 ;
  {
    \__fp_trim_zeros_loop:w #1
      ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
  }
\cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
```

```
\cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
\cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }
```

(*End definition for* `\__fp_trim_zeros:w` *and others.*)

30.2 Scientific notation

\fp_to_scientific:N
\fp_to_scientific:c
\fp_to_scientific:n

The three public functions evaluate their argument, then pass it to `\__fp_to_scientific_dispatch:w`.

```
\cs_new:Npn \fp_to_scientific:N #1
  { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_scientific:N { c }
\cs_new:Npn \fp_to_scientific:n
  {
    \exp_after:wN \__fp_to_scientific_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End definition for* `\fp_to_scientific:N` *and* `\fp_to_scientific:n`*. These functions are documented on page 180.*)

__fp_to_scientific_dispatch:w
__fp_to_scientific_normal:wnnnnn
__fp_to_scientific_normal:wNw

Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2` $= 2$) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases: ±0 are represented as `0`; infinities are converted to a number slightly larger than the largest after an "invalid_operation" exception; `nan` is represented as `0` after an "invalid_operation" exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```
\cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { 0.000000000000000e0 }
    \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_scientific:N \c__fp_overflowing_fp }
            { fp_to_scientific }
        }
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_scientific:N \c_zero_fp }
            { fp_to_scientific }
        }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
\cs_new:Npn \__fp_to_scientific_normal:wnnnnn
  \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \exp_after:wN \__fp_to_scientific_normal:wNw
    \exp_after:wN e
    \__int_value:w \__int_eval:w #2 - 1
    ; #3 #4 #5 #6 ;
  }
\cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
  { #2.#3 #1 }
```

(*End definition for* `\__fp_to_scientific_dispatch:w`*,* `\__fp_to_scientific_normal:wnnnnn`*, and* `\__fp_to_scientific_normal:wNw`*.*)

30.3 Decimal representation

\fp_to_decimal:N
\fp_to_decimal:c
\fp_to_decimal:n

All three public variants are based on the same `\__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```
\cs_new:Npn \fp_to_decimal:N #1
  { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_decimal:N { c }
\cs_new:Npn \fp_to_decimal:n
  {
    \exp_after:wN \__fp_to_decimal_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End definition for* `\fp_to_decimal:N` *and* `\fp_to_decimal:n`*. These functions are documented on page 179.*)

__fp_to_decimal_dispatch:w
__fp_to_decimal_normal:wnnnnn
__fp_to_decimal_large:Nnnw
__fp_to_decimal_huge:wnnnn

The structure is similar to `\__fp_to_scientific_dispatch:w`. Insert `-` for negative numbers. Zero gives 0, ±∞ and `NaN` yield an "invalid operation" exception; note that ±∞ produces a very large output, which we don't expand now since it most likely won't be needed. Normal numbers with an exponent in the range $[1, 15]$ have that number of digits before the decimal separator: "decimate" them, and remove leading zeros with `\__int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.⟨*zeros*⟩⟨*digits*⟩, trimmed.

```
\cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { 0 }
    \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_decimal:N \c__fp_overflowing_fp }
            { fp_to_decimal }
        }
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { 0 }
            { fp_to_decimal }
        }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
\cs_new:Npn \__fp_to_decimal_normal:wnnnnn
    \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \int_compare:nNnTF {#2} > 0
      {
        \int_compare:nNnTF {#2} < \c__fp_prec_int
          {
            \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
              \__fp_to_decimal_large:Nnnw
          }
          {
            \exp_after:wN \exp_after:wN
            \exp_after:wN \__fp_to_decimal_huge:wnnnn
            \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
          }
        {#3} {#4} {#5} {#6}
      }
      {
        \exp_after:wN \__fp_trim_zeros:w
        \exp_after:wN 0
        \exp_after:wN .
        \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
        #3#4#5#6 ;
      }
  }
\cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
  {
    \exp_after:wN \__fp_trim_zeros:w \__int_value:w
      \if_int_compare:w #2 > 0 \exp_stop_f:
        #2
      \fi:
      \exp_stop_f:
      #3.#4 ;
  }
\cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }
```

(*End definition for* `\__fp_to_decimal_dispatch:w` *and others.*)

30.4 Token list representation

\fp_to_tl:N
\fp_to_tl:c
\fp_to_tl:n

These three public functions evaluate their argument, then pass it to `\__fp_to_tl_dispatch:w`.

```
\cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_tl:N { c }
\cs_new:Npn \fp_to_tl:n
  {
    \exp_after:wN \__fp_to_tl_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End definition for* `\fp_to_tl:N` *and* `\fp_to_tl:n`*. These functions are documented on page 180.*)

`\__fp_to_tl_dispatch:w`
`\__fp_to_tl_normal:nnnnn`
`\__fp_to_tl_scientific:wnnnnn`
`\__fp_to_tl_scientific:wNw`

A structure similar to `\__fp_to_scientific_dispatch:w` and `\__fp_to_decimal_dispatch:w`, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```
\cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return:nw { 0 }
    \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
    \or:   \__fp_case_return:nw { inf }
    \else: \__fp_case_return:nw { nan }
    \fi:
  }
\cs_new:Npn \__fp_to_tl_normal:nnnnn #1
  {
    \int_compare:nTF
      { -2 <= #1 <= \c__fp_prec_int }
      { \__fp_to_decimal_normal:wnnnnn }
      { \__fp_to_tl_scientific:wnnnnn }
    \s__fp \__fp_chk:w 1 0 {#1}
  }
\cs_new:Npn \__fp_to_tl_scientific:wnnnnn
  \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \exp_after:wN \__fp_to_tl_scientific:wNw
    \exp_after:wN e
    \__int_value:w \__int_eval:w #2 - 1
    ; #3 #4 #5 #6 ;
  }
\cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
  { \__fp_trim_zeros:w #2.#3 ; #1 }
```

(*End definition for* `\__fp_to_tl_dispatch:w` *and others.*)

30.5 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

30.6 Convert to dimension or integer

`\fp_to_dim:N`
`\fp_to_dim:c`
`\fp_to_dim:n`

These three public functions rely on `\fp_to_decimal:n` internally.

```
\cs_new:Npn \fp_to_dim:N #1
  { \fp_to_decimal:N #1 pt }
\cs_generate_variant:Nn \fp_to_dim:N { c }
\cs_new:Npn \fp_to_dim:n #1
  { \fp_to_decimal:n {#1} pt }
```

(*End definition for* `\fp_to_dim:N` *and* `\fp_to_dim:n`*. These functions are documented on page 179.*)

\fp_to_int:N
\fp_to_int:c
\fp_to_int:n

These three public functions evaluate their argument, then pass it to `\fp_to_int_dispatch:w`.

```
\cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_int:N { c }
\cs_new:Npn \fp_to_int:n
  {
    \exp_after:wN \__fp_to_int_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End definition for* `\fp_to_int:N` *and* `\fp_to_int:n`*. These functions are documented on page 180.*)

__fp_to_int_dispatch:w

To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `\__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```
\cs_new:Npn \__fp_to_int_dispatch:w #1;
  {
    \exp_after:wN \__fp_to_decimal_dispatch:w \exp:w \exp_end_continue_f:w
    \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
  }
```

(*End definition for* `\__fp_to_int_dispatch:w`*.*)

30.7 Convert from a dimension

\dim_to_fp:n
__fp_from_dim_test:ww
__fp_from_dim:wNw
__fp_from_dim:wNNnnnnnn
__fp_from_dim:wnnnnwNw

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `\__fp_mul_npos_o:Nww` expects the desired ⟨*final sign*⟩ and two floating point operands (of the form `\s__fp` ... `;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16}. There is also a need to expand afterwards: this is performed by `\__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new:Npn \dim_to_fp:n #1
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \exp_after:wN 0
    \exp_after:wN ,
    \__int_value:w \etex_glueexpr:D #1 ;
  }
\cs_new:Npn \__fp_from_dim_test:ww #1, #2
  {
    \if_meaning:w 0 #2
      \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
    \else:
      \exp_after:wN \__fp_from_dim:wNw
      \__int_value:w \__int_eval:w #1 - 4
        \if_meaning:w - #2
          \exp_after:wN , \exp_after:wN 2 \__int_value:w
        \else:
          \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
        \fi:
    \fi:
  }
\cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
  {
    \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
    #3 000 0000 00 {10}987654321; #2 {#1}
  }
\cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
  { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
\cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
  {
    \__fp_mul_npos_o:Nww #7
      \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
      \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
      \prg_do_nothing:
  }
```

(*End definition for* `\dim_to_fp:n` *and others. These functions are documented on page 156.*)

30.8 Use and eval

\fp_use:N
\fp_use:c
\fp_eval:n

Those public functions are simple copies of the decimal conversions.

```
\cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\cs_generate_variant:Nn \fp_use:N { c }
\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n
```

(*End definition for* `\fp_use:N` *and* `\fp_eval:n`*. These functions are documented on page 180.*)

\fp_abs:n

Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `\__fp_parse:n`, namely, for better error reporting.

```
\cs_new:Npn \fp_abs:n #1
  { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(*End definition for* `\fp_abs:n`*. This function is documented on page 193.*)

\fp_max:nn
\fp_min:nn

Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, *etc.*

```
\cs_new:Npn \fp_max:nn #1#2
  { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
\cs_new:Npn \fp_min:nn #1#2
  { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(*End definition for* `\fp_max:nn` *and* `\fp_min:nn`*. These functions are documented on page 193.*)

30.9 Convert an array of floating points to a comma list

__fp_array_to_clist:n
__fp_array_to_clist_loop:Nw

Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
  {
    \use_none:n #1
    { , ~ } \fp_to_tl:n { #1 #2 ; }
    \__fp_array_to_clist_loop:Nw
  }
```

The `\use_ii:nn` function is expanded after `\__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
\cs_new:Npn \__fp_array_to_clist:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \__fp_expand:n
          {
            { \use_ii:nn }
            \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
            \__prg_break_point:
          }
      }
  }
\cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
  {
    \exp_not:N \use_none:n #1
    \exp_not:N \exp_after:wN
                 {
    \exp_not:N     \exp_after:wN ,
    \exp_not:N     \exp_after:wN \c_space_tl
    \exp_not:N     \exp:w
    \exp_not:N     \exp_end_continue_f:w
    \exp_not:N     \__fp_to_tl_dispatch:w #1 #2 ;
                 }
    \exp_not:N \__fp_array_to_clist_loop:Nw
  }
```

(*End definition for* `\__fp_array_to_clist:n` *and* `\__fp_array_to_clist_loop:Nw`.)

```
⟨/initex | package⟩
```

31 l3fp-random Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

`\__fp_parse_word_rand:N`
`\__fp_parse_word_randint:N`

Those functions may receive a variable number of arguments. We won't use the argument `?`.

```
\cs_new:Npn \__fp_parse_word_rand:N
  { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
\cs_new:Npn \__fp_parse_word_randint:N
  { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }
```

(*End definition for* `\__fp_parse_word_rand:N` *and* `\__fp_parse_word_randint:N`.)

31.1 Engine support

At present, XƎTEX, pTEX and upTEX do not provide random numbers, while LuaTEX and pdfTEX provide the primitive `\pdftex_uniformdeviate:D` (`\pdfuniformdeviate` in pdfTEX and `\uniformdeviate` in LuaTEX). We write the test twice simply in order to write the `false` branch first.

```
\cs_if_exist:NF \pdftex_uniformdeviate:D
```

```
  {
    \__msg_kernel_new:nnn { kernel } { fp-no-random }
      { Random~numbers~unavailable }
    \cs_new:Npn \__fp_rand_o:Nw ? #1 @
      {
        \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
        \exp_after:wN \c_nan_fp
      }
    \cs_new_eq:NN \__fp_randint_o:Nw \__fp_rand_o:Nw
  }
\cs_if_exist:NT \pdftex_uniformdeviate:D
  {
```

__fp_rand_uniform:
\c__fp_rand_size_int
\c__fp_rand_four_int
\c__fp_rand_eight_int

The \pdftex_uniformdeviate:D primitive gives a pseudo-random integer in a range $[0,n-1]$ of the user's choice. This number is meant to be uniformly distributed, but is produced by rescaling a uniform pseudo-random integer in $[0,2^{28}-1]$. For instance, setting n to (any multiple of) 2^{29} gives only even values. Thus it is only safe to call \pdftex_uniformdeviate:D with argument 2^{28}. This integer is also used in the implementation of \int_rand:nn. We also use variants of this number rounded down to multiples of 10^4 and 10^8.

```
\cs_new:Npn \__fp_rand_uniform:
  { \pdftex_uniformdeviate:D \c__fp_rand_size_int }
\int_const:Nn \c__fp_rand_size_int   { 268 435 456 }
\int_const:Nn \c__fp_rand_four_int   { 268 430 000 }
\int_const:Nn \c__fp_rand_eight_int  { 200 000 000 }
```

(*End definition for __fp_rand_uniform: and others.*)

__fp_rand_myriads:n
__fp_rand_myriads_loop:nn
__fp_rand_myriads_get:w
__fp_rand_myriads_last:
__fp_rand_myriads_last:w

Used as __fp_rand_myriads:n {XXX} with one input character per block of four digit we want. Given a pseudo-random integer from the primitive, we extract 2 blocks of digits if possible, namely if the integer is less than 2×10^8. If that's not possible, we try to extract 1 block, which succeeds in the range $[2\times10^8, 26843\times10^4)$. For the 5456 remaining possible values we just throw away the random integer and get a new one. Depending on whether we got 2, 1, or 0 blocks, remove the same number of characters from the input stream with \use_i:nnn, \use_i:nn or nothing.

```
\cs_new:Npn \__fp_rand_myriads:n #1
  {
    \__fp_rand_myriads_loop:nn #1
      { ? \use_i_delimit_by_q_stop:nw \__fp_rand_myriads_last: }
      { ? \use_none_delimit_by_q_stop:w } \q_stop
  }
\cs_new:Npn \__fp_rand_myriads_loop:nn #1#2
  {
    \use_none:n #2
    \exp_after:wN \__fp_rand_myriads_get:w
    \__int_value:w \__fp_rand_uniform: ; {#1}{#2}
  }
\cs_new:Npn \__fp_rand_myriads_get:w #1 ;
  {
    \if_int_compare:w #1 < \c__fp_rand_eight_int
      \exp_after:wN \use_none:n
      \__int_value:w \__int_eval:w
        \c__fp_rand_eight_int + #1 \__int_eval_end:
```

```
      \exp_after:wN \use_i:nnn
    \else:
      \if_int_compare:w #1 < \c__fp_rand_four_int
        \exp_after:wN \use_none:nnnnn
        \__int_value:w \__int_eval:w
          \c__fp_rand_four_int + #1 \__int_eval_end:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \fi:
    \fi:
    \__fp_rand_myriads_loop:nn
  }
\cs_new:Npn \__fp_rand_myriads_last:
  {
    \exp_after:wN \__fp_rand_myriads_last:w
    \__int_value:w \__fp_rand_uniform: ;
  }
\cs_new:Npn \__fp_rand_myriads_last:w #1 ;
  {
    \if_int_compare:w #1 < \c__fp_rand_four_int
      \exp_after:wN \use_none:nnnnn
      \__int_value:w \__int_eval:w
        \c__fp_rand_four_int + #1 \__int_eval_end:
    \else:
      \exp_after:wN \__fp_rand_myriads_last:
    \fi:
  }
```

(*End definition for* `\__fp_rand_myriads:n` *and others.*)

31.2 Random floating point

`\__fp_rand_o:Nw`
`\__fp_rand_o:`
`\__fp_rand_o:w`

First we check that `random` was called without argument. Then get four blocks of four digits.

```
\cs_new:Npn \__fp_rand_o:Nw ? #1 @
  {
    \tl_if_empty:nTF {#1}
      { \__fp_rand_o: }
      {
        \__msg_kernel_expandable_error:nnnnn
          { kernel } { fp-num-args } { rand() } { 0 } { 0 }
        \exp_after:wN \c_nan_fp
      }
  }
\cs_new:Npn \__fp_rand_o:
  { \__fp_parse_o:n { . \__fp_rand_myriads:n { xxxx } } }
```

(*End definition for* `\__fp_rand_o:Nw`*,* `\__fp_rand_o:`*, and* `\__fp_rand_o:w`*.*)

31.3 Random integer

`\__fp_randint_o:Nw`
`\__fp_randint_badarg:w`
`\__fp_randint_e:w`
`\__fp_randint_e:wnn`
`\__fp_randint_e:wwNnn`
`\__fp_randint_e:wwwNnn`
`\__fp_randint_narrow_e:nnnn`
`\__fp_randint_wide_e:nnnn`
`\__fp_randint_wide_e:wnnn`

Enforce that there is one argument (then add first argument 1) or two arguments. Enforce that they are integers in $(-10^{16}, 10^{16})$ and ordered. We distinguish narrow ranges (less than 2^{28}) from wider ones.

For narrow ranges, compute the number n of possible outputs as an integer using `\fp_to_int:n`, and reduce a pseudo-random 28-bit integer r modulo n. On its own, this is not uniform when $[0, 2^{28} - 1]$ does not divide evenly into intervals of size n. The auxiliary `\__fp_randint_e:wwwNnn` discards the pseudo-random integer if it lies in an incomplete interval, and repeats.

For wide ranges we use the same code except for the last eight digits which use `\__fp_rand_myriads:n`. It is not safe to combine the first digits with the last eight as a single string of digits, as this may exceed 16 digits and be rounded. Instead, we first add the first few digits (times 10^8) to the lower bound. The result is compared to the upper bound and the process repeats if needed.

```
\cs_new:Npn \__fp_randint_o:Nw ? #1 @
  {
    \if_case:w
      \__int_eval:w \__fp_array_count:n {#1} - 1 \__int_eval_end:
         \exp_after:wN \__fp_randint_e:w \c_one_fp #1
    \or: \__fp_randint_e:w #1
    \else:
      \__msg_kernel_expandable_error:nnnnn
        { kernel } { fp-num-args } { randint() } { 1 } { 2 }
      \exp_after:wN \c_nan_fp \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
\cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
  {
    \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
      {
        \if_meaning:w 1 #1
          \if_int_compare:w
            \use_i_delimit_by_q_stop:nw #3 \q_stop > \c__fp_prec_int
            1 \exp_stop_f:
          \fi:
        \fi:
      }
      { 1 \exp_stop_f: }
  }
\cs_new:Npn \__fp_randint_e:w #1; #2;
  {
    \if_case:w
        \__fp_randint_badarg:w #1;
        \__fp_randint_badarg:w #2;
        \fp_compare:nNnTF { #1; } > { #2; } { 1 } { 0 } \exp_stop_f:
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_randint_e:wnn
        \__fp_parse:n { #2; - #1; } { #1; } { #2; }
    \or:
      \__fp_invalid_operation_tl_o:ff
        { randint } { \__fp_array_to_clist:n { #1; #2; } }
      \exp:w
    \fi:
  }
\cs_new:Npn \__fp_randint_e:wnn #1;
  {
    \exp_after:wN \__fp_randint_e:wwNnn
    \__int_value:w \__fp_rand_uniform: \exp_after:wN ;
    \exp:w \exp_end_continue_f:w
      \fp_compare:nNnTF { #1 ; } < \c__fp_rand_size_int
        { \fp_to_int:n { #1 ; + 1 } ; \__fp_randint_narrow_e:nnnn }
        { \fp_to_int:n { floor(#1 ; * 1e-8 + 1) } ; \__fp_randint_wide_e:nnnn }
  }
\cs_new:Npn \__fp_randint_e:wwNnn #1 ; #2 ;
  {
    \exp_after:wN \__fp_randint_e:wwwNnn
    \__int_value:w \int_mod:nn {#1} {#2} ; #1 ; #2 ;
  }
\cs_new:Npn \__fp_randint_e:wwwNnn #1 ; #2 ; #3 ; #4
  {
    \int_compare:nNnTF { #2 - #1 + #3 } > \c__fp_rand_size_int
      {
        \exp_after:wN \__fp_randint_e:wwNnn
          \__int_value:w \__fp_rand_uniform: ; #3 ; #4
      }
      { #4 {#1} {#3} }
  }
\cs_new:Npn \__fp_randint_narrow_e:nnnn #1#2#3#4
  { \__fp_parse_o:n { #3 + #1 } \exp:w }
\cs_new:Npn \__fp_randint_wide_e:nnnn #1#2#3#4
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_randint_wide_e:wnnn
      \__fp_parse:n { #3 + #1e8 + \__fp_rand_myriads:n { xx } }
      {#2} {#3} {#4}
  }
\cs_new:Npn \__fp_randint_wide_e:wnnn #1 ; #2#3#4
  {
    \fp_compare:nNnTF { #1 ; } > {#4}
      {
        \exp_after:wN \__fp_randint_e:wwNnn
          \__int_value:w \__fp_rand_uniform: ; #2 ;
          \__fp_randint_wide_e:nnnn {#3} {#4}
      }
      { \__fp_exp_after_o:w #1 ; \exp:w }
  }
```

(*End definition for* `\__fp_randint_o:Nw` *and others.*)

End the initial conditional that ensures these commands are only defined in pdfTEX and LuaTEX.

```
  }
⟨/initex | package⟩
```

32 l3fp-assign implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

32.1 Assigning values

\fp_new:N

Floating point variables are initialized to be +0.

```
\cs_new_protected:Npn \fp_new:N #1
  { \cs_new_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_new:N {c}
```

(*End definition for* `\fp_new:N`. *This function is documented on page 178.*)

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn
\fp_const:Nn
\fp_const:cn

Simply use `\__fp_parse:n` within various `f`-expanding assignments.

```
\cs_new_protected:Npn \fp_set:Nn   #1#2
  { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_gset:Nn  #1#2
  { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_const:Nn #1#2
  { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_generate_variant:Nn \fp_set:Nn {c}
\cs_generate_variant:Nn \fp_gset:Nn {c}
\cs_generate_variant:Nn \fp_const:Nn {c}
```

(*End definition for* `\fp_set:Nn`, `\fp_gset:Nn`, *and* `\fp_const:Nn`. *These functions are documented on page 178.*)

\fp_set_eq:NN
\fp_set_eq:cN
\fp_set_eq:Nc
\fp_set_eq:cc
\fp_gset_eq:NN
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc

Copying a floating point is the same as copying the underlying token list.

```
\cs_new_eq:NN \fp_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \fp_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(*End definition for* `\fp_set_eq:NN` *and* `\fp_gset_eq:NN`. *These functions are documented on page 179.*)

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Setting a floating point to zero: copy `\c_zero_fp`.

```
\cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_zero:N  { c }
\cs_generate_variant:Nn \fp_gzero:N { c }
```

(*End definition for* `\fp_zero:N` *and* `\fp_gzero:N`. *These functions are documented on page 178.*)

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Set the floating point to zero, or define it if needed.

```
\cs_new_protected:Npn \fp_zero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\cs_new_protected:Npn \fp_gzero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
\cs_generate_variant:Nn \fp_zero_new:N  { c }
\cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(*End definition for* `\fp_zero_new:N` *and* `\fp_gzero_new:N`. *These functions are documented on page 178.*)

32.2 Updating values

These match the equivalent functions in l3int and l3skip.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn
\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn
__fp_add:NNNn

For the sake of error recovery we should not simply set `#1` to `#1` ± (`#2`): for instance, if `#2` is `0)+2`, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate `#2` instead of just putting parentheses. As an optimization we use `\__fp_parse:n` rather than `\fp_eval:n`, which would convert the result away from the internal representation and back.

```
\cs_new_protected:Npn \fp_add:Nn  { \__fp_add:NNNn \fp_set:Nn  + }
\cs_new_protected:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\cs_new_protected:Npn \fp_sub:Nn  { \__fp_add:NNNn \fp_set:Nn  - }
\cs_new_protected:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
\cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
  { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
\cs_generate_variant:Nn \fp_add:Nn  { c }
\cs_generate_variant:Nn \fp_gadd:Nn { c }
\cs_generate_variant:Nn \fp_sub:Nn  { c }
\cs_generate_variant:Nn \fp_gsub:Nn { c }
```

(*End definition for `\fp_add:Nn` and others. These functions are documented on page 179.*)

32.3 Showing values

\fp_show:N
\fp_show:c
\fp_show:n

This shows the result of computing its argument. The input of `\__msg_show_variable:NNNnn` must start with `>~` (or be empty).

```
\cs_new_protected:Npn \fp_show:N #1
  {
    \__msg_show_variable:NNNnn #1 \fp_if_exist:NTF ? { }
      { > ~ \token_to_str:N #1 = \fp_to_tl:N #1 }
  }
\cs_new_protected:Npn \fp_show:n
  { \__msg_show_wrap:Nn \fp_to_tl:n }
\cs_generate_variant:Nn \fp_show:N { c }
```

(*End definition for `\fp_show:N` and `\fp_show:n`. These functions are documented on page 185.*)

\fp_log:N
\fp_log:c
\fp_log:n

Redirect output of `\fp_show:N` and `\fp_show:n` to the log.

```
\cs_new_protected:Npn \fp_log:N
  { \__msg_log_next: \fp_show:N }
\cs_new_protected:Npn \fp_log:n
  { \__msg_log_next: \fp_show:n }
\cs_generate_variant:Nn \fp_log:N { c }
```

(*End definition for `\fp_log:N` and `\fp_log:n`. These functions are documented on page 185.*)

32.4 Some useful constants and scratch variables

\c_one_fp
\c_e_fp

Some constants.

```
\fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
\fp_const:Nn \c_one_fp        { 1 }
```

(*End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 184.*)

\c_pi_fp
\c_one_degree_fp

We simply round π to and $\pi/180$ to 16 significant digits.

```
\fp_const:Nn \c_pi_fp         { 3.141 5926 5358 9793 }
\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
```

(*End definition for* `\c_pi_fp` *and* `\c_one_degree_fp`*. These variables are documented on page 184.*)

\l_tmpa_fp
\l_tmpb_fp
\g_tmpa_fp
\g_tmpb_fp

Scratch variables are simply initialized there.

```
\fp_new:N \l_tmpa_fp
\fp_new:N \l_tmpb_fp
\fp_new:N \g_tmpa_fp
\fp_new:N \g_tmpb_fp
```

(*End definition for* `\l_tmpa_fp` *and others. These variables are documented on page 184.*)

```
⟨/initex | package⟩
```

33 l3sort implementation

```
⟨*initex | package⟩
⟨@@=sort⟩
```

33.1 Variables

\l__sort_length_int
\l__sort_min_int
\l__sort_top_int
\l__sort_max_int
\l__sort_true_max_int

The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int` $-$ 1. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `\__sort_compute_range:`. That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```
\int_new:N \l__sort_length_int
\int_new:N \l__sort_min_int
\int_new:N \l__sort_top_int
\int_new:N \l__sort_max_int
\int_new:N \l__sort_true_max_int
```

(*End definition for* `\l__sort_length_int` *and others.*)

\l__sort_block_int

Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k+1, 2^{k+1}]$, reaches 2^k in the last pass.

```
\int_new:N \l__sort_block_int
```

(*End definition for* `\l__sort_block_int`*.*)

\l__sort_begin_int
\l__sort_end_int

When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```
\int_new:N \l__sort_begin_int
\int_new:N \l__sort_end_int
```

(*End definition for* `\l__sort_begin_int` *and* `\l__sort_end_int`*.*)

\l__sort_A_int
\l__sort_B_int
\l__sort_C_int

When merging two blocks (whose end-points are beg and end), A starts from the high end of the low block, and decreases until reaching beg. The index B starts from the top of the range and marks the register in which a sorted item should be put. Finally, C points to the copy of the high block in the interval of registers starting at \l__sort_length_int, upwards. C starts from the upper limit of that range.

```
\int_new:N \l__sort_A_int
\int_new:N \l__sort_B_int
\int_new:N \l__sort_C_int
```

(*End definition for \l__sort_A_int , \l__sort_B_int , and \l__sort_C_int.*)

33.2 Finding available \toks registers

__sort_shrink_range:
__sort_shrink_range_loop:

After __sort_compute_range: (defined below) determines that \toks registers between \l__sort_min_int (included) and \l__sort_true_max_int (excluded) have not yet been assigned, __sort_shrink_range: computes \l__sort_max_int to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving \l__sort_block_int, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set \l__sort_max_int.

```
\cs_new_protected:Npn \__sort_shrink_range:
  {
    \int_set:Nn \l__sort_A_int
      { \l__sort_true_max_int - \l__sort_min_int + 1 }
    \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
    \__sort_shrink_range_loop:
    \int_set:Nn \l__sort_max_int
      {
        \int_compare:nNnTF
          { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
          {
            \l__sort_min_int
            + ( \l__sort_A_int - 1 ) / 2
            + \l__sort_block_int / 4
            - 1
          }
          { \l__sort_true_max_int - \l__sort_block_int / 2 }
      }
  }
\cs_new_protected:Npn \__sort_shrink_range_loop:
  {
    \if_int_compare:w \l__sort_A_int < \l__sort_block_int
      \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
      \exp_after:wN \__sort_shrink_range_loop:
    \fi:
  }
```

(*End definition for __sort_shrink_range: and __sort_shrink_range_loop:.*)

__sort_compute_range:
__sort_redefine_compute_range:
\c__sort_max_length_int

First find out what \toks have not yet been assigned. There are many cases. In LaTeX 2_ε with no package, available \toks range from $\count15 + 1$ to \c_max_register_int included (this was not altered despite the 2015 changes). When \loctoks is defined, namely in plain (e)TeX, or when the package etex is loaded in LaTeX 2$_\varepsilon$, redefine `\__sort_compute_range:` to use the range `\count265` to `\count275` − 1. The elocalloc package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as etex or elocalloc could be loaded.

In ConTeXt MkIV the range is from `\c_syst_last_allocated_toks` + 1 to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks` + 1 to `\c_max_register_int`. In all these cases, call `\__sort_shrink_range:`. The LaTeX3 format mode is easiest: no `\toks` are ever allocated so available `\toks` range from 0 to `\c_max_register_int` and we precompute the result of `\__sort_shrink_range:`.

```
⟨*package⟩
\cs_new_protected:Npn \__sort_compute_range:
  {
    \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
    \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
    \__sort_shrink_range:
    \if_meaning:w \loctoks \tex_undefined:D \else:
      \if_meaning:w \loctoks \scan_stop: \else:
        \__sort_redefine_compute_range:
        \__sort_compute_range:
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__sort_redefine_compute_range:
  {
    \cs_if_exist:cTF { ver@elocalloc.sty }
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
            \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
            \__sort_shrink_range:
          }
      }
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
            \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
            \__sort_shrink_range:
          }
      }
  }
\cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
\tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
  {
    \cs_if_exist:NT #1
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { #1 + 1 }
            \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
            \__sort_shrink_range:
          }
      }
  }
⟨/package⟩
⟨*initex⟩
\int_const:Nn \c__sort_max_length_int
  { ( \c_max_register_int + 1 ) * 3 / 4 }
\cs_new_protected:Npn \__sort_compute_range:
  {
    \int_set:Nn \l__sort_min_int { 0 }
    \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
    \int_set:Nn \l__sort_max_int { \c__sort_max_length_int }
  }
⟨/initex⟩
```

(*End definition for* `\__sort_compute_range:`, `\__sort_redefine_compute_range:`, *and* `\c__sort_max_length_int`.)

33.3 Protected user commands

`\__sort_main:NNNnNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int` − 1, while checking that the list is not too long. If we reach the maximum length, all further items are entirely ignored after raising an error. Secondly, sort the array of `\toks` registers, using the user-defined sorting function, `#6`. Finally, unpack the `\toks` registers (now sorted) into a variable of the right type, by `x`-expanding the code in `#4`, specific to each type of list.

```
\cs_new_protected:Npn \__sort_main:NNNnNn #1#2#3#4#5#6
  {
    \group_begin:
⟨package⟩      \__sort_disable_toksdef:
      \__sort_compute_range:
      \int_set_eq:NN \l__sort_top_int \l__sort_min_int
      #2 #5
        {
          \if_int_compare:w \l__sort_top_int = \l__sort_max_int
            \__sort_too_long_error:NNw #3 #5
          \fi:
          \tex_toks:D \l__sort_top_int {##1}
          \int_incr:N \l__sort_top_int
        }
      \int_set:Nn \l__sort_length_int
        { \l__sort_top_int - \l__sort_min_int }
      \cs_set:Npn \__sort_compare:nn ##1 ##2 { #6 }
      \int_set:Nn \l__sort_block_int { 1 }
      \__sort_level:
      \use:x
        {
          \group_end:
          #1 \exp_not:N #5 {#4}
        }
  }
```

(*End definition for* `\__sort_main:NNNnNn`.)

\seq_sort:Nn
\seq_gsort:Nn

The first argument to __sort_main:NNNnNn is the final assignment function used, either \tl_set:Nn or \tl_gset:Nn to control local versus global results. The second argument is what mapping function is used when storing items to \toks registers, and the third breaks away from the loop. The fourth is used to build back the correct kind of list from the contents of the \toks registers, including the leading \s__seq. Fifth and sixth arguments are the variable to sort, and the sorting method as inline code.

```
\cs_new_protected:Npn \seq_sort:Nn
  {
    \__sort_main:NNNnNn \tl_set:Nn
      \seq_map_inline:Nn \seq_map_break:n
      { \s__seq \__sort_toks:NN \exp_not:N \__seq_item:n }
  }
\cs_generate_variant:Nn \seq_sort:Nn { c }
\cs_new_protected:Npn \seq_gsort:Nn
  {
    \__sort_main:NNNnNn \tl_gset:Nn
      \seq_map_inline:Nn \seq_map_break:n
      { \s__seq \__sort_toks:NN \exp_not:N \__seq_item:n }
  }
\cs_generate_variant:Nn \seq_gsort:Nn { c }
```

(*End definition for \seq_sort:Nn and \seq_gsort:Nn. These functions are documented on page 62.*)

\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn

Again, use \tl_set:Nn or \tl_gset:Nn to control the scope of the assignment. Mapping through the token list is done with \tl_map_inline:Nn, and producing the token list is very similar to sequences, removing __seq_item:n.

```
\cs_new_protected:Npn \tl_sort:Nn
  {
    \__sort_main:NNNnNn \tl_set:Nn
      \tl_map_inline:Nn \tl_map_break:n
      { \__sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
  }
\cs_generate_variant:Nn \tl_sort:Nn { c }
\cs_new_protected:Npn \tl_gsort:Nn
  {
    \__sort_main:NNNnNn \tl_gset:Nn
      \tl_map_inline:Nn \tl_map_break:n
      { \__sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
  }
\cs_generate_variant:Nn \tl_gsort:Nn { c }
```

(*End definition for \tl_sort:Nn and \tl_gsort:Nn. These functions are documented on page 44.*)

\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn
__sort_clist:NNn

The case of empty comma-lists is a little bit special as usual, and filtered out: there is nothing to sort in that case. Otherwise, the input is done with \clist_map_inline:Nn, and the output requires some more elaborate processing than for sequences and token lists. The first comma must be removed. An item must be wrapped in an extra set of braces if it contains either the space or the comma characters. This is taken care of by \clist_wrap_item:n, but __sort_toks:NN would simply feed \tex_the:D \tex_toks:D ⟨*number*⟩ as an argument to that function; hence we need to expand this argument once to unpack the register.

```
\cs_new_protected:Npn \clist_sort:Nn
  { \__sort_clist:NNn \tl_set:Nn }
```

```
\cs_new_protected:Npn \clist_gsort:Nn
  { \__sort_clist:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \clist_sort:Nn  { c }
\cs_generate_variant:Nn \clist_gsort:Nn { c }
\cs_new_protected:Npn \__sort_clist:NNn #1#2#3
  {
    \clist_if_empty:NF #2
      {
        \__sort_main:NNNnNn #1
          \clist_map_inline:Nn \clist_map_break:n
          {
            \exp_last_unbraced:Nf \use_none:n
              { \__sort_toks:NN \exp_args:No \__clist_wrap_item:n }
          }
          #2 {#3}
      }
  }
```

(*End definition for* `\clist_sort:Nn`, `\clist_gsort:Nn`, *and* `\__sort_clist:NNn`. *These functions are documented on page 102.*)

`\__sort_toks:NN`
`\__sort_toks:NNw`

Unpack the various `\toks` registers, from `\l__sort_min_int` to `\l__sort_top_int` − 1. The functions `#1` and `#2` allow us to treat the three data structures in a unified way:

- for sequences, they are `\exp_not:N \__seq_item:n`, expanding to the `\__seq_item:n` separator, as expected;
- for token lists, they expand to nothing;
- for comma lists, they expand to `\exp_args:No \clist_wrap_item:n`, taking care of unpacking the register before letting the undocumented internal clist function `\clist_wrap_item:n` do the work of putting a comma and possibly braces.

```
\cs_new:Npn \__sort_toks:NN #1#2
  { \__sort_toks:NNw #1 #2 \l__sort_min_int ; }
\cs_new:Npn \__sort_toks:NNw #1#2#3 ;
  {
    \if_int_compare:w #3 < \l__sort_top_int
      #1 #2 { \tex_the:D \tex_toks:D #3 }
      \exp_after:wN \__sort_toks:NNw \exp_after:wN #1 \exp_after:wN #2
      \__int_value:w \__int_eval:w #3 + 1 \exp_after:wN ;
    \fi:
  }
```

(*End definition for* `\__sort_toks:NN` *and* `\__sort_toks:NNw`.)

33.4 Merge sort

`\__sort_level:`

This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```
\cs_new_protected:Npn \__sort_level:
  {
    \if_int_compare:w \l__sort_block_int < \l__sort_length_int
```

```
      \l__sort_end_int \l__sort_min_int
      \__sort_merge_blocks:
      \tex_advance:D \l__sort_block_int \l__sort_block_int
      \exp_after:wN \__sort_level:
    \fi:
  }
```

(*End definition for* `\__sort_level:`.)

`\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in A, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it $\leq$ `top`. Copy this upper block of `\toks` registers in registers above `length`, indexed by C: this is covered by `\__sort_copy_block:`. Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:`, after shifting A, B and C so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```
\cs_new_protected:Npn \__sort_merge_blocks:
  {
    \l__sort_begin_int \l__sort_end_int
    \tex_advance:D \l__sort_end_int \l__sort_block_int
    \if_int_compare:w \l__sort_end_int < \l__sort_top_int
      \l__sort_A_int \l__sort_end_int
      \tex_advance:D \l__sort_end_int \l__sort_block_int
      \if_int_compare:w \l__sort_end_int > \l__sort_top_int
        \l__sort_end_int \l__sort_top_int
      \fi:
      \l__sort_B_int \l__sort_A_int
      \l__sort_C_int \l__sort_top_int
      \__sort_copy_block:
      \int_decr:N \l__sort_A_int
      \int_decr:N \l__sort_B_int
      \int_decr:N \l__sort_C_int
      \exp_after:wN \__sort_merge_blocks_aux:
      \exp_after:wN \__sort_merge_blocks:
    \fi:
  }
```

(*End definition for* `\__sort_merge_blocks:`.)

`\__sort_copy_block:` We wish to store a copy of the "upper" block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```
\cs_new_protected:Npn \__sort_copy_block:
  {
    \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
    \int_incr:N \l__sort_C_int
    \int_incr:N \l__sort_B_int
    \if_int_compare:w \l__sort_B_int = \l__sort_end_int
      \use_i:nn
    \fi:
    \__sort_copy_block:
  }
```

(*End definition for* `\__sort_copy_block:`*.*)

`\__sort_merge_blocks_aux:`

At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int` -1 and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```
\cs_new_protected:Npn \__sort_merge_blocks_aux:
  {
    \exp_after:wN \__sort_compare:nn \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
      \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
    \prg_do_nothing:
    \__sort_return_mark:N
    \__sort_return_mark:N
    \__sort_return_none_error:
  }
```

(*End definition for* `\__sort_merge_blocks_aux:`*.*)

`\sort_return_same:`
`\sort_return_swapped:`
`\__sort_return_mark:N`
`\__sort_return_none_error:`
`\__sort_return_two_error:w`

The marker removes one token. Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `\__sort_return_none_error:` is called.

```
\cs_new_protected:Npn \sort_return_same: #1 \__sort_return_mark:N
  { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_same: }
\cs_new_protected:Npn \sort_return_swapped: #1 \__sort_return_mark:N
  { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_swapped: }
\cs_new_protected:Npn \__sort_return_mark:N #1 { }
\cs_new_protected:Npn \__sort_return_none_error:
  {
    \__msg_kernel_error:nnxx { kernel } { return-none }
      { \tex_the:D \tex_toks:D \l__sort_A_int }
      { \tex_the:D \tex_toks:D \l__sort_C_int }
    \__sort_return_same:
  }
\cs_new_protected:Npn \__sort_return_two_error:w
    #1 \__sort_return_none_error:
  { \__msg_kernel_error:nn { kernel } { return-two } }
```

(*End definition for* `\sort_return_same:` *and others. These functions are documented on page* **??**.)

`\__sort_return_same:`

If the comparison function returns `same`, then the second argument fed to `\__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers B and C, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```
\cs_new_protected:Npn \__sort_return_same:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \int_decr:N \l__sort_B_int
    \int_decr:N \l__sort_C_int
    \if_int_compare:w \l__sort_C_int < \l__sort_top_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End definition for* `\__sort_return_same:`.)

`\__sort_return_swapped:` If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register A. Then shift the pointers A and B to the left, and go for one more step for the merger, unless the left block was exhausted (A goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by C, are copied to the merger by `\__sort_merge_blocks_end:`.

```
\cs_new_protected:Npn \__sort_return_swapped:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
    \int_decr:N \l__sort_B_int
    \int_decr:N \l__sort_A_int
    \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
      \__sort_merge_blocks_end: \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End definition for* `\__sort_return_swapped:`.)

`\__sort_merge_blocks_end:` This function's task is to copy the `\toks` registers in the block indexed by C to the merger indexed by B. The end can equally be detected by checking when B reaches the threshold `begin`, or when C reaches `top`.

```
\cs_new_protected:Npn \__sort_merge_blocks_end:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \int_decr:N \l__sort_B_int
    \int_decr:N \l__sort_C_int
    \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_end:
  }
```

(*End definition for* `\__sort_merge_blocks_end:`.)

33.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of __sort:nnNnn). The arguments of __sort:nnNnn are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of the list, call \tl_sort:nN on #1 and on #2, placing #4 in between; \use:ff expands the parts to make \tl_sort:nN f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the "greater" items, otherwise amongst the "lesser" items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \tl_if_blank:nF {#1}
      {
        \__sort:nnNnn { } { } #2
          #1 \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
  {
    \quark_if_recursion_tail_stop_do:nn {#5}
      { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
    #3 {#4} {#5}
      { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
      { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
  }
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, __sort:nnNnn is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to __sort:nnNnn. For this, the list is prepared by changing each ⟨*item*⟩ of the original token list into ⟨*command*⟩ {⟨*item*⟩}, just like sequences are stored. We arrange things such that the ⟨*command*⟩ is the ⟨*conditional*⟩ provided by the user: the loop over the ⟨*prepared tokens*⟩ then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
#6 {⟨pivot⟩} {#7} ⟨loop big⟩ ⟨loop small⟩
⟨extra arguments⟩
}
\__sort_loop:wNn ... ⟨prepared tokens⟩
⟨end-loop⟩ {} \q_stop
```

In this example, which matches the structure of __sort_quick_split_i:NnnnnNn and a few other functions below, the __sort_loop:wNn auxiliary normally receives the user's ⟨*conditional*⟩ as #6 and an ⟨*item*⟩ as #7. This is compared to the ⟨*pivot*⟩ (the argument #5, not shown here), and the ⟨*conditional*⟩ leaves the ⟨*loop big*⟩ or ⟨*loop small*⟩ auxiliary, which both have the same form as __sort_loop:wNn, receiving the next pair

⟨*conditional*⟩ `{`⟨*item*⟩`}` as `#6` and `#7`. At the end, `#6` is the ⟨*end-loop*⟩ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `\__sort:nnNnn`, which receive the new item as `#1` and place it either into the list `#2` of items less than the pivot `#4` or into the list `#3` of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} { #2 {#1} } {#3} {#4}
  }
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} {#2} { #3 {#1} } {#4}
  }
```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as `#5` the conditional or a function to end the loop. In fact, the lists `#2` and `#3` must be made of pairs ⟨*conditional*⟩ `{`⟨*item*⟩`}`, so we have to replace `{#6}` above by `{ #5 {#6} }`, and `{#1}` by `#1`. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\q_mark {`⟨*code*⟩`}`, and expands to ⟨*code*⟩ ⟨*sorted list*⟩. Sorting the two parts of the list around the pivot is done with

```
    \__sort_quick_split:NnNn #2 ...  \q_mark
    {
    \__sort_quick_split:NnNn #1 ...  \q_mark {⟨code⟩}
    {⟨pivot⟩}
    }
```

Items which are larger than the ⟨*pivot*⟩ are sorted, then placed after code that sorts the smaller items, and after the (braced) ⟨*pivot*⟩.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:nnnnNn` of the last example, but aware of whether the list of ⟨*conditional*⟩ `{`⟨*item*⟩`}` read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the ⟨*end-loop*⟩ function (that is initially placed after the "prepared" list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the ⟨*end-loop*⟩ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when TeX encounters

```
  \use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first \use:n is not completely read by the second \use:n, hence must remain in memory; then the argument of the second \use:n is not completely read when grabbing the argument of the third \use:n, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested \use:n. In practice, this means that we must read everything until a trailing \q_stop once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical TeX's memory.

\tl_sort:nN
__sort_quick_prepare:Nnnn
__sort_quick_prepare_end:NNNnw
__sort_quick_cleanup:w

The code within the \exp_not:f sorts the list, leaving in most cases a leading \exp_not:f, which stops the expansion, letting the result be return within \exp_not:n. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The prepare auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains __prg_break_point:, then the prepare_end auxiliary finds the prepared token list as #4. The scene is then set up for __sort_quick_split:NnNn, which sorts the prepared list and perform the post action placed after \q_mark, namely removing the trailing \s__stop and \q_stop and leaving \exp_stop_f: to stop f-expansion.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \exp_not:f
      {
        \tl_if_blank:nF {#1}
          {
            \__sort_quick_prepare:Nnnn #2 { } { }
              #1
              { \__prg_break_point: \__sort_quick_prepare_end:NNNnw }
            \q_stop
          }
      }
  }
\cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
  {
    \__prg_break: #4 \__prg_break_point:
    \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
  }
\cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
  {
    \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
    \q_mark { \__sort_quick_cleanup:w \exp_stop_f: }
    \s__stop \q_stop
  }
\cs_new:Npn \__sort_quick_cleanup:w #1 \s__stop \q_stop {#1}
```

(*End definition for \tl_sort:nN and others. These functions are documented on page 44.*)

__sort_quick_split:NnNn
__sort_quick_only_i:NnnnnNn
__sort_quick_only_ii:NnnnnNn
__sort_quick_split_i:NnnnnNn
__sort_quick_split_ii:NnnnnNn

The only_i, only_ii, split_i and split_ii auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a ⟨*function*⟩ #6, and an item #7. The ⟨*function*⟩ is the user's ⟨*conditional*⟩ except at the end of the list where it is __sort_quick_end:nnTFNn. The comparison is applied to the ⟨*pivot*⟩ and the ⟨*item*⟩, and calls the only_i or split_i auxiliaries if the ⟨*item*⟩ is smaller, and the only_ii or split_ii auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form ⟨*conditional*⟩ {⟨*item*⟩}, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The split auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's ⟨*conditional*⟩ rather than an ending function.

```
\cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
  {
    #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
      \__sort_quick_single_end:nnnwnw
      { #3 {#4} } { } { } {#2}
  }
\cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
      \__sort_quick_only_i_end:nnnwnw
      { #6 {#7} } { #3 #2 } { } {#5}
  }
\cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_only_ii_end:nnnwnw
      { #6 {#7} } { } { #4 #2 } {#5}
  }
\cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } { #3 #2 } {#4} {#5}
  }
\cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } {#3} { #4 #2 } {#5}
  }
```

(*End definition for* `\__sort_quick_split:NnNn` *and others.*)

__sort_quick_end:nnTFNn
__sort_quick_single_end:nnnwnw
__sort_quick_only_i_end:nnnwnw
__sort_quick_only_ii_end:nnnwnw
__sort_quick_split_end:nnnwnw

The __sort_quick_end:nnTFNn appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a true and a false branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after \q_mark. To avoid a memory problem described earlier, all of the ending functions read #6 until \q_stop and place #6 back into the input stream. When the lists #1 and #2 are empty, the single auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```
\cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
\cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  { #5 {#3} #6 \q_stop }
\cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #1
      \__sort_quick_end:nnTFNn { } \q_mark {#5}
    {#3}
    #6 \q_stop
  }
\cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #2
      \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
    #6 \q_stop
  }
\cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
      {
        \__sort_quick_split:NnNn #1
          \__sort_quick_end:nnTFNn { } \q_mark {#5}
        {#3}
      }
    #6 \q_stop
  }
```

(*End definition for* `\__sort_quick_end:nnTFNn` *and others.*)

33.6 Messages

`\__sort_error:`

Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with `\__sort_level:` getting rid of the final assignment. This error recovery won't work in a group.

```
\cs_new_protected:Npn \__sort_error:
  {
    \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
    \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
    \cs_set_protected:Npn \__sort_level: \use:x ##1 { \group_end: }
  }
```

(*End definition for* `\__sort_error:`.)

`\__sort_disable_toksdef:`
`\__sort_disabled_toksdef:n`

While sorting, `\toksdef` is locally disabled to prevent users from using `\newtoks` or similar commands in their comparison code: the `\toks` registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no `\toks` allocator.

```
⟨*package⟩
\cs_new_protected:Npn \__sort_disable_toksdef:
  { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
\cs_new_protected:Npn \__sort_disabled_toksdef:n #1
  {
    \__msg_kernel_error:nnx { kernel } { toksdef }
      { \token_to_str:N #1 }
    \__sort_error:
    \tex_toksdef:D #1
  }
\__msg_kernel_new:nnnn { kernel } { toksdef }
  { Allocation~of~\iow_char:N\\toks~registers~impossible~while~sorting. }
  {
    The~comparison~code~used~for~sorting~a~list~has~attempted~to~
    define~#1~as~a~new~\iow_char:N\\toks~register~using~\iow_char:N\\newtoks~
    or~a~similar~command.~The~list~will~not~be~sorted.
  }
⟨/package⟩
```

(*End definition for* `\__sort_disable_toksdef:` *and* `\__sort_disabled_toksdef:n`.)

`\__sort_too_long_error:NNw`

When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function `#1`.

```
\cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
  {
    \fi:
    \__msg_kernel_error:nnxxx { kernel } { too-large }
      { \token_to_str:N #2 }
      { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
      { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
    #1 \__sort_error:
  }
\__msg_kernel_new:nnnn { kernel } { too-large }
  { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
  {
    TeX~has~#2~toks~registers~still~available:~
    this~only~allows~to~sort~with~up~to~#3~
    items.~All~extra~items~will~be~deleted.
  }
```

(*End definition for* `\__sort_too_long_error:NNw`.)

```
\__msg_kernel_new:nnnn { kernel } { return-none }
  { The~comparison~code~did~not~return. }
  {
    When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
    did~not~call~
    \iow_char:N\\sort_return_same: ~nor~
    \iow_char:N\\sort_return_swapped: .~
    Exactly~one~of~these~should~be~called.
  }
\__msg_kernel_new:nnnn { kernel } { return-two }
  { The~comparison~code~returned~multiple~times. }
  {
    When~sorting~a~list,~the~code~to~compare~items~called~
    \iow_char:N\\sort_return_same: ~or~
    \iow_char:N\\sort_return_swapped: ~multiple~times.~
    Exactly~one~of~these~should~be~called.
  }
```

33.7 Deprecated functions

\sort_ordered:
\sort_reversed:

These functions were renamed for consistency.

```
\__debug_deprecation:nnNNpn { 2018-12-31 } { \sort_return_same: }
\cs_new_protected:Npn \sort_ordered: { \sort_return_same: }
\__debug_deprecation:nnNNpn { 2018-12-31 } { \sort_return_swapped: }
\cs_new_protected:Npn \sort_reversed: { \sort_return_swapped: }
```

(*End definition for* `\sort_ordered:` *and* `\sort_reversed:`.)

```
⟨/initex | package⟩
```

34 l3tl-build implementation

```
⟨*initex | package⟩
⟨@@=tl_build⟩
```

34.1 Variables and helper functions

\l__tl_build_start_index_int
\l__tl_build_index_int

Integers pointing to the starting index (currently always starts at zero), and the current index. The corresponding `\toks` are accessed directly by number.

```
\int_new:N \l__tl_build_start_index_int
\int_new:N \l__tl_build_index_int
```

(*End definition for* `\l__tl_build_start_index_int` *and* `\l__tl_build_index_int`.)

\l__tl_build_result_tl

The resulting token list is normally built in one go by unpacking all `\toks` in some range. In the rare cases where there are too many `\__tl_build_one:n` commands, leading to the depletion of registers, the contents of the current set of `\toks` is unpacked into `\l__tl_build_result_tl`. This prevents overflow from affecting the end-user (beyond an obvious performance hit).

```
\tl_new:N \l__tl_build_result_tl
```

(*End definition for* `\l__tl_build_result_tl`.)

__tl_build_unpack:
__tl_build_unpack_loop:w

The various pieces of the token list are built in `\toks` from the `start_index` (inclusive) to the (current) `index` (excluded). Those `\toks` are unpacked and stored in order in the `result` token list. Optimizations would be possible here, for instance, unpacking 10 `\toks` at a time with a macro expanding to `\the\toks#10...\the\toks#19`, but this should be kept for much later.

```
\cs_new_protected:Npn \__tl_build_unpack:
  {
    \tl_put_right:Nx \l__tl_build_result_tl
      {
        \exp_after:wN \__tl_build_unpack_loop:w
          \int_use:N \l__tl_build_start_index_int ;
        \__prg_break_point:
      }
  }
\cs_new:Npn \__tl_build_unpack_loop:w #1 ;
  {
    \if_int_compare:w #1 = \l__tl_build_index_int
      \exp_after:wN \__prg_break:
    \fi:
```

```
    \tex_the:D \tex_toks:D #1 \exp_stop_f:
    \exp_after:wN \__tl_build_unpack_loop:w
      \int_use:N \__int_eval:w #1 + 1 ;
  }
```

(*End definition for* `\__tl_build_unpack:` *and* `\__tl_build_unpack_loop:w`*.*)

34.2 Building the token list

`\__tl_build:Nw`
`\__tl_build_x:Nw`
`\__tl_gbuild:Nw`
`\__tl_gbuild_x:Nw`
`\__tl_build_aux:NNw`

Similar to what is done for coffins: redefine some command, here `\__tl_build_end_aux:n` to hold the relevant assignment (see `\__tl_build_end:` for details). Then initialize the start index and the current index at zero, and empty the `result` token list.

```
\cs_new_protected:Npn \__tl_build:Nw
  { \__tl_build_aux:NNw \tl_set:Nn }
\cs_new_protected:Npn \__tl_build_x:Nw
  { \__tl_build_aux:NNw \tl_set:Nx }
\cs_new_protected:Npn \__tl_gbuild:Nw
  { \__tl_build_aux:NNw \tl_gset:Nn }
\cs_new_protected:Npn \__tl_gbuild_x:Nw
  { \__tl_build_aux:NNw \tl_gset:Nx }
\cs_new_protected:Npn \__tl_build_aux:NNw #1#2
  {
    \group_begin:
      \cs_set:Npn \__tl_build_end_assignment:n
        { \group_end: #1 #2 }
      \int_zero:N \l__tl_build_start_index_int
      \int_zero:N \l__tl_build_index_int
      \tl_clear:N \l__tl_build_result_tl
  }
```

(*End definition for* `\__tl_build:Nw` *and others.*)

`\__tl_build_end:`
`\__tl_build_end_assignment:n`

When we are done building a token list, unpack all `\toks` into the `result` token list, and expand this list before closing the group. The `\__tl_build_end_assignment:n` function is defined by `\__tl_build_aux:NNw` to end the group and hold the relevant assignment. Its value outside is irrelevant, but just in case, we set it to a function which would clean up the contents of `\l__tl_build_result_tl`.

```
\cs_new_protected:Npn \__tl_build_end:
  {
      \__tl_build_unpack:
      \exp_args:No
    \__tl_build_end_assignment:n \l__tl_build_result_tl
  }
\cs_new_eq:NN \__tl_build_end_assignment:n \use_none:n
```

(*End definition for* `\__tl_build_end:` *and* `\__tl_build_end_assignment:n`*.*)

`\__tl_build_one:n`
`\__tl_build_one:o`
`\__tl_build_one:x`

Store the tokens in a free `\toks`, then move the pointer to the next one. If we overflow, unpack the current `\toks`, and reset the current index, preparing to fill more `\toks`. This could be optimized by avoiding to read `#1`, using `\afterassignment`.

```
\cs_new_protected:Npn \__tl_build_one:n #1
  {
    \tex_toks:D \l__tl_build_index_int {#1}
    \int_incr:N \l__tl_build_index_int
    \if_int_compare:w \l__tl_build_index_int > \c_max_register_int
      \__tl_build_unpack:
      \l__tl_build_index_int \l__tl_build_start_index_int
    \fi:
  }
\cs_new_protected:Npn \__tl_build_one:o #1
  {
    \tex_toks:D \l__tl_build_index_int \exp_after:wN {#1}
    \int_incr:N \l__tl_build_index_int
    \if_int_compare:w \l__tl_build_index_int > \c_max_register_int
      \__tl_build_unpack:
      \l__tl_build_index_int \l__tl_build_start_index_int
    \fi:
  }
\cs_new_protected:Npn \__tl_build_one:x #1
  { \use:x { \__tl_build_one:n {#1} } }
```

(*End definition for* `\__tl_build_one:n`.)

⟨/initex | package⟩

35 l3tl-analysis implementation

35.1 Internal functions

`\s__tl` The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

(*End definition for* `\s__tl`.)

`\__tl_analysis_map_inline:nn` `\__tl_analysis_map_inline:nn {⟨token list⟩} {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to each individual ⟨*token*⟩ in the ⟨*token list*⟩. The ⟨*inline function*⟩ receives three arguments:

- ⟨*tokens*⟩, which both o-expand and x-expand to the ⟨*token*⟩. The detailed form of ⟨*token*⟩ may change in later releases.
- ⟨*catcode*⟩, a capital hexadecimal digit which denotes the category code of the ⟨*token*⟩ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C:other, D:active).
- ⟨*char code*⟩, a decimal representation of the character code of the token, −1 if it is a control sequence (with ⟨*catcode*⟩ 0).

For optimizations in l3regex (when matching control sequences), it may be useful to provide a `\__tl_analysis_from_str_map_inline:nn` function, perhaps named `\__str_analysis_map_inline:nn`.

35.2 Internal format

The task of the l3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in l3regex where we need to support arbitrary tokens, and it is used in conversion functions in l3str-convert, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any ⟨*token*⟩ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find ⟨*tokens*⟩ which both o-expand and x-expand to the given ⟨*token*⟩. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

⟨*tokens*⟩ `\s__tl` ⟨*catcode*⟩ ⟨*char code*⟩ `\s__tl`

The ⟨*tokens*⟩ o- *and* x-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for l3str-convert). The ⟨*catcode*⟩ is given as a single hexadecimal digit, 0 for control sequences. The ⟨*char code*⟩ is given as a decimal number, −1 for control sequences.

Using delimited arguments lets us build the ⟨*tokens*⟩ progressively when doing an encoding conversion in l3str-convert. On the other hand, the delimiter `\s__tl` may not appear unbraced in ⟨*tokens*⟩. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a ⟨*token*⟩ to a balanced set of ⟨*tokens*⟩ which both o-expands and x-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl` 0 −1 `\s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl` 1 ⟨*char code*⟩ `\s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl` 2 ⟨*char code*⟩ `\s__tl`.
- A character with any other category code becomes `\exp_not:n {`⟨*character*⟩`} \s__tl` ⟨*hex catcode*⟩ ⟨*char code*⟩ `\s__tl`.

```
⟨*initex | package⟩
⟨@@=tl_analysis⟩
```

35.3 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\__int_value:w '#1 \s__tl` with `\__int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an x-expansion.

```
\__scan_new:N \s__tl
```

(*End definition for* `\s__tl`*.*)

`\l__tl_analysis_internal_tl` This token list variable is used to hand the argument of `\tl_show_analysis:n` to `\tl_show_analysis:N`.

```
\tl_new:N \l__tl_analysis_internal_tl
```

(*End definition for* `\l__tl_analysis_internal_tl`*.*)

\l__tl_analysis_token
\l__tl_analysis_char_token

The tokens in the token list are probed with the TeX primitive \futurelet. We use \l__tl_analysis_token in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is \l__tl_analysis_char_token.

```
\cs_new_eq:NN \l__tl_analysis_token ?
\cs_new_eq:NN \l__tl_analysis_char_token ?
```

(*End definition for \l__tl_analysis_token and \l__tl_analysis_char_token.*)

\l__tl_analysis_normal_int

The number of normal (N-type argument) tokens since the last special token.

```
\int_new:N \l__tl_analysis_normal_int
```

(*End definition for \l__tl_analysis_normal_int.*)

\l__tl_analysis_index_int

During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
\int_new:N \l__tl_analysis_index_int
```

(*End definition for \l__tl_analysis_index_int.*)

\l__tl_analysis_nesting_int

Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
\int_new:N \l__tl_analysis_nesting_int
```

(*End definition for \l__tl_analysis_nesting_int.*)

\l__tl_analysis_type_int

When encountering special characters, we record their “type” in this integer.

```
\int_new:N \l__tl_analysis_type_int
```

(*End definition for \l__tl_analysis_type_int.*)

\g__tl_analysis_result_tl

The result of the conversion is stored in this token list, with a succession of items of the form

⟨*tokens*⟩ \s__tl ⟨*catcode*⟩ ⟨*char code*⟩ \s__tl

```
\tl_new:N \g__tl_analysis_result_tl
```

(*End definition for \g__tl_analysis_result_tl.*)

__tl_analysis_extract_charcode:
__tl_analysis_extract_charcode_aux:w

Extracting the character code from the meaning of \l__tl_analysis_token. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form `⟨*char*⟩.

```
\cs_new:Npn \__tl_analysis_extract_charcode:
  {
    \exp_after:wN \__tl_analysis_extract_charcode_aux:w
      \token_to_meaning:N \l__tl_analysis_token
  }
\cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ` }
```

(*End definition for __tl_analysis_extract_charcode: and __tl_analysis_extract_charcode_aux:w.*)

__tl_analysis_cs_space_count:NN
__tl_analysis_cs_space_count:w
__tl_analysis_cs_space_count_end:w

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```
\cs_new:Npn \__tl_analysis_cs_space_count:NN #1 #2
  {
    \exp_after:wN #1
    \__int_value:w \__int_eval:w 0
      \exp_after:wN \__tl_analysis_cs_space_count:w
        \token_to_str:N #2
        \fi: \__tl_analysis_cs_space_count_end:w ; ~ !
  }
\cs_new:Npn \__tl_analysis_cs_space_count:w #1 ~
  {
    \if_false: #1 #1 \fi:
    + 1
    \__tl_analysis_cs_space_count:w
  }
\cs_new:Npn \__tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
  { \exp_after:wN ; \__int_value:w \str_count_ignore_spaces:n {#1} ; }
```

(*End definition for* `\__tl_analysis_cs_space_count:NN`, `\__tl_analysis_cs_space_count:w`, *and* `\__tl_analysis_cs_space_count_end:w`.)

35.4 Plan of attack

Our goal is to produce a token list of the form roughly

> ⟨*token 1*⟩ `\s__tl` ⟨*catcode 1*⟩ ⟨*char code 1*⟩ `\s__tl`
> ⟨*token 2*⟩ `\s__tl` ⟨*catcode 2*⟩ ⟨*char code 2*⟩ `\s__tl`
> ... ⟨*token N*⟩ `\s__tl` ⟨*catcode N*⟩ ⟨*char code N*⟩ `\s__tl`

Most but not all tokens can be grabbed as an undelimited (`N`-type) argument by TEX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an `x`-expanding assignment, normal tokens are taken in as `N`-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for TEX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an `N`-type argument).

The token itself can "look like" one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it "true" character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`\__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `\__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```
\cs_new_protected:Npn \__tl_analysis:n #1
  {
    \group_begin:
      \group_align_safe_begin:
        \__tl_analysis_a:n {#1}
        \__tl_analysis_b:n {#1}
      \group_align_safe_end:
    \group_end:
  }
```

(*End definition for* `\__tl_analysis:n`.)

35.5 Disabling active characters

`\__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For pTEX and upTEX we skip characters beyond $[0, 255]$ because `\lccode` only allows those values.

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_new_protected:Npn \__tl_analysis_disable:n #1
    {
      \tex_lccode:D 0 = #1 \exp_stop_f:
      \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
    }
  \cs_if_exist:NT \ptex_kanjiskip:D
    {
      \cs_gset_protected:Npn \__tl_analysis_disable:n #1
        {
          \if_int_compare:w 256 > #1 \exp_stop_f:
            \tex_lccode:D 0 = #1 \exp_stop_f:
            \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
          \fi:
        }
    }
\group_end:
```

(*End definition for* `\__tl_analysis_disable:n`.)

35.6 First pass

The goal of this pass is to detect special (non-`N`-type) tokens, and count how many `N`-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid `N`-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {`⟨*token*⟩`}` is non-empty, because the escape character is printable.

`\__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches -1 when we read the closing brace.

```
\cs_new_protected:Npn \__tl_analysis_a:n #1
  {
    \__tl_analysis_disable:n { 32 }
    \int_set:Nn \tex_escapechar:D { 92 }
    \int_zero:N \l__tl_analysis_normal_int
    \int_zero:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_nesting_int
    \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
    \int_decr:N \l__tl_analysis_index_int
  }
```

(*End definition for* `\__tl_analysis_a:n`.)

__tl_analysis_a_loop:w Read one character and check its type.

```
\cs_new_protected:Npn \__tl_analysis_a_loop:w
  { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }
```

(*End definition for* `\__tl_analysis_a_loop:w`*.*)

__tl_analysis_a_type:w At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```
\cs_new_protected:Npn \__tl_analysis_a_type:w
  {
    \l__tl_analysis_type_int =
      \if_meaning:w \l__tl_analysis_token \c_space_token
        0
      \else:
        \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
          1
        \else:
          \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
            - 1
          \else:
            2
          \fi:
        \fi:
      \fi:
      \exp_stop_f:
    \if_case:w \l__tl_analysis_type_int
         \exp_after:wN \__tl_analysis_a_space:w
    \or: \exp_after:wN \__tl_analysis_a_bgroup:w
    \or: \exp_after:wN \__tl_analysis_a_safe:N
    \else: \exp_after:wN \__tl_analysis_a_egroup:w
    \fi:
  }
```

(*End definition for* `\__tl_analysis_a_type:w`*.*)

__tl_analysis_a_space:w
__tl_analysis_a_space_test:w

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `\__tl_analysis_a_space_test:w`. Also, since `\__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```
\cs_new_protected:Npn \__tl_analysis_a_space:w
  {
    \tex_afterassignment:D \__tl_analysis_a_space_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
  }
\cs_new_protected:Npn \__tl_analysis_a_space_test:w
  {
    \if_meaning:w \l__tl_analysis_char_token \c_space_token
      \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
      \__tl_analysis_a_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_a_loop:w
  }
```

(*End definition for* `\__tl_analysis_a_space:w` *and* `\__tl_analysis_a_space_test:w`.)

`\__tl_analysis_a_bgroup:w`
`\__tl_analysis_a_egroup:w`
`\__tl_analysis_a_group:nw`
`\__tl_analysis_a_group_aux:w`
`\__tl_analysis_a_group_auxii:w`
`\__tl_analysis_a_group_test:w`

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```
\group_begin:
  \char_set_catcode_group_begin:N \^^@ % {
  \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
    { \__tl_analysis_a_group:nw { \exp_after:wN ^^@ \if_false: } \fi: } }
  \char_set_catcode_group_end:N \^^@
  \cs_new_protected:Npn \__tl_analysis_a_egroup:w
    { \__tl_analysis_a_group:nw { \if_false: { \fi: ^^@ } } % }
\group_end:
\cs_new_protected:Npn \__tl_analysis_a_group:nw #1
  {
    \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
    \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
    \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
      \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
    \fi:
    \__tl_analysis_disable:n { \tex_lccode:D 0 }
    \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
  }
\cs_new_protected:Npn \__tl_analysis_a_group_aux:w
  {
    \if_meaning:w \l__tl_analysis_token \tex_undefined:D
      \exp_after:wN \__tl_analysis_a_safe:N
    \else:
      \exp_after:wN \__tl_analysis_a_group_auxii:w
    \fi:
  }
\cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
  {
    \tex_afterassignment:D \__tl_analysis_a_group_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
  }
\cs_new_protected:Npn \__tl_analysis_a_group_test:w
  {
    \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
      \__tl_analysis_a_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_a_loop:w
  }
```

(*End definition for* `\__tl_analysis_a_bgroup:w` *and others.*)

`\__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a \skip register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```
\cs_new_protected:Npn \__tl_analysis_a_store:
  {
    \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
    \if_int_compare:w \tex_lccode:D 0 = ‘\ \exp_stop_f:
      \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
    \fi:
    \tex_skip:D \l__tl_analysis_index_int
      = \l__tl_analysis_normal_int sp plus \l__tl_analysis_type_int sp \scan_stop:
    \int_incr:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_normal_int
    \if_int_compare:w \l__tl_analysis_nesting_int = -1 \exp_stop_f:
      \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
    \fi:
  }
```

(*End definition for* __tl_analysis_a_store:*.*)

__tl_analysis_a_safe:N
__tl_analysis_a_cs:ww

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the \if_charcode:w test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one "normal" token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing \l__tl_analysis_index_int (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of "normal" tokens.

```
\cs_new_protected:Npn \__tl_analysis_a_safe:N #1
  {
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
        \scan_stop:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \__tl_analysis_disable:n { ‘#1 }
        \int_incr:N \l__tl_analysis_normal_int
      }
      { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
    \__tl_analysis_a_loop:w
  }
\cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
  {
    \if_int_compare:w #1 > 0 \exp_stop_f:
      \tex_skip:D \l__tl_analysis_index_int
        = \__int_eval:w \l__tl_analysis_normal_int + 1 sp \scan_stop:
```

```
      \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
    \else:
      \tex_advance:D
    \fi:
    \l__tl_analysis_normal_int #2 \exp_stop_f:
  }
```

(*End definition for* `\__tl_analysis_a_safe:N` *and* `\__tl_analysis_a_cs:ww`.)

35.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`\__tl_analysis_b:n`
`\__tl_analysis_b_loop:w`

Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```
\cs_new_protected:Npn \__tl_analysis_b:n #1
  {
    \tl_gset:Nx \g__tl_analysis_result_tl
      {
        \__tl_analysis_b_loop:w 0; #1
        \__prg_break_point:
      }
  }
\cs_new:Npn \__tl_analysis_b_loop:w #1;
  {
    \exp_after:wN \__tl_analysis_b_normals:ww
      \__int_value:w \tex_skip:D #1 ; #1 ;
  }
```

(*End definition for* `\__tl_analysis_b:n` *and* `\__tl_analysis_b_loop:w`.)

`\__tl_analysis_b_normals:ww`
`\__tl_analysis_b_normal:wwN`

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n {`⟨*token*⟩`} \s__tl` in the input stream (after x-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because #3 could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```
\cs_new:Npn \__tl_analysis_b_normals:ww #1;
  {
    \if_int_compare:w #1 = 0 \exp_stop_f:
      \__tl_analysis_b_special:w
    \fi:
    \__tl_analysis_b_normal:wwN #1;
  }
\cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
  {
    \exp_not:n { \exp_not:n { #3 } } \s__tl
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
```

```
        \scan_stop:
      \exp_after:wN \__tl_analysis_b_char:Nww
    \else:
      \exp_after:wN \__tl_analysis_b_cs:Nww
    \fi:
    #3 #1; #2;
  }
```

(*End definition for* `\__tl_analysis_b_normals:ww` *and* `\__tl_analysis_b_normal:wwN`.)

`\__tl_analysis_b_char:Nww`

If the normal token we grab is a character, leave ⟨*catcode*⟩ ⟨*charcode*⟩ followed by `\s__tl` in the input stream, and call `\__tl_analysis_b_normals:ww` with its first argument decremented.

```
\cs_new:Npx \__tl_analysis_b_char:Nww #1
  {
    \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
       \token_to_str:N D \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_catcode_other_token
      \token_to_str:N C \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
      \token_to_str:N B \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_math_toggle_token      3 \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_alignment_token        4 \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7 \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_math_subscript_token   8 \exp_not:N \else:
    \exp_not:N \if_catcode:w #1 \c_space_token
      \token_to_str:N A \exp_not:N \else:
      6
    \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
    \exp_not:N \__int_value:w `#1 \s__tl
   \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
     \exp_not:N \__int_value:w \exp_not:N \__int_eval:w - 1 +
  }
```

(*End definition for* `\__tl_analysis_b_char:Nww`.)

`\__tl_analysis_b_cs:Nww`
`\__tl_analysis_b_cs_test:ww`

If the token we grab is a control sequence, leave `0 -1` (as category code and character code) in the input stream, followed by `\s__tl`, and call `\__tl_analysis_b_normals:ww` with updated arguments.

```
\cs_new:Npn \__tl_analysis_b_cs:Nww #1
  {
    0 -1 \s__tl
    \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
  }
\cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
  {
    \exp_after:wN \__tl_analysis_b_normals:ww
    \__int_value:w \__int_eval:w
    \if_int_compare:w #1 = 0 \exp_stop_f:
      #3
    \else:
      \tex_skip:D \__int_eval:w #4 + #1 \__int_eval_end:
    \fi:
    - #2
    \exp_after:wN ;
    \__int_value:w \__int_eval:w #4 + #1 ;
  }
```

(*End definition for* `\__tl_analysis_b_cs:Nww` *and* `\__tl_analysis_b_cs_test:ww`.)

`\__tl_analysis_b_special:w`
`\__tl_analysis_b_special_char:wN`
`\__tl_analysis_b_special_space:w`

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when `x`-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `\__tl_analysis_b_loop:w` with the next index.

```
\group_begin:
  \char_set_catcode_other:N A
  \cs_new:Npn \__tl_analysis_b_special:w
      \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
    {
      \fi:
      \if_int_compare:w #1 = \l__tl_analysis_index_int
        \exp_after:wN \__prg_break:
      \fi:
      \tex_the:D \tex_toks:D #1 \s__tl
      \if_case:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
             \token_to_str:N A
      \or:   1
      \or:   1
      \else: 2
      \fi:
      \if_int_odd:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
        \exp_after:wN \__tl_analysis_b_special_char:wN \__int_value:w
      \else:
        \exp_after:wN \__tl_analysis_b_special_space:w \__int_value:w
      \fi:
      \__int_eval:w 1 + #1 \exp_after:wN ;
      \token_to_str:N
    }
\group_end:
\cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
  {
    \__int_value:w `#2 \s__tl
    \__tl_analysis_b_loop:w #1 ;
  }
\cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
  {
    32 \s__tl
    \__tl_analysis_b_loop:w #1 ;
  }
```

(*End definition for* `\__tl_analysis_b_special:w`*,* `\__tl_analysis_b_special_char:wN`*, and* `\__tl_analysis_b_special_space:w`.)

35.8 Mapping through the analysis

`\__tl_analysis_map_inline:nn`
`\__tl_analysis_map_inline_aux:Nn`

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__prg_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth.

That looping grabs the ⟨*tokens*⟩, ⟨*catcode*⟩ and ⟨*char code*⟩; it checks for the end of the loop with \use_none:n ##2, normally empty, but which becomes \tl_map_break: at the end; it then performs the user's code #2, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```
\cs_new_protected:Npn \__tl_analysis_map_inline:nn #1
  {
    \__tl_analysis:n {#1}
    \int_gincr:N \g__prg_map_int
    \exp_args:Nc \__tl_analysis_map_inline_aux:Nn
      { __tl_analysis_map_inline_ \int_use:N \g__prg_map_int :wNw }
  }
\cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
  {
    \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
      {
        \use_none:n ##2
        #2
        #1
      }
    \exp_after:wN #1
      \g__tl_analysis_result_tl
      \s__tl { ? \tl_map_break: } \s__tl
    \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
  }
```

(*End definition for* __tl_analysis_map_inline:nn *and* __tl_analysis_map_inline_aux:Nn.)

35.9 Showing the results

\tl_show_analysis:N
\tl_show_analysis:n
__tl_analysis_show:

Add to __tl_analysis:n a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as \tl_show:N by simply calling that function.

```
\cs_new_protected:Npn \tl_show_analysis:N #1
  {
    \tl_if_exist:NTF #1
      {
        \exp_args:No \__tl_analysis:n {#1}
        \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-tl-analysis }
          { \token_to_str:N #1 } { \tl_if_empty:NTF #1 { } { ? } } { } { }
        \__tl_analysis_show:
      }
      { \tl_show:N #1 }
  }
\cs_new_protected:Npn \tl_show_analysis:n #1
  {
    \__tl_analysis:n {#1}
    \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-tl-analysis }
      { } { \tl_if_empty:nTF {#1} { } { ? } } { } { }
    \__tl_analysis_show:
  }
\cs_new_protected:Npn \__tl_analysis_show:
  {
    \group_begin:
```

```
    \exp_args:NNx
    \group_end:
    \__msg_show_wrap:n
      {
        \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
          \s__tl { ? \__prg_break: } \s__tl
        \__prg_break_point:
      }
  }
```

(*End definition for* `\tl_show_analysis:N`*,* `\tl_show_analysis:n`*, and* `\__tl_analysis_show:`*. These functions are documented on page 198.*)

`\__tl_analysis_show_loop:wNw` Here, `#1` o- and x-expands to the token; `#2` is the category code (one uppercase hexadecimal digit), 0 for control sequences; `#3` is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```
\cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
  {
    \use_none:n #2
    \exp_not:n { \\ > \ \  }
    \if_int_compare:w "#2 = 0 \exp_stop_f:
      \exp_after:wN \__tl_analysis_show_cs:n
    \else:
      \if_int_compare:w "#2 = 13 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_active:n
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_normal:n
      \fi:
    \fi:
    {#1}
    \__tl_analysis_show_loop:wNw
  }
```

(*End definition for* `\__tl_analysis_show_loop:wNw`*.*)

`\__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TEX's alignment status.

```
\cs_new:Npn \__tl_analysis_show_normal:n #1
  {
    \exp_after:wN \token_to_str:N #1 ~
    ( \exp_after:wN \token_to_meaning:N #1 )
  }
```

(*End definition for* `\__tl_analysis_show_normal:n`*.*)

`\__tl_analysis_show_value:N` This expands to the value of `#1` if it has any.

```
\cs_new:Npn \__tl_analysis_show_value:N #1
  {
    \token_if_expandable:NF #1
      {
        \token_if_chardef:NTF       #1 \__prg_break: { }
        \token_if_mathchardef:NTF   #1 \__prg_break: { }
        \token_if_dim_register:NTF  #1 \__prg_break: { }
        \token_if_int_register:NTF  #1 \__prg_break: { }
        \token_if_skip_register:NTF #1 \__prg_break: { }
        \token_if_toks_register:NTF #1 \__prg_break: { }
        \use_none:nnn
        \__prg_break_point:
        \use:n { \exp_after:wN = \tex_the:D #1 }
      }
  }
```

(*End definition for* `\__tl_analysis_show_value:N`*.*)

`\__tl_analysis_show_cs:n`
`\__tl_analysis_show_active:n`
`\__tl_analysis_show_long:nn`
`\__tl_analysis_show_long_aux:nnnn`

Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c__tl_analysis_show_etc_str`.

```
\cs_new:Npn \__tl_analysis_show_cs:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
\cs_new:Npn \__tl_analysis_show_active:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }
\cs_new:Npn \__tl_analysis_show_long:nn #1
  {
    \__tl_analysis_show_long_aux:oofn
      { \token_to_str:N #1 }
      { \token_to_meaning:N #1 }
      { \__tl_analysis_show_value:N #1 }
  }
\cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
  {
     \int_compare:nNnTF
      { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
      > { \l_iow_line_count_int - 3 }
      {
        \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
          {
            \l_iow_line_count_int - 3
            - \str_count:N \c__tl_analysis_show_etc_str
          }
        \c__tl_analysis_show_etc_str
      }
      { #1 ~ ( #4 #2 #3 ) }
  }
\cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }
```

(*End definition for* `\__tl_analysis_show_cs:n` *and others.*)

35.10 Messages

`\c__tl_analysis_show_etc_str`

When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```
\tl_const:Nx \c__tl_analysis_show_etc_str % (
  { \token_to_str:N \ETC.) }
```

(*End definition for* `\c__tl_analysis_show_etc_str`.)

```
\__msg_kernel_new:nnn { kernel } { show-tl-analysis }
  {
    The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
    \tl_if_empty:nTF {#2}
      { is~empty }
      { contains~the~tokens: }
  }
⟨/initex | package⟩
```

36 l3regex implementation

```
⟨*initex | package⟩
⟨@@=regex⟩
```

36.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (backreferences come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since TEX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one "position") at a time, exploring in parallel every possible path ("active thread") through the NFA, considering active threads in an order determined by the quantifiers' greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer ⟨*position*⟩, with `min_pos` $- 1 \leq$ ⟨*position*⟩ $\leq$ `max_pos`. The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer ⟨*state*⟩ with `min_state` $\leq$ ⟨*state*⟩ $<$ `max_state`.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use l3intarray to manipulate arrays of integers (stored into some dimension registers in scaled points). We also abuse TeX’s `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When compiling, `\toks` registers are used under the hood by functions from the l3tl-build module. When building, `\toks`⟨*state*⟩ holds the tests and actions to perform in the ⟨*state*⟩ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last ⟨*step*⟩ in which each ⟨*state*⟩ was active.
- `\g__regex_thread_state_intarray` maps each ⟨*thread*⟩ (with `min_active` ≤ ⟨*thread*⟩ < `max_active`) to the ⟨*state*⟩ in which the ⟨*thread*⟩ currently is. The ⟨*threads*⟩ or ordered starting from the best to the least preferred.
- `\toks`⟨*thread*⟩ holds the submatch information for the ⟨*thread*⟩, as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each ⟨*position*⟩ in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.
- `\toks`⟨*position*⟩ holds ⟨*tokens*⟩ which `o`- and `x`-expand to the ⟨*position*⟩-th token in the query.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

`\count` registers are not abused, which means that we can safely use named integers in this module. Note that `\box` registers are not abused either; maybe we could leverage those for some purpose.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

36.2 Helpers

__regex_standard_escapechar:

Make the `\escapechar` into the standard backslash.

```
\cs_new_protected:Npn \__regex_standard_escapechar:
  { \int_set:Nn \tex_escapechar:D { `\\ } }
```

(*End definition for* `\__regex_standard_escapechar:`.)

`\__regex_toks_use:w`

Unpack a `\toks` given its number.

```
\cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(*End definition for* `\__regex_toks_use:w`*.*)

`\__regex_toks_clear:N`
`\__regex_toks_set:Nn`
`\__regex_toks_set:No`

Empty a `\toks` or set it to a value, given its number.

```
\cs_new_protected:Npn \__regex_toks_clear:N #1
  { \tex_toks:D #1 { } }
\cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
\cs_new_protected:Npn \__regex_toks_set:No #1
  { \__regex_toks_set:Nn #1 \exp_after:wN }
```

(*End definition for* `\__regex_toks_clear:N` *and* `\__regex_toks_set:Nn`*.*)

`\__regex_toks_memcpy:NNn`

Copy `#3` `\toks` registers from `#2` onwards to `#1` onwards, like `C`'s `memcpy`.

```
\cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
  {
    \prg_replicate:nn {#3}
      {
        \tex_toks:D #1 = \tex_toks:D #2
        \int_incr:N #1
        \int_incr:N #2
      }
  }
```

(*End definition for* `\__regex_toks_memcpy:NNn`*.*)

`\__regex_toks_put_left:Nx`
`\__regex_toks_put_right:Nx`
`\__regex_toks_put_right:Nn`

During the building phase we wish to add `x`-expanded material to `\toks`, either to the left or to the right. The expansion is done "by hand" for optimization (these operations are used quite a lot). The `Nn` version of `\__regex_toks_put_right:Nx` is provided because it is more efficient than `x`-expanding with `\exp_not:n`.

```
\cs_new_protected:Npn \__regex_toks_put_left:Nx #1#2
  {
    \cs_set:Npx \__regex_tmp:w { #2 }
    \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
      { \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Nx #1#2
  {
    \cs_set:Npx \__regex_tmp:w {#2}
    \tex_toks:D #1 \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
  { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(*End definition for* `\__regex_toks_put_left:Nx` *and* `\__regex_toks_put_right:Nx`*.*)

`\__regex_curr_cs_to_str:`

Expands to the string representation of the token (known to be a control sequence) at the current position `\l__regex_curr_pos_int`. It should only be used in `x`-expansion to avoid losing a leading space.

```
\cs_new:Npn \__regex_curr_cs_to_str:
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
    \tex_the:D \tex_toks:D \l__regex_curr_pos_int
  }
```

(*End definition for* __regex_curr_cs_to_str:.)

36.2.1 Constants and variables

__regex_tmp:w

Temporary function used for various short-term purposes.

```
\cs_new:Npn \__regex_tmp:w { }
```

(*End definition for* __regex_tmp:w.)

\l__regex_internal_a_tl
\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl

Temporary variables used for various purposes.

```
\tl_new:N   \l__regex_internal_a_tl
\tl_new:N   \l__regex_internal_b_tl
\int_new:N  \l__regex_internal_a_int
\int_new:N  \l__regex_internal_b_int
\int_new:N  \l__regex_internal_c_int
\bool_new:N \l__regex_internal_bool
\seq_new:N  \l__regex_internal_seq
\tl_new:N   \g__regex_internal_tl
```

(*End definition for* \l__regex_internal_a_tl *and others.*)

\c__regex_no_match_regex

This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using \regex_new:N.

```
\tl_const:Nn \c__regex_no_match_regex
  {
    \__regex_branch:n
      { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
  }
```

(*End definition for* \c__regex_no_match_regex.)

\g__regex_charcode_intarray
\g__regex_catcode_intarray
\g__regex_balance_intarray

The first thing we do when matching is to go once through the query token list and store the information for each token into \g__regex_charcode_intarray, \g__regex_catcode_intarray and \toks registers. We also store the balance of begin-group/end-group characters into \g__regex_balance_intarray.

```
\__intarray_new:Nn \g__regex_charcode_intarray { 65536 }
\__intarray_new:Nn \g__regex_catcode_intarray { 65536 }
\__intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(*End definition for* \g__regex_charcode_intarray, \g__regex_catcode_intarray, *and* \g__regex_balance_intarray.)

\l__regex_balance_int

During this phase, \l__regex_balance_int counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
\int_new:N \l__regex_balance_int
```

(*End definition for* \l__regex_balance_int.)

\l__regex_cs_name_tl

This variable is used in __regex_item_cs:n to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.

```
\tl_new:N \l__regex_cs_name_tl
```

(*End definition for* \l__regex_cs_name_tl.)

36.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int

```
\int_const:Nn \c__regex_ascii_min_int { 0 }
\int_const:Nn \c__regex_ascii_max_control_int { 31 }
\int_const:Nn \c__regex_ascii_max_int { 127 }
```

(*End definition for* `\c__regex_ascii_min_int`*,* `\c__regex_ascii_max_control_int`*, and* `\c__regex_ascii_max_int`*.*)

\c__regex_ascii_lower_int

```
\int_const:Nn \c__regex_ascii_lower_int { `a - `A }
```

(*End definition for* `\c__regex_ascii_lower_int`*.*)

__regex_break_point:TF
__regex_break_true:w

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

⟨*test1*⟩ . . . ⟨*test$_n$*⟩
`\__regex_break_point:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

If any of the tests succeeds, it calls `\__regex_break_true:w`, which cleans up and leaves ⟨*true code*⟩ in the input stream. Otherwise, `\__regex_break_point:TF` leaves the ⟨*false code*⟩ in the input stream.

```
\cs_new_protected:Npn \__regex_break_true:w
   #1 \__regex_break_point:TF #2 #3 {#2}
\cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(*End definition for* `\__regex_break_point:TF` *and* `\__regex_break_true:w`*.*)

__regex_item_reverse:n

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by -2, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
\cs_new_protected:Npn \__regex_item_reverse:n #1
  {
    #1
    \__regex_break_point:TF { } \__regex_break_true:w
  }
```

(*End definition for* `\__regex_item_reverse:n`*.*)

__regex_item_caseful_equal:n
__regex_item_caseful_range:nn

Simple comparisons triggering `\__regex_break_true:w` when true.

```
\cs_new_protected:Npn \__regex_item_caseful_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_item_caseful_equal:n` *and* `\__regex_item_caseful_range:nn`*.*)

`\__regex_item_caseless_equal:n`
`\__regex_item_caseless_range:nn`

For caseless matching, we perform the test both on the `current_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```
\cs_new_protected:Npn \__regex_item_caseless_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
    \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
      \__regex_compute_case_changed_char:
    \fi:
    \if_int_compare:w #1 = \l__regex_case_changed_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
    \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
      \__regex_compute_case_changed_char:
    \fi:
    \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_item_caseless_equal:n` *and* `\__regex_item_caseless_range:nn`*.*)

`\__regex_compute_case_changed_char:`

This function is called when `\l__regex_case_changed_char_int` has not yet been computed (or rather, when it is set to the marker value `\c_max_int`). If the current character code is in the range $[65, 90]$ (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range $[97, 122]$, subtract 32.

```
\cs_new_protected:Npn \__regex_compute_case_changed_char:
  {
    \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
    \if_int_compare:w \l__regex_curr_char_int > ‘Z \exp_stop_f:
      \if_int_compare:w \l__regex_curr_char_int > ‘z \exp_stop_f: \else:
        \if_int_compare:w \l__regex_curr_char_int < ‘a \exp_stop_f: \else:
          \int_sub:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
        \fi:
      \fi:
    \else:
      \if_int_compare:w \l__regex_curr_char_int < ‘A \exp_stop_f: \else:
        \int_add:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_compute_case_changed_char:`.)

`\__regex_item_equal:n`
`\__regex_item_range:nn`

Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```
\cs_new_eq:NN \__regex_item_equal:n ?
\cs_new_eq:NN \__regex_item_range:nn ?
```

(*End definition for* `\__regex_item_equal:n` *and* `\__regex_item_range:nn`.)

`\__regex_item_catcode:nT`
`\__regex_item_catcode_reverse:nT`
`\__regex_item_catcode:`

The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```
\cs_new_protected:Npn \__regex_item_catcode:
  {
    "
    \if_case:w \l__regex_curr_catcode_int
         1       \or: 4       \or: 10      \or: 40
    \or: 100     \or:         \or: 1000    \or: 4000
    \or: 10000   \or:         \or: 100000  \or: 400000
    \or: 1000000 \or: 4000000 \else: 1*0
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode:nT #1
  {
    \if_int_odd:w \__int_eval:w #1 / \__regex_item_catcode: \__int_eval_end:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
  { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }
```

(*End definition for* `\__regex_item_catcode:nT`, `\__regex_item_catcode_reverse:nT`, *and* `\__regex_item_catcode:`.)

`\__regex_item_exact:nn`
`\__regex_item_exact_cs:n`

This matches an exact ⟨*category*⟩-⟨*character code*⟩ pair, or an exact control sequence, more precisely one of several possible control sequences.

```
\cs_new_protected:Npn \__regex_item_exact:nn #1#2
  {
    \if_int_compare:w #1 = \l__regex_curr_catcode_int
      \if_int_compare:w #2 = \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_item_exact_cs:n #1
  {
    \int_compare:nNnTF \l__regex_curr_catcode_int = 0
      {
        \tl_set:Nx \l__regex_internal_a_tl
          { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
        \tl_if_in:noTF { \scan_stop: #1 \scan_stop: } \l__regex_internal_a_tl
          { \__regex_break_true:w } { }
      }
      { }
  }
```

(*End definition for* `\__regex_item_exact:nn` *and* `\__regex_item_exact_cs:n`.)

`\__regex_item_cs:n`

Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches. The three `\exp_after:wN` expand the contents of the `\toks`⟨*current position*⟩ (of the form `\exp_not:n {`⟨*control sequence*⟩`}`) to ⟨*control sequence*⟩. We store the cs name before building states for the cs, as those states may overlap with toks registers storing the user's input.

```
\cs_new_protected:Npn \__regex_item_cs:n #1
  {
    \int_compare:nNnT \l__regex_curr_catcode_int = 0
      {
        \group_begin:
          \tl_set:Nx \l__regex_cs_name_tl { \__regex_curr_cs_to_str: }
          \__regex_single_match:
          \__regex_disable_submatches:
          \__regex_build_for_cs:n {#1}
          \bool_set_eq:NN \l__regex_saved_success_bool \g__regex_success_bool
          \exp_args:NV \__regex_match:n \l__regex_cs_name_tl
          \if_meaning:w \c_true_bool \g__regex_success_bool
            \group_insert_after:N \__regex_break_true:w
          \fi:
          \bool_gset_eq:NN \g__regex_success_bool \l__regex_saved_success_bool
        \group_end:
      }
  }
```

(*End definition for* `\__regex_item_cs:n`.)

36.2.3 Character property tests

`\__regex_prop_d:`
`\__regex_prop_h:`
`\__regex_prop_s:`
`\__regex_prop_v:`
`\__regex_prop_w:`
`\__regex_prop_N:`

Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`, `\w=[0-9A-Z_a-z]`, `\s=[\␣\^^I\^^J\^^L\^^M]`, `\h=[\␣\^^I]`, `\v=[\^^J-\^^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```
\cs_new_protected:Npn \__regex_prop_d:
  { \__regex_item_caseful_range:nn { `0 } { `9 } }
\cs_new_protected:Npn \__regex_prop_h:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
  }
\cs_new_protected:Npn \__regex_prop_s:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
    \__regex_item_caseful_equal:n { `\^^J }
    \__regex_item_caseful_equal:n { `\^^L }
    \__regex_item_caseful_equal:n { `\^^M }
  }
\cs_new_protected:Npn \__regex_prop_v:
  { \__regex_item_caseful_range:nn { `\^^J } { `\^^M } } % lf, vtab, ff, cr
\cs_new_protected:Npn \__regex_prop_w:
  {
    \__regex_item_caseful_range:nn { `a } { `z }
    \__regex_item_caseful_range:nn { `A } { `Z }
    \__regex_item_caseful_range:nn { `0 } { `9 }
    \__regex_item_caseful_equal:n { `_ }
  }
\cs_new_protected:Npn \__regex_prop_N:
  {
    \__regex_item_reverse:n
      { \__regex_item_caseful_equal:n { `\^^J } }
  }
```

(*End definition for* `\__regex_prop_d:` *and others.*)

__regex_posix_alnum:
__regex_posix_alpha:
__regex_posix_ascii:
__regex_posix_blank:
__regex_posix_cntrl:
__regex_posix_digit:
__regex_posix_graph:
__regex_posix_lower:
__regex_posix_print:
__regex_posix_punct:
__regex_posix_space:
__regex_posix_upper:
__regex_posix_word:
__regex_posix_xdigit:

POSIX properties. No surprise.

```
\cs_new_protected:Npn \__regex_posix_alnum:
  { \__regex_posix_alpha: \__regex_posix_digit: }
\cs_new_protected:Npn \__regex_posix_alpha:
  { \__regex_posix_lower: \__regex_posix_upper: }
\cs_new_protected:Npn \__regex_posix_ascii:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\cs_new_protected:Npn \__regex_posix_cntrl:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_control_int
    \__regex_item_caseful_equal:n \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
\cs_new_protected:Npn \__regex_posix_graph:
  { \__regex_item_caseful_range:nn { `! } { `\~ } }
\cs_new_protected:Npn \__regex_posix_lower:
  { \__regex_item_caseful_range:nn { `a } { `z } }
\cs_new_protected:Npn \__regex_posix_print:
  { \__regex_item_caseful_range:nn { `\  } { `\~ } }
\cs_new_protected:Npn \__regex_posix_punct:
  {
    \__regex_item_caseful_range:nn { `! } { `/ }
    \__regex_item_caseful_range:nn { `: } { `@ }
    \__regex_item_caseful_range:nn { `[ } { `` }
    \__regex_item_caseful_range:nn { `\{ } { `\~ }
  }
\cs_new_protected:Npn \__regex_posix_space:
  {
    \__regex_item_caseful_equal:n { ‘\  }
    \__regex_item_caseful_range:nn { ‘\^^I } { ‘\^^M }
  }
\cs_new_protected:Npn \__regex_posix_upper:
  { \__regex_item_caseful_range:nn { ‘A } { ‘Z } }
\cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
\cs_new_protected:Npn \__regex_posix_xdigit:
  {
    \__regex_posix_digit:
    \__regex_item_caseful_range:nn { ‘A } { ‘F }
    \__regex_item_caseful_range:nn { ‘a } { ‘f }
  }
```

(*End definition for* `\__regex_posix_alnum:` *and others.*)

36.2.4 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\__regex_escape_use:nnnn` ⟨*inline 1*⟩ ⟨*inline 2*⟩ ⟨*inline 3*⟩ `{`⟨*token list*⟩`}` The ⟨*token list*⟩ is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function ⟨*inline 1*⟩, and escaped characters are fed to the function ⟨*inline 2*⟩ within an `x`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to ⟨*inline 3*⟩. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is mostly done within an `x`-expanding assignment, except for the `\x` escape sequence, which is not amenable to that in general. For this, we use the general framework of `\__tl_build:Nw`.

`\__regex_escape_use:nnnn`

The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Go through `#4` once, applying `#1`, `#2`, or `#3` as relevant to each character (after de-escaping it). Note that we cannot replace `\tl_set:Nx` and `\__tl_build_one:o` by a single call to `\__tl_build_one:x`, because the `x`-expanding assignment may be interrupted by `\x`.

```
\__debug_patch:nnNNpn
  {
    \__debug_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
    \__tl_build:Nw \l__regex_internal_a_tl
      \__tl_build_one:n { \__debug_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
      \use_none:nn
  }
  { }
\cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
```

```
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
      \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
      \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
      \__regex_standard_escapechar:
      \tl_gset:Nx \g__regex_internal_tl { \__str_to_other_fast:n {#4} }
      \tl_set:Nx \l__regex_internal_b_tl
        {
          \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
          { break } \__prg_break_point:
        }
      \__tl_build_one:o \l__regex_internal_b_tl
    \__tl_build_end:
    \l__regex_internal_a_tl
  }
```

(*End definition for* `\__regex_escape_use:nnnn`.)

__regex_escape_loop:N
__regex_escape_\:w

`\__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are "escaped".

```
\cs_new:Npn \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
      { \__regex_escape_unescaped:N #1 }
    \__regex_escape_loop:N
  }
\cs_new:cpn { __regex_escape_ \c_backslash_str :w }
    \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
      { \__regex_escape_escaped:N #1 }
    \__regex_escape_loop:N
  }
```

(*End definition for* `\__regex_escape_loop:N` *and* `\__regex_escape_\:w`.)

__regex_escape_unescaped:N
__regex_escape_escaped:N
__regex_escape_raw:N

Those functions are never called before being given a new meaning, so their definitions here don't matter.

```
\cs_new_eq:NN \__regex_escape_unescaped:N ?
\cs_new_eq:NN \__regex_escape_escaped:N   ?
\cs_new_eq:NN \__regex_escape_raw:N       ?
```

(*End definition for* `\__regex_escape_unescaped:N`, `\__regex_escape_escaped:N`, *and* `\__regex_escape_raw:N`.)

__regex_escape_break:w
__regex_escape_/break:w
__regex_escape_/a:w
__regex_escape_/e:w
__regex_escape_/f:w
__regex_escape_/n:w
__regex_escape_/r:w
__regex_escape_/t:w
__regex_escape_␣:w

The loop is ended upon seeing the end-marker "`break`", with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```
\cs_new_eq:NN \__regex_escape_break:w \__prg_break:
\cs_new:cpn { __regex_escape_/break:w }
  {
    \if_false: { \fi: }
    \__msg_kernel_error:nn { kernel } { trailing-backslash }
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
\cs_new:cpn { __regex_escape_~:w } { }
\cs_new:cpx { __regex_escape_/a:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
\cs_new:cpx { __regex_escape_/t:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
\cs_new:cpx { __regex_escape_/n:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
\cs_new:cpx { __regex_escape_/f:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
\cs_new:cpx { __regex_escape_/r:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
\cs_new:cpx { __regex_escape_/e:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }
```

(*End definition for* `\__regex_escape_break:w` *and others.*)

__regex_escape_/x:w
__regex_escape_x_end:w
__regex_escape_x_large:n

When `\x` is encountered, `\__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `\__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `\__regex_escape_raw:N` on the corresponding character token.

```
\cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
  {
    \exp_after:wN \__regex_escape_x_end:w
    \__int_value:w "0 \__regex_escape_x_test:N
  }
\cs_new:Npn \__regex_escape_x_end:w #1 ;
  {
    \int_compare:nNnTF {#1} > \c_max_char_int
      {
        \if_false: { \fi: }
        \__tl_build_one:o \l__regex_internal_b_tl
        \__msg_kernel_error:nnx { kernel } { x-overflow } {#1}
        \tl_set:Nx \l__regex_internal_b_tl
          { \if_false: } \fi:
      }
      {
        \exp_last_unbraced:Nf \__regex_escape_raw:N
          { \char_generate:nn {#1} { 12 } }
      }
  }
```

(*End definition for* `\__regex_escape_/x:w`, `\__regex_escape_x_end:w`, *and* `\__regex_escape_x_large:n`.)

__regex_escape_x_test:N
__regex_escape_x_testii:N

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `\__regex_escape_x_loop:N` or `\__regex_escape_x:N`.

```
\cs_new:Npn \__regex_escape_x_test:N #1
  {
    \str_if_eq_x:nnTF {#1} { break } { ; }
      {
        \if_charcode:w \c_space_token #1
```

```
          \exp_after:wN \__regex_escape_x_test:N
        \else:
          \exp_after:wN \__regex_escape_x_testii:N
          \exp_after:wN #1
        \fi:
      }
  }
\cs_new:Npn \__regex_escape_x_testii:N #1
  {
    \if_charcode:w \c_left_brace_str #1
      \exp_after:wN \__regex_escape_x_loop:N
    \else:
      \__regex_hexadecimal_use:NTF #1
        { \exp_after:wN \__regex_escape_x:N }
        { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
    \fi:
  }
```

(*End definition for* `\__regex_escape_x_test:N` *and* `\__regex_escape_x_testii:N`*.*)

__regex_escape_x:N

This looks for the second digit in the unbraced case.

```
\cs_new:Npn \__regex_escape_x:N #1
  {
    \str_if_eq_x:nnTF {#1} { break } { ; }
      {
        \__regex_hexadecimal_use:NTF #1
          { ; \__regex_escape_loop:N }
          { ; \__regex_escape_loop:N #1 }
      }
  }
```

(*End definition for* `\__regex_escape_x:N`*.*)

__regex_escape_x_loop:N
__regex_escape_x_loop_error:

Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```
\cs_new:Npn \__regex_escape_x_loop:N #1
  {
    \str_if_eq_x:nnTF {#1} { break }
      { ; \__regex_escape_x_loop_error:n { } {#1} }
      {
        \__regex_hexadecimal_use:NTF #1
          { \__regex_escape_x_loop:N }
          {
            \token_if_eq_charcode:NNTF \c_space_token #1
              { \__regex_escape_x_loop:N }
              {
                ;
                \exp_after:wN
                \token_if_eq_charcode:NNTF \c_right_brace_str #1
                  { \__regex_escape_loop:N }
                  { \__regex_escape_x_loop_error:n {#1} }
              }
          }
      }
  }
\cs_new:Npn \__regex_escape_x_loop_error:n #1
  {
    \if_false: { \fi: }
    \__tl_build_one:o \l__regex_internal_b_tl
    \__msg_kernel_error:nnx { kernel } { x-missing-rbrace } {#1}
    \tl_set:Nx \l__regex_internal_b_tl
      { \if_false: } \fi: \__regex_escape_loop:N #1
  }
```

(*End definition for* `\__regex_escape_x_loop:N` *and* `\__regex_escape_x_loop_error:`.)

__regex_hexadecimal_use:NTF

TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
\prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
  {
    \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w \__int_eval:w
          \exp_after:wN ` \token_to_str:N #1 - `a
        \__int_eval_end:
          A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\__regex_hexadecimal_use:NTF`.)

__regex_char_if_alphanumeric:NTF
__regex_char_if_special:NTF

These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumerics are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means `0`–`9`, `A`–`Z`, `a`–`z`; “special” character means non-alphanumeric but printable ascii, from space (hex `20`) to `del` (hex `7E`).

```
\prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
```

```
  {
    \if_int_compare:w `#1 > `Z \exp_stop_f:
      \if_int_compare:w `#1 > `z \exp_stop_f:
        \if_int_compare:w `#1 < \c__regex_ascii_max_int
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w `#1 < `a \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 > `9 \exp_stop_f:
        \if_int_compare:w `#1 < `A \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w `#1 < `0 \exp_stop_f:
          \if_int_compare:w `#1 < `\ \exp_stop_f:
            \prg_return_false: \else: \prg_return_true: \fi:
        \else: \prg_return_false: \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
  {
    \if_int_compare:w `#1 > `Z \exp_stop_f:
      \if_int_compare:w `#1 > `z \exp_stop_f:
        \prg_return_false:
      \else:
        \if_int_compare:w `#1 < `a \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 > `9 \exp_stop_f:
        \if_int_compare:w `#1 < `A \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \else:
        \if_int_compare:w `#1 < `0 \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_char_if_alphanumeric:NTF` *and* `\__regex_char_if_special:NTF`.)

36.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a "compiled" regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\__regex_class:NnnnN` ⟨*boolean*⟩ `{`⟨*tests*⟩`}` `{`⟨*min*⟩`}` `{`⟨*more*⟩`}` ⟨*lazyness*⟩
- `\__regex_group:nnnN` `{`⟨*branches*⟩`}` `{`⟨*min*⟩`}` `{`⟨*more*⟩`}` ⟨*lazyness*⟩, also `\__regex_group_no_capture:nnnN` and `\__regex_group_resetting:nnnN` with the same syntax.

- `\__regex_branch:n` {⟨*contents*⟩}

- `\__regex_command_K:`

- `\__regex_assertion:Nn` ⟨*boolean*⟩ {⟨*assertion test*⟩}, where the ⟨*assertion test*⟩ is `\__regex_b_test:` or {`\__regex_anchor:N` ⟨*integer*⟩}

Tests can be the following:

- `\__regex_item_caseful_equal:n` {⟨*char code*⟩}

- `\__regex_item_caseless_equal:n` {⟨*char code*⟩}

- `\__regex_item_caseful_range:nn` {⟨*min*⟩} {⟨*max*⟩}

- `\__regex_item_caseless_range:nn` {⟨*min*⟩} {⟨*max*⟩}

- `\__regex_item_catcode:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}

- `\__regex_item_catcode_reverse:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}

- `\__regex_item_reverse:n` {⟨*tests*⟩}

- `\__regex_item_exact:nn` {⟨*catcode*⟩} {⟨*char code*⟩}

- `\__regex_item_exact_cs:n` {⟨*csnames*⟩}, more precisely given as ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ and so on in a brace group.

- `\__regex_item_cs:n` {⟨*compiled regex*⟩}

36.3.1 Variables used when compiling

`\l__regex_group_level_int`

We make sure to open the same number of groups as we close.

```
\int_new:N \l__regex_group_level_int
```

(*End definition for* `\l__regex_group_level_int`*.*)

`\l__regex_mode_int`
`\c__regex_cs_in_class_mode_int`
`\c__regex_cs_mode_int`
`\c__regex_outer_mode_int`
`\c__regex_catcode_mode_int`
`\c__regex_class_mode_int`
`\c__regex_catcode_in_class_mode_int`

While compiling, ten modes are recognized, labelled -63, -23, -6, -2, 0, 2, 3, 6, 23, 63. See section 36.3.3. We only define some of these as constants.

```
\int_new:N \l__regex_mode_int
\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\int_const:Nn \c__regex_cs_mode_int { -2 }
\int_const:Nn \c__regex_outer_mode_int { 0 }
\int_const:Nn \c__regex_catcode_mode_int { 2 }
\int_const:Nn \c__regex_class_mode_int { 3 }
\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(*End definition for* `\l__regex_mode_int` *and others.*)

`\l__regex_catcodes_int`
`\l__regex_default_catcodes_int`
`\l__regex_catcodes_bool`

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c, for all allowed catcodes c. The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape.

The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
\int_new:N \l__regex_catcodes_int
\int_new:N \l__regex_default_catcodes_int
\bool_new:N \l__regex_catcodes_bool
```

(*End definition for* `\l__regex_catcodes_int`*,* `\l__regex_default_catcodes_int`*, and* `\l__regex_catcodes_bool`*.*)

`\c__regex_catcode_C_int`
`\c__regex_catcode_B_int`
`\c__regex_catcode_E_int`
`\c__regex_catcode_M_int`
`\c__regex_catcode_T_int`
`\c__regex_catcode_P_int`
`\c__regex_catcode_U_int`
`\c__regex_catcode_D_int`
`\c__regex_catcode_S_int`
`\c__regex_catcode_L_int`
`\c__regex_catcode_O_int`
`\c__regex_catcode_A_int`
`\c__regex_all_catcodes_int`

Constants: 4^c for each category, and the sum of all powers of 4.

```
\int_const:Nn \c__regex_catcode_C_int { "1 }
\int_const:Nn \c__regex_catcode_B_int { "4 }
\int_const:Nn \c__regex_catcode_E_int { "10 }
\int_const:Nn \c__regex_catcode_M_int { "40 }
\int_const:Nn \c__regex_catcode_T_int { "100 }
\int_const:Nn \c__regex_catcode_P_int { "1000 }
\int_const:Nn \c__regex_catcode_U_int { "4000 }
\int_const:Nn \c__regex_catcode_D_int { "10000 }
\int_const:Nn \c__regex_catcode_S_int { "100000 }
\int_const:Nn \c__regex_catcode_L_int { "400000 }
\int_const:Nn \c__regex_catcode_O_int { "1000000 }
\int_const:Nn \c__regex_catcode_A_int { "4000000 }
\int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(*End definition for* `\c__regex_catcode_C_int` *and others.*)

`\l__regex_internal_regex`

The compilation step stores its result in this variable.

```
\cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(*End definition for* `\l__regex_internal_regex`*.*)

`\l__regex_show_prefix_seq`

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
\seq_new:N \l__regex_show_prefix_seq
```

(*End definition for* `\l__regex_show_prefix_seq`*.*)

`\l__regex_show_lines_int`

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
\int_new:N \l__regex_show_lines_int
```

(*End definition for* `\l__regex_show_lines_int`*.*)

36.3.2 Generic helpers used when compiling

`\__regex_get_digits:NTFw`
`\__regex_get_digits_loop:w`

If followed by some raw digits, collect them one by one in the integer variable `#1`, and take the `true` branch. Otherwise, take the `false` branch.

```
\cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
  {
    \__regex_if_raw_digit:NNTF #4 #5
      { #1 = #5 \__regex_get_digits_loop:nw {#2} }
      { #3 #4 #5 }
  }
```

```
\cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
  {
    \__regex_if_raw_digit:NNTF #2 #3
      { #3 \__regex_get_digits_loop:nw {#1} }
      { \scan_stop: #1 #2 #3 }
  }
```

(*End definition for* `\__regex_get_digits:NTFw` *and* `\__regex_get_digits_loop:w`.)

`\__regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
\prg_new_conditional:Npnn \__regex_if_raw_digit:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \if_int_compare:w 1 < 1 #2 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__regex_if_raw_digit:NNTF`.)

36.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

-6 `[\c{...}]` control sequence in a class,

-2 `\c{...}` control sequence,

0 `...` outer,

2 `\c...` catcode test,

6 `[\c...]` catcode test in a class,

-63 `[\c{[...]}]` class inside mode −6,

-23 `\c{[...]}` class inside mode −2,

3 `[...]` class inside mode 0,

23 `\c[...]` class inside mode 2,

63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \to (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`\__regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
\cs_new:Npn \__regex_if_in_class:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End definition for* `\__regex_if_in_class:TF`.)

`\__regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
\cs_new:Npn \__regex_if_in_cs:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_ii:nn
    \else:
      \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_if_in_cs:TF`.)

`\__regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```
\cs_new:Npn \__regex_if_in_class_or_catcode:TF
  {
    \if_int_odd:w \l__regex_mode_int
```

```
      \exp_after:wN \use_i:nn
    \else:
      \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_if_in_class_or_catcode:TF`*.*)

__regex_if_within_catcode:TF

This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
\cs_new:Npn \__regex_if_within_catcode:TF
  {
    \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End definition for* `\__regex_if_within_catcode:TF`*.*)

__regex_chk_c_allowed:T

The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
\cs_new_protected:Npn \__regex_chk_c_allowed:T
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \exp_after:wN \use:n
    \else:
      \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use:n
      \else:
        \__msg_kernel_error:nn { kernel } { c-bad-mode }
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_chk_c_allowed:T`*.*)

__regex_mode_quit_c:

This function changes the mode as it is needed just after a catcode test.

```
\cs_new_protected:Npn \__regex_mode_quit_c:
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_in_class_mode_int
        \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_mode_quit_c:`*.*)

36.3.4 Framework

__regex_compile:w
__regex_compile_end:

Used when compiling a user regex or a regex for the \c{...} escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building \l__regex_internal_regex.

```
\cs_new_protected:Npn \__regex_compile:w
  {
    \__tl_build_x:Nw \l__regex_internal_regex
      \int_zero:N \l__regex_group_level_int
      \int_set_eq:NN \l__regex_default_catcodes_int \c__regex_all_catcodes_int
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
      \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      \__tl_build_one:n { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_end:
  {
      \__regex_if_in_class:TF
        {
          \__msg_kernel_error:nn { kernel } { missing-rbrack }
          \use:c { __regex_compile_]: }
          \prg_do_nothing: \prg_do_nothing:
        }
        { }
      \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
        \__msg_kernel_error:nnx { kernel } { missing-rparen }
          { \int_use:N \l__regex_group_level_int }
        \prg_replicate:nn
          { \l__regex_group_level_int }
          {
              \__tl_build_one:n
                {
                  \if_false: { \fi: }
                  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
                }
            \__tl_build_end:
            \__tl_build_one:o \l__regex_internal_regex
          }
      \fi:
      \__tl_build_one:n { \if_false: { \fi: } }
    \__tl_build_end:
  }
```

(*End definition for __regex_compile:w and __regex_compile_end:.*)

__regex_compile:n

The compilation is done between __regex_compile:w and __regex_compile_end:, starting in mode 0. Then __regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since __regex_compile_end: does that. However, catch the case of a trailing \cL construction.

```
\cs_new_protected:Npn \__regex_compile:n #1
  {
    \__regex_compile:w
      \__regex_standard_escapechar:
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
      \__regex_escape_use:nnnn
        {
          \__regex_char_if_special:NTF ##1
            \__regex_compile_special:N \__regex_compile_raw:N ##1
        }
        {
          \__regex_char_if_alphanumeric:NTF ##1
            \__regex_compile_escaped:N \__regex_compile_raw:N ##1
        }
        { \__regex_compile_raw:N ##1 }
        { #1 }
      \prg_do_nothing: \prg_do_nothing:
      \prg_do_nothing: \prg_do_nothing:
      \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
        { \__msg_kernel_error:nn { kernel } { c-trailing } }
      \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
        {
          \__msg_kernel_error:nn { kernel } { c-missing-rbrace }
          \__regex_compile_end_cs:
          \prg_do_nothing: \prg_do_nothing:
          \prg_do_nothing: \prg_do_nothing:
        }
    \__regex_compile_end:
  }
```

(*End definition for `\__regex_compile:n`.*)

`\__regex_compile_escaped:N`
`\__regex_compile_special:N`

If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```
\cs_new_protected:Npn \__regex_compile_special:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_#1: }
      { \__regex_compile_raw:N #1 }
  }
\cs_new_protected:Npn \__regex_compile_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_/#1: }
      { \__regex_compile_raw:N #1 }
  }
```

(*End definition for `\__regex_compile_escaped:N` and `\__regex_compile_special:N`.*)

`\__regex_compile_one:x`

This is used after finding one "test", such as `\d`, or a raw character. If that followed a catcode test (*e.g.*, `\cL`), then restore the mode. If we are not in a class, then the test is "standalone", and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```
\cs_new_protected:Npn \__regex_compile_one:x #1
```

```
  {
    \__regex_mode_quit_c:
    \__regex_if_in_class:TF { }
      {
        \__tl_build_one:n
          { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
      }
    \__tl_build_one:x
      {
        \if_int_compare:w \l__regex_catcodes_int < \c__regex_all_catcodes_int
          \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
            { \exp_not:N \exp_not:n {#1} }
        \else:
          \exp_not:N \exp_not:n {#1}
        \fi:
      }
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
  }
```

(*End definition for* `\__regex_compile_one:x`.)

__regex_compile_abort_tokens:n
__regex_compile_abort_tokens:x

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```
\cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
  {
    \use:x
      {
        \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
          \__regex_compile_raw:N
      }
  }
\cs_generate_variant:Nn \__regex_compile_abort_tokens:n { x }
```

(*End definition for* `\__regex_compile_abort_tokens:n`.)

36.3.5 Quantifiers

__regex_compile_quantifier:w

This looks ahead and finds any quantifier (special character equal to either of `?+*{`).

```
\cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
          { \__regex_compile_quantifier_none: #1 #2 }
      }
      { \__regex_compile_quantifier_none: #1 #2 }
  }
```

(*End definition for* `\__regex_compile_quantifier:w`.)

__regex_compile_quantifier_none:
__regex_compile_quantifier_abort:xNN

Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```
\cs_new_protected:Npn \__regex_compile_quantifier_none:
  { \__tl_build_one:n { \if_false: { \fi: } { 1 } { 0 } \c_false_bool } }
\cs_new_protected:Npn \__regex_compile_quantifier_abort:xNN #1#2#3
  {
    \__regex_compile_quantifier_none:
    \__msg_kernel_warning:nnxx { kernel } { invalid-quantifier } {#1} {#3}
    \__regex_compile_abort_tokens:x {#1}
    #2 #3
  }
```

(*End definition for* `\__regex_compile_quantifier_none:` *and* `\__regex_compile_quantifier_abort:xNN`*.*)

__regex_compile_quantifier_lazyness:nnNN

Once the "main" quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `\__regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
\cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
  {
    \str_if_eq:nnTF { #3 #4 } { \__regex_compile_special:N ? }
      { \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_true_bool } }
      {
        \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
        #3 #4
      }
  }
```

(*End definition for* `\__regex_compile_quantifier_lazyness:nnNN`*.*)

__regex_compile_quantifier_?:w
__regex_compile_quantifier_*:w
__regex_compile_quantifier_+:w

For each "basic" quantifier, `?`, `*`, `+`, feed the correct arguments to `\__regex_compile_quantifier_lazyness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```
\cs_new_protected:cpn { __regex_compile_quantifier_?:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_*:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_+:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(*End definition for* `\__regex_compile_quantifier_?:w`*,* `\__regex_compile_quantifier_*:w`*, and* `\__regex_compile_quantifier_+:w`*.*)

__regex_compile_quantifier_{:w
__regex_compile_quantifier_braced_auxi:w
__regex_compile_quantifier_braced_auxii:w
__regex_compile_quantifier_braced_auxiii:w

Three possible syntaxes: `{`⟨*int*⟩`}`, `{`⟨*int*⟩`,}`, or `{`⟨*int*⟩`,`⟨*int*⟩`}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is $[a,a]$. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a,\infty]$ or $[a,b]$, encoded as $\{a\}\{-1\}$ and $\{a\}\{b-a\}$.

```
\cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
  {
    \__regex_get_digits:NTFw \l__regex_internal_a_int
      { \__regex_compile_quantifier_braced_auxi:w }
      { \__regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
  {
    \str_case_x:nnF { #1 #2 }
      {
        { \__regex_compile_special:N \c_right_brace_str }
          {
            \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
              { \int_use:N \l__regex_internal_a_int } { 0 }
          }
        { \__regex_compile_special:N , }
          {
            \__regex_get_digits:NTFw \l__regex_internal_b_int
              { \__regex_compile_quantifier_braced_auxiii:w }
              { \__regex_compile_quantifier_braced_auxii:w }
          }
      }
      {
        \__regex_compile_quantifier_abort:xNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
  {
    \str_if_eq_x:nnTF
      { #1 #2 } { \__regex_compile_special:N \c_right_brace_str }
      {
        \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int } { -1 }
      }
      {
        \__regex_compile_quantifier_abort:xNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
  {
    \str_if_eq_x:nnTF
      { #1 #2 } { \__regex_compile_special:N \c_right_brace_str }
      {
        \if_int_compare:w \l__regex_internal_a_int > \l__regex_internal_b_int
          \__msg_kernel_error:nnxx { kernel } { backwards-quantifier }
            { \int_use:N \l__regex_internal_a_int }
            { \int_use:N \l__regex_internal_b_int }
          \int_zero:N \l__regex_internal_b_int
        \else:
          \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
        \fi:
        \exp_args:Noo \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int }
          { \int_use:N \l__regex_internal_b_int }
      }
      {
        \__regex_compile_quantifier_abort:xNN
          {
            \c_left_brace_str
            \int_use:N \l__regex_internal_a_int ,
            \int_use:N \l__regex_internal_b_int
          }
        #1 #2
      }
  }
```

(*End definition for* `\__regex_compile_quantifier_{:w` *and others.*)

36.3.6 Raw characters

`\__regex_compile_raw_error:N`

Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```
\cs_new_protected:Npn \__regex_compile_raw_error:N #1
  {
    \__msg_kernel_error:nnx { kernel } { bad-escape } {#1}
    \__regex_compile_raw:N #1
  }
```

(*End definition for* `\__regex_compile_raw_error:N`.)

`\__regex_compile_raw:N`

If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```
\cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
  {
    \__regex_if_in_class:TF
      {
        \str_if_eq:nnTF {#2#3} { \__regex_compile_special:N - }
          { \__regex_compile_range:Nw #1 }
          {
            \__regex_compile_one:x
              { \__regex_item_equal:n { \__int_value:w `#1 ~ } }
            #2 #3
          }
      }
      {
        \__regex_compile_one:x
          { \__regex_item_equal:n { \__int_value:w `#1 ~ } }
        #2 #3
      }
  }
```

(*End definition for* `\__regex_compile_raw:N`.)

`\__regex_compile_range:Nw`
`\__regex_if_end_range:NNTF`

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```
\prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \prg_return_true:
    \else:
      \if_meaning:w \__regex_compile_special:N #1
        \if_charcode:w ] #2
          \prg_return_false:
        \else:
          \prg_return_true:
        \fi:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
  {
    \__regex_if_end_range:NNTF #2 #3
      {
        \if_int_compare:w ‘#1 > ‘#3 \exp_stop_f:
          \__msg_kernel_error:nnxx { kernel } { range-backwards } {#1} {#3}
        \else:
          \__tl_build_one:x
            {
              \if_int_compare:w ‘#1 = ‘#3 \exp_stop_f:
                \__regex_item_equal:n
              \else:
                \__regex_item_range:nn { \__int_value:w ‘#1 ~ }
              \fi:
              { \__int_value:w ‘#3 ~ }
            }
        \fi:
      }
      {
        \__msg_kernel_warning:nnxx { kernel } { range-missing-end }
          {#1} { \c_backslash_str #3 }
        \__tl_build_one:x
          {
            \__regex_item_equal:n { \__int_value:w ‘#1 ~ }
            \__regex_item_equal:n { \__int_value:w ‘- ~ }
          }
        #2#3
      }
  }
```

(*End definition for* `\__regex_compile_range:Nw` *and* `\__regex_if_end_range:NNTF`.)

36.3.7 Character properties

__regex_compile_.:
__regex_prop_.:

In a class, the dot has no special meaning. Outside, insert `\__regex_prop_.:`, which matches any character or control sequence, and refuses −2 (end-marker).

```
\cs_new_protected:cpx { __regex_compile_.: }
  {
    \exp_not:N \__regex_if_in_class:TF
      { \__regex_compile_raw:N . }
      { \__regex_compile_one:x \exp_not:c { __regex_prop_.: } }
  }
\cs_new_protected:cpn { __regex_prop_.: }
```

```
  {
    \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End definition for* `\__regex_compile_.:` *and* `\__regex_prop_.:`*.*)

__regex_compile_/d:
__regex_compile_/D:
__regex_compile_/h:
__regex_compile_/H:
__regex_compile_/s:
__regex_compile_/S:
__regex_compile_/v:
__regex_compile_/V:
__regex_compile_/w:
__regex_compile_/W:
__regex_compile_/N:

The constants `\__regex_prop_d:`, *etc.* hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpx { __regex_compile_/#1: }
      { \__regex_compile_one:x \exp_not:c { __regex_prop_#1: } }
    \cs_new_protected:cpx { __regex_compile_/#2: }
      {
        \__regex_compile_one:x
          { \__regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
      }
  }
\__regex_tmp:w d D
\__regex_tmp:w h H
\__regex_tmp:w s S
\__regex_tmp:w v V
\__regex_tmp:w w W
\cs_new_protected:cpn { __regex_compile_/N: }
  { \__regex_compile_one:x \__regex_prop_N: }
```

(*End definition for* `\__regex_compile_/d:` *and others.*)

36.3.8 Anchoring and simple assertions

__regex_compile_anchor:NF
__regex_compile_^:
__regex_compile_/A:
__regex_compile_/G:
__regex_compile_$:
__regex_compile_/Z:
__regex_compile_/z:

In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer `#1`. In other modes, `#2` treats the character as raw, with an error for escaped letters (`$` is valid in a class, but `\A` is definitely a mistake on the user's part).

```
\cs_new_protected:Npn \__regex_compile_anchor:NF #1#2
  {
    \__regex_if_in_class_or_catcode:TF {#2}
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_true_bool { \__regex_anchor:N #1 } }
      }
  }
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpn { __regex_compile_/#1: }
      { \__regex_compile_anchor:NF #2 { \__regex_compile_raw_error:N #1 } }
  }
\__regex_tmp:w A \l__regex_min_pos_int
\__regex_tmp:w G \l__regex_start_pos_int
\__regex_tmp:w Z \l__regex_max_pos_int
\__regex_tmp:w z \l__regex_max_pos_int
```

```
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpn { __regex_compile_#1: }
      { \__regex_compile_anchor:NF #2 { \__regex_compile_raw:N #1 } }
  }
\exp_args:Nx \__regex_tmp:w { \iow_char:N \^ } \l__regex_min_pos_int
\exp_args:Nx \__regex_tmp:w { \iow_char:N \$ } \l__regex_max_pos_int
```

(*End definition for* `\__regex_compile_anchor:NF` *and others.*)

__regex_compile_/b:
__regex_compile_/B:

Contrarily to ^ and $, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

```
\cs_new_protected:cpn { __regex_compile_/b: }
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N b }
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_true_bool { \__regex_b_test: } }
      }
  }
\cs_new_protected:cpn { __regex_compile_/B: }
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N B }
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_false_bool { \__regex_b_test: } }
      }
  }
```

(*End definition for* `\__regex_compile_/b:` *and* `\__regex_compile_/B:`*.*)

36.3.9 Character classes

__regex_compile_]:

Outside a class, right brackets have no meaning. In a class, change the mode ($m \to (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[...\cL[...]...]`). quantifiers.

```
\cs_new_protected:cpn { __regex_compile_]: }
  {
    \__regex_if_in_class:TF
      {
        \if_int_compare:w \l__regex_mode_int > \c__regex_catcode_in_class_mode_int
          \__tl_build_one:n { \if_false: { \fi: } }
        \fi:
        \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
        \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
        \if_int_odd:w \l__regex_mode_int \else:
          \exp_after:wN \__regex_compile_quantifier:w
        \fi:
      }
      { \__regex_compile_raw:N ] }
  }
```

(*End definition for* `\__regex_compile_]:`.)

`\__regex_compile_[:` In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c`⟨*category*⟩, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, −2 and −6) just parse the class. The mode is updated later.

```
\cs_new_protected:cpn { __regex_compile_[: }
  {
    \__regex_if_in_class:TF
      { \__regex_compile_class_posix_test:w }
      {
        \__regex_if_within_catcode:TF
          {
            \exp_after:wN \__regex_compile_class_catcode:w
              \int_use:N \l__regex_catcodes_int ;
          }
          { \__regex_compile_class_normal:w }
      }
  }
```

(*End definition for* `\__regex_compile_[:`.)

`\__regex_compile_class_normal:w` In the “normal” case, we insert `\__regex_class:NnnnN` ⟨*boolean*⟩ in the compiled code. The ⟨*boolean*⟩ is true for positive classes, and false for negative classes, characterized by a leading `^`. The auxiliary `\__regex_compile_class:TFNN` also checks for a leading `]` which has a special meaning.

```
\cs_new_protected:Npn \__regex_compile_class_normal:w
  {
    \__regex_compile_class:TFNN
      { \__regex_class:NnnnN \c_true_bool }
      { \__regex_class:NnnnN \c_false_bool }
  }
```

(*End definition for* `\__regex_compile_class_normal:w`.)

`\__regex_compile_class_catcode:w` This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting `\__regex_item_catcode:nT` or the `reverse` variant as appropriate, each with the current catcodes bitmap `#1` as an argument, and reset the catcodes.

```
\cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \__tl_build_one:n
        { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
    \fi:
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_compile_class:TFNN
      { \__regex_item_catcode:nT {#1} }
      { \__regex_item_catcode_reverse:nT {#1} }
  }
```

(*End definition for* `\__regex_compile_class_catcode:w`.)

__regex_compile_class:TFNN
__regex_compile_class:NN

If the first character is ^, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```
\cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
  {
    \l__regex_mode_int = \__int_value:w \l__regex_mode_int 3 \exp_stop_f:
    \str_if_eq:nnTF { #3 #4 } { \__regex_compile_special:N ^ }
      {
        \__tl_build_one:n { #2 { \if_false: } \fi: }
        \__regex_compile_class:NN
      }
      {
        \__tl_build_one:n { #1 { \if_false: } \fi: }
        \__regex_compile_class:NN #3 #4
      }
  }
\cs_new_protected:Npn \__regex_compile_class:NN #1#2
  {
    \token_if_eq_charcode:NNTF #2 ]
      { \__regex_compile_raw:N #2 }
      { #1 #2 }
  }
```

(*End definition for __regex_compile_class:TFNN and __regex_compile_class:NN.*)

__regex_compile_class_posix_test:w
__regex_compile_class_posix:NNNNw
__regex_compile_class_posix_loop:w
__regex_compile_class_posix_end:w

Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra __regex_item_reverse:n for negative classes.

```
\cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
  {
    \token_if_eq_meaning:NNT \__regex_compile_special:N #1
      {
        \str_case:nn { #2 }
          {
            : { \__regex_compile_class_posix:NNNNw }
            = { \__msg_kernel_warning:nnx { kernel } { posix-unsupported } { = } }
            . { \__msg_kernel_warning:nnx { kernel } { posix-unsupported } { . } }
          }
      }
    \__regex_compile_raw:N [ #1 #2
  }
\cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
  {
    \str_if_eq:nnTF { #5 #6 } { \__regex_compile_special:N ^ }
      {
        \bool_set_false:N \l__regex_internal_bool
        \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w
      }
      {
        \bool_set_true:N \l__regex_internal_bool
        \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w #5 #6
```

```
   }
 }
\cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
 {
  \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
   { #2 \__regex_compile_class_posix_loop:w }
   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
 }
\cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
 {
  \str_if_eq:nnTF { #1 #2 #3 #4 }
   { \__regex_compile_special:N : \__regex_compile_special:N ] }
   {
    \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
     {
      \__regex_compile_one:x
       {
        \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
        \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
       }
     }
     {
      \__msg_kernel_warning:nnx { kernel } { posix-unknown }
       { \l__regex_internal_a_tl }
      \__regex_compile_abort_tokens:x
       {
        [: \bool_if:NF \l__regex_internal_bool { ^ }
        \l__regex_internal_a_tl :]
       }
     }
   }
   {
    \__msg_kernel_error:nnxx { kernel } { posix-missing-close }
     { [: \l__regex_internal_a_tl } { #2 #4 }
    \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl }
    #1 #2 #3 #4
   }
 }
```

(*End definition for* `\__regex_compile_class_posix_test:w` *and others.*)

36.3.10 Groups and alternations

__regex_compile_group_begin:N
__regex_compile_group_end:

The contents of a regex group are turned into compiled code in `\l__regex_internal_regex`, which ends up with items of the form `\__regex_branch:n` {⟨*concatenation*⟩}. This construction is done using l3tl-build within a TEX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument `#1` is `\__regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```
\cs_new_protected:Npn \__regex_compile_group_begin:N #1
 {
  \__tl_build_one:n { #1 { \if_false: } \fi: }
```

```
    \__regex_mode_quit_c:
    \__tl_build:Nw \l__regex_internal_regex
      \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
      \int_incr:N \l__regex_group_level_int
      \__tl_build_one:n { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_group_end:
  {
    \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
        \__tl_build_one:n { \if_false: { \fi: } }
      \__tl_build_end:
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \__tl_build_one:o \l__regex_internal_regex
      \exp_after:wN \__regex_compile_quantifier:w
    \else:
      \__msg_kernel_warning:nn { kernel } { extra-rparen }
      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
    \fi:
  }
```

(*End definition for* `\__regex_compile_group_begin:N` *and* `\__regex_compile_group_end:`.)

`\__regex_compile_(:` In a class, parentheses are not special. Outside, check for a `?`, denoting special groups, and run the code for the corresponding special group.

```
\cs_new_protected:cpn { __regex_compile_(: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
      { \__regex_compile_lparen:w }
  }
\cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
  {
    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ? }
      {
        \cs_if_exist_use:cF
          { __regex_compile_special_group_\token_to_str:N #4 :w }
          {
            \__msg_kernel_warning:nnx { kernel } { special-group-unknown }
              { (? #4 }
            \__regex_compile_group_begin:N \__regex_group:nnnN
              \__regex_compile_raw:N ? #3 #4
          }
      }
      {
        \__regex_compile_group_begin:N \__regex_group:nnnN
          #1 #2 #3 #4
      }
  }
```

(*End definition for* `\__regex_compile_(:`.)

`\__regex_compile_|:` In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```
\cs_new_protected:cpn { __regex_compile_|: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N | }
      {
        \__tl_build_one:n
          { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
      }
  }
```

(*End definition for* `\__regex_compile_|:`.)

`\__regex_compile_):`

Within a class, parentheses are not special. Outside, close a group.

```
\cs_new_protected:cpn { __regex_compile_): }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
      { \__regex_compile_group_end: }
  }
```

(*End definition for* `\__regex_compile_):`.)

`\__regex_compile_special_group_::w`
`\__regex_compile_special_group_|:w`

Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```
\cs_new_protected:cpn { __regex_compile_special_group_::w }
  { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
\cs_new_protected:cpn { __regex_compile_special_group_|:w }
  { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }
```

(*End definition for* `\__regex_compile_special_group_::w` *and* `\__regex_compile_special_group_|:w`.)

`\__regex_compile_special_group_i:w`
`\__regex_compile_special_group_-:w`

The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```
\cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
  {
    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ) }
      {
        \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseless_equal:n }
        \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseless_range:nn }
      }
      {
        \__msg_kernel_warning:nnx { kernel } { unknown-option } { (?i #2 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N i
        #1 #2
      }
  }
\cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
  {
    \str_if_eq:nnTF { #1 #2 #3 #4 }
      { \__regex_compile_raw:N i \__regex_compile_special:N ) }
      {
        \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
        \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      }
      {
        \__msg_kernel_warning:nnx { kernel } { unknown-option } { (?-#2#4 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N -
        #1 #2 #3 #4
      }
  }
```

(*End definition for* `\__regex_compile_special_group_i:w` *and* `\__regex_compile_special_group_-:w`.)

36.3.11 Catcodes and csnames

`\__regex_compile_/c:`
`\__regex_compile_c_test:NN`

The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```
\cs_new_protected:cpn { __regex_compile_/c: }
  { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
\cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \int_set_eq:Nc \l__regex_catcodes_int { c__regex_catcode_#2_int }
            \l__regex_mode_int
              = \if_case:w \l__regex_mode_int
                  \c__regex_catcode_mode_int
                \else:
                  \c__regex_catcode_in_class_mode_int
                \fi:
            \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
          }
      }
      { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
          {
            \__msg_kernel_error:nnx { kernel } { c-missing-category } {#2}
            #1 #2
          }
  }
```

(*End definition for* `\__regex_compile_/c:` *and* `\__regex_compile_c_test:NN`.)

`\__regex_compile_c_C:NN`

If `\cC` is not followed by `.` or `(...)` then complain because that construction cannot match anything, except in cases like `\cC[\c{...}]`, where it has no effect.

```
\cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \token_if_eq_charcode:NNTF #2 .
          { \use_none:n }
          { \token_if_eq_charcode:NNF #2 ( } % )
      }
      { \use:n }
    { \__msg_kernel_error:nnn { kernel } { c-C-invalid } {#2} }
    #1 #2
  }
```

(*End definition for* `\__regex_compile_c_C:NN`.)

__regex_compile_c_[:w
__regex_compile_c_lbrack_loop:NN
__regex_compile_c_lbrack_add:N
__regex_compile_c_lbrack_end:

When encountering `\c[`, the task is to collect uppercase letters representing character categories. First check for `^` which negates the list of category codes.

```
\cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
  {
    \l__regex_mode_int
      = \if_case:w \l__regex_mode_int
          \c__regex_catcode_mode_int
        \else:
          \c__regex_catcode_in_class_mode_int
        \fi:
    \int_zero:N \l__regex_catcodes_int
    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ^ }
      {
        \bool_set_false:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
      }
      {
        \bool_set_true:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \exp_args:Nc \__regex_compile_c_lbrack_add:N
              { c__regex_catcode_#2_int }
            \__regex_compile_c_lbrack_loop:NN
          }
      }
      {
        \token_if_eq_charcode:NNTF #2 ]
          { \__regex_compile_c_lbrack_end: }
      }
          {
            \__msg_kernel_error:nnx { kernel } { c-missing-rbrack } {#2}
            \__regex_compile_c_lbrack_end:
            #1 #2
          }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
  {
    \if_int_odd:w \__int_eval:w \l__regex_catcodes_int / #1 \__int_eval_end:
    \else:
      \int_add:Nn \l__regex_catcodes_int {#1}
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_end:
  {
    \if_meaning:w \c_false_bool \l__regex_catcodes_bool
      \int_set:Nn \l__regex_catcodes_int
        { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
    \fi:
  }
```

(*End definition for __regex_compile_c_[:w and others.*)

__regex_compile_c_{:

The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```
\cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
  {
    \__regex_compile:w
      \__regex_disable_submatches:
      \l__regex_mode_int
        = \if_case:w \l__regex_mode_int
            \c__regex_cs_mode_int
          \else:
            \c__regex_cs_in_class_mode_int
          \fi:
  }
```

(*End definition for __regex_compile_c_{:.*)

__regex_compile_}:
__regex_compile_end_cs:
__regex_compile_cs_aux:Nn
__regex_compile_cs_aux:NNnnnN

Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use __regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```
\flag_new:n { __regex_cs }
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \__regex_compile_end_cs: }
      { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
  }
\cs_new_protected:Npn \__regex_compile_end_cs:
  {
    \__regex_compile_end:
    \flag_clear:n { __regex_cs }
    \tl_set:Nx \l__regex_internal_a_tl
      {
        \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
        \q_nil \q_nil \q_recursion_stop
      }
    \exp_args:Nx \__regex_compile_one:x
      {
        \flag_if_raised:nTF { __regex_cs }
          { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
          { \__regex_item_exact_cs:n { \tl_tail:N \l__regex_internal_a_tl } }
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
  {
    \cs_if_eq:NNTF #1 \__regex_branch:n
      {
        \scan_stop:
        \__regex_compile_cs_aux:NNnnnN #2
        \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil \q_recursion_stop
        \__regex_compile_cs_aux:Nn
      }
      {
        \quark_if_nil:NF #1 { \flag_raise:n { __regex_cs } }
        \use_none_delimit_by_q_recursion_stop:w
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
  {
    \bool_lazy_all:nTF
      {
        { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
        {#2}
        { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
        { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
        { \int_compare_p:nNn {#5} = { 0 } }
      }
      {
        \prg_replicate:nn {#4}
          { \char_generate:nn { \use_ii:nn #3 } {12} }
        \__regex_compile_cs_aux:NNnnnN
      }
      {
        \quark_if_nil:NF #1
          {
            \flag_raise:n { __regex_cs }
            \use_i_delimit_by_q_recursion_stop:nw
          }
        \use_none_delimit_by_q_recursion_stop:w
      }
  }
```

(*End definition for* `\__regex_compile_}:` *and others.*)

36.3.12 Raw token lists with \u

__regex_compile_/u:
__regex_compile_u_loop:NN

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of `\u` within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```
\cs_new_protected:cpn { __regex_compile_/u: } #1#2
  {
    \__regex_if_in_class_or_catcode:TF
```

```
      { \__regex_compile_raw_error:N u #1 #2 }
      {
        \str_if_eq_x:nnTF {#1#2} { \__regex_compile_special:N \c_left_brace_str }
          {
            \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
            \__regex_compile_u_loop:NN
          }
          {
            \__msg_kernel_error:nn { kernel } { u-missing-lbrace }
            \__regex_compile_raw:N u #1 #2
          }
      }
  }
\cs_new:Npn \__regex_compile_u_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      { #2 \__regex_compile_u_loop:NN }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
          {
            \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
              { \if_false: { \fi: } \__regex_compile_u_end: }
              { #2 \__regex_compile_u_loop:NN }
          }
          {
            \if_false: { \fi: }
            \__msg_kernel_error:nnx { kernel } { u-missing-rbrace } {#2}
            \__regex_compile_u_end:
            #1 #2
          }
      }
  }
```

(*End definition for* `\__regex_compile_/u:` *and* `\__regex_compile_u_loop:NN`.)

`\__regex_compile_u_end:` Once we have extracted the variable's name, we store the contents of that variable in `\l__regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not.

```
\cs_new_protected:Npn \__regex_compile_u_end:
  {
    \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \__regex_compile_u_not_cs:
    \else:
      \__regex_compile_u_in_cs:
    \fi:
  }
```

(*End definition for* `\__regex_compile_u_end:`.)

`\__regex_compile_u_in_cs:` When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```
\cs_new_protected:Npn \__regex_compile_u_in_cs:
  {
    \tl_gset:Nx \g__regex_internal_tl
      { \exp_args:No \__str_to_other_fast:n { \l__regex_internal_a_tl } }
    \__tl_build_one:x
      {
        \tl_map_function:NN \g__regex_internal_tl
          \__regex_compile_u_in_cs_aux:n
      }
  }
\cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
  {
    \__regex_class:NnnnN \c_true_bool
      { \__regex_item_caseful_equal:n { \__int_value:w ‘#1 } }
      { 1 } { 0 } \c_false_bool
  }
```

(*End definition for* `\__regex_compile_u_in_cs:`*.*)

`\__regex_compile_u_not_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l__regex_internal_a_tl`. If a given ⟨*token*⟩ is a control sequence, then insert a string comparison test, otherwise, `\__regex_item_exact:nn` which compares catcode and character code.

```
\cs_new_protected:Npn \__regex_compile_u_not_cs:
  {
    \exp_args:No \__tl_analysis_map_inline:nn { \l__regex_internal_a_tl }
      {
        \__tl_build_one:n
          {
            \__regex_class:NnnnN \c_true_bool
              {
                \if_int_compare:w "##2 = 0 \exp_stop_f:
                  \__regex_item_exact_cs:n { \exp_after:wN \cs_to_str:N ##1 }
                \else:
                  \__regex_item_exact:nn { \__int_value:w "##2 } { ##3 }
                \fi:
              }
              { 1 } { 0 } \c_false_bool
          }
      }
  }
```

(*End definition for* `\__regex_compile_u_not_cs:`*.*)

36.3.13 Other

`\__regex_compile_/K:` The `\K` control sequence is currently the only "command", which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```
\cs_new_protected:cpn { __regex_compile_/K: }
  {
    \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
      { \__tl_build_one:n { \__regex_command_K: } }
      { \__regex_compile_raw_error:N K }
  }
```

(*End definition for* `\__regex_compile_/K:`*.*)

36.3.14 Showing regexes

__regex_show:Nn

Within a __tl_build:Nw ... __tl_build_end: group, we redefine all the function that can appear in a compiled regex, then run the regex. The result is then shown.

```
\cs_new_protected:Npn \__regex_show:Nn #1#2
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \cs_set_protected:Npn \__regex_branch:n
        {
          \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl
          \__regex_show_one:n { +-branch }
          \seq_put_right:No \l__regex_show_prefix_seq \l__regex_internal_a_tl
          \use:n
        }
      \cs_set_protected:Npn \__regex_group:nnnN
        { \__regex_show_group_aux:nnnnN { } }
      \cs_set_protected:Npn \__regex_group_no_capture:nnnN
        { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
      \cs_set_protected:Npn \__regex_group_resetting:nnnN
        { \__regex_show_group_aux:nnnnN { ~(resetting) } }
      \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
      \cs_set_protected:Npn \__regex_command_K:
        { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
      \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
        { \__regex_show_one:n { \bool_if:NF ##1 { negative~ } assertion:~##2 } }
      \cs_set:Npn \__regex_b_test: { word~boundary }
      \cs_set_eq:NN \__regex_anchor:N \__regex_show_anchor_to_str:N
      \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
        { \__regex_show_one:n { char~code~\int_eval:n{##1} } }
      \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
        { \__regex_show_one:n { range~[\int_eval:n{##1}, \int_eval:n{##2}] } }
      \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
        { \__regex_show_one:n { char~code~\int_eval:n{##1}~(caseless) } }
      \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
        {
          \__regex_show_one:n
            { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }
        }
      \cs_set_protected:Npn \__regex_item_catcode:nT
        { \__regex_show_item_catcode:NnT \c_true_bool }
      \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
        { \__regex_show_item_catcode:NnT \c_false_bool }
      \cs_set_protected:Npn \__regex_item_reverse:n
        { \__regex_show_scope:nn { Reversed~match } }
      \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
        { \__regex_show_one:n { char~##2,~catcode~##1 } }
      \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
      \cs_set_protected:Npn \__regex_item_cs:n
        { \__regex_show_scope:nn { control~sequence } }
      \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
      \seq_clear:N \l__regex_show_prefix_seq
      \__regex_show_push:n { ~ }
      \cs_if_exist_use:N #1
    \__tl_build_end:
```

```
    \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { }
      { >~Compiled~regex~#2: \l__regex_internal_a_tl }
  }
```

(*End definition for* `\__regex_show:Nn`.)

__regex_show_one:n

Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```
\cs_new_protected:Npn \__regex_show_one:n #1
  {
    \int_incr:N \l__regex_show_lines_int
    \__tl_build_one:x
      {
        \exp_not:N \\
        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
        #1
      }
  }
```

(*End definition for* `\__regex_show_one:n`.)

__regex_show_push:n
__regex_show_pop:
__regex_show_scope:nn

Enter and exit levels of nesting. The `scope` function prints its first argument as an "introduction", then performs its second argument in a deeper level of nesting.

```
\cs_new_protected:Npn \__regex_show_push:n #1
  { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
\cs_new_protected:Npn \__regex_show_pop:
  { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
\cs_new_protected:Npn \__regex_show_scope:nn #1#2
  {
    \__regex_show_one:n {#1}
    \__regex_show_push:n { ~ }
    #2
    \__regex_show_pop:
  }
```

(*End definition for* `\__regex_show_push:n` *,* `\__regex_show_pop:` *, and* `\__regex_show_scope:nn`.)

__regex_show_group_aux:nnnnN

We display all groups in the same way, simply adding a message, `(no capture)` or `(resetting)`, to special groups. The odd `\use_ii:nn` avoids printing a spurious `+-branch` for the first branch.

```
\cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
  {
    \__regex_show_one:n { ,-group~begin #1 }
    \__regex_show_push:n { | }
    \use_ii:nn #2
    \__regex_show_pop:
    \__regex_show_one:n
      { '-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
  }
```

(*End definition for* `\__regex_show_group_aux:nnnnN`.)

__regex_show_class:NnnnN I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write Match or Don't match on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```
\cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \int_zero:N \l__regex_show_lines_int
      \__regex_show_push:n {~}
      #2
      \exp_last_unbraced:Nf
    \int_case:nnF { \l__regex_show_lines_int }
      {
        {0}
          {
            \__tl_build_end:
            \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
          }
        {1}
          {
            \__tl_build_end:
            \bool_if:NTF #1
              {
                #2
                \__tl_build_one:n { \__regex_msg_repeated:nnN {#3} {#4} #5 }
              }
              {
                \__regex_show_one:n
                  { Don't~match~\__regex_msg_repeated:nnN {#3} {#4} #5 }
                \__tl_build_one:o \l__regex_internal_a_tl
              }
          }
      }
      {
        \__tl_build_end:
        \__regex_show_one:n
          {
            \bool_if:NTF #1 { M } { Don't~m } atch
            \__regex_msg_repeated:nnN {#3} {#4} #5
          }
        \__tl_build_one:o \l__regex_internal_a_tl
      }
  }
```

(*End definition for __regex_show_class:NnnnN.*)

__regex_show_anchor_to_str:N The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```
\cs_new:Npn \__regex_show_anchor_to_str:N #1
  {
    anchor~at~
    \str_case:nnF { #1 }
```

```
      {
        { \l__regex_min_pos_int   } { start~(\iow_char:N\\A) }
        { \l__regex_start_pos_int } { start~of~match~(\iow_char:N\\G) }
        { \l__regex_max_pos_int   } { end~(\iow_char:N\\Z) }
      }
      { <error:~'#1'~not~recognized> }
  }
```

(*End definition for* `\__regex_show_anchor_to_str:N`.)

__regex_show_item_catcode:NnT

Produce a sequence of categories which the catcode bitmap `#2` contains, and show it, indenting the tests on which this catcode constraint applies.

```
\cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
  {
    \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
    \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
      { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
    \__regex_show_scope:nn
      {
        categories~
        \seq_map_function:NN \l__regex_internal_seq \use:n
        , ~
        \bool_if:NF #1 { negative~ } class
      }
  }
```

(*End definition for* `\__regex_show_item_catcode:NnT`.)

__regex_show_item_exact_cs:n

```
\cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
  {
    \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
    \seq_set_map:NNn \l__regex_internal_seq
      \l__regex_internal_seq { \iow_char:N\\##1 }
    \__regex_show_one:n
      { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
  }
```

(*End definition for* `\__regex_show_item_exact_cs:n`.)

36.4 Building

36.4.1 Variables used while building

`\l__regex_min_state_int`
`\l__regex_max_state_int`

The last state that was allocated is `\l__regex_max_state_int` -1, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1, but is included to avoid hard-coding this value everywhere.

```
\int_new:N  \l__regex_min_state_int
\int_set:Nn \l__regex_min_state_int { 1 }
\int_new:N  \l__regex_max_state_int
```

(*End definition for* `\l__regex_min_state_int` *and* `\l__regex_max_state_int`.)

\l__regex_left_state_int
\l__regex_right_state_int
\l__regex_left_state_seq
\l__regex_right_state_seq

Alternatives are implemented by branching from a left state into the various choices, then merging those into a right state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```
\int_new:N  \l__regex_left_state_int
\int_new:N  \l__regex_right_state_int
\seq_new:N  \l__regex_left_state_seq
\seq_new:N  \l__regex_right_state_seq
```

(*End definition for* \l__regex_left_state_int *and others.*)

\l__regex_capturing_group_int

\l__regex_capturing_group_int is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering resetting groups.

```
\int_new:N  \l__regex_capturing_group_int
```

(*End definition for* \l__regex_capturing_group_int.)

36.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a \toks. The operations which can appear in the \toks are

- __regex_action_start_wildcard: inserted at the start of the regular expression to make it unanchored.
- __regex_action_success: marks the exit state of the NFA.
- __regex_action_cost:n {⟨*shift*⟩} is a transition from the current ⟨*state*⟩ to ⟨*state*⟩ + ⟨*shift*⟩, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- __regex_action_free:n {⟨*shift*⟩}, and __regex_action_free_group:n {⟨*shift*⟩} are free transitions, which immediately perform the actions for the state ⟨*state*⟩ + ⟨*shift*⟩ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the group variant must be used for transitions back to the start of a group.
- __regex_action_submatch:n {⟨*key*⟩} where the ⟨*key*⟩ is a group number followed by < or > for the beginning or end of group. This causes the current position in the query to be stored as the ⟨*key*⟩ submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is capturing_group − 1, and if a group opened now it would be labelled capturing_group.
- The last allocated state is max_state − 1, so max_state is a free state.
- The left_state points to a state to the left of the current group or of the last class.
- The right_state points to a newly created, empty state, with some transitions leading to it.

- The left/right sequences hold a list of the corresponding end-points of nested groups.

__regex_build:n
__regex_build:N

The n-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of capturing_group). Finally, if the match reaches the last state, it is successful.

```
\cs_new_protected:Npn \__regex_build:n #1
  {
    \__regex_compile:n {#1}
    \__regex_build:N \l__regex_internal_regex
  }
\__debug_patch:nnNNpn
  { \__debug_trace_push:nnN { regex } { 1 } \__regex_build:N }
  {
    \__regex_trace_states:n { 2 }
    \__debug_trace_pop:nnN { regex } { 1 } \__regex_build:N
  }
\cs_new_protected:Npn \__regex_build:N #1
  {
    \__regex_standard_escapechar:
    \int_zero:N \l__regex_capturing_group_int
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard: }
    \__regex_group:nnnN {#1} { 1 } { 0 } \c_false_bool
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      { \__regex_action_success: }
  }
```

(*End definition for __regex_build:n and __regex_build:N.*)

__regex_build_for_cs:n

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate left and right states in their sequence.

```
\__debug_patch:nnNNpn
  { \__debug_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
  {
    \__regex_trace_states:n { 2 }
    \__debug_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
  }
\cs_new_protected:Npn \__regex_build_for_cs:n #1
  {
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_push_lr_states:
    #1
    \__regex_pop_lr_states:
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      {
        \if_int_compare:w \l__regex_curr_pos_int = \l__regex_max_pos_int
          \exp_after:wN \__regex_action_success:
        \fi:
      }
  }
```

(*End definition for* `\__regex_build_for_cs:n`.)

36.4.3 Helpers for building an nfa

`\__regex_push_lr_states:`
`\__regex_pop_lr_states:`

When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from TeX's grouping.

```
\cs_new_protected:Npn \__regex_push_lr_states:
  {
    \seq_push:No \l__regex_left_state_seq
      { \int_use:N \l__regex_left_state_int }
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_right_state_int }
  }
\cs_new_protected:Npn \__regex_pop_lr_states:
  {
    \seq_pop:NN \l__regex_left_state_seq  \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int  \l__regex_internal_a_tl
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End definition for* `\__regex_push_lr_states:` *and* `\__regex_pop_lr_states:`.)

`\__regex_build_transition_left:NNN`
`\__regex_build_transition_right:nNn`

Add a transition from `#2` to `#3` using the function `#1`. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```
\cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
  { \__regex_toks_put_left:Nx  #2 { #1 { \int_eval:n { #3 - #2 } } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
  { \__regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(*End definition for* `\__regex_build_transition_left:NNN` *and* `\__regex_build_transition_right:nNn`.)

`\__regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously "current" state.

```
\__debug_patch:nnNNpn
  {
    \__debug_trace:nnx { regex } { 2 }
      {
        regex~new~state~
        L=\int_use:N \l__regex_left_state_int ~ -> ~
        R=\int_use:N \l__regex_right_state_int ~ -> ~
        M=\int_use:N \l__regex_max_state_int ~ -> ~
        \int_eval:n { \l__regex_max_state_int + 1 }
      }
  }
  { }
\cs_new_protected:Npn \__regex_build_new_state:
  {
    \__regex_toks_clear:N \l__regex_max_state_int
    \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
    \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
    \int_incr:N \l__regex_max_state_int
  }
```

(*End definition for* `\__regex_build_new_state:`.)

__regex_build_transitions_lazyness:NNNNN

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by `#1`, true for lazy quantifiers, and false for greedy quantifiers.

```
\cs_new_protected:Npn \__regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Nx \l__regex_left_state_int
      {
        \if_meaning:w \c_true_bool #1
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
        \else:
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
        \fi:
      }
  }
```

(*End definition for* `\__regex_build_transitions_lazyness:NNNNN`.)

36.4.4 Building classes

__regex_class:NnnnN
__regex_tests_action_cost:n

The arguments are: ⟨*boolean*⟩ `{`⟨*tests*⟩`}` `{`⟨*min*⟩`}` `{`⟨*more*⟩`}` ⟨*lazyness*⟩. First store the tests with a trailing `\__regex_action_cost:n`, in the true branch of `\__regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer ⟨*more*⟩ is 0 for fixed repetitions, -1 for unbounded repetitions, and ⟨*max*⟩ − ⟨*min*⟩ for a range of repetitions.

```
\cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
  {
    \cs_set:Npx \__regex_tests_action_cost:n ##1
      {
        \exp_not:n { \exp_not:n {#2} }
        \bool_if:NTF #1
          { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
          { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
      }
    \if_case:w - #4 \exp_stop_f:
           \__regex_class_repeat:n   {#3}
    \or:   \__regex_class_repeat:nN  {#3}      #5
    \else: \__regex_class_repeat:nnN {#3} {#4} #5
    \fi:
  }
\cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }
```

(*End definition for* `\__regex_class:NnnnN` *and* `\__regex_tests_action_cost:n`*.*)

__regex_class_repeat:n

This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for #1 = 0 repetitions: nothing is built.

```
\cs_new_protected:Npn \__regex_class_repeat:n #1
  {
    \prg_replicate:nn {#1}
      {
        \__regex_build_new_state:
        \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
          \l__regex_left_state_int \l__regex_right_state_int
      }
  }
```

(*End definition for* `\__regex_class_repeat:n`*.*)

__regex_class_repeat:nN

This implements unbounded repetitions of a single class (*e.g.* the * and + quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `\__regex_class_repeat:n` for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean #2.

```
\cs_new_protected:Npn \__regex_class_repeat:nN #1#2
  {
    \if_int_compare:w #1 = 0 \exp_stop_f:
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n       \l__regex_right_state_int
        \__regex_tests_action_cost:n \l__regex_left_state_int
    \else:
      \__regex_class_repeat:n {#1}
      \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n \l__regex_right_state_int
        \__regex_action_free:n \l__regex_internal_a_int
    \fi:
  }
```

(*End definition for* `\__regex_class_repeat:nN`*.*)

__regex_class_repeat:nnN

We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 $>$ 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from max_state.

```
\cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
  {
    \__regex_class_repeat:n {#1}
    \int_set:Nn \l__regex_internal_a_int
      { \l__regex_max_state_int + #2 - 1 }
    \prg_replicate:nn { #2 }
      {
        \__regex_build_transitions_lazyness:NNNNN #3
```

```
          \__regex_action_free:n       \l__regex_internal_a_int
          \__regex_tests_action_cost:n \l__regex_right_state_int
      }
  }
```

(*End definition for* `\__regex_class_repeat:nnN`.)

36.4.5 Building groups

__regex_group_aux:nnnnN

Arguments: {⟨*label*⟩} {⟨*contents*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩. If ⟨*min*⟩ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents `#2` of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly `#3` times, or `#3` or more times, or between `#3` and `#3` + `#4` times, with lazyness `#5`. The ⟨*label*⟩ `#1` is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```
\__debug_patch:nnNNpn
  { \__debug_trace_push:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
  { \__debug_trace_pop:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
  {
      \if_int_compare:w #3 = 0 \exp_stop_f:
        \__regex_build_new_state:
⟨assert⟩\assert_int:n { \l__regex_max_state_int = \l__regex_right_state_int + 1 }
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_left_state_int \l__regex_right_state_int
      \fi:
      \__regex_build_new_state:
      \__regex_push_lr_states:
      #2
      \__regex_pop_lr_states:
      \if_case:w - #4 \exp_stop_f:
             \__regex_group_repeat:nn   {#1} {#3}
      \or:   \__regex_group_repeat:nnN  {#1} {#3}      #5
      \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
      \fi:
  }
```

(*End definition for* `\__regex_group_aux:nnnnN`.)

__regex_group:nnnN
__regex_group_no_capture:nnnN

Hand to `\__regex_group_aux:nnnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```
\cs_new_protected:Npn \__regex_group:nnnN #1
  {
    \exp_args:No \__regex_group_aux:nnnnN
      { \int_use:N \l__regex_capturing_group_int }
      {
        \int_incr:N \l__regex_capturing_group_int
        #1
      }
  }
\cs_new_protected:Npn \__regex_group_no_capture:nnnN
  { \__regex_group_aux:nnnnN { -1 } }
```

(*End definition for* `\__regex_group:nnnN` *and* `\__regex_group_no_capture:nnnN`*.*)

`\__regex_group_resetting:nnnN`
`\__regex_group_resetting_loop:nnNn`

Again, hand the label -1 to `\__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `\__regex_branch:n {`⟨*branch*⟩`}`.

```
\cs_new_protected:Npn \__regex_group_resetting:nnnN #1
  {
    \__regex_group_aux:nnnnN { -1 }
      {
        \exp_args:Noo \__regex_group_resetting_loop:nnNn
          { \int_use:N \l__regex_capturing_group_int }
          { \int_use:N \l__regex_capturing_group_int }
          #1
          { ?? \__prg_break:n } { }
        \__prg_break_point:
      }
  }
\cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
  {
    \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
    \int_set:Nn \l__regex_capturing_group_int {#2}
    #3 {#4}
    \exp_args:Nf \__regex_group_resetting_loop:nnNn
      { \int_max:nn {#1} { \l__regex_capturing_group_int } }
      {#2}
  }
```

(*End definition for* `\__regex_group_resetting:nnnN` *and* `\__regex_group_resetting_loop:nnNn`*.*)

`\__regex_branch:n`

Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```
\__debug_patch:nnNNpn
  { \__debug_trace_push:nnN { regex } { 1 } \__regex_branch:n }
  { \__debug_trace_pop:nnN { regex } { 1 } \__regex_branch:n }
\cs_new_protected:Npn \__regex_branch:n #1
  {
    \__regex_build_new_state:
    \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_left_state_int \l__regex_right_state_int
    #1
    \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End definition for* `\__regex_branch:n`*.*)

__regex_group_repeat:nn

This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the capturing_group label). Otherwise, the auxiliary __regex_group_repeat_aux:n copies #2 times the \toks for the group, and leaves internal_a pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary.

```
\cs_new_protected:Npn \__regex_group_repeat:nn #1#2
  {
    \if_int_compare:w #2 = 0 \exp_stop_f:
      \int_set:Nn \l__regex_max_state_int
        { \l__regex_left_state_int - 1 }
      \__regex_build_new_state:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \__regex_build_new_state:
    \fi:
  }
```

(*End definition for __regex_group_repeat:nn.*)

__regex_group_submatches:nNN

This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of −1.

```
\cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
  {
    \if_int_compare:w #1 > - 1 \exp_stop_f:
      \__regex_toks_put_left:Nx #2 { \__regex_action_submatch:n { #1 < } }
      \__regex_toks_put_left:Nx #3 { \__regex_action_submatch:n { #1 > } }
    \fi:
  }
```

(*End definition for __regex_group_submatches:nNN.*)

__regex_group_repeat_aux:n

Here we repeat \toks ranging from left_state to max_state, #1 > 0 times. First add a transition so that the copies "chain" properly. Compute the shift c between the original copy and the last copy we want. Shift the right_state and max_state to their final values. We then want to perform c copy operations. At the end, b is equal to the max_state, and a points to the left of the last copy of the group.

```
\cs_new_protected:Npn \__regex_group_repeat_aux:n #1
  {
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
    \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
    \if_int_compare:w \__int_eval:w #1 > 1 \exp_stop_f:
      \int_set:Nn \l__regex_internal_c_int
        {
          ( #1 - 1 )
          * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
        }
      \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
      \int_add:Nn \l__regex_max_state_int   { \l__regex_internal_c_int }
      \__regex_toks_memcpy:NNn
        \l__regex_internal_b_int
        \l__regex_internal_a_int
        \l__regex_internal_c_int
    \fi:
  }
```

(*End definition for* __regex_group_repeat_aux:n.)

__regex_group_repeat:nnN

This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the "true" left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from __regex_group_repeat_aux:n.

```
\cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
  {
    \if_int_compare:w #2 = 0 \exp_stop_f:
      \__regex_group_submatches:nNN {#1}
        \l__regex_left_state_int \l__regex_right_state_int
      \int_set:Nn \l__regex_internal_a_int
        { \l__regex_left_state_int - 1 }
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_internal_a_int
      \__regex_build_new_state:
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_left:NNN \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \else:
        \__regex_build_transition_right:nNn \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \fi:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \else:
        \__regex_build_transition_left:NNN \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \fi:
      \__regex_build_new_state:
    \fi:
  }
```

(*End definition for* __regex_group_repeat:nnN.)

__regex_group_repeat:nnnN

We wish to repeat the group between #2 and #2 + #3 times, with a lazyness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it "by hand" earlier.

```
\cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
  {
    \__regex_group_submatches:nNN {#1}
      \l__regex_left_state_int \l__regex_right_state_int
    \__regex_group_repeat_aux:n { #2 + #3 }
    \if_meaning:w \c_true_bool #4
      \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
      \prg_replicate:nn { #3 }
        {
          \int_sub:Nn \l__regex_left_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_left:NNN \__regex_action_free:n
            \l__regex_left_state_int \l__regex_max_state_int
        }
    \else:
      \prg_replicate:nn { #3 - 1 }
        {
          \int_sub:Nn \l__regex_right_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_right:nNn \__regex_action_free:n
            \l__regex_right_state_int \l__regex_max_state_int
        }
      \if_int_compare:w #2 = 0 \exp_stop_f:
        \int_set:Nn \l__regex_right_state_int
          { \l__regex_left_state_int - 1 }
      \else:
        \int_sub:Nn \l__regex_right_state_int
          { \l__regex_internal_b_int - \l__regex_internal_a_int }
      \fi:
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_max_state_int
    \fi:
    \__regex_build_new_state:
  }
```

(*End definition for __regex_group_repeat:nnnN.*)

36.4.6 Others

__regex_assertion:Nn
__regex_b_test:
__regex_anchor:N

Usage: __regex_assertion:Nn ⟨*boolean*⟩ {⟨*test*⟩}, where the ⟨*test*⟩ is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use `\__regex_anchor:N`, with a position controlled by the integer `#1`.

```
\cs_new_protected:Npn \__regex_assertion:Nn #1#2
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Nx \l__regex_left_state_int
      {
        \exp_not:n {#2}
        \__regex_break_point:TF
          \bool_if:NF #1 { { } }
          {
            \__regex_action_free:n
              {
                \int_eval:n
                  { \l__regex_right_state_int - \l__regex_left_state_int }
              }
          }
          \bool_if:NT #1 { { } }
      }
  }
\cs_new_protected:Npn \__regex_anchor:N #1
  {
    \if_int_compare:w #1 = \l__regex_curr_pos_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_b_test:
  {
    \group_begin:
      \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
      \__regex_prop_w:
      \__regex_break_point:TF
        { \group_end: \__regex_item_reverse:n \__regex_prop_w: }
        { \group_end: \__regex_prop_w: }
  }
```

(*End definition for* `\__regex_assertion:Nn`*,* `\__regex_b_test:`*, and* `\__regex_anchor:N`*.*)

`\__regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```
\cs_new_protected:Npn \__regex_command_K:
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Nx \l__regex_left_state_int
      {
        \__regex_action_submatch:n { 0< }
        \bool_set_true:N \l__regex_fresh_thread_bool
        \__regex_action_free:n
          { \int_eval:n { \l__regex_right_state_int - \l__regex_left_state_int } }
        \bool_set_false:N \l__regex_fresh_thread_bool
      }
  }
```

(*End definition for* `\__regex_command_K:`*.*)

36.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains "free" transitions to other states, and transitions which "consume" the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of "active states", stored in `\g__regex_thread_state_intarray`: this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA "collide" in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn't it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing "normal" free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a "completed" thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

36.5.1 Variables used when matching

`\l__regex_min_pos_int`
`\l__regex_max_pos_int`
`\l__regex_curr_pos_int`
`\l__regex_start_pos_int`
`\l__regex_success_pos_int`

The tokens in the query are indexed from `min_pos` for the first to `max_pos` -1 for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We don't start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the `current_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```
\int_new:N \l__regex_min_pos_int
\int_new:N \l__regex_max_pos_int
\int_new:N \l__regex_curr_pos_int
\int_new:N \l__regex_start_pos_int
\int_new:N \l__regex_success_pos_int
```

(*End definition for* `\l__regex_min_pos_int` *and others.*)

`\l__regex_curr_char_int`
`\l__regex_curr_catcode_int`
`\l__regex_last_char_int`
`\l__regex_case_changed_char_int`

The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token (`A-Z`↔`a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `current_char` variable is also used in various other phases to hold a character code.

```
\int_new:N \l__regex_curr_char_int
\int_new:N \l__regex_curr_catcode_int
\int_new:N \l__regex_last_char_int
\int_new:N \l__regex_case_changed_char_int
```

(*End definition for* `\l__regex_curr_char_int` *and others.*)

`\l__regex_curr_state_int`

For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
\int_new:N \l__regex_curr_state_int
```

(*End definition for* `\l__regex_curr_state_int`.)

`\l__regex_curr_submatches_prop`
`\l__regex_success_submatches_prop`

The submatches for the thread which is currently active are stored in the `current_submatches` property list variable. This property list is stored by `\__regex_action_cost:n` into the `\toks` register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to `\l__regex_success_submatches_prop`: only the last successful thread remains there.

```
\prop_new:N \l__regex_curr_submatches_prop
\prop_new:N \l__regex_success_submatches_prop
```

(*End definition for* `\l__regex_curr_submatches_prop` *and* `\l__regex_success_submatches_prop`.)

`\l__regex_step_int`

This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each ⟨*state*⟩ in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks`⟨*state*⟩, but not finished yet. However, once we finish, we store `step` + 1 in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
\int_new:N \l__regex_step_int
```

(*End definition for* `\l__regex_step_int`.)

`\l__regex_min_active_int`
`\l__regex_max_active_int`

All the currently active threads are kept in order of precedence in `\g__regex_thread_state_intarray`, and the corresponding submatches in the `\toks`. For our purposes, those serve as an array, indexed from `min_active` (inclusive) to `max_active` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_active` is reset to `min_active`, effectively clearing the array.

```
\int_new:N \l__regex_min_active_int
\int_new:N \l__regex_max_active_int
```

(*End definition for* `\l__regex_min_active_int` *and* `\l__regex_max_active_int`.)

`\g__regex_state_active_intarray`
`\g__regex_thread_state_intarray`

`\g__regex_state_active_intarray` stores the last ⟨*step*⟩ in which each ⟨*state*⟩ was active. `\g__regex_thread_state_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
\__intarray_new:Nn \g__regex_state_active_intarray { 65536 }
\__intarray_new:Nn \g__regex_thread_state_intarray { 65536 }
```

(*End definition for \g__regex_state_active_intarray and \g__regex_thread_state_intarray.*)

\l__regex_every_match_tl

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See __regex_single_match: and __regex_multi_match:n.

```
\tl_new:N \l__regex_every_match_tl
```

(*End definition for \l__regex_every_match_tl.*)

\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
__regex_if_two_empty_matches:F

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting \l__regex_fresh_thread_bool to true for threads which directly come from the start of the regex or from the \K command, and testing that boolean whenever a thread succeeds. The function __regex_if_two_empty_matches:F is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is \use:n.

```
\bool_new:N \l__regex_fresh_thread_bool
\bool_new:N \l__regex_empty_success_bool
\cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(*End definition for \l__regex_fresh_thread_bool , \l__regex_empty_success_bool , and __regex_if_two_empty_matches:F.*)

\g__regex_success_bool
\l__regex_saved_success_bool
\l__regex_match_success_bool

The boolean \l__regex_match_success_bool is true if the current match attempt was successful, and \g__regex_success_bool is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with \c{...}. This is done by saving the global variable into \l__regex_saved_success_bool, which is local, hence not affected by the changes due to inner regex functions.

```
\bool_new:N \g__regex_success_bool
\bool_new:N \l__regex_saved_success_bool
\bool_new:N \l__regex_match_success_bool
```

(*End definition for \g__regex_success_bool , \l__regex_saved_success_bool , and \l__regex_match_success_bool.*)

36.5.2 Matching: framework

__regex_match:n
__regex_match_init:

First store the query into \toks registers and arrays (see __regex_query_set:nnn). Then initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (\g__regex_state_active_intarray), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
\__debug_patch:nnNNpn
  {
    \__debug_trace_push:nnN { regex } { 1 } \__regex_match:n
    \__debug_trace:nnx { regex } { 1 } { analyzing~query~token~list }
  }
  { \__debug_trace_pop:nnN { regex } { 1 } \__regex_match:n }
\cs_new_protected:Npn \__regex_match:n #1
  {
    \int_zero:N \l__regex_balance_int
    \int_set:Nn \l__regex_curr_pos_int { 2 * \l__regex_max_state_int }
    \__regex_query_set:nnn { } { -1 } { -2 }
    \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
    \__tl_analysis_map_inline:nn {#1}
      { \__regex_query_set:nnn {##1} {"##2} {##3} }
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
    \__regex_query_set:nnn { } { -1 } { -2 }
    \__regex_match_init:
    \__regex_match_once:
  }
\__debug_patch:nnNNpn
  { \__debug_trace:nnx { regex } { 1 } { initializing } }
  { }
\cs_new_protected:Npn \__regex_match_init:
  {
    \bool_gset_false:N \g__regex_success_bool
    \int_step_inline:nnnn
      \l__regex_min_state_int { 1 } { \l__regex_max_state_int - 1 }
      { \__intarray_gset_fast:Nnn \g__regex_state_active_intarray {##1} { 1 } }
    \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
    \int_zero:N \l__regex_step_int
    \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
    \int_set:Nn \l__regex_min_submatch_int
      { 2 * \l__regex_max_state_int }
    \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
    \bool_set_false:N \l__regex_empty_success_bool
  }
```

(*End definition for* `\__regex_match:n` *and* `\__regex_match_init:`.)

`\__regex_match_once:` This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `\__regex_match_once:`. First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start, and `get` that token, to set `last_char` properly for word boundaries. Then call `\__regex_match_loop:`, which runs through the query until the end or until a successful match breaks early.

```
\cs_new_protected:Npn \__regex_match_once:
  {
    \if_meaning:w \c_true_bool \l__regex_empty_success_bool
      \cs_set:Npn \__regex_if_two_empty_matches:F
        { \int_compare:nNnF \l__regex_start_pos_int = \l__regex_curr_pos_int }
    \else:
      \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
    \fi:
    \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
    \bool_set_false:N \l__regex_match_success_bool
    \prop_clear:N \l__regex_curr_submatches_prop
    \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
    \__regex_store_state:n { \l__regex_min_state_int }
    \int_set:Nn \l__regex_curr_pos_int
      { \l__regex_start_pos_int - 1 }
    \__regex_query_get:
    \__regex_match_loop:
    \l__regex_every_match_tl
  }
```

(*End definition for* `\__regex_match_once:`*.*)

`\__regex_single_match:`
`\__regex_multi_match:n`

For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action `#1`, then find the next match.

```
\cs_new_protected:Npn \__regex_single_match:
  {
    \tl_set:Nn \l__regex_every_match_tl
      { \bool_gset_eq:NN \g__regex_success_bool \l__regex_match_success_bool }
  }
\cs_new_protected:Npn \__regex_multi_match:n #1
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \if_meaning:w \c_true_bool \l__regex_match_success_bool
          \bool_gset_true:N \g__regex_success_bool
          #1
          \exp_after:wN \__regex_match_once:
        \fi:
      }
  }
```

(*End definition for* `\__regex_single_match:` *and* `\__regex_multi_match:n`*.*)

`\__regex_match_loop:`
`\__regex_match_one_active:n`

At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_active`). This results in a sequence of `\__regex_use_state_and_submatches:nn` `{`⟨*state*⟩`}` `{`⟨*prop*⟩`}`, and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is what `\__regex_match_once:` matches. We explain the `fresh_thread` business when describing `\__regex_action_wildcard:`.

```
\cs_new_protected:Npn \__regex_match_loop:
  {
    \int_add:Nn \l__regex_step_int { 2 }
    \int_incr:N \l__regex_curr_pos_int
    \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
    \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
    \__regex_query_get:
    \use:x
      {
        \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
        \int_step_function:nnnN
          { \l__regex_min_active_int }
          { 1 }
          { \l__regex_max_active_int - 1 }
          \__regex_match_one_active:n
      }
    \__prg_break_point:
    \bool_set_false:N \l__regex_fresh_thread_bool %^^A was arg of break_point:n
    \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
      \if_int_compare:w \l__regex_curr_pos_int < \l__regex_max_pos_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_match_loop:
      \fi:
    \fi:
  }
\cs_new:Npn \__regex_match_one_active:n #1
  {
    \__regex_use_state_and_submatches:nn
      { \__intarray_item_fast:Nn \g__regex_thread_state_intarray {#1} }
      { \__regex_toks_use:w #1 }
  }
```

(*End definition for* `\__regex_match_loop:` *and* `\__regex_match_one_active:n`.)

`\__regex_query_set:nnn`

The arguments are: tokens that `o` and `x` expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\toks` register and some arrays, then update the `balance`.

```
\cs_new_protected:Npn \__regex_query_set:nnn #1#2#3
  {
    \__intarray_gset_fast:Nnn \g__regex_charcode_intarray
      { \l__regex_curr_pos_int } {#3}
    \__intarray_gset_fast:Nnn \g__regex_catcode_intarray
      { \l__regex_curr_pos_int } {#2}
    \__intarray_gset_fast:Nnn \g__regex_balance_intarray
      { \l__regex_curr_pos_int } { \l__regex_balance_int }
    \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
    \int_incr:N \l__regex_curr_pos_int
    \if_case:w #2 \exp_stop_f:
    \or: \int_incr:N \l__regex_balance_int
    \or: \int_decr:N \l__regex_balance_int
    \fi:
  }
```

(*End definition for* `\__regex_query_set:nnn`.)

`\__regex_query_get:`

Extract the current character and category codes at the current position from the appropriate arrays.

```
\cs_new_protected:Npn \__regex_query_get:
  {
    \l__regex_curr_char_int
      = \__intarray_item_fast:Nn \g__regex_charcode_intarray
          { \l__regex_curr_pos_int } \scan_stop:
    \l__regex_curr_catcode_int
      = \__intarray_item_fast:Nn \g__regex_catcode_intarray
          { \l__regex_curr_pos_int } \scan_stop:
  }
```

(*End definition for* `\__regex_query_get:`.)

36.5.3 Using states of the nfa

__regex_use_state: Use the current NFA instruction. The state is initially marked as belonging to the current step: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as step + 1: any thread hitting it at that point will be terminated.

```
\__debug_patch:nnNNpn
  { \__debug_trace:nnx { regex } { 2 } { state~\int_use:N \l__regex_curr_state_int } }
  { }
\cs_new_protected:Npn \__regex_use_state:
  {
    \__intarray_gset_fast:Nnn \g__regex_state_active_intarray
      { \l__regex_curr_state_int } { \l__regex_step_int }
    \__regex_toks_use:w \l__regex_curr_state_int
    \__intarray_gset_fast:Nnn \g__regex_state_active_intarray
      { \l__regex_curr_state_int } { \l__regex_step_int + 1 }
  }
```

(*End definition for __regex_use_state:.*)

__regex_use_state_and_submatches:nn This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the current_state and current_submatches and use the state if it has not yet been encountered at this step.

```
\cs_new_protected:Npn \__regex_use_state_and_submatches:nn #1 #2
  {
    \int_set:Nn \l__regex_curr_state_int {#1}
    \if_int_compare:w
        \__intarray_item_fast:Nn \g__regex_state_active_intarray
          { \l__regex_curr_state_int }
                      < \l__regex_step_int
      \tl_set:Nn \l__regex_curr_submatches_prop {#2}
      \exp_after:wN \__regex_use_state:
    \fi:
    \scan_stop:
  }
```

(*End definition for __regex_use_state_and_submatches:nn.*)

36.5.4 Actions when matching

__regex_action_start_wildcard: For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting \l__regex_fresh_thread_bool may be skipped by a successful thread, hence we had to add it to __regex_match_loop: too.

```
\cs_new_protected:Npn \__regex_action_start_wildcard:
  {
    \bool_set_true:N \l__regex_fresh_thread_bool
    \__regex_action_free:n {1}
    \bool_set_false:N \l__regex_fresh_thread_bool
    \__regex_action_cost:n {0}
  }
```

(*End definition for __regex_action_start_wildcard:.*)

__regex_action_free:n
__regex_action_free_group:n
__regex_action_free_aux:nn

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of \l__regex_curr_state_int and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the group version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the "normal" version will revisit a state even within the thread itself.

```
\cs_new_protected:Npn \__regex_action_free:n
  { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
\cs_new_protected:Npn \__regex_action_free_group:n
  { \__regex_action_free_aux:nn { < \l__regex_step_int } }
\cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
  {
    \use:x
      {
        \int_add:Nn \l__regex_curr_state_int {#2}
        \exp_not:n
          {
            \if_int_compare:w
                \__intarray_item_fast:Nn \g__regex_state_active_intarray
                  { \l__regex_curr_state_int }
                #1
              \exp_after:wN \__regex_use_state:
            \fi:
          }
        \int_set:Nn \l__regex_curr_state_int
          { \int_use:N \l__regex_curr_state_int }
        \tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
          { \exp_not:o \l__regex_curr_submatches_prop }
      }
  }
```

(*End definition for __regex_action_free:n, __regex_action_free_group:n, and __regex_action_free_aux:nn.*)

__regex_action_cost:n

A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```
\cs_new_protected:Npn \__regex_action_cost:n #1
  {
    \exp_args:No \__regex_store_state:n
      { \__int_value:w \__int_eval:w \l__regex_curr_state_int + #1 }
  }
```

(*End definition for __regex_action_cost:n.*)

__regex_store_state:n
__regex_store_submatches:

Put the given state in \g__regex_thread_state_intarray, and increment the length of the array. Also store the current submatch in the appropriate \toks.

```
\cs_new_protected:Npn \__regex_store_state:n #1
  {
    \__regex_store_submatches:
    \__intarray_gset_fast:Nnn \g__regex_thread_state_intarray
      { \l__regex_max_active_int } {#1}
    \int_incr:N \l__regex_max_active_int
  }
\cs_new_protected:Npn \__regex_store_submatches:
  {
    \__regex_toks_set:No \l__regex_max_active_int
      { \l__regex_curr_submatches_prop }
  }
```

(*End definition for* `\__regex_store_state:n` *and* `\__regex_store_submatches:`.)

`\__regex_disable_submatches:` Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```
\cs_new_protected:Npn \__regex_disable_submatches:
  {
    \cs_set_protected:Npn \__regex_store_submatches: { }
    \cs_set_protected:Npn \__regex_action_submatch:n ##1 { }
  }
```

(*End definition for* `\__regex_disable_submatches:`.)

`\__regex_action_submatch:n` Update the current submatches with the information from the current position. Maybe a bottleneck.

```
\cs_new_protected:Npn \__regex_action_submatch:n #1
  {
    \prop_put:Nno \l__regex_curr_submatches_prop {#1}
      { \int_use:N \l__regex_curr_pos_int }
  }
```

(*End definition for* `\__regex_action_submatch:n`.)

`\__regex_action_success:` There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is "fresh"; and we store the current position and submatches. The current step is then interrupted with `\__prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```
\cs_new_protected:Npn \__regex_action_success:
  {
    \__regex_if_two_empty_matches:F
      {
        \bool_set_true:N \l__regex_match_success_bool
        \bool_set_eq:NN \l__regex_empty_success_bool
          \l__regex_fresh_thread_bool
        \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
        \prop_set_eq:NN \l__regex_success_submatches_prop
          \l__regex_curr_submatches_prop
        \__prg_break:
      }
  }
```

(*End definition for* `\__regex_action_success:`.)

36.6 Replacement

36.6.1 Variables and helpers used in replacement

\l__regex_replacement_csnames_int

The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of "open" such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```
\int_new:N \l__regex_replacement_csnames_int
```

(*End definition for* `\l__regex_replacement_csnames_int`.)

\l__regex_replacement_category_tl
\l__regex_replacement_category_seq

This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

```
\tl_new:N \l__regex_replacement_category_tl
\seq_new:N \l__regex_replacement_category_seq
```

(*End definition for* `\l__regex_replacement_category_tl` *and* `\l__regex_replacement_category_seq`.)

\l__regex_balance_tl

This token list holds the replacement text for `\__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
\tl_new:N \l__regex_balance_tl
```

(*End definition for* `\l__regex_balance_tl`.)

__regex_replacement_balance_one_match:n

This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading `+` in the actual definition).

```
\cs_new:Npn \__regex_replacement_balance_one_match:n #1
  { - \__regex_submatch_balance:n {#1} }
```

(*End definition for* `\__regex_replacement_balance_one_match:n`.)

__regex_replacement_do_one_match:n

The input is the same as `\__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
\cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \__intarray_item_fast:Nn \g__regex_submatch_prev_intarray {#1} }
      { \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray {#1} }
  }
```

(*End definition for* `\__regex_replacement_do_one_match:n`.)

`\__regex_replacement_exp_not:N`

This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an `x`-expanding assignment, `\exp_not:N #` behaves as a single `#`, whereas `\exp_not:n {#}` behaves as a doubled `##`.

```
\cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(*End definition for* `\__regex_replacement_exp_not:N`*.*)

36.6.2 Query and brace balance

`\__regex_query_range:nn`
`\__regex_query_range_loop:ww`

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `\__regex_query_range:nn {`⟨*min*⟩`} {`⟨*max*⟩`}` unpacks registers from the position ⟨*min*⟩ to the position ⟨*max*⟩ $-$ 1 included. Once this is expanded, a second `x`-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```
\cs_new:Npn \__regex_query_range:nn #1#2
  {
    \exp_after:wN \__regex_query_range_loop:ww
    \__int_value:w \__int_eval:w #1 \exp_after:wN ;
    \__int_value:w \__int_eval:w #2 ;
    \__prg_break_point:
  }
\cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
  {
    \if_int_compare:w #1 < #2 \exp_stop_f:
    \else:
      \exp_after:wN \__prg_break:
    \fi:
    \__regex_toks_use:w #1 \exp_stop_f:
    \exp_after:wN \__regex_query_range_loop:ww
      \__int_value:w \__int_eval:w #1 + 1 ; #2 ;
  }
```

(*End definition for* `\__regex_query_range:nn` *and* `\__regex_query_range_loop:ww`*.*)

`\__regex_query_submatch:n`

Find the start and end positions for a given submatch (of a given match).

```
\cs_new:Npn \__regex_query_submatch:n #1
  {
    \__regex_query_range:nn
      { \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray {#1} }
      { \__intarray_item_fast:Nn \g__regex_submatch_end_intarray {#1} }
  }
```

(*End definition for* `\__regex_query_submatch:n`*.*)

`\__regex_submatch_balance:n`

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the ⟨*max pos*⟩ and ⟨*min pos*⟩. These two positions are found in the corresponding "submatch" arrays.

```
\cs_new_protected:Npn \__regex_submatch_balance:n #1
  {
    \__int_eval:w
      \int_compare:nNnTF
        { \__intarray_item_fast:Nn \g__regex_submatch_end_intarray {#1} } = 0
        { 0 }
        {
          \__intarray_item_fast:Nn \g__regex_balance_intarray
            { \__intarray_item_fast:Nn \g__regex_submatch_end_intarray {#1} }
        }
      -
      \int_compare:nNnTF
        { \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray {#1} } = 0
        { 0 }
        {
          \__intarray_item_fast:Nn \g__regex_balance_intarray
            { \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray {#1} }
        }
    \__int_eval_end:
  }
```

(*End definition for* `\__regex_submatch_balance:n`.)

36.6.3 Framework

`\__regex_replacement:n`
`\__regex_replacement_aux:n`

The replacement text is built incrementally by abusing `\toks` within a group (see l3tl-build). We keep track in `\l__regex_balance_int` of the balance of explicit begin- and end-group tokens and we store in `\l__regex_balance_tl` some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing `\prg_do_nothing:` because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the `balance_one_match` and `do_one_match` functions.

```
\__debug_patch:nnNNpn
  { \__debug_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
  { \__debug_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
\cs_new_protected:Npn \__regex_replacement:n #1
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \int_zero:N \l__regex_balance_int
      \tl_clear:N \l__regex_balance_tl
      \__regex_escape_use:nnnn
        {
          \if_charcode:w \c_right_brace_str ##1
            \__regex_replacement_rbrace:N
          \else:
            \__regex_replacement_normal:n
          \fi:
          ##1
        }
        { \__regex_replacement_escaped:N ##1 }
        { \__regex_replacement_normal:n ##1 }
        {#1}
      \prg_do_nothing: \prg_do_nothing:
      \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
        \__msg_kernel_error:nnx { kernel } { replacement-missing-rbrace }
          { \int_use:N \l__regex_replacement_csnames_int }
        \__tl_build_one:x
          { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
      \fi:
      \seq_if_empty:NF \l__regex_replacement_category_seq
        {
          \__msg_kernel_error:nnx { kernel } { replacement-missing-rparen }
            { \seq_count:N \l__regex_replacement_category_seq }
          \seq_clear:N \l__regex_replacement_category_seq
        }
      \cs_gset:Npx \__regex_replacement_balance_one_match:n ##1
        {
          + \int_use:N \l__regex_balance_int
          \l__regex_balance_tl
          - \__regex_submatch_balance:n {##1}
        }
    \__tl_build_end:
    \exp_args:No \__regex_replacement_aux:n \l__regex_internal_a_tl
  }
\cs_new_protected:Npn \__regex_replacement_aux:n #1
  {
    \cs_set:Npn \__regex_replacement_do_one_match:n ##1
      {
        \__regex_query_range:nn
          { \__intarray_item_fast:Nn \g__regex_submatch_prev_intarray {##1} }
          { \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray {##1} }
        #1
      }
  }
```

(*End definition for* `\__regex_replacement:n` *and* `\__regex_replacement_aux:n`.)

__regex_replacement_normal:n

Most characters are simply sent to the output by `\__tl_build_one:n`, unless a particular category code has been requested: then `\__regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`.

```
\cs_new_protected:Npn \__regex_replacement_normal:n #1
  {
    \tl_if_empty:NTF \l__regex_replacement_category_tl
      { \__tl_build_one:n {#1} }
      { % (
        \token_if_eq_charcode:NNTF #1 )
          {
            \seq_pop:NN \l__regex_replacement_category_seq
              \l__regex_replacement_category_tl
          }
          {
            \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
              \__regex_replacement_normal:n {#1}
          }
      }
  }
```

(*End definition for* `\__regex_replacement_normal:n`.)

__regex_replacement_escaped:N

As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character. We use `\token_to_str:N` to give spaces the right category code.

```
\cs_new_protected:Npn \__regex_replacement_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_replacement_#1:w }
      {
        \if_int_compare:w 1 < 1#1 \exp_stop_f:
          \__regex_replacement_put_submatch:n {#1}
        \else:
          \exp_args:No \__regex_replacement_normal:n
            { \token_to_str:N #1 }
        \fi:
      }
  }
```

(*End definition for* `\__regex_replacement_escaped:N`.)

36.6.4 Submatches

__regex_replacement_put_submatch:n

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match. We cannot use `\int_eval:n` because it is expandable, and would be expanded too early (short of adding `\exp_not:N`, making the code messy again).

```
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
  {
    \if_int_compare:w #1 < \l__regex_capturing_group_int
      \__tl_build_one:n { \__regex_query_submatch:n { #1 + ##1 } }
      \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
        \tl_put_right:Nn \l__regex_balance_tl
          { + \__regex_submatch_balance:n { \__int_eval:w #1+##1 \__int_eval_end: } }
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_replacement_put_submatch:n`.)

__regex_replacement_g:w
__regex_replacement_g_digits:NN

Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```
\cs_new_protected:Npn \__regex_replacement_g:w #1#2
  {
    \str_if_eq_x:nnTF { #1#2 } { \__regex_replacement_normal:n \c_left_brace_str }
      { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
      { \__regex_replacement_error:NNN g #1 #2 }
  }
\cs_new:Npn \__regex_replacement_g_digits:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
```

```
      {
        \if_int_compare:w 1 < 1#2 \exp_stop_f:
          #2
          \exp_after:wN \use_i:nnn
          \exp_after:wN \__regex_replacement_g_digits:NN
        \else:
          \exp_stop_f:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
      {
        \exp_stop_f:
        \if_meaning:w \__regex_replacement_rbrace:N #1
          \exp_args:No \__regex_replacement_put_submatch:n
            { \int_use:N \l__regex_internal_a_int }
          \exp_after:wN \use_none:nn
        \else:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
    #1 #2
  }
```

(*End definition for* `\__regex_replacement_g:w` *and* `\__regex_replacement_g_digits:NN`.)

36.6.5 Csnames in replacement

`\__regex_replacement_c:w`

`\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```
\cs_new_protected:Npn \__regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
          { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
          {
            \cs_if_exist:cTF { __regex_replacement_c_#2:w }
              { \__regex_replacement_cat:NNN #2 }
              { \__regex_replacement_error:NNN c #1#2 }
          }
      }
      { \__regex_replacement_error:NNN c #1#2 }
  }
```

(*End definition for* `\__regex_replacement_c:w`.)

`\__regex_replacement_cu_aux:Nw`

Start a control sequence with `\cs:w`, protected from expansion by `#1` (either `\__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```
\cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
```

```
  {
    \if_case:w \l__regex_replacement_csnames_int
      \__tl_build_one:n { \exp_not:n { \exp_after:wN #1 \cs:w } }
    \else:
      \__tl_build_one:n { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }
```

(*End definition for* `\__regex_replacement_cu_aux:Nw`.)

__regex_replacement_u:w

Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```
\cs_new_protected:Npn \__regex_replacement_u:w #1#2
  {
    \str_if_eq_x:nnTF { #1#2 } { \__regex_replacement_normal:n \c_left_brace_str }
      { \__regex_replacement_cu_aux:Nw \exp_not:V }
      { \__regex_replacement_error:NNN u #1#2 }
  }
```

(*End definition for* `\__regex_replacement_u:w`.)

__regex_replacement_rbrace:N

Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```
\cs_new_protected:Npn \__regex_replacement_rbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
      \__tl_build_one:n \cs_end:
      \int_decr:N \l__regex_replacement_csnames_int
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*End definition for* `\__regex_replacement_rbrace:N`.)

36.6.6 Characters in replacement

__regex_replacement_cat:NNN

Here, `#1` is a letter among `BEMTPUDSLOA` and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```
\cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
  {
    \token_if_eq_meaning:NNTF \prg_do_nothing: #3
      { \__msg_kernel_error:nn { kernel } { replacement-catcode-end } }
      {
        \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
          {
            \__msg_kernel_error:nnnn
              { kernel } { replacement-catcode-in-cs } {#1} {#3}
            #2 #3
          }
          {
            \str_if_eq:nnTF { #2 #3 } { \__regex_replacement_normal:n ( } % )
              {
                \seq_push:NV \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
                \tl_set:Nn \l__regex_replacement_category_tl {#1}
              }
              {
                \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
                  {
                    \__regex_char_if_alphanumeric:NTF #3
                      {
                        \__msg_kernel_error:nnnn
                          { kernel } { replacement-catcode-escaped }
                          {#1} {#3}
                      }
                      { }
                  }
                \use:c { __regex_replacement_c_#1:w } #2 #3
              }
          }
      }
  }
```

(*End definition for* `\__regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
\group_begin:
```

__regex_replacement_char:nNN

The only way to produce an arbitrary character–catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```
\cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
  {
    \tex_lccode:D 0 = `#3 \scan_stop:
    \tex_lowercase:D { \__tl_build_one:n {#1} }
  }
```

(*End definition for* `\__regex_replacement_char:nNN`.)

__regex_replacement_c_A:w

For an active character, expansion must be avoided, twice because we later do two `x`-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```
\char_set_catcode_active:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_A:w
  { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^^@ } } }
```

(*End definition for* `\__regex_replacement_c_A:w`.)

__regex_replacement_c_B:w

An explicit begin-group token increases the balance, unless within a \c{...} or \u{...} construction. Add the desired begin-group character, using the standard \if_false: trick. We eventually x-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The \exp_after:wN is not strictly needed, but is more consistent with l3tl-analysis.

```
\char_set_catcode_group_begin:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_B:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
      \int_incr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
  }
```

(*End definition for* __regex_replacement_c_B:w.)

__regex_replacement_c_C:w

This is not quite catcode-related: when the user requests a character with category "control sequence", the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

```
\cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
  { \__tl_build_one:n { \exp_not:N \exp_not:N \exp_not:c {#2} } }
```

(*End definition for* __regex_replacement_c_C:w.)

__regex_replacement_c_D:w

Subscripts fit the mould: \lowercase the null byte with the correct category.

```
\char_set_catcode_math_subscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_D:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_D:w.)

__regex_replacement_c_E:w

Similar to the begin-group case, the second x-expansion produces the bare end-group token.

```
\char_set_catcode_group_end:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_E:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
      \int_decr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \if_false: { \fi:  ^^@ } }
  }
```

(*End definition for* __regex_replacement_c_E:w.)

__regex_replacement_c_L:w

Simply \lowercase a letter null byte to produce an arbitrary letter.

```
\char_set_catcode_letter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_L:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_L:w.)

__regex_replacement_c_M:w No surprise here, we lowercase the null math toggle.

```
\char_set_catcode_math_toggle:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_M:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_M:w*.*)

__regex_replacement_c_O:w Lowercase an other null byte.

```
\char_set_catcode_other:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_O:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_O:w*.*)

__regex_replacement_c_P:w For macro parameters, expansion is a tricky issue. We need to prepare for two x-expansions and passing through various macro definitions. Note that we cannot replace one \exp_not:n by doubling the macro parameter characters because this would misbehave if a mischievous user asks for \c{\cP\#}, since that macro parameter character would be doubled.

```
\char_set_catcode_parameter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_P:w
  {
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
  }
```

(*End definition for* __regex_replacement_c_P:w*.*)

__regex_replacement_c_S:w Spaces are normalized on input by TEX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```
\cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
  {
    \if_int_compare:w ‘#2 = 0 \exp_stop_f:
      \__msg_kernel_error:nn { kernel } { replacement-null-space }
    \fi:
    \tex_lccode:D ‘\ = ‘#2 \scan_stop:
    \tex_lowercase:D { \__tl_build_one:n {~} }
  }
```

(*End definition for* __regex_replacement_c_S:w*.*)

__regex_replacement_c_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```
\char_set_catcode_alignment:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_T:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_T:w*.*)

__regex_replacement_c_U:w Simple call to __regex_replacement_char:nNN which lowercases the math superscript ^^@.

```
\char_set_catcode_math_superscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_U:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End definition for* __regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```
\group_end:
```

36.6.7 An error

__regex_replacement_error:NNN

Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
\cs_new_protected:Npn \__regex_replacement_error:NNN #1#2#3
  {
    \__msg_kernel_error:nnx { kernel } { replacement-#1 } {#3}
    #2 #3
  }
```

(*End definition for* __regex_replacement_error:NNN.)

36.7 User functions

\regex_new:N

Before being assigned a sensible value, a regex variable matches nothing.

```
\cs_new_protected:Npn \regex_new:N #1
  { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(*End definition for* \regex_new:N. *This function is documented on page 206.*)

\regex_set:Nn
\regex_gset:Nn
\regex_const:Nn

Compile, then store the result in the user variable with the appropriate assignment function.

```
\cs_new_protected:Npn \regex_set:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_set_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_gset:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_gset_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_const:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
  }
```

(*End definition for* \regex_set:Nn, \regex_gset:Nn, *and* \regex_const:Nn. *These functions are documented on page 206.*)

\regex_show:N
\regex_show:n

User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `\__regex_show:Nx` is defined in a different section.

```
\cs_new_protected:Npn \regex_show:n #1
  {
    \__regex_compile:n {#1}
    \__regex_show:Nn \l__regex_internal_regex
      { { \tl_to_str:n {#1} } }
  }
\cs_new_protected:Npn \regex_show:N #1
  { \__regex_show:Nn #1 { variable~\token_to_str:N #1 } }
```

(*End definition for* `\regex_show:N` *and* `\regex_show:n`*. These functions are documented on page 206.*)

`\regex_match:nnTF`
`\regex_match:NnTF`

Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or `false`.

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
    \__regex_return:
  }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:N #1 } {#2}
    \__regex_return:
  }
```

(*End definition for* `\regex_match:nnTF` *and* `\regex_match:NnTF`*. These functions are documented on page 206.*)

`\regex_count:nnN`
`\regex_count:NnN`

Again, use an auxiliary whose first argument builds the NFA.

```
\cs_new_protected:Npn \regex_count:nnN #1
  { \__regex_count:nnN { \__regex_build:n {#1} } }
\cs_new_protected:Npn \regex_count:NnN #1
  { \__regex_count:nnN { \__regex_build:N #1 } }
```

(*End definition for* `\regex_count:nnN` *and* `\regex_count:NnN`*. These functions are documented on page 207.*)

`\regex_extract_once:nnN`
`\regex_extract_once:NnN`
`\regex_extract_all:nnN`
`\regex_extract_all:NnN`
`\regex_replace_once:nnN`
`\regex_replace_once:NnN`
`\regex_replace_all:nnN`
`\regex_replace_all:NnN`
`\regex_split:nnN`
`\regex_split:NnN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:NnNTF`
`\regex_extract_all:nnNTF`
`\regex_extract_all:NnNTF`
`\regex_replace_once:nnNTF`
`\regex_replace_once:NnNTF`
`\regex_replace_all:nnNTF`
`\regex_replace_all:NnNTF`
`\regex_split:nnNTF`
`\regex_split:NnNTF`

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `\__regex_build:n` or `\__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call `\__regex_return:` to return either `true` or `false` once matching has been performed.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
  {
    \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
    \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N   ##1   } }
    \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
    \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:N   ##1   } {##2} ##3 \__regex_return: }
  }
\__regex_tmp:w \__regex_extract_once:nnN
  \regex_extract_once:nnN \regex_extract_once:NnN
\__regex_tmp:w \__regex_extract_all:nnN
  \regex_extract_all:nnN \regex_extract_all:NnN
\__regex_tmp:w \__regex_replace_once:nnN
  \regex_replace_once:nnN \regex_replace_once:NnN
\__regex_tmp:w \__regex_replace_all:nnN
  \regex_replace_all:nnN \regex_replace_all:NnN
\__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN
```

(*End definition for* `\regex_extract_once:nnN` *and others. These functions are documented on page* ??.)

36.7.1 Variables and helpers for user functions

\l__regex_match_count_int

The number of matches found so far is stored in \l__regex_match_count_int. This is only used in the \regex_count:nnN functions.

```
\int_new:N \l__regex_match_count_int
```

(*End definition for \l__regex_match_count_int.*)

__regex_begin
__regex_end

Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.

```
\flag_new:n { __regex_begin }
\flag_new:n { __regex_end }
```

(*End definition for __regex_begin and __regex_end.*)

\l__regex_min_submatch_int
\l__regex_submatch_int
\l__regex_zeroth_submatch_int

The end-points of each submatch are stored in two arrays whose index ⟨*submatch*⟩ ranges from \l__regex_min_submatch_int (inclusive) to \l__regex_submatch_int (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at zeroth_submatch. The entry \l__regex_zeroth_submatch_int in \g__regex_submatch_prev_intarray holds the position at which that match attempt started: this is used for splitting and replacements.

```
\int_new:N \l__regex_min_submatch_int
\int_new:N \l__regex_submatch_int
\int_new:N \l__regex_zeroth_submatch_int
```

(*End definition for \l__regex_min_submatch_int, \l__regex_submatch_int, and \l__regex_zeroth_submatch_int.*)

\g__regex_submatch_prev_intarray
\g__regex_submatch_begin_intarray
\g__regex_submatch_end_intarray

Hold the place where the match attempt begun and the end-points of each submatch.

```
\__intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\__intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
\__intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
```

(*End definition for \g__regex_submatch_prev_intarray, \g__regex_submatch_begin_intarray, and \g__regex_submatch_end_intarray.*)

__regex_return:

This function triggers either \prg_return_false: or \prg_return_true: as appropriate to whether a match was found or not. It is used by all user conditionals.

```
\cs_new_protected:Npn \__regex_return:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for __regex_return:.*)

36.7.2 Matching

__regex_if_match:nn

We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```
\cs_new_protected:Npn \__regex_if_match:nn #1#2
  {
    \group_begin:
      \__regex_disable_submatches:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
    \group_end:
  }
```

(*End definition for* __regex_if_match:nn*.*)

__regex_count:nnN

Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing \l__regex_match_count_int every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```
\cs_new_protected:Npn \__regex_count:nnN #1#2#3
  {
    \group_begin:
      \__regex_disable_submatches:
      \int_zero:N \l__regex_match_count_int
      \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
      #1
      \__regex_match:n {#2}
      \exp_args:NNNo
    \group_end:
    \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
  }
```

(*End definition for* __regex_count:nnN*.*)

36.7.3 Extracting submatches

__regex_extract_once:nnN
__regex_extract_all:nnN

Match once or multiple times. After each match (or after the only match), extract the submatches using __regex_extract:. At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```
\cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
      \__regex_extract:
    \__regex_group_end_extract_seq:N #3
  }
\cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
```

```
   \__regex_match:n {#2}
  \__regex_group_end_extract_seq:N #3
 }
```

(*End definition for* `\__regex_extract_once:nnN` *and* `\__regex_extract_all:nnN`.)

`\__regex_split:nnN`

Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```
\cs_new_protected:Npn \__regex_split:nnN #1#2#3
 {
  \group_begin:
   \__regex_multi_match:n
    {
     \if_int_compare:w \l__regex_start_pos_int < \l__regex_success_pos_int
      \__regex_extract:
      \__intarray_gset_fast:Nnn \g__regex_submatch_prev_intarray
       { \l__regex_zeroth_submatch_int } { 0 }
      \__intarray_gset_fast:Nnn \g__regex_submatch_end_intarray
       { \l__regex_zeroth_submatch_int }
       {
        \__intarray_item_fast:Nn \g__regex_submatch_begin_intarray
         { \l__regex_zeroth_submatch_int }
       }
      \__intarray_gset_fast:Nnn \g__regex_submatch_begin_intarray
       { \l__regex_zeroth_submatch_int }
       { \l__regex_start_pos_int }
     \fi:
    }
   #1
   \__regex_match:n {#2}
⟨assert⟩\assert_int:n { \l__regex_curr_pos_int = \l__regex_max_pos_int }
   \__intarray_gset_fast:Nnn \g__regex_submatch_prev_intarray
    { \l__regex_submatch_int } { 0 }
   \__intarray_gset_fast:Nnn \g__regex_submatch_end_intarray
    { \l__regex_submatch_int }
    { \l__regex_max_pos_int }
   \__intarray_gset_fast:Nnn \g__regex_submatch_begin_intarray
    { \l__regex_submatch_int }
    { \l__regex_start_pos_int }
   \int_incr:N \l__regex_submatch_int
   \if_meaning:w \c_true_bool \l__regex_empty_success_bool
    \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
     \int_decr:N \l__regex_submatch_int
    \fi:
   \fi:
  \__regex_group_end_extract_seq:N #3
 }
```

(*End definition for* `\__regex_split:nnN`.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \l__regex_internal_a_tl. We detect unbalanced results using the two flags @@_begin and @@_end, raised whenever we see too many begin-group or end-group tokens in a submatch. We disable __seq_item:n to prevent two x-expansions.

```
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
  {
      \cs_set_eq:NN \__seq_item:n \scan_stop:
      \flag_clear:n { __regex_begin }
      \flag_clear:n { __regex_end }
      \tl_set:Nx \l__regex_internal_a_tl
        {
          \s__seq
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            { 1 }
            { \l__regex_submatch_int - 1 }
            \__regex_extract_seq_aux:n
        }
      \int_compare:nNnF
        { \flag_height:n { __regex_begin } + \flag_height:n { __regex_end } }
        = 0
        {
          \__msg_kernel_error:nnxxx { kernel } { result-unbalanced }
            { splitting~or~extracting~submatches }
            { \flag_height:n { __regex_end } }
            { \flag_height:n { __regex_begin } }
        }
      \use:x
        {
          \group_end:
          \tl_set:Nn \exp_not:N #1 { \l__regex_internal_a_tl }
        }
  }
```

(*End definition for __regex_group_end_extract_seq:N.*)

__regex_extract_seq_aux:n
__regex_extract_seq_aux:ww The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```
\cs_new:Npn \__regex_extract_seq_aux:n #1
  {
    \__seq_item:n
      {
        \exp_after:wN \__regex_extract_seq_aux:ww
        \__int_value:w \__regex_submatch_balance:n {#1} ; #1;
      }
  }
\cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
  {
    \if_int_compare:w #1 < 0 \exp_stop_f:
      \flag_raise:n { __regex_end }
      \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
    \fi:
```

```
    \__regex_query_submatch:n {#2}
    \if_int_compare:w #1 > 0 \exp_stop_f:
      \flag_raise:n { __regex_begin }
      \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
    \fi:
  }
```

(*End definition for* `\__regex_extract_seq_aux:n` *and* `\__regex_extract_seq_aux:ww`*.*)

`\__regex_extract:`
`\__regex_extract_b:wn`
`\__regex_extract_e:wn`

Our task here is to extract from the property list `\l__regex_success_submatches_prop` the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. We begin by emptying those entries. Then for each ⟨*key*⟩–⟨*value*⟩ pair in the property list update the appropriate entry. This is somewhat a hack: the ⟨*key*⟩ is a non-negative integer followed by `<` or `>`, which we use in a comparison to -1. At the end, store the information about the position at which the match attempt started, in `\g__regex_submatch_prev_intarray`.

```
\cs_new_protected:Npn \__regex_extract:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
      \prg_replicate:nn \l__regex_capturing_group_int
        {
          \__intarray_gset_fast:Nnn \g__regex_submatch_begin_intarray
            { \l__regex_submatch_int } { 0 }
          \__intarray_gset_fast:Nnn \g__regex_submatch_end_intarray
            { \l__regex_submatch_int } { 0 }
          \__intarray_gset_fast:Nnn \g__regex_submatch_prev_intarray
            { \l__regex_submatch_int } { 0 }
          \int_incr:N \l__regex_submatch_int
        }
      \prop_map_inline:Nn \l__regex_success_submatches_prop
        {
          \if_int_compare:w ##1 - 1 \exp_stop_f:
            \exp_after:wN \__regex_extract_e:wn \__int_value:w
          \else:
            \exp_after:wN \__regex_extract_b:wn \__int_value:w
          \fi:
          \__int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
        }
      \__intarray_gset_fast:Nnn \g__regex_submatch_prev_intarray
        { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
    \fi:
  }
\cs_new_protected:Npn \__regex_extract_b:wn #1 < #2
  { \__intarray_gset_fast:Nnn \g__regex_submatch_begin_intarray {#1} {#2} }
\cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
  { \__intarray_gset_fast:Nnn \g__regex_submatch_end_intarray {#1} {#2} }
```

(*End definition for* `\__regex_extract:` *,* `\__regex_extract_b:wn` *, and* `\__regex_extract_e:wn`*.*)

36.7.4 Replacement

`\__regex_replace_once:nnN`

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```
\cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \__regex_replacement:n {#2}
      \exp_args:No \__regex_match:n { #3 }
      \if_meaning:w \c_false_bool \g__regex_success_bool
        \group_end:
      \else:
        \__regex_extract:
        \int_set:Nn \l__regex_balance_int
          {
            \__regex_replacement_balance_one_match:n
              { \l__regex_zeroth_submatch_int }
          }
        \tl_set:Nx \l__regex_internal_a_tl
          {
            \__regex_replacement_do_one_match:n { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              {
                \__intarray_item_fast:Nn \g__regex_submatch_end_intarray
                  { \l__regex_zeroth_submatch_int }
              }
              { \l__regex_max_pos_int }
          }
        \__regex_group_end_replace:N #3
      \fi:
  }
```

(*End definition for* __regex_replace_once:nnN.)

__regex_replace_all:nnN

Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from \l__regex_min_submatch_int to \l__regex_submatch_int hold information about submatches of every match in order; each match corresponds to \l__regex_capturing_group_int consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```
\cs_new_protected:Npn \__regex_replace_all:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \__regex_replacement:n {#2}
      \exp_args:No \__regex_match:n {#3}
      \int_set:Nn \l__regex_balance_int
        {
          0
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_balance_one_match:n
        }
      \tl_set:Nx \l__regex_internal_a_tl
        {
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_do_one_match:n
          \__regex_query_range:nn
            \l__regex_start_pos_int \l__regex_max_pos_int
        }
    \__regex_group_end_replace:N #3
  }
```

(*End definition for* `\__regex_replace_all:nnN`.)

`\__regex_group_end_replace:N`

If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the x-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```
\cs_new_protected:Npn \__regex_group_end_replace:N #1
  {
    \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
    \else:
      \__msg_kernel_error:nnxxx { kernel } { result-unbalanced }
        { replacing }
        { \int_max:nn { - \l__regex_balance_int } { 0 } }
        { \int_max:nn { \l__regex_balance_int } { 0 } }
    \fi:
    \use:x
      {
        \group_end:
        \tl_set:Nn \exp_not:N #1
          {
            \if_int_compare:w \l__regex_balance_int < 0 \exp_stop_f:
              \prg_replicate:nn { - \l__regex_balance_int }
                { { \if_false: } \fi: }
            \fi:
            \l__regex_internal_a_tl
            \if_int_compare:w \l__regex_balance_int > 0 \exp_stop_f:
              \prg_replicate:nn { \l__regex_balance_int }
                { \if_false: { \fi: } }
            \fi:
          }
      }
  }
```

(*End definition for* `\__regex_group_end_replace:N`.)

36.7.5 Storing and showing compiled patterns

36.8 Messages

Messages for the preparsing phase.

```
\__msg_kernel_new:nnnn { kernel } { trailing-backslash }
  { Trailing~escape~character~’\iow_char:N\\’. }
  {
    A~regular~expression~or~its~replacement~text~ends~with~
    the~escape~character~’\iow_char:N\\’.~It~will~be~ignored.
  }
\__msg_kernel_new:nnnn { kernel } { x-missing-rbrace }
  { Missing~closing~brace~in~’\iow_char:N\\x’~hexadecimal~sequence. }
  {
    You~wrote~something~like~
    ’\iow_char:N\\x\{...#1’.~
    The~closing~brace~is~missing.
  }
\__msg_kernel_new:nnnn { kernel } { x-overflow }
  { Character~code~’#1’~too~large~in~’\iow_char:N\\x’~hexadecimal~sequence. }
  {
    You~wrote~something~like~
    ’\iow_char:N\\x\{\int_to_Hex:n{#1}\}’.~
    The~character~code~#1~is~larger~than~
    the~maximum~value~\int_use:N \c_max_char_int.
  }
```

Invalid quantifier.

```
\__msg_kernel_new:nnnn { kernel } { invalid-quantifier }
  { Braced~quantifier~’#1’~may~not~be~followed~by~’#2’. }
  {
    The~character~’#2’~is~invalid~in~the~braced~quantifier~’#1’.~
    The~only~valid~quantifiers~are~’*’,~’?’,~’+’,~’{<int>}’,~
    ’{<min>,}’~and~’{<min>,<max>}’,~optionally~followed~by~’?’.
  }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```
\__msg_kernel_new:nnnn { kernel } { missing-rbrack }
  { Missing~right~bracket~inserted~in~regular~expression. }
  {
    LaTeX~was~given~a~regular~expression~where~a~character~class~
    was~started~with~’[’,~but~the~matching~’]’~is~missing.
  }
\__msg_kernel_new:nnnn { kernel } { missing-rparen }
  {
    Missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
    inserted~in~regular~expression.
  }
  {
    LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
    more~left~parentheses~than~right~parentheses.
  }
\__msg_kernel_new:nnnn { kernel } { extra-rparen }
```

```
  { Extra~right~parenthesis~ignored~in~regular~expression. }
  {
    LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
    was~open.~The~parenthesis~will~be~ignored.
  }
```

Some escaped alphanumerics are not allowed everywhere.

```
\__msg_kernel_new:nnnn { kernel } { bad-escape }
  {
    Invalid~escape~'\iow_char:N\\#1'~
    \__regex_if_in_cs:TF { within~a~control~sequence. }
      {
        \__regex_if_in_class:TF
          { in~a~character~class. }
          { following~a~category~test. }
      }
  }
  {
    The~escape~sequence~'\iow_char:N\\#1'~may~not~appear~
    \__regex_if_in_cs:TF
      {
        within~a~control~sequence~test~introduced~by~
        '\iow_char:N\\c\iow_char:N\{'.
      }
      {
        \__regex_if_in_class:TF
          { within~a~character~class~ }
          { following~a~category~test~such~as~'\iow_char:N\\cL'~ }
        because~it~does~not~match~exactly~one~character.
      }
  }
```

Range errors.

```
\__msg_kernel_new:nnnn { kernel } { range-missing-end }
  { Invalid~end-point~for~range~'#1-#2'~in~character~class. }
  {
    The~end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
    end-point~for~a~range:~alphanumeric~characters~should~not~be~
    escaped,~and~non-alphanumeric~characters~should~be~escaped.
  }
\__msg_kernel_new:nnnn { kernel } { range-backwards }
  { Range~'[#1-#2]'~out~of~order~in~character~class. }
  {
    In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
    the~first~character~code~must~not~be~larger~than~the~second.~
    Here,~'#1'~has~character~code~\int_eval:n {`#1},~while~
    '#2'~has~character~code~\int_eval:n {`#2}.
  }
```

Errors related to \c and \u.

```
\__msg_kernel_new:nnnn { kernel } { c-bad-mode }
  { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
  {
    The~'\iow_char:N\\c'~escape~cannot~be~used~within~
    a~control~sequence~test~'\iow_char:N\\c{...}'.~
    To~combine~several~category~tests,~use~’\iow_char:N\\c[...]’.
  }
\__msg_kernel_new:nnnn { kernel } { c-C-invalid }
  { ’\iow_char:N\\cC’~should~be~followed~by~’.’~or~’(’,~not~’#1’. }
  {
    The~’\iow_char:N\\cC’~construction~restricts~the~next~item~to~be~a~
    control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
    It~only~makes~sense~to~follow~it~by~’.’~or~by~a~group.
  }
\__msg_kernel_new:nnnn { kernel } { c-missing-rbrace }
  { Missing~right~brace~inserted~for~’\iow_char:N\\c’~escape. }
  {
    LaTeX~was~given~a~regular~expression~where~a~
    ’\iow_char:N\\c\iow_char:N\{...’~construction~was~not~ended~
    with~a~closing~brace~’\iow_char:N\}’.
  }
\__msg_kernel_new:nnnn { kernel } { c-missing-rbrack }
  { Missing~right~bracket~inserted~for~’\iow_char:N\\c’~escape. }
  {
    A~construction~’\iow_char:N\\c[...’~appears~in~a~
    regular~expression,~but~the~closing~’]’~is~not~present.
  }
\__msg_kernel_new:nnnn { kernel } { c-missing-category }
  { Invalid~character~’#1’~following~’\iow_char:N\\c’~escape. }
  {
    In~regular~expressions,~the~’\iow_char:N\\c’~escape~sequence~
    may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
    capital~letter~representing~a~character~category,~namely~
    one~of~’ABCDELMOPSTU’.
  }
\__msg_kernel_new:nnnn { kernel } { c-trailing }
  { Trailing~category~code~escape~’\iow_char:N\\c’... }
  {
    A~regular~expression~ends~with~’\iow_char:N\\c’~followed~
    by~a~letter.~It~will~be~ignored.
  }
\__msg_kernel_new:nnnn { kernel } { u-missing-lbrace }
  { Missing~left~brace~following~’\iow_char:N\\u’~escape. }
  {
    The~’\iow_char:N\\u’~escape~sequence~must~be~followed~by~
    a~brace~group~with~the~name~of~the~variable~to~use.
  }
\__msg_kernel_new:nnnn { kernel } { u-missing-rbrace }
  { Missing~right~brace~inserted~for~’\iow_char:N\\u’~escape. }
  {
    LaTeX~
    \str_if_eq_x:nnTF { } {#2}
      { reached~the~end~of~the~string~ }
      { encountered~an~escaped~alphanumeric~character ’\iow_char:N\\#2’~ }
    when~parsing~the~argument~of~an~’\iow_char:N\\u\iow_char:N\{...\}’~escape.
  }
```

Errors when encountering the POSIX syntax [:...:].

```
\__msg_kernel_new:nnnn { kernel } { posix-unsupported }
  { POSIX~collating~element~’[#1 ~ #1]’~not~supported. }
  {
    The~’[.foo.]’~and~’[=bar=]’~syntaxes~have~a~special~meaning~
    in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
    Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
  }
\__msg_kernel_new:nnnn { kernel } { posix-unknown }
  { POSIX~class~’[:#1:]’~unknown. }
  {
    ’[:#1:]’~is~not~among~the~known~POSIX~classes~
    ’[:alnum:]’,~’[:alpha:]’,~’[:ascii:]’,~’[:blank:]’,~
    ’[:cntrl:]’,~’[:digit:]’,~’[:graph:]’,~’[:lower:]’,~
    ’[:print:]’,~’[:punct:]’,~’[:space:]’,~’[:upper:]’,~
    ’[:word:]’,~and~’[:xdigit:]’.
  }
\__msg_kernel_new:nnnn { kernel } { posix-missing-close }
  { Missing~closing~’:]’~for~POSIX~class. }
  { The~POSIX~syntax~’#1’~must~be~followed~by~’:]’,~not~’#2’. }
```

In various cases, the result of a l3regex operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```
\__msg_kernel_new:nnnn { kernel } { result-unbalanced }
  { Missing~brace~inserted~when~#1. }
  {
    LaTeX~was~asked~to~do~some~regular~expression~operation,~
    and~the~resulting~token~list~would~not~have~the~same~number~
    of~begin-group~and~end-group~tokens.~Braces~were~inserted:~
    #2~left,~#3~right.
  }
```

Error message for unknown options.

```
\__msg_kernel_new:nnnn { kernel } { unknown-option }
  { Unknown~option~’#1’~for~regular~expressions. }
  {
    The~only~available~option~is~’case-insensitive’,~toggled~by~
    ’(?i)’~and~’(?-i)’.
  }
\__msg_kernel_new:nnnn { kernel } { special-group-unknown }
  { Unknown~special~group~’#1~...’~in~a~regular~expression. }
  {
    The~only~valid~constructions~starting~with~’(?’~are~
    ’(?:~...~)’,~’(?|~...~)’,~’(?i)’,~and~’(?-i)’.
  }
```

Errors in the replacement text.

```
\__msg_kernel_new:nnnn { kernel } { replacement-c }
  { Misused~’\iow_char:N\\c’~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~’\iow_char:N\\c’~escape~sequence~
    can~be~followed~by~one~of~the~letters~’ABCDELMOPSTU’~
    or~a~brace~group,~not~by~’#1’.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-u }
  { Misused~’\iow_char:N\\u’~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
    must~be~~followed~by~a~brace~group~holding~the~name~of~the~
    variable~to~use.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-g }
  {
    Missing~brace~for~the~'\iow_char:N\\g'~construction~
    in~a~replacement~text.
  }
  {
    In~the~replacement~text~for~a~regular~expression~search,~
    submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
    or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-catcode-end }
  {
    Missing~character~for~the~'\iow_char:N\\c<category><character>'~
    construction~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow.~LaTeX~
    reached~the~end~of~the~replacement~when~looking~for~that.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-catcode-escaped }
  {
    Escaped~letter~or~digit~after~category~code~in~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow,~not~
    '\iow_char:N\\#2'.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-catcode-in-cs }
  {
    Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
    '\iow_char:N\\c\{...\}'~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~category~codes~of~the~argument~of~
    '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
    sequence~name.
  }
\__msg_kernel_new:nnnn { kernel } { replacement-null-space }
  { TeX~cannot~build~a~space~token~with~character~code~0. }
  {
    You~asked~for~a~character~token~with~category~space,~
    and~character~code~0,~for~instance~through~
    '\iow_char:N\\cS\iow_char:N\\x00'.~
    This~specific~case~is~impossible~and~will~be~replaced~
    by~a~normal~space.
  }
```

```
\__msg_kernel_new:nnnn { kernel } { replacement-missing-rbrace }
  { Missing~right~brace~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
  }
\__msg_kernel_new:nnnn { kernel } { replacement-missing-rparen }
  { Missing~right~parenthesis~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~\int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
  }
```

__regex_msg_repeated:nnN

This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (−1 for infinite number), and #3 tells us about lazyness.

```
\cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
  {
    \str_if_eq_x:nnF { #1 #2 } { 1 0 }
      {
        , ~ repeated ~
        \int_case:nnF {#2}
          {
            { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
            {  0 } { #1~times }
          }
          {
            between~#1~and~\int_eval:n {#1+#2}~times,~
            \bool_if:NTF #3 { lazy } { greedy }
          }
      }
  }
```

(*End definition for* `\__regex_msg_repeated:nnN`*.*)

36.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

__debug_trace_push:nnN
__debug_trace_pop:nnN
__debug_trace:nnx

Here #1 is the module name (regex) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```
\__debug:TF
  {
    \cs_new_protected:Npn \__debug_trace_push:nnN #1#2#3
      { \__debug_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
    \cs_new_protected:Npn \__debug_trace_pop:nnN #1#2#3
      { \__debug_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
    \cs_new_protected:Npn \__debug_trace:nnx #1#2#3
      {
        \int_compare:nNnF
          { \int_use:c { g__debug_trace_#1_int } } < {#2}
          { \iow_term:x { Trace:~#3 } }
      }
  }
  { }
```

(*End definition for* `\__debug_trace_push:nnN`*,* `\__debug_trace_pop:nnN`*, and* `\__debug_trace:nnx`*.*)

`\g__debug_trace_regex_int` No tracing when that is zero.

```
\int_new:N \g__debug_trace_regex_int
```

(*End definition for* `\g__debug_trace_regex_int`*.*)

`\__regex_trace_states:n` This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__regex_max_state_int` (excluded).

```
\__debug:TF
  {
    \cs_new_protected:Npn \__regex_trace_states:n #1
      {
        \int_step_inline:nnnn
          \l__regex_min_state_int
          { 1 }
          { \l__regex_max_state_int - 1 }
          {
            \__debug_trace:nnx { regex } {#1}
              { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
          }
      }
  }
  { }
```

(*End definition for* `\__regex_trace_states:n`*.*)

```
⟨/initex | package⟩
```

37 l3box implementation

```
⟨*initex | package⟩
⟨@@=box⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

37.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

`\box_new:N`
`\box_new:c` Defining a new ⟨*box*⟩ register: remember that box 255 is not generally available.

```
⟨*package⟩
\cs_new_protected:Npn \box_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newbox \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \box_new:N { c }
```

Clear a ⟨*box*⟩ register.

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

```
\cs_new_protected:Npn \box_clear:N #1
  { \box_set_eq:NN  #1 \c_empty_box }
\cs_new_protected:Npn \box_gclear:N #1
  { \box_gset_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box_clear:N  { c }
\cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

```
\cs_new_protected:Npn \box_clear_new:N #1
  { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\cs_new_protected:Npn \box_gclear_new:N #1
  { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\cs_generate_variant:Nn \box_clear_new:N  { c }
\cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

\box_set_eq:NN
\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc
\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

```
\cs_new_protected:Npn \box_set_eq:NN #1#2
  { \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box_gset_eq:NN
  { \tex_global:D \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

Assigning the contents of a box to be another box. This clears the second box globally (that's how TEX does it).

\box_set_eq_clear:NN
\box_set_eq_clear:cN
\box_set_eq_clear:Nc
\box_set_eq_clear:cc
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc

```
\cs_new_protected:Npn \box_set_eq_clear:NN #1#2
  { \tex_setbox:D #1 \tex_box:D #2 }
\cs_new_protected:Npn \box_gset_eq_clear:NN
  { \tex_global:D  \box_set_eq_clear:NN }
\cs_generate_variant:Nn \box_set_eq_clear:NN  { c , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
```

Copies of the cs functions defined in l3basics.

\box_if_exist_p:N
\box_if_exist_p:c
\box_if_exist:NTF
\box_if_exist:cTF

```
\prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

37.2 Measuring and setting box dimensions

Accessing the height, depth, and width of a ⟨*box*⟩ register.

\box_ht:N
\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c

```
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N { c }
\cs_generate_variant:Nn \box_dp:N { c }
\cs_generate_variant:Nn \box_wd:N { c }
```

Setting the size is easy: all primitive work. These primitives are not expandable, so the derived functions are not either. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

\box_set_ht:Nn
\box_set_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn

```
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new_protected:Npn \box_set_dp:Nn #1#2
  { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new_protected:Npn \box_set_ht:Nn #1#2
  { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new_protected:Npn \box_set_wd:Nn #1#2
  { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\cs_generate_variant:Nn \box_set_ht:Nn { c }
\cs_generate_variant:Nn \box_set_dp:Nn { c }
\cs_generate_variant:Nn \box_set_wd:Nn { c }
```

37.3 Using boxes

Using a ⟨*box*⟩. These are just TEX primitives with meaningful names.

\box_use_drop:N
\box_use_drop:c
\box_use:N
\box_use:c

```
\cs_new_eq:NN \box_use_drop:N \tex_box:D
\cs_new_eq:NN \box_use:N \tex_copy:D
\cs_generate_variant:Nn \box_use_drop:N { c }
\cs_generate_variant:Nn \box_use:N { c }
```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

```
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \box_move_left:nn #1#2
  { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \box_move_right:nn #1#2
  { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \box_move_up:nn #1#2
  { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \box_move_down:nn #1#2
  { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }
```

37.4 Box conditionals

The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

\if_hbox:N
\if_vbox:N
\if_box_empty:N

```
\cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N*TF*
\box_if_horizontal:c*TF*
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N*TF*
\box_if_vertical:c*TF*

```
\prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
  { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
  { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\cs_generate_variant:Nn \box_if_horizontal:NT  { c }
\cs_generate_variant:Nn \box_if_horizontal:NF  { c }
\cs_generate_variant:Nn \box_if_horizontal:NTF { c }
\cs_generate_variant:Nn \box_if_vertical_p:N { c }
```

```
\cs_generate_variant:Nn \box_if_vertical:NT  { c }
\cs_generate_variant:Nn \box_if_vertical:NF  { c }
\cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

Testing if a ⟨*box*⟩ is empty/void.

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:N*TF*
\box_if_empty:c*TF*

```
\prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
  { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_empty_p:N { c }
\cs_generate_variant:Nn \box_if_empty:NT  { c }
\cs_generate_variant:Nn \box_if_empty:NF  { c }
\cs_generate_variant:Nn \box_if_empty:NTF { c }
```

(*End definition for* `\box_new:N` *and others. These functions are documented on page 212.*)

37.5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

Set a box to the previous box.

```
\cs_new_protected:Npn \box_set_to_last:N #1
  { \tex_setbox:D #1 \tex_lastbox:D }
\cs_new_protected:Npn \box_gset_to_last:N
  { \tex_global:D \box_set_to_last:N }
\cs_generate_variant:Nn \box_set_to_last:N  { c }
\cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(*End definition for* `\box_set_to_last:N` *and* `\box_gset_to_last:N`*. These functions are documented on page 215.*)

37.6 Constant boxes

\c_empty_box

A box we never use.

```
\box_new:N \c_empty_box
```

(*End definition for* `\c_empty_box`*. This variable is documented on page 215.*)

37.7 Scratch boxes

\l_tmpa_box
\l_tmpb_box
\g_tmpa_box
\g_tmpb_box

Scratch boxes.

```
\box_new:N \l_tmpa_box
\box_new:N \l_tmpb_box
\box_new:N \g_tmpa_box
\box_new:N \g_tmpb_box
```

(*End definition for* `\l_tmpa_box` *and others. These variables are documented on page 215.*)

37.8 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other LaTeX3 `show` functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

\box_show:N
\box_show:c
\box_show:Nnn
\box_show:cnn

Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

```
\cs_new_protected:Npn \box_show:N #1
  { \box_show:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_show:N { c }
\cs_new_protected:Npn \box_show:Nnn #1#2#3
  { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
\cs_generate_variant:Nn \box_show:Nnn { c }
```

(*End definition for* `\box_show:N` *and* `\box_show:Nnn`*. These functions are documented on page 215.*)

\box_log:N
\box_log:c
\box_log:Nnn
\box_log:cnn
__box_log:nNnn

Getting TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the ε-TeX extensions are needed.

```
\cs_new_protected:Npn \box_log:N #1
  { \box_log:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_log:N { c }
\cs_new_protected:Npn \box_log:Nnn
  { \exp_args:No \__box_log:nNnn { \tex_the:D \etex_interactionmode:D } }
\cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
  {
    \int_set:Nn \etex_interactionmode:D { 0 }
    \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
    \int_set:Nn \etex_interactionmode:D {#1}
  }
\cs_generate_variant:Nn \box_log:Nnn { c }
```

(*End definition for* `\box_log:N`, `\box_log:Nnn`, *and* `\__box_log:nNnn`*. These functions are documented on page 216.*)

__box_show:NNnn
__box_show:NNff

The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline` and `\errorcontextlines` is used to control what appears in the terminal.

```
\cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
  {
    \box_if_exist:NTF #2
      {
        \group_begin:
          \int_set:Nn \tex_showboxbreadth:D {#3}
          \int_set:Nn \tex_showboxdepth:D   {#4}
          \int_set:Nn \tex_tracingonline:D  {#1}
          \int_set:Nn \tex_errorcontextlines:D { -1 }
          \tex_showbox:D \use:n {#2}
        \group_end:
      }
      {
        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #2 }
      }
  }
\cs_generate_variant:Nn \__box_show:NNnn { NNff }
```

(*End definition for* `\__box_show:NNnn`*.*)

37.9 Horizontal mode boxes

\hbox:n

(*The test suite for this command, and others in this file, is* m3box002.lvt.)

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox:n #1
  { \tex_hbox:D \scan_stop: { \group_begin: #1 \group_end: } }
```

(*End definition for* `\hbox:n`*. This function is documented on page 216.*)

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn

```
\cs_new_protected:Npn \hbox_set:Nn #1#2
  { \tex_setbox:D #1 \tex_hbox:D { \group_begin: #2 \group_end: } }
\cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
\cs_generate_variant:Nn \hbox_set:Nn { c }
\cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(*End definition for* `\hbox_set:Nn` *and* `\hbox_gset:Nn`*. These functions are documented on page 216.*)

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } {#3} }
\cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
  {
    \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end:
      { \group_begin: #3 \group_end: }
  }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnn
  { \tex_global:D \hbox_set_to_wd:Nnn }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(*End definition for* `\hbox_set_to_wd:Nnn` *and* `\hbox_gset_to_wd:Nnn`*. These functions are documented on page 216.*)

\hbox_set:Nw
\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:

Storing material in a horizontal box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \hbox_set:Nw  #1
  {
    \tex_setbox:D #1 \tex_hbox:D
      \c_group_begin_token
        \group_begin:
  }
\cs_new_protected:Npn \hbox_gset:Nw
  { \tex_global:D \hbox_set:Nw }
\cs_generate_variant:Nn \hbox_set:Nw  { c }
\cs_generate_variant:Nn \hbox_gset:Nw { c }
\cs_new_protected:Npn \hbox_set_end:
  {
      \group_end:
    \c_group_end_token
  }
\cs_new_eq:NN \hbox_gset_end: \hbox_set_end:
```

(*End definition for* `\hbox_set:Nw` *and others. These functions are documented on page 217.*)

\hbox_set_to_wd:Nnw
\hbox_set_to_wd:cnw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cnw

Combining the above ideas.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
  {
    \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end:
      \c_group_begin_token
        \group_begin:
  }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnw
  { \tex_global:D \hbox_set_to_wd:Nnw }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnw  { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }
```

(*End definition for* `\hbox_set_to_wd:Nnw` *and* `\hbox_gset_to_wd:Nnw`*. These functions are documented on page 217.*)

\hbox_to_wd:nn
\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \hbox_to_wd:nn #1#2
   {
     \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end:
       { \group_begin: #2 \group_end: }
   }
\cs_new_protected:Npn \hbox_to_zero:n #1
  { \tex_hbox:D to \c_zero_dim { \group_begin: #1 \group_end: } }
```

(*End definition for* `\hbox_to_wd:nn` *and* `\hbox_to_zero:n`*. These functions are documented on page 216.*)

\hbox_overlap_left:n
\hbox_overlap_right:n

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```
\cs_new_protected:Npn \hbox_overlap_left:n  #1
  { \hbox_to_zero:n { \tex_hss:D #1 } }
\cs_new_protected:Npn \hbox_overlap_right:n #1
  { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(*End definition for* `\hbox_overlap_left:n` *and* `\hbox_overlap_right:n`*. These functions are documented on page 217.*)

\hbox_unpack:N
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
\cs_generate_variant:Nn \hbox_unpack:N { c }
\cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(*End definition for* `\hbox_unpack:N` *and* `\hbox_unpack_clear:N`*. These functions are documented on page 217.*)

37.10 Vertical mode boxes

TeX ends these boxes directly with the internal *end_graf* routine. This means that there is no `\par` at the end of vertical boxes unless we insert one.

\vbox:n

The following test files are used for this code: m3box003.lvt*.*

The following test files are used for this code: m3box003.lvt*.*

\vbox_top:n

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox:n #1
  { \tex_vbox:D { \group_begin: #1 \par \group_end: } }
\cs_new_protected:Npn \vbox_top:n #1
  { \tex_vtop:D { \group_begin: #1 \par \group_end: } }
```

(*End definition for* `\vbox:n` *and* `\vbox_top:n`*. These functions are documented on page 217.*)

\vbox_to_ht:nn
\vbox_to_zero:n
\vbox_to_ht:nn
\vbox_to_zero:n

Put a vertical box directly into the input stream.

```
\__debug_patch_args:nNNpn { { (#1) } {#2} }
\cs_new_protected:Npn \vbox_to_ht:nn #1#2
  {
    \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end:
      { \group_begin: #2 \par \group_end: }
  }
\cs_new_protected:Npn \vbox_to_zero:n #1
  {
    \tex_vbox:D to \c_zero_dim
      { \group_begin: #1 \par \group_end: }
  }
```

(*End definition for* `\vbox_to_ht:nn` *and others. These functions are documented on page 218.*)

\vbox_set:Nn
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

Storing material in a vertical box with a natural height.

```
\cs_new_protected:Npn \vbox_set:Nn #1#2
  {
    \tex_setbox:D #1 \tex_vbox:D
      { \group_begin: #2 \par \group_end: }
  }
\cs_new_protected:Npn \vbox_gset:Nn  { \tex_global:D \vbox_set:Nn }
\cs_generate_variant:Nn \vbox_set:Nn  { c }
\cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(*End definition for* `\vbox_set:Nn` *and* `\vbox_gset:Nn`*. These functions are documented on page 218.*)

\vbox_set_top:Nn
\vbox_set_top:cn
\vbox_gset_top:Nn
\vbox_gset_top:cn

Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
\cs_new_protected:Npn \vbox_set_top:Nn #1#2
  {
    \tex_setbox:D #1 \tex_vtop:D
      { \group_begin: #2 \par \group_end: }
  }
\cs_new_protected:Npn \vbox_gset_top:Nn
  { \tex_global:D \vbox_set_top:Nn }
\cs_generate_variant:Nn \vbox_set_top:Nn { c }
\cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(*End definition for* `\vbox_set_top:Nn` *and* `\vbox_gset_top:Nn`*. These functions are documented on page 218.*)

\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn

Storing material in a vertical box with a specified height.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } {#3} }
\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
  {
    \tex_setbox:D #1 \tex_vbox:D to \__dim_eval:w #2 \__dim_eval_end:
      { \group_begin: #3 \par \group_end: }
  }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnn
  { \tex_global:D \vbox_set_to_ht:Nnn }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnn  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(*End definition for* `\vbox_set_to_ht:Nnn` *and* `\vbox_gset_to_ht:Nnn`*. These functions are documented on page 218.*)

\vbox_set:Nw
\vbox_set:cw
\vbox_gset:Nw
\vbox_gset:cw
\vbox_set_end:
\vbox_gset_end:

Storing material in a vertical box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \vbox_set:Nw #1
  {
    \tex_setbox:D #1 \tex_vbox:D
      \c_group_begin_token
        \group_begin:
  }
\cs_new_protected:Npn \vbox_gset:Nw
  { \tex_global:D \vbox_set:Nw }
\cs_generate_variant:Nn \vbox_set:Nw  { c }
\cs_generate_variant:Nn \vbox_gset:Nw { c }
\cs_new_protected:Npn \vbox_set_end:
  {
        \par
      \group_end:
    \c_group_end_token
  }
\cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(*End definition for* `\vbox_set:Nw` *and others. These functions are documented on page 218.*)

\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw

A combination of the above ideas.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } }
\cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
  {
    \tex_setbox:D #1 \tex_vbox:D to \__dim_eval:w #2 \__dim_eval_end:
      \c_group_begin_token
        \group_begin:
  }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnw
  { \tex_global:D \vbox_set_to_ht:Nnw }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnw  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }
```

(*End definition for* `\vbox_set_to_ht:Nnw` *and* `\vbox_gset_to_ht:Nnw`*. These functions are documented on page 218.*)

\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_clear:N
\vbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
```

```
\cs_generate_variant:Nn \vbox_unpack:N { c }
\cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(*End definition for* `\vbox_unpack:N` *and* `\vbox_unpack_clear:N`*. These functions are documented on page 219.*)

`\vbox_set_split_to_ht:NNn`

Splitting a vertical box in two.

```
\__debug_patch_args:nNNpn { {#1} {#2} { (#3) } }
\cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
  { \tex_setbox:D #1 \tex_vsplit:D #2 to \__dim_eval:w #3 \__dim_eval_end: }
```

(*End definition for* `\vbox_set_split_to_ht:NNn`*. This function is documented on page 218.*)

37.11 Affine transformations

`\l__box_angle_fp`

When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
\fp_new:N \l__box_angle_fp
```

(*End definition for* `\l__box_angle_fp`*.*)

`\l__box_cos_fp`
`\l__box_sin_fp`

These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\fp_new:N \l__box_cos_fp
\fp_new:N \l__box_sin_fp
```

(*End definition for* `\l__box_cos_fp` *and* `\l__box_sin_fp`*.*)

`\l__box_top_dim`
`\l__box_bottom_dim`
`\l__box_left_dim`
`\l__box_right_dim`

These are the positions of the four edges of a box before manipulation.

```
\dim_new:N \l__box_top_dim
\dim_new:N \l__box_bottom_dim
\dim_new:N \l__box_left_dim
\dim_new:N \l__box_right_dim
```

(*End definition for* `\l__box_top_dim` *and others.*)

`\l__box_top_new_dim`
`\l__box_bottom_new_dim`
`\l__box_left_new_dim`
`\l__box_right_new_dim`

These are the positions of the four edges of a box after manipulation.

```
\dim_new:N \l__box_top_new_dim
\dim_new:N \l__box_bottom_new_dim
\dim_new:N \l__box_left_new_dim
\dim_new:N \l__box_right_new_dim
```

(*End definition for* `\l__box_top_new_dim` *and others.*)

`\l__box_internal_box`

Scratch space, but also needed by some parts of the driver.

```
\box_new:N \l__box_internal_box
```

(*End definition for* `\l__box_internal_box`*.*)

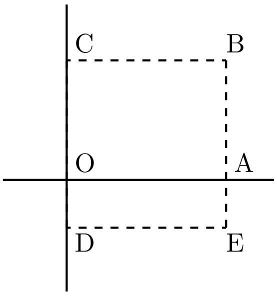

Figure 1: Co-ordinates of a box prior to rotation.

\box_rotate:Nn
__box_rotate:N
__box_rotate_x:nnN
__box_rotate_y:nnN
__box_rotate_quadrant_one:
__box_rotate_quadrant_two:
__box_rotate_quadrant_three:
__box_rotate_quadrant_four:

Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

```
\cs_new_protected:Npn \box_rotate:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \fp_set:Nn \l__box_angle_fp {#2}
        \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
        \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
        \__box_rotate:N #1
      }
  }
```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
\cs_new_protected:Npn \__box_rotate:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B, C, D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and angle α:

$$\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}$$

The "extra" horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x-coordinate 0. This is desirable as TEX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```
    \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
      {
        \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
          { \__box_rotate_quadrant_one: }
          { \__box_rotate_quadrant_two: }
      }
      {
        \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
          { \__box_rotate_quadrant_three: }
          { \__box_rotate_quadrant_four: }
      }
```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TEX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```
    \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\l__box_left_new_dim
        \hbox:n
          {
            \__driver_box_use_rotate:Nn
              \l__box_internal_box
              \l__box_angle_fp
          }
      }
```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```
    \box_set_ht:Nn \l__box_internal_box {  \l__box_top_new_dim }
    \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
    \box_set_wd:Nn \l__box_internal_box
      { \l__box_right_new_dim - \l__box_left_new_dim }
    \box_use_drop:N \l__box_internal_box
  }
```

These functions take a general point (#1, #2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```
\cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \l__box_cos_fp * \dim_to_fp:n {#1}
            - \l__box_sin_fp * \dim_to_fp:n {#2}
          }
      }
  }
\cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \l__box_sin_fp * \dim_to_fp:n {#1}
            + \l__box_cos_fp * \dim_to_fp:n {#2}
          }
      }
  }
```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y-values, whereas the left and right edges need the x-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```
\cs_new_protected:Npn \__box_rotate_quadrant_one:
  {
    \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_two:
  {
    \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_right_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_left_dim   \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_three:
  {
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_left_dim   \l__box_top_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_four:
  {
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
      \l__box_right_new_dim
  }
```

(*End definition for* `\box_rotate:Nn` *and others. These functions are documented on page 221.*)

\l__box_scale_x_fp
\l__box_scale_y_fp

Scaling is potentially-different in the two axes.

```
\fp_new:N \l__box_scale_x_fp
\fp_new:N \l__box_scale_y_fp
```

(*End definition for* `\l__box_scale_x_fp` *and* `\l__box_scale_y_fp`*.*)

\box_resize_to_wd_and_ht_plus_dp:Nnn
\box_resize_to_wd_and_ht_plus_dp:cnn
__box_resize_set_corners:N
__box_resize:N
__box_resize:NNN

Resizing a box starts by working out the various dimensions of the existing box.

```
\cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \__box_resize_set_corners:N #1
```

The x-scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
```

The y-scaling needs both the height and the depth of the current box.

```
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#3}
            / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
          }
```

Hand off to the auxiliary which does the rest of the work.

```
        \__box_resize:N #1
      }
  }
\cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \__box_resize_set_corners:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
  }
```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```
\cs_new_protected:Npn \__box_resize:N #1
  {
    \__box_resize:NNN \l__box_right_new_dim
      \l__box_scale_x_fp \l__box_right_dim
    \__box_resize:NNN \l__box_bottom_new_dim
      \l__box_scale_y_fp \l__box_bottom_dim
    \__box_resize:NNN \l__box_top_new_dim
      \l__box_scale_y_fp \l__box_top_dim
    \__box_resize_common:N #1
  }
\cs_new_protected:Npn \__box_resize:NNN #1#2#3
  {
    \dim_set:Nn #1
      { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
  }
```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` *and others. These functions are documented on page 221.)*

\box_resize_to_ht:Nn
\box_resize_to_ht:cn
\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```
\cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#2}
            / \dim_to_fp:n { \l__box_top_dim }
          }
        \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
        \__box_resize:N #1
      }
  }
\cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
\cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#2}
            / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
          }
        \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
        \__box_resize:N #1
      }
  }
\cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
\cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
        \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
        \__box_resize:N #1
      }
  }
\cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
\cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#3}
            / \dim_to_fp:n { \l__box_top_dim }
          }
        \__box_resize:N #1
      }
  }
\cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
```

(*End definition for* `\box_resize_to_ht:Nn` *and others. These functions are documented on page 220.*)

`\box_scale:Nnn`
`\box_scale:cnn`
`\__box_scale_aux:N`

When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```
\cs_new_protected:Npn \box_scale:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \fp_set:Nn \l__box_scale_x_fp {#2}
        \fp_set:Nn \l__box_scale_y_fp {#3}
        \__box_scale_aux:N #1
      }
  }
\cs_generate_variant:Nn \box_scale:Nnn { c }
\cs_new_protected:Npn \__box_scale_aux:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
    \dim_set:Nn \l__box_top_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
    \dim_set:Nn \l__box_bottom_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
    \dim_set:Nn \l__box_right_new_dim
      { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
    \__box_resize_common:N #1
  }
```

(*End definition for* `\box_scale:Nnn` *and* `\__box_scale_aux:N`*. These functions are documented on page 221.*)

\box_autosize_to_wd_and_ht:Nnn
\box_autosize_to_wd_and_ht:cnn
\box_autosize_to_wd_and_ht_plus_dp:cnn
\box_autosize_to_wd_and_ht_plus_dp:Nnn
__box_autosize:Nnnn

Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```
\cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
  { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 } }
\cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
\cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 } }
\cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \__box_autosize:Nnnn #1#2#3#4
  {
    \hbox_set:Nn #1
      {
        \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
        \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
        \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
          { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
          { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
        \__box_scale_aux:N #1
      }
  }
```

(*End definition for* \box_autosize_to_wd_and_ht:Nnn *,* \box_autosize_to_wd_and_ht_plus_dp:cnn *, and* __box_autosize:Nnnn*. These functions are documented on page 219.*)

__box_resize_common:N

The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```
\cs_new_protected:Npn \__box_resize_common:N #1
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \__driver_box_use_scale:Nnn
          #1
          \l__box_scale_x_fp
          \l__box_scale_y_fp
      }
```

The new height and depth can be applied directly.

```
    \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
      {
        \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
        \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
      }
      {
        \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
        \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
      }
```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```
    \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
      {
        \hbox_to_wd:nn { \l__box_right_new_dim }
```

```
     {
      \tex_kern:D \l__box_right_new_dim
      \box_use_drop:N \l__box_internal_box
      \tex_hss:D
     }
   }
   {
    \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
    \hbox:n
     {
      \tex_kern:D \c_zero_dim
      \box_use_drop:N \l__box_internal_box
      \tex_hss:D
     }
   }
 }
```

(*End definition for* `\__box_resize_common:N`*.*)

37.12 Deprecated functions

\box_resize:Nnn
\box_resize:cnn
\box_use_clear:N
\box_use_clear:c

```
\__debug_deprecation:nnNNpn
 { 2018-12-31 } { \box_resize_to_wd_and_ht_plus_dp:Nnn }
\cs_new_protected:Npn \box_resize:Nnn
 { \box_resize_to_wd_and_ht_plus_dp:Nnn }
\__debug_deprecation:nnNNpn
 { 2018-12-31 } { \box_resize_to_wd_and_ht_plus_dp:cnn }
\cs_new_protected:Npn \box_resize:cnn
 { \box_resize_to_wd_and_ht_plus_dp:cnn }
\__debug_deprecation:nnNNpn
 { 2018-12-31 } { \box_use_clear:N }
\cs_new_protected:Npn \box_use_clear:N { \box_use_drop:N }
\__debug_deprecation:nnNNpn
 { 2018-12-31 } { \box_use_clear:c }
\cs_new_protected:Npn \box_use_clear:c { \box_use_drop:c }
```

(*End definition for* `\box_resize:Nnn` *and* `\box_use_clear:N`*.*)

⟨/initex | package⟩

38 l3coffins Implementation

⟨*initex | package⟩

⟨@@=coffin⟩

38.1 Coffins: data structures and general variables

\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl

Scratch variables.

```
\box_new:N \l__coffin_internal_box
\dim_new:N \l__coffin_internal_dim
\tl_new:N  \l__coffin_internal_tl
```

(*End definition for* `\l__coffin_internal_box`*,* `\l__coffin_internal_dim`*, and* `\l__coffin_internal_tl`*.*)

\c__coffin_corners_prop

The “corners”; of a coffin define the real content, as opposed to the TEX bounding box. They all start off in the same place, of course.

```
\prop_new:N \c__coffin_corners_prop
\prop_put:Nnn \c__coffin_corners_prop { tl } { { 0pt } { 0pt } }
\prop_put:Nnn \c__coffin_corners_prop { tr } { { 0pt } { 0pt } }
\prop_put:Nnn \c__coffin_corners_prop { bl } { { 0pt } { 0pt } }
\prop_put:Nnn \c__coffin_corners_prop { br } { { 0pt } { 0pt } }
```

(*End definition for* `\c__coffin_corners_prop`*.*)

\c__coffin_poles_prop

Pole positions are given for horizontal, vertical and reference-point based values.

```
\prop_new:N \c__coffin_poles_prop
\tl_set:Nn \l__coffin_internal_tl { { 0pt } { 0pt } { 0pt } { 1000pt } }
\prop_put:Nno \c__coffin_poles_prop { l }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { r }  { \l__coffin_internal_tl }
\tl_set:Nn \l__coffin_internal_tl { { 0pt } { 0pt } { 1000pt } { 0pt } }
\prop_put:Nno \c__coffin_poles_prop { b }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { t }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { B }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { H }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { T }  { \l__coffin_internal_tl }
```

(*End definition for* `\c__coffin_poles_prop`*.*)

\l__coffin_slope_x_fp
\l__coffin_slope_y_fp

Used for calculations of intersections.

```
\fp_new:N \l__coffin_slope_x_fp
\fp_new:N \l__coffin_slope_y_fp
```

(*End definition for* `\l__coffin_slope_x_fp` *and* `\l__coffin_slope_y_fp`*.*)

\l__coffin_error_bool

For propagating errors so that parts of the code can work around them.

```
\bool_new:N \l__coffin_error_bool
```

(*End definition for* `\l__coffin_error_bool`*.*)

\l__coffin_offset_x_dim
\l__coffin_offset_y_dim

The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

```
\dim_new:N \l__coffin_offset_x_dim
\dim_new:N \l__coffin_offset_y_dim
```

(*End definition for* `\l__coffin_offset_x_dim` *and* `\l__coffin_offset_y_dim`*.*)

\l__coffin_pole_a_tl
\l__coffin_pole_b_tl

Needed for finding the intersection of two poles.

```
\tl_new:N \l__coffin_pole_a_tl
\tl_new:N \l__coffin_pole_b_tl
```

(*End definition for* `\l__coffin_pole_a_tl` *and* `\l__coffin_pole_b_tl`*.*)

\l__coffin_x_dim
\l__coffin_y_dim
\l__coffin_x_prime_dim
\l__coffin_y_prime_dim

For calculating intersections and so forth.

```
\dim_new:N \l__coffin_x_dim
\dim_new:N \l__coffin_y_dim
\dim_new:N \l__coffin_x_prime_dim
\dim_new:N \l__coffin_y_prime_dim
```

(*End definition for* `\l__coffin_x_dim` *and others.*)

38.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

\coffin_if_exist_p:N
\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF

Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```
\prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
          { \prg_return_true: }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \coffin_if_exist_p:N { c }
\cs_generate_variant:Nn \coffin_if_exist:NT  { c }
\cs_generate_variant:Nn \coffin_if_exist:NF  { c }
\cs_generate_variant:Nn \coffin_if_exist:NTF { c }
```

(*End definition for* `\coffin_if_exist:NTF`. *This function is documented on page 223.*)

\__coffin_if_exist:NT

Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```
\cs_new_protected:Npn \__coffin_if_exist:NT #1#2
  {
    \coffin_if_exist:NTF #1
      { #2 }
      {
        \__msg_kernel_error:nnx { kernel } { unknown-coffin }
          { \token_to_str:N #1 }
      }
  }
```

(*End definition for* `\__coffin_if_exist:NT`.)

\coffin_clear:N
\coffin_clear:c

Clearing coffins means emptying the box and resetting all of the structures.

```
\cs_new_protected:Npn \coffin_clear:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \box_clear:N #1
        \__coffin_reset_structure:N #1
      }
  }
\cs_generate_variant:Nn \coffin_clear:N { c }
```

(*End definition for* `\coffin_clear:N`. *This function is documented on page 223.*)

\coffin_new:N
\coffin_new:c

Creating a new coffin means making the underlying box and adding the data structures. These are created globally, as there is a need to avoid any strange effects if the coffin is created inside a group. This means that the usual rule about \l_... variables has to be broken. The __debug_suspend_log: and __debug_resume_log: functions prevent \prop_clear_new:c from writing useless information to the log file; however they only exist if debugging is enabled.

```
\__debug:TF
  {
    \cs_new_protected:Npn \coffin_new:N #1
      {
        \box_new:N #1
        \__debug_suspend_log:
        \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
        \prop_clear_new:c { l__coffin_poles_   \__int_value:w #1 _prop }
        \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
          \c__coffin_corners_prop
        \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
          \c__coffin_poles_prop
        \__debug_resume_log:
      }
  }
  {
    \cs_new_protected:Npn \coffin_new:N #1
      {
        \box_new:N #1
        \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
        \prop_clear_new:c { l__coffin_poles_   \__int_value:w #1 _prop }
        \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
          \c__coffin_corners_prop
        \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
          \c__coffin_poles_prop
      }
  }
\cs_generate_variant:Nn \coffin_new:N { c }
```

(*End definition for \coffin_new:N. This function is documented on page 223.*)

\hcoffin_set:Nn
\hcoffin_set:cn

Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
\cs_new_protected:Npn \hcoffin_set:Nn #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nn #1
          {
            \color_ensure_current:
            #2
          }
        \__coffin_reset_structure:N #1
        \__coffin_update_poles:N #1
        \__coffin_update_corners:N #1
      }
  }
\cs_generate_variant:Nn \hcoffin_set:Nn { c }
```

(*End definition for* \hcoffin_set:Nn. *This function is documented on page 223.*)

\vcoffin_set:Nnn
\vcoffin_set:cnn

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No \color_ensure_current: here as that would add a whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the \vtop primitive, used to do the measuring).

```
\cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \vbox_set:Nn #1
          {
            \dim_set:Nn \tex_hsize:D {#2}
⟨*package⟩
            \dim_set_eq:NN \linewidth   \tex_hsize:D
            \dim_set_eq:NN \columnwidth \tex_hsize:D
⟨/package⟩
            #3
          }
        \__coffin_reset_structure:N #1
        \__coffin_update_poles:N #1
        \__coffin_update_corners:N #1
        \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
        \__coffin_set_pole:Nnx #1 { T }
          {
            { 0pt }
            {
              \dim_eval:n
                { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
            }
            { 1000pt }
            { 0pt }
          }
        \box_clear:N \l__coffin_internal_box
      }
  }
\cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(*End definition for* \vcoffin_set:Nnn. *This function is documented on page 224.*)

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:

These are the “begin”/“end” versions of the above: watch the grouping!

```
\cs_new_protected:Npn \hcoffin_set:Nw #1
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nw #1 \color_ensure_current:
          \cs_set_protected:Npn \hcoffin_set_end:
            {
              \hbox_set_end:
              \__coffin_reset_structure:N #1
              \__coffin_update_poles:N #1
              \__coffin_update_corners:N #1
            }
      }
  }
\cs_new_protected:Npn \hcoffin_set_end: { }
\cs_generate_variant:Nn \hcoffin_set:Nw { c }
```

(*End definition for* `\hcoffin_set:Nw` *and* `\hcoffin_set_end:`*. These functions are documented on page 223.*)

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:

The same for vertical coffins.

```
\cs_new_protected:Npn \vcoffin_set:Nnw #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \vbox_set:Nw #1
          \dim_set:Nn \tex_hsize:D {#2}
⟨*package⟩
            \dim_set_eq:NN \linewidth   \tex_hsize:D
            \dim_set_eq:NN \columnwidth \tex_hsize:D
⟨/package⟩
          \cs_set_protected:Npn \vcoffin_set_end:
            {
              \vbox_set_end:
              \__coffin_reset_structure:N #1
              \__coffin_update_poles:N #1
              \__coffin_update_corners:N #1
              \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
              \__coffin_set_pole:Nnx #1 { T }
                {
                  { 0pt }
                  {
                    \dim_eval:n
                      { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
                  }
                  { 1000pt }
                  { 0pt }
                }
              \box_clear:N \l__coffin_internal_box
            }
      }
  }
\cs_new_protected:Npn \vcoffin_set_end: { }
\cs_generate_variant:Nn \vcoffin_set:Nnw { c }
```

(*End definition for* `\vcoffin_set:Nnw` *and* `\vcoffin_set_end:`*. These functions are documented on page 224.*)

\coffin_set_eq:NN
\coffin_set_eq:Nc
\coffin_set_eq:cN
\coffin_set_eq:cc

Setting two coffins equal is just a wrapper around other functions.

```
\cs_new_protected:Npn \coffin_set_eq:NN #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \box_set_eq:NN #1 #2
        \__coffin_set_eq_structure:NN #1 #2
      }
  }
\cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(*End definition for* \coffin_set_eq:NN*. This function is documented on page 223.*)

\c_empty_coffin
\l__coffin_aligned_coffin
\l__coffin_aligned_internal_coffin

Special coffins: these cannot be set up earlier as they need \coffin_new:N. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available.

```
\coffin_new:N \c_empty_coffin
\hbox_set:Nn  \c_empty_coffin { }
\coffin_new:N \l__coffin_aligned_coffin
\coffin_new:N \l__coffin_aligned_internal_coffin
```

(*End definition for* \c_empty_coffin*,* \l__coffin_aligned_coffin*, and* \l__coffin_aligned_internal_coffin*. These variables are documented on page 226.*)

\l_tmpa_coffin
\l_tmpb_coffin

The usual scratch space.

```
\coffin_new:N \l_tmpa_coffin
\coffin_new:N \l_tmpb_coffin
```

(*End definition for* \l_tmpa_coffin *and* \l_tmpb_coffin*. These variables are documented on page 226.*)

38.3 Measuring coffins

\coffin_dp:N
\coffin_dp:c
\coffin_ht:N
\coffin_ht:c
\coffin_wd:N
\coffin_wd:c

Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```
\cs_new_eq:NN \coffin_dp:N \box_dp:N
\cs_new_eq:NN \coffin_dp:c \box_dp:c
\cs_new_eq:NN \coffin_ht:N \box_ht:N
\cs_new_eq:NN \coffin_ht:c \box_ht:c
\cs_new_eq:NN \coffin_wd:N \box_wd:N
\cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(*End definition for* \coffin_dp:N*,* \coffin_ht:N*, and* \coffin_wd:N*. These functions are documented on page 225.*)

38.4 Coffins: handle and pole management

__coffin_get_pole:NnN

A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
\cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
  {
    \prop_get:cnNF
      { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
      {
        \__msg_kernel_error:nnxx { kernel } { unknown-coffin-pole }
          {#2} { \token_to_str:N #1 }
        \tl_set:Nn #3 { { 0pt } { 0pt } { 0pt } { 0pt } }
      }
  }
```

(*End definition for* __coffin_get_pole:NnN*.*)

__coffin_reset_structure:N

Resetting the structure is a simple copy job.

```
\cs_new_protected:Npn \__coffin_reset_structure:N #1
  {
    \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
      \c__coffin_corners_prop
    \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
      \c__coffin_poles_prop
  }
```

(*End definition for __coffin_reset_structure:N.*)

__coffin_set_eq_structure:NN
__coffin_gset_eq_structure:NN

Setting coffin structures equal simply means copying the property list.

```
\cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
  {
    \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
      { l__coffin_corners_ \__int_value:w #2 _prop }
    \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
      { l__coffin_poles_ \__int_value:w #2 _prop }
  }
\cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
  {
    \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
      { l__coffin_corners_ \__int_value:w #2 _prop }
    \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
      { l__coffin_poles_ \__int_value:w #2 _prop }
  }
```

(*End definition for __coffin_set_eq_structure:NN and __coffin_gset_eq_structure:NN.*)

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
__coffin_set_pole:Nnn
__coffin_set_pole:Nnx

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```
\cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \__coffin_set_pole:Nnx #1 {#2}
          {
            { 0pt } { \dim_eval:n {#3} }
            { 1000pt } { 0pt }
          }
      }
  }
\cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \__coffin_set_pole:Nnx #1 {#2}
          {
            { \dim_eval:n {#3} } { 0pt }
            { 0pt } { 1000pt }
          }
      }
  }
```

```
\cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
  { \prop_put:cnn { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
\cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
\cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
\cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }
```

(*End definition for `\coffin_set_horizontal_pole:Nnn`, `\coffin_set_vertical_pole:Nnn`, and `\__coffin_set_pole:Nnn`. These functions are documented on page 224.*)

__coffin_update_corners:N

Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TEX box.

```
\cs_new_protected:Npn \__coffin_update_corners:N #1
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
      { { 0pt } { \dim_eval:n { \box_ht:N #1 } } }
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
      { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
      { { 0pt } { \dim_eval:n { -\box_dp:N #1 } } }
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
      { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { -\box_dp:N #1 } } }
  }
```

(*End definition for `\__coffin_update_corners:N`.*)

__coffin_update_poles:N

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```
\cs_new_protected:Npn \__coffin_update_poles:N #1
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
      {
        { \dim_eval:n { 0.5 \box_wd:N #1 } }
        { 0pt } { 0pt } { 1000pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
      {
        { \dim_eval:n { \box_wd:N #1 } }
        { 0pt } { 0pt } { 1000pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
      {
        { 0pt }
        { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
        { 1000pt }
        { 0pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
      {
        { 0pt }
        { \dim_eval:n { \box_ht:N #1 } }
        { 1000pt }
        { 0pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
      {
        { 0pt }
        { \dim_eval:n { -\box_dp:N #1 } }
        { 1000pt }
        { 0pt }
      }
  }
```

(*End definition for* `\__coffin_update_poles:N`.)

38.5 Coffins: calculation of pole intersections

`\__coffin_calculate_intersection:Nnn`
`\__coffin_calculate_intersection:nnnnnnnn`
`\__coffin_calculate_intersection_aux:nnnnnN`

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
  {
    \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
    \bool_set_false:N \l__coffin_error_bool
    \exp_last_two_unbraced:Noo
      \__coffin_calculate_intersection:nnnnnnnn
        \l__coffin_pole_a_tl \l__coffin_pole_b_tl
    \bool_if:NT \l__coffin_error_bool
      {
        \__msg_kernel_error:nn { kernel } { no-pole-intersection }
        \dim_zero:N \l__coffin_x_dim
        \dim_zero:N \l__coffin_y_dim
      }
  }
```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d'). These are arguments 1–4 and 5–8, respectively. In both cases a and b are the co-ordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c'. However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
  #1#2#3#4#5#6#7#8
  {
    \dim_compare:nNnTF {#3} = { \c_zero_dim }
```

The case where the first pole is vertical. So the x-component of the interaction is at a. There is then a test on the second pole: if it is also vertical then there is an error.

```
      {
        \dim_set:Nn \l__coffin_x_dim {#1}
        \dim_compare:nNnTF {#7} = \c_zero_dim
          { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y-component of the intersection is b'. If not,

$$y = \frac{d'}{c'}(x - a') + b'$$

with the x-component already known to be #1. This calculation is done as a generalised auxiliary.

```
          {
            \dim_compare:nNnTF {#8} = \c_zero_dim
              { \dim_set:Nn \l__coffin_y_dim {#6} }
              {
                \__coffin_calculate_intersection_aux:nnnnnN
                  {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
              }
          }
      }
```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x- and y-components interchanged.

```
      {
        \dim_compare:nNnTF {#4} = \c_zero_dim
          {
            \dim_set:Nn \l__coffin_y_dim {#2}
            \dim_compare:nNnTF {#8} = { \c_zero_dim }
              { \bool_set_true:N \l__coffin_error_bool }
              {
                \dim_compare:nNnTF {#7} = \c_zero_dim
                  { \dim_set:Nn \l__coffin_x_dim {#5} }
```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'}\,(y - b') + a'$$

which is again handled by the same auxiliary.

```
                  {
                    \__coffin_calculate_intersection_aux:nnnnnN
                      {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
                  }
              }
          }
```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```
          {
            \dim_compare:nNnTF {#7} = \c_zero_dim
              {
                \dim_set:Nn \l__coffin_x_dim {#5}
                \__coffin_calculate_intersection_aux:nnnnnN
                  {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
              }
              {
                \dim_compare:nNnTF {#8} = \c_zero_dim
                  {
                    \dim_set:Nn \l__coffin_y_dim {#6}
                    \__coffin_calculate_intersection_aux:nnnnnN
                      {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
                  }
```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```
                  {
                    \fp_set:Nn \l__coffin_slope_x_fp
                      { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
                    \fp_set:Nn \l__coffin_slope_y_fp
                      { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
                    \fp_compare:nNnTF
                      \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
                      { \bool_set_true:N \l__coffin_error_bool }
```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y-values is then worked out using the standard auxiliary starting from the x-position.

```
                      {
                        \dim_set:Nn \l__coffin_x_dim
                          {
                            \fp_to_dim:n
                              {
                                (
                                      \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
                                  - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
                                  -   \dim_to_fp:n {#2}
                                  +   \dim_to_fp:n {#6}
                                )
                                /
                                ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
                            }
                          }
                        \__coffin_calculate_intersection_aux:nnnnnN
                          { \l__coffin_x_dim }
                          {#5} {#6} {#8} {#7} \l__coffin_y_dim
                      }
                  }
              }
          }
      }
  }
```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left(\frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```
\cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN
    #1#2#3#4#5#6
  {
    \dim_set:Nn #6
```

```
        {
          \fp_to_dim:n
            {
              \dim_to_fp:n {#4} *
              ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
              \dim_to_fp:n {#5}
              + \dim_to_fp:n {#3}
            }
        }
    }
```

(*End definition for* `\__coffin_calculate_intersection:Nnn`, `\__coffin_calculate_intersection:nnnnnnnn`, *and* `\__coffin_calculate_intersection_aux:nnnnnN`.)

38.6 Aligning and typesetting of coffins

\coffin_join:NnnNnnnn
\coffin_join:cnnNnnnn
\coffin_join:Nnncnnnn
\coffin_join:cnncnnnn

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```
\cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
```

Correct the placement of the reference point. If the x-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x-offset and the width of the second box. So a second kern may be needed.

```
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
          { \tex_kern:D -\l__coffin_offset_x_dim }
        \hbox_unpack:N \l__coffin_aligned_coffin
        \dim_set:Nn \l__coffin_internal_dim
          { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
        \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
          { \tex_kern:D -\l__coffin_internal_dim }
      }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
   \__coffin_reset_structure:N \l__coffin_aligned_coffin
   \prop_clear:c
     { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
   \__coffin_update_poles:N \l__coffin_aligned_coffin
```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```
    \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
      {
        \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt }
        \__coffin_offset_poles:Nnn #4 { 0pt } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt }
        \__coffin_offset_corners:Nnn #4 { 0pt } { \l__coffin_offset_y_dim }
       }
       {
        \__coffin_offset_poles:Nnn #1 { 0pt } { 0pt }
        \__coffin_offset_poles:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { 0pt } { 0pt }
        \__coffin_offset_corners:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
       }
    \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
    \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
   }
\cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_join:NnnNnnnn`*. This function is documented on page 225.*)

`\coffin_attach:NnnNnnnn`
`\coffin_attach:cnnNnnnn`
`\coffin_attach:Nnncnnnn`
`\coffin_attach:cnncnnnn`
`\coffin_attach_mark:NnnNnnnn`

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```
\cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
 {
  \__coffin_align:NnnNnnnnN
   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
  \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
  \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
  \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
  \__coffin_reset_structure:N \l__coffin_aligned_coffin
  \prop_set_eq:cc
   { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
   { l__coffin_corners_ \__int_value:w #1 _prop }
  \__coffin_update_poles:N  \l__coffin_aligned_coffin
  \__coffin_offset_poles:Nnn #1 { 0pt } { 0pt }
  \__coffin_offset_poles:Nnn #4
   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
  \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
  \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
 }
\cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
 {
  \__coffin_align:NnnNnnnnN
   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
  \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
  \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
  \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
  \box_set_eq:NN #1 \l__coffin_aligned_coffin
 }
\cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_attach:NnnNnnnn` *and* `\coffin_attach_mark:NnnNnnnn`*. These functions are documented on page 224.*)

__coffin_align:NnnNnnnnN

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the 'primed' storage area to be used for the second coffin. The 'real' box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```
\cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_offset_x_dim
      { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
    \hbox_set:Nn \l__coffin_aligned_internal_coffin
      {
        \box_use:N #1
        \tex_kern:D -\box_wd:N #1
        \tex_kern:D \l__coffin_offset_x_dim
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
      }
    \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
  }
```

(*End definition for* `\__coffin_align:NnnNnnnnN`.)

__coffin_offset_poles:Nnn
__coffin_offset_pole:Nnnnnnn

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```
\cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
  {
    \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
    \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
    \tl_if_in:nnTF {#2} { - }
      { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
      { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
    \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
      { \l__coffin_internal_tl }
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.*)

__coffin_offset_corners:Nnn
__coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```
\cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx
      { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
      { #1 - #2 }
      {
        { \dim_eval:n { #3 + #5 } }
        { \dim_eval:n { #4 + #6 } }
      }
  }
```

(*End definition for __coffin_offset_corners:Nnn and __coffin_offset_corner:Nnnnn.*)

__coffin_update_vertical_poles:NNN
__coffin_update_T:nnnnnnnnN
__coffin_update_B:nnnnnnnnN

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```
\cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
  {
    \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
    \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
  }
\cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \__coffin_set_pole:Nnx #9 { T }
          { { 0pt } {#6} { 1000pt } { 0pt } }
      }
      {
        \__coffin_set_pole:Nnx #9 { T }
          { { 0pt } {#2} { 1000pt } { 0pt } }
      }
  }
\cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
```

```
      {
        \__coffin_set_pole:Nnx #9 { B }
          { { 0pt } {#2}  { 1000pt } { 0pt } }
      }
      {
        \__coffin_set_pole:Nnx #9 { B }
          { { 0pt } {#6} { 1000pt } { 0pt } }
      }
  }
```

(*End definition for* `\__coffin_update_vertical_poles:NNN`, `\__coffin_update_T:nnnnnnnnN`, *and* `\__coffin_update_B:nnnnnnnnN`.)

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
\cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
  {
    \mode_leave_vertical:
    \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { l }
      #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
    \box_use_drop:N \l__coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(*End definition for* `\coffin_typeset:Nnnnn`. *This function is documented on page 225.*)

38.7 Coffin diagnostics

`\l__coffin_display_coffin`
`\l__coffin_display_coord_coffin`
`\l__coffin_display_pole_coffin`

Used for printing coffins with data structures attached.

```
\coffin_new:N \l__coffin_display_coffin
\coffin_new:N \l__coffin_display_coord_coffin
\coffin_new:N \l__coffin_display_pole_coffin
```

(*End definition for* `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, *and* `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop`

This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
\prop_new:N \l__coffin_display_handles_prop
\prop_put:Nnn \l__coffin_display_handles_prop { tl }
  { { b } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { thc }
  { { b } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { tr }
  { { b } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcl }
  { { vc } { r } { -1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vchc }
  { { vc } { hc } { 0 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcr }
  { { vc } { l } { 1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bl }
  { { t } { r } { -1 } { -1 } }
```

```
\prop_put:Nnn \l__coffin_display_handles_prop { bhc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { br }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Thc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tr }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hl }
  { { vc } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
  { { vc } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hr }
  { { vc } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bl }
  { { b } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
  { { b } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Br }
  { { b } { l } { 1 } { -1 } }
```

(*End definition for* `\l__coffin_display_handles_prop`.)

`\l__coffin_display_offset_dim`

The standard offset for the label from the handle position when displaying handles.

```
\dim_new:N  \l__coffin_display_offset_dim
\dim_set:Nn \l__coffin_display_offset_dim { 2pt }
```

(*End definition for* `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim`
`\l__coffin_display_y_dim`

As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```
\dim_new:N \l__coffin_display_x_dim
\dim_new:N \l__coffin_display_y_dim
```

(*End definition for* `\l__coffin_display_x_dim` *and* `\l__coffin_display_y_dim`.)

`\l__coffin_display_poles_prop`

A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```
\prop_new:N \l__coffin_display_poles_prop
```

(*End definition for* `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_tl`

Stores the settings used to print coffin data: this keeps things flexible.

```
\tl_new:N  \l__coffin_display_font_tl
⟨*initex⟩
\tl_set:Nn \l__coffin_display_font_tl { } % TODO
⟨/initex⟩
⟨*package⟩
\tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }
⟨/package⟩
```

(*End definition for* `\l__coffin_display_font_tl`.)

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
__coffin_mark_handle_aux:nnnnNnn

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
\cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
        \rule { 1pt } { 1pt }
⟨/package⟩
      }
    \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
      \l__coffin_display_pole_coffin { hc } { vc } { 0pt } { 0pt }
    \hcoffin_set:Nn \l__coffin_display_coord_coffin
      {
⟨*initex⟩
        % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
⟨/package⟩
        \l__coffin_display_font_tl
        ( \tl_to_str:n { #2 , #3 } )
      }
    \prop_get:NnN \l__coffin_display_handles_prop
      { #2 #3 } \l__coffin_internal_tl
    \quark_if_no_value:NTF \l__coffin_internal_tl
      {
        \prop_get:NnN \l__coffin_display_handles_prop
          { #3 #2 } \l__coffin_internal_tl
        \quark_if_no_value:NTF \l__coffin_internal_tl
          {
            \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
              \l__coffin_display_coord_coffin { l } { vc }
                { 1pt } { 0pt }
          }
          {
            \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
              \l__coffin_internal_tl #1 {#2} {#3}
          }
      }
      {
        \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
          \l__coffin_internal_tl #1 {#2} {#3}
      }
  }
\cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
  {
    \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
```

```
      \l__coffin_display_coord_coffin {#1} {#2}
      { #3 \l__coffin_display_offset_dim }
      { #4 \l__coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(*End definition for* `\coffin_mark_handle:Nnnn` *and* `\__coffin_mark_handle_aux:nnnnNnn`*. These functions are documented on page 226.*)

\coffin_display_handles:Nn
\coffin_display_handles:cn
__coffin_display_handles_aux:nnnnnn
__coffin_display_handles_aux:nnnn
__coffin_display_attach:Nnnnn

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```
\cs_new_protected:Npn \coffin_display_handles:Nn #1#2
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#2}
        \rule { 1pt } { 1pt }
⟨/package⟩
      }
    \prop_set_eq:Nc \l__coffin_display_poles_prop
      { l__coffin_poles_ \__int_value:w #1 _prop }
    \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
      { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
    \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
      { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
    \coffin_set_eq:NN \l__coffin_display_coffin #1
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \prop_remove:Nn \l__coffin_display_poles_prop {##1}
        \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
      }
    \box_use_drop:N \l__coffin_display_coffin
  }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
  {
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \bool_set_false:N \l__coffin_error_bool
        \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
        \bool_if:NF \l__coffin_error_bool
          {
```

```
      \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
      \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
      \__coffin_display_attach:Nnnnn
       \l__coffin_display_pole_coffin { hc } { vc }
       { 0pt } { 0pt }
      \hcoffin_set:Nn \l__coffin_display_coord_coffin
       {
⟨*initex⟩
        % TODO
⟨/initex⟩
⟨*package⟩
        \color {#6}
⟨/package⟩
        \l__coffin_display_font_tl
        ( \tl_to_str:n { #1 , ##1 } )
       }
      \prop_get:NnN \l__coffin_display_handles_prop
       { #1 ##1 } \l__coffin_internal_tl
      \quark_if_no_value:NTF \l__coffin_internal_tl
       {
        \prop_get:NnN \l__coffin_display_handles_prop
         { ##1 #1 } \l__coffin_internal_tl
        \quark_if_no_value:NTF \l__coffin_internal_tl
         {
          \__coffin_display_attach:Nnnnn
           \l__coffin_display_coord_coffin { l } { vc }
           { 1pt } { 0pt }
         }
         {
          \exp_last_unbraced:No
           \__coffin_display_handles_aux:nnnn
           \l__coffin_internal_tl
         }
       }
       {
        \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
         \l__coffin_internal_tl
       }
     }
   }
 }
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
 {
  \__coffin_display_attach:Nnnnn
   \l__coffin_display_coord_coffin {#1} {#2}
   { #3 \l__coffin_display_offset_dim }
   { #4 \l__coffin_display_offset_dim }
 }
\cs_generate_variant:Nn \coffin_display_handles:Nn { c }
```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```
\cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
```

```
  {
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \dim_set:Nn \l__coffin_offset_x_dim
      { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \box_use:N \l__coffin_display_coffin
        \tex_kern:D -\box_wd:N \l__coffin_display_coffin
        \tex_kern:D \l__coffin_offset_x_dim
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
      }
    \box_set_ht:Nn \l__coffin_aligned_coffin
      { \box_ht:N \l__coffin_display_coffin }
    \box_set_dp:Nn \l__coffin_aligned_coffin
      { \box_dp:N \l__coffin_display_coffin }
    \box_set_wd:Nn \l__coffin_aligned_coffin
      { \box_wd:N \l__coffin_display_coffin }
    \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
  }
```

(*End definition for* `\coffin_display_handles:Nn` *and others. These functions are documented on page 225.*)

`\coffin_show_structure:N`
`\coffin_show_structure:c`

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```
\cs_new_protected:Npn \coffin_show_structure:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-coffin }
          { \token_to_str:N #1 }
          { \dim_eval:n { \coffin_ht:N #1 } }
          { \dim_eval:n { \coffin_dp:N #1 } }
          { \dim_eval:n { \coffin_wd:N #1 } }
        \__msg_show_wrap:n
          {
            \prop_map_function:cN
              { l__coffin_poles_ \__int_value:w #1 _prop }
              \__msg_show_item_unbraced:nn
          }
      }
  }
\cs_generate_variant:Nn \coffin_show_structure:N { c }
```

(*End definition for* `\coffin_show_structure:N`*. This function is documented on page 226.*)

`\coffin_log_structure:N`
`\coffin_log_structure:c`

Redirect output of `\coffin_show_structure:N` to the log.

```
\cs_new_protected:Npn \coffin_log_structure:N
  { \__msg_log_next: \coffin_show_structure:N }
\cs_generate_variant:Nn \coffin_log_structure:N { c }
```

(*End definition for* `\coffin_log_structure:N`*. This function is documented on page 226.*)

38.8 Messages

```
\__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
  { No~intersection~between~coffin~poles. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
    but~they~do~not~have~a~unique~meeting~point:~
    the~value~(0~pt,~0~pt)~will~be~used.
  }
\__msg_kernel_new:nnnn { kernel } { unknown-coffin }
  { Unknown~coffin~'#1'. }
  { The~coffin~'#1'~was~never~defined. }
\__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
  { Pole~'#1'~unknown~for~coffin~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
    but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
  }
\__msg_kernel_new:nnn { kernel } { show-coffin }
  {
    Size~of~coffin~#1 : \\
    > ~ ht~=~#2 \\
    > ~ dp~=~#3 \\
    > ~ wd~=~#4 \\
    Poles~of~coffin~#1 :
  }
```

```
⟨/initex | package⟩
```

39 l3color Implementation

```
⟨*initex | package⟩
```

\color_group_begin:
\color_group_end:

Grouping for color is almost the same as using the basic \group_begin: and \group_end: functions. However, in vertical mode the end-of-group needs a \par, which in horizontal mode does nothing.

```
\cs_new_eq:NN \color_group_begin: \group_begin:
\cs_new_protected:Npn \color_group_end:
  {
      \par
    \group_end:
  }
```

(*End definition for* \color_group_begin: *and* \color_group_end:. *These functions are documented on page 227.*)

\color_ensure_current:

A driver-independent wrapper for setting the foreground color to the current color "now".

```
\cs_new_protected:Npn \color_ensure_current:
  {
    \__driver_color_ensure_current:
    \group_insert_after:N \__driver_color_reset:
  }
```

(*End definition for* \color_ensure_current:. *This function is documented on page 227.*)

\l__color_current_tl

The current color: the format here is taken from dvips but it is easy enough to convert to pdfmode as required.

```
\tl_new:N \l__color_current_tl
\tl_set:Nn \l__color_current_tl { gray~0 }
```

(*End definition for* \l__color_current_tl.)

```
⟨/initex | package⟩
```

40 l3sys implementation

```
⟨*initex | package⟩
```

40.1 The name of the job

\c_sys_jobname_str

Inherited from the LaTeX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
  }
⟨/initex⟩
⟨*package⟩
\str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
⟨/package⟩
```

(*End definition for* \c_sys_jobname_str. *This variable is documented on page 228.*)

40.2 Time and date

\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int

Copies of the information provided by TeX

```
\int_const:Nn \c_sys_minute_int
  { \int_mod:nn { \tex_time:D } { 60 } }
\int_const:Nn \c_sys_hour_int
  { \int_div_truncate:nn { \tex_time:D } { 60 } }
\int_const:Nn \c_sys_day_int    { \tex_day:D }
\int_const:Nn \c_sys_month_int { \tex_month:D }
\int_const:Nn \c_sys_year_int   { \tex_year:D }
```

(*End definition for* \c_sys_minute_int *and others. These variables are documented on page 228.*)

40.3 Detecting the engine

\sys_if_engine_luatex_p:
\sys_if_engine_luatex:*TF*
\sys_if_engine_pdftex_p:
\sys_if_engine_pdftex:*TF*
\sys_if_engine_ptex_p:
\sys_if_engine_ptex:*TF*
\sys_if_engine_uptex_p:
\sys_if_engine_uptex:*TF*
\sys_if_engine_xetex_p:
\sys_if_engine_xetex:*TF*
\c_sys_engine_str

Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive. For upTeX, there is a complexity in that setting -kanji-internal=sjis or -kanji-internal=euc effective makes it more like pTeX. In those cases we therefore report pTeX rather than upTeX.

```
\clist_map_inline:nn { lua , pdf , p , up , xe }
  {
    \cs_new_eq:cN { sys_if_engine_ #1 tex:T }  \use_none:n
    \cs_new_eq:cN { sys_if_engine_ #1 tex:F }  \use:n
    \cs_new_eq:cN { sys_if_engine_ #1 tex:TF } \use_ii:nn
    \cs_new_eq:cN { sys_if_engine_ #1 tex_p: } \c_false_bool
  }
\cs_if_exist:NT \luatex_luatexversion:D
  {
    \cs_gset_eq:NN \sys_if_engine_luatex:T  \use:n
    \cs_gset_eq:NN \sys_if_engine_luatex:F  \use_none:n
    \cs_gset_eq:NN \sys_if_engine_luatex:TF \use_i:nn
    \cs_gset_eq:NN \sys_if_engine_luatex_p: \c_true_bool
    \str_const:Nn \c_sys_engine_str { luatex }
  }
\cs_if_exist:NT \pdftex_pdftexversion:D
  {
    \cs_gset_eq:NN \sys_if_engine_pdftex:T  \use:n
    \cs_gset_eq:NN \sys_if_engine_pdftex:F  \use_none:n
    \cs_gset_eq:NN \sys_if_engine_pdftex:TF \use_i:nn
    \cs_gset_eq:NN \sys_if_engine_pdftex_p: \c_true_bool
    \str_const:Nn \c_sys_engine_str { pdftex }
  }
\cs_if_exist:NT \ptex_kanjiskip:D
  {
    \bool_lazy_and:nnTF
      { \cs_if_exist_p:N \uptex_disablecjktoken:D }
      { \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 } }
      {
        \cs_gset_eq:NN \sys_if_engine_uptex:T  \use:n
        \cs_gset_eq:NN \sys_if_engine_uptex:F  \use_none:n
        \cs_gset_eq:NN \sys_if_engine_uptex:TF \use_i:nn
        \cs_gset_eq:NN \sys_if_engine_uptex_p: \c_true_bool
        \str_const:Nn \c_sys_engine_str { uptex }
      }
      {
        \cs_gset_eq:NN \sys_if_engine_ptex:T  \use:n
        \cs_gset_eq:NN \sys_if_engine_ptex:F  \use_none:n
        \cs_gset_eq:NN \sys_if_engine_ptex:TF \use_i:nn
        \cs_gset_eq:NN \sys_if_engine_ptex_p: \c_true_bool
        \str_const:Nn \c_sys_engine_str { ptex }
      }
  }
\cs_if_exist:NT \xetex_XeTeXversion:D
  {
    \cs_gset_eq:NN \sys_if_engine_xetex:T  \use:n
    \cs_gset_eq:NN \sys_if_engine_xetex:F  \use_none:n
    \cs_gset_eq:NN \sys_if_engine_xetex:TF \use_i:nn
    \cs_gset_eq:NN \sys_if_engine_xetex_p: \c_true_bool
    \str_const:Nn \c_sys_engine_str { xetex }
  }
```

(*End definition for* `\sys_if_engine_luatex:TF` *and others. These functions are documented on page 228.*)

40.4 Detecting the output

\sys_if_output_dvi_p:
\sys_if_output_dvi:*TF*
\sys_if_output_pdf_p:
\sys_if_output_pdf:*TF*
\c_sys_output_str

This is a simple enough concept: the two views here are complementary.

```
\int_compare:nNnTF
  { \cs_if_exist_use:NF \pdftex_pdfoutput:D { 0 } } > { 0 }
  {
    \cs_new_eq:NN \sys_if_output_dvi:T  \use_none:n
    \cs_new_eq:NN \sys_if_output_dvi:F  \use:n
    \cs_new_eq:NN \sys_if_output_dvi:TF \use_ii:nn
    \cs_new_eq:NN \sys_if_output_dvi_p: \c_false_bool
    \cs_new_eq:NN \sys_if_output_pdf:T  \use:n
    \cs_new_eq:NN \sys_if_output_pdf:F  \use_none:n
    \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn
    \cs_new_eq:NN \sys_if_output_pdf_p: \c_true_bool
    \str_const:Nn \c_sys_output_str { pdf }
  }
  {
    \cs_new_eq:NN \sys_if_output_dvi:T  \use:n
    \cs_new_eq:NN \sys_if_output_dvi:F  \use_none:n
    \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
    \cs_new_eq:NN \sys_if_output_dvi_p: \c_true_bool
    \cs_new_eq:NN \sys_if_output_pdf:T  \use_none:n
    \cs_new_eq:NN \sys_if_output_pdf:F  \use:n
    \cs_new_eq:NN \sys_if_output_pdf:TF \use_ii:nn
    \cs_new_eq:NN \sys_if_output_pdf_p: \c_false_bool
    \str_const:Nn \c_sys_output_str { dvi }
  }
```

(*End definition for* `\sys_if_output_dvi:TF`*,* `\sys_if_output_pdf:TF`*, and* `\c_sys_output_str`*. These functions are documented on page 229.*)

```
⟨/initex | package⟩
```

41 l3deprecation implementation

```
⟨*initex | package⟩
⟨@@=deprecation⟩
```

__deprecation_error:Nnn

The `\outer` definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```
\cs_new_protected:Npn \__deprecation_error:Nnn #1#2#3
  {
    \etex_protected:D \tex_outer:D \tex_edef:D #1
      {
        \exp_not:N \__msg_kernel_expandable_error:nnnnn
          { kernel } { deprecated-command }
          { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
        \exp_not:N \__msg_kernel_error:nnxxx
          { kernel } { deprecated-command }
          { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
      }
  }
\__deprecation_error:Nnn \c_job_name_tl { \c_sys_jobname_str } { 2017-01-01 }
\__deprecation_error:Nnn \dim_case:nnn { \dim_case:nnF } { 2015-07-14 }
\__deprecation_error:Nnn \int_case:nnn { \int_case:nnF } { 2015-07-14 }
\__deprecation_error:Nnn \int_from_binary:n { \int_from_bin:n } { 2016-01-05 }
\__deprecation_error:Nnn \int_from_hexadecimal:n { \int_from_hex:n } { 2016-01-05 }
```

```
\__deprecation_error:Nnn \int_from_octal:n { \int_from_oct:n } { 2016-01-05 }
\__deprecation_error:Nnn \int_to_binary:n { \int_to_bin:n } { 2016-01-05 }
\__deprecation_error:Nnn \int_to_hexadecimal:n { \int_to_hex:n } { 2016-01-05 }
\__deprecation_error:Nnn \int_to_octal:n { \int_to_oct:n } { 2016-01-05 }
\__deprecation_error:Nnn \luatex_if_engine_p: { \sys_if_engine_luatex_p: } { 2017-01-01 }
\__deprecation_error:Nnn \luatex_if_engine:F { \sys_if_engine_luatex:F } { 2017-01-01 }
\__deprecation_error:Nnn \luatex_if_engine:T { \sys_if_engine_luatex:T } { 2017-01-01 }
\__deprecation_error:Nnn \luatex_if_engine:TF { \sys_if_engine_luatex:TF } { 2017-01-01 }
\__deprecation_error:Nnn \pdftex_if_engine_p: { \sys_if_engine_pdftex_p: } { 2017-01-01 }
\__deprecation_error:Nnn \pdftex_if_engine:F { \sys_if_engine_pdftex:F } { 2017-01-01 }
\__deprecation_error:Nnn \pdftex_if_engine:T { \sys_if_engine_pdftex:T } { 2017-01-01 }
\__deprecation_error:Nnn \pdftex_if_engine:TF { \sys_if_engine_pdftex:TF } { 2017-01-01 }
\__deprecation_error:Nnn \prop_get:cn { \prop_item:cn } { 2016-01-05 }
\__deprecation_error:Nnn \prop_get:Nn { \prop_item:Nn } { 2016-01-05 }
\__deprecation_error:Nnn \quark_if_recursion_tail_break:N { } { 2015-07-14 }
\__deprecation_error:Nnn \quark_if_recursion_tail_break:n { }{ 2015-07-14 }
\__deprecation_error:Nnn \scan_align_safe_stop: { protected~commands } { 2017-01-01 }
\__deprecation_error:Nnn \str_case:nnn { \str_case:nnF } { 2015-07-14 }
\__deprecation_error:Nnn \str_case:onn { \str_case:onF } { 2015-07-14 }
\__deprecation_error:Nnn \str_case_x:nnn { \str_case_x:nnF } { 2015-07-14 }
\__deprecation_error:Nnn \tl_case:cnn { \tl_case:cnF } { 2015-07-14 }
\__deprecation_error:Nnn \tl_case:Nnn { \tl_case:NnF } { 2015-07-14 }
\__deprecation_error:Nnn \xetex_if_engine_p: { \sys_if_engine_xetex_p: } { 2017-01-01 }
\__deprecation_error:Nnn \xetex_if_engine:F { \sys_if_engine_xetex:F } { 2017-01-01 }
\__deprecation_error:Nnn \xetex_if_engine:T { \sys_if_engine_xetex:T } { 2017-01-01 }
\__deprecation_error:Nnn \xetex_if_engine:TF { \sys_if_engine_xetex:TF } { 2017-01-01 }
```

(*End definition for* `\__deprecation_error:Nnn`.)

```
⟨/initex | package⟩
```

42 l3candidates Implementation

```
⟨*initex | package⟩
```

42.1 Additions to l3basics

\mode_leave_vertical: The approach here is different to that used by LaTeX2ε or plain TeX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the reinserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the LaTeX2ε version, the availability of ε-TeX means using a mode test can be done at for example the start of an `\halign`. The `\quitvmode` primitive essentially wraps the same code up at the engine level.

```
\cs_new_protected:Npx \mode_leave_vertical:
  {
    \cs_if_exist:NTF \pdftex_quitvmode:D
      { \pdftex_quitvmode:D }
      {
        \exp_not:n
          {
            \if_mode_vertical:
              \exp_after:wN \tex_indent:D
          \fi:
        }
    }
  }
```

(*End definition for* `\mode_leave_vertical:`. *This function is documented on page 232.*)

42.2 Additions to l3box

```
⟨@@=box⟩
```

42.3 Viewing part of a box

`\box_clip:N`
`\box_clip:c`

A wrapper around the driver-dependent code.

```
\cs_new_protected:Npn \box_clip:N #1
  { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
\cs_generate_variant:Nn \box_clip:N { c }
```

(*End definition for* `\box_clip:N`. *This function is documented on page 232.*)

`\box_trim:Nnnnn`
`\box_trim:cnnnn`

Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } {#3} { (#4) } {#5} }
\cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
        \box_use:N #1
        \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
      }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```
    \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn { (#3) - \box_dp:N #1 }
              { \box_use:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
```

Same thing, this time from the top of the box.

```
    \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn \c_zero_dim
              { \box_use:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box
          { \box_ht:N \l__box_internal_box - (#5) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
              { \box_use:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box \c_zero_dim
      }
    \box_set_eq:NN #1 \l__box_internal_box
  }
\cs_generate_variant:Nn \box_trim:Nnnnn { c }
```

(*End definition for* `\box_trim:Nnnnn`*. This function is documented on page 233.*)

\box_viewport:Nnnnn
\box_viewport:cnnnn

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```
\__debug_patch_args:nNNpn { {#1} { (#2) } {#3} { (#4) } {#5} }
\cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
        \box_use:N #1
        \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
      }
    \dim_compare:nNnTF {#3} < \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
    \dim_compare:nNnTF {#5} > \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          {
```

```
      \box_move_up:nn \c_zero_dim
       { \box_use:N \l__box_internal_box }
     }
    \box_set_ht:Nn \l__box_internal_box
     {
      (#5)
      \dim_compare:nNnT {#3} > \c_zero_dim
       { - (#3) }
     }
   }
   {
    \hbox_set:Nn \l__box_internal_box
     {
      \box_move_up:nn { -\dim_eval:n {#5} }
       { \box_use:N \l__box_internal_box }
     }
    \box_set_ht:Nn \l__box_internal_box \c_zero_dim
   }
  \box_set_eq:NN #1 \l__box_internal_box
 }
\cs_generate_variant:Nn \box_viewport:Nnnnn { c }
```

(*End definition for* \box_viewport:Nnnnn*. This function is documented on page 233.*)

42.4 Additions to l3clist

```
⟨@@=clist⟩
```

\clist_rand_item:n
\clist_rand_item:N
\clist_rand_item:c
__clist_rand_item:nn

The N-type function is not implemented through the n-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptyness of an n-type comma-list is slow, so we count items first and use that both for the emptyness test and the pseudo-random integer. Importantly, \clist_item:Nn and \clist_item:nn only evaluate their argument once.

```
\cs_new:Npn \clist_rand_item:n #1
 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
\cs_new:Npn \__clist_rand_item:nn #1#2
 {
  \int_compare:nNnF {#1} = 0
   { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
 }
\cs_new:Npn \clist_rand_item:N #1
 {
  \clist_if_empty:NF #1
   { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
 }
\cs_generate_variant:Nn \clist_rand_item:N { c }
```

(*End definition for* \clist_rand_item:n *,* \clist_rand_item:N*, and* __clist_rand_item:nn*. These functions are documented on page 233.*)

42.5 Additions to l3coffins

```
⟨@@=coffin⟩
```

42.6 Rotating coffins

\l__coffin_sin_fp
\l__coffin_cos_fp

Used for rotations to get the sine and cosine values.

```
\fp_new:N \l__coffin_sin_fp
\fp_new:N \l__coffin_cos_fp
```

(*End definition for \l__coffin_sin_fp and \l__coffin_cos_fp.*)

\l__coffin_bounding_prop

A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
\prop_new:N \l__coffin_bounding_prop
```

(*End definition for \l__coffin_bounding_prop.*)

\l__coffin_bounding_shift_dim

The shift of the bounding box of a coffin from the real content.

```
\dim_new:N \l__coffin_bounding_shift_dim
```

(*End definition for \l__coffin_bounding_shift_dim.*)

\l__coffin_left_corner_dim
\l__coffin_right_corner_dim
\l__coffin_bottom_corner_dim
\l__coffin_top_corner_dim

These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
\dim_new:N \l__coffin_left_corner_dim
\dim_new:N \l__coffin_right_corner_dim
\dim_new:N \l__coffin_bottom_corner_dim
\dim_new:N \l__coffin_top_corner_dim
```

(*End definition for \l__coffin_left_corner_dim and others.*)

\coffin_rotate:Nn
\coffin_rotate:cn

Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set \l__coffin_sin_fp and \l__coffin_cos_fp, which are carried through unchanged for the rest of the procedure.

```
\cs_new_protected:Npn \coffin_rotate:Nn #1#2
  {
    \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
    \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
    \__coffin_set_bounding:N #1
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
    \__coffin_find_corner_maxima:N #1
    \__coffin_find_bounding_shift:
    \box_rotate:Nn #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The *x*-direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The *y*-direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```
    \hbox_set:Nn \l__coffin_internal_box
      {
        \tex_kern:D
          \__dim_eval:w
            \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
          \__dim_eval_end:
        \box_move_down:nn { \l__coffin_bottom_corner_dim }
          { \box_use:N #1 }
      }
```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```
    \box_set_ht:Nn \l__coffin_internal_box
      { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
    \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
    \box_set_wd:Nn \l__coffin_internal_box
      { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
    \hbox_set:Nn #1 { \box_use:N \l__coffin_internal_box }
```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
  }
\cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

(*End definition for* `\coffin_rotate:Nn`*. This function is documented on page 233.*)

`\__coffin_set_bounding:N`

The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```
\cs_new_protected:Npn \__coffin_set_bounding:N #1
  {
    \prop_put:Nnx \l__coffin_bounding_prop { tl }
      { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
    \prop_put:Nnx \l__coffin_bounding_prop { tr }
      { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
    \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
    \prop_put:Nnx \l__coffin_bounding_prop { bl }
      { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
    \prop_put:Nnx \l__coffin_bounding_prop { br }
      { { \dim_eval:n { \box_wd:N #1 } } { \dim_use:N \l__coffin_internal_dim } }
  }
```

(*End definition for* `\__coffin_set_bounding:N`*.*)

`\__coffin_rotate_bounding:nnn`
`\__coffin_rotate_corner:Nnnn`

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```
\cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
  {
    \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nnx \l__coffin_bounding_prop {#1}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
```

(*End definition for* `\__coffin_rotate_bounding:nnn` *and* `\__coffin_rotate_corner:Nnnn`*.*)

`\__coffin_rotate_pole:Nnnnnn`

Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```
\cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \__coffin_rotate_vector:nnNN {#5} {#6}
      \l__coffin_x_prime_dim \l__coffin_y_prime_dim
    \__coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        { \dim_use:N \l__coffin_x_prime_dim }
        { \dim_use:N \l__coffin_y_prime_dim }
      }
  }
```

(*End definition for* `\__coffin_rotate_pole:Nnnnnn`*.*)

`\__coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```
\cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \dim_to_fp:n {#1} * \l__coffin_cos_fp
            - \dim_to_fp:n {#2} * \l__coffin_sin_fp
          }
      }
    \dim_set:Nn #4
      {
        \fp_to_dim:n
```

```
          {
              \dim_to_fp:n {#1} * \l__coffin_sin_fp
            + \dim_to_fp:n {#2} * \l__coffin_cos_fp
          }
      }
  }
```

(*End definition for* `\__coffin_rotate_vector:nnNN`*.*)

__coffin_find_corner_maxima:N
__coffin_find_corner_maxima_aux:nn

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```
\cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
  {
    \dim_set:Nn \l__coffin_top_corner_dim   { -\c_max_dim }
    \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
    \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
    \dim_set:Nn \l__coffin_left_corner_dim   { \c_max_dim }
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_find_corner_maxima_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_left_corner_dim
     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_right_corner_dim
     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_bottom_corner_dim
     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
    \dim_set:Nn \l__coffin_top_corner_dim
     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
  }
```

(*End definition for* `\__coffin_find_corner_maxima:N` *and* `\__coffin_find_corner_maxima_aux:nn`*.*)

__coffin_find_bounding_shift:
__coffin_find_bounding_shift_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```
\cs_new_protected:Npn \__coffin_find_bounding_shift:
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_find_bounding_shift_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim
      { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
  }
```

(*End definition for* `\__coffin_find_bounding_shift:` *and* `\__coffin_find_bounding_shift_aux:nn`*.*)

__coffin_shift_corner:Nnnn
__coffin_shift_pole:Nnnnnn

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x- and y-components. For the poles, this means that the direction vector is unchanged.

```
\cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
      }
  }
\cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
        {#5} {#6}
      }
  }
```

(*End definition for __coffin_shift_corner:Nnnn and __coffin_shift_pole:Nnnnnn.*)

42.7 Resizing coffins

\l__coffin_scale_x_fp
\l__coffin_scale_y_fp

Storage for the scaling factors in x and y, respectively.

```
\fp_new:N \l__coffin_scale_x_fp
\fp_new:N \l__coffin_scale_y_fp
```

(*End definition for \l__coffin_scale_x_fp and \l__coffin_scale_y_fp.*)

\l__coffin_scaled_total_height_dim
\l__coffin_scaled_width_dim

When scaling, the values given have to be turned into absolute values.

```
\dim_new:N \l__coffin_scaled_total_height_dim
\dim_new:N \l__coffin_scaled_width_dim
```

(*End definition for \l__coffin_scaled_total_height_dim and \l__coffin_scaled_width_dim.*)

\coffin_resize:Nnn
\coffin_resize:cnn

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```
\cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
  {
    \fp_set:Nn \l__coffin_scale_x_fp
      { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
    \fp_set:Nn \l__coffin_scale_y_fp
      {
          \dim_to_fp:n {#3}
        / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
      }
    \box_resize_to_wd_and_ht_plus_dp:Nnn #1 {#2} {#3}
    \__coffin_resize_common:Nnn #1 {#2} {#3}
  }
\cs_generate_variant:Nn \coffin_resize:Nnn { c }
```

(*End definition for* `\coffin_resize:Nnn`*. This function is documented on page 233.*)

`\__coffin_resize_common:Nnn`

The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
\cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative x-scaling values place the poles in the wrong location: this is corrected here.

```
    \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
      {
        \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
          { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
        \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
          { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
      }
  }
```

(*End definition for* `\__coffin_resize_common:Nnn`*.*)

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the `fp` module.

```
\cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
  {
    \fp_set:Nn \l__coffin_scale_x_fp {#2}
    \fp_set:Nn \l__coffin_scale_y_fp {#3}
    \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
    \dim_set:Nn \l__coffin_internal_dim
      { \coffin_ht:N #1 + \coffin_dp:N #1 }
    \dim_set:Nn \l__coffin_scaled_total_height_dim
      { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
    \dim_set:Nn \l__coffin_scaled_width_dim
      { -\fp_abs:n { \l__coffin_scale_x_fp  } \coffin_wd:N #1 }
    \__coffin_resize_common:Nnn #1
      { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
  }
\cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

(*End definition for* `\coffin_scale:Nnn`*. This function is documented on page 233.*)

`\__coffin_scale_vector:nnNN`

This functions scales a vector from the origin using the pre-set scale factors in x and y. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```
\cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
    \dim_set:Nn #4
      { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
  }
```

(*End definition for* `\__coffin_scale_vector:nnNN`.)

`\__coffin_scale_corner:Nnnn`
`\__coffin_scale_pole:Nnnnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```
\cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
  {
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \__coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_scale_corner:Nnnn` *and* `\__coffin_scale_pole:Nnnnnn`.)

`\__coffin_x_shift_corner:Nnnn`
`\__coffin_x_shift_pole:Nnnnnn`

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```
\cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
      }
  }
\cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_x_shift_corner:Nnnn` *and* `\__coffin_x_shift_pole:Nnnnnn`.)

42.8 Additions to l3file

```
⟨@@=file⟩
```

`\file_get_mdfive_hash:nN`
`\file_get_size:nN`
`\file_get_timestamp:nN`
`\__file_get_details:nnN`

These are all wrappers around the pdfTEX primitives doing the same jobs: as we want consistent file paths to be found, they are all set up using `\file_get_full_name:nN` and so are non-expandable `get` functions. Much of the code is repetitive but we need to branch for LuaTEX (emulation in Lua), for the slightly different syntax needed for `\pdftex_mdfivesum:D` and for the fact that primitive coverage varies in other engines.

```
\cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
  { \__file_get_details:nnN {#1} { mdfivesum } {#2} }
\cs_new_protected:Npn \file_get_size:nN #1#2
  { \__file_get_details:nnN {#1} { size } {#2} }
```

```
\cs_new_protected:Npn \file_get_timestamp:nN #1#2
  { \__file_get_details:nnN {#1} { moddate } {#2} }
\cs_new_protected:Npn \__file_get_details:nnN #1#2#3
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_set:Nx #3
      {
        \use:c { pdftex_file #2 :D } \exp_after:wN
          { \l__file_full_name_str }
      }
  }
\cs_if_exist:NTF \luatex_directlua:D
  {
    \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
      {
        \file_get_full_name:nN {#1} \l__file_full_name_str
        \str_set:Nx #3
          {
            \lua_now_x:n
              {
                l3kernel.file#2
                  ( " \lua_escape_x:n { \l__file_full_name_str } " )
              }
          }
       }
  }
  {
    \cs_set_protected:Npn \file_get_mdfive_hash:nN #1#2
      {
        \file_get_full_name:nN {#1} \l__file_full_name_str
        \tl_set:Nx #2
          {
            \pdftex_mdfivesum:D file \exp_after:wN
              { \l__file_full_name_str }
          }
      }
    \cs_if_exist:NT \xetex_XeTeXversion:D
      {
        \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
          {
            \tl_clear:N #3
            \__msg_kernel_error:nnx
              { kernel } { xetex-primitive-not-available }
              { \exp_not:c { pdffile #2 } }
          }
      }
  }
\__msg_kernel_new:nnnn { kernel } { xetex-primitive-not-available }
  { Primitive~\token_to_str:N #1 not~available }
  {
    XeTeX~does~not~currently~provide~functionality~equivalent~to~the~
    \token_to_str:N #1 primitive.
  }
```

(*End definition for* `\file_get_mdfive_hash:nN` *and others. These functions are documented on page*

234.)

\file_if_exist_input:n
\file_if_exist_input:nF

Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the ⟨*true code*⟩ would be inconsistent with other conditionals.

```
\cs_new_protected:Npn \file_if_exist_input:n #1
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_if_empty:NF \l__file_full_name_str
      { \__file_input:V \l__file_full_name_str }
  }
\cs_new_protected:Npn \file_if_exist_input:nF #1#2
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      {#2}
      { \__file_input:V \l__file_full_name_str }
  }
```

(*End definition for* `\file_if_exist_input:n` *and* `\file_if_exist_input:nF`*. These functions are documented on page 234.*)

\file_if_exist_input:nT
\file_if_exist_input:nTF

For removal after 2017-12-31.

```
\__debug_deprecation:nnNNpn { 2017-12-31 }
  { \file_if_exist:nTF and~ \file_input:n }
\cs_new_protected:Npn \file_if_exist_input:nTF #1#2#3
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      {#3} { #2 \__file_input:V \l__file_full_name_str }
  }
\__debug_deprecation:nnNNpn { 2017-12-31 }
  { \file_if_exist:nT and~ \file_input:n }
\cs_new_protected:Npn \file_if_exist_input:nT #1#2
  {
    \file_get_full_name:nN {#1} \l__file_full_name_str
    \str_if_empty:NF \l__file_full_name_str
      { #2 \__file_input:V \l__file_full_name_str }
  }
```

(*End definition for* `\file_if_exist_input:nT` *and* `\file_if_exist_input:nTF`*.*)

\file_input_stop:

A simple rename.

```
\cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(*End definition for* `\file_input_stop:`*. This function is documented on page 234.*)

42.9 Additions to l3int

```
⟨@@=int⟩
```

\int_rand:nn
__int_rand:ww
__int_rand_narrow:n
__int_rand_narrow:nnn
__int_rand_narrow:nnnn

Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than \c__fp_rand_size_int; #2-#1 may overflow for very large positive #2 and negative #1. If the range is wide, use slower code from l3fp. If the range is narrow, call __int_rand_narrow:nn

{⟨*choices*⟩} {#1} where ⟨*choices*⟩ is the number of possible outcomes. Then __int_rand_narrow:nnnn receives a random number reduced modulo ⟨*choices*⟩, the random number itself, ⟨*choices*⟩ and #1. To avoid bias, throw away the random number if it lies in the last, incomplete, interval of size ⟨*choices*⟩ in [0, \c__fp_rand_size_int − 1], and try again.

```
\cs_if_exist:NTF \pdftex_uniformdeviate:D
 {
  \__debug_patch_args:nNNpn { { (#1) } { (#2) } }
  \cs_new:Npn \int_rand:nn #1#2
   {
    \exp_after:wN \__int_rand:ww
    \__int_value:w \__int_eval:w #1 \exp_after:wN ;
    \__int_value:w \__int_eval:w #2 ;
   }
  \cs_new:Npn \__int_rand:ww #1; #2;
   {
    \int_compare:nNnTF {#1} > {#2}
     {
      \__msg_kernel_expandable_error:nnnn
       { kernel } { backward-range } {#1} {#2}
      \__int_rand:ww #2; #1;
     }
     {
      \int_compare:nNnTF {#1} > 0
       { \int_compare:nNnTF { #2 - #1 } < \c__fp_rand_size_int }
       { \int_compare:nNnTF {#2} < { #1 + \c__fp_rand_size_int } }
         {
          \exp_args:Nf \__int_rand_narrow:nn
           { \int_eval:n { #2 - #1 + 1 } } {#1}
         }
         { \fp_to_int:n { randint(#1,#2) } }
     }
   }
  \cs_new:Npn \__int_rand_narrow:nn
   {
    \exp_args:No \__int_rand_narrow:nnn
     { \pdftex_uniformdeviate:D \c__fp_rand_size_int }
   }
  \cs_new:Npn \__int_rand_narrow:nnn #1#2
   {
    \exp_args:Nf \__int_rand_narrow:nnnn
     { \int_mod:nn {#1} {#2} } {#1} {#2}
   }
  \cs_new:Npn \__int_rand_narrow:nnnn #1#2#3#4
   {
    \int_compare:nNnTF { #2 - #1 + #3 } > \c__fp_rand_size_int
     { \__int_rand_narrow:nn {#3} {#4} }
     { \int_eval:n { #4 + #1 } }
   }
 }
 {
  \cs_new:Npn \int_rand:nn #1#2
   {
        \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
        \int_eval:n {#1}
      }
  }
```

(*End definition for* `\int_rand:nn` *and others. These functions are documented on page 235.*)

The following must be added to l3msg.

```
\cs_if_exist:NT \pdftex_uniformdeviate:D
  {
    \__msg_kernel_new:nnn { kernel } { backward-range }
      { Bounds~ordered~backwards~in~\int_rand:nn {#1}~{#2}. }
  }
```

42.10 Additions to l3msg

```
⟨@@=msg⟩
```

\msg_expandable_error:nnnnnn
\msg_expandable_error:nnffff
\msg_expandable_error:nnnnn
\msg_expandable_error:nnfff
\msg_expandable_error:nnnn
\msg_expandable_error:nnff
\msg_expandable_error:nnn
\msg_expandable_error:nnf
\msg_expandable_error:nn
__msg_expandable_error_module:nn

Pass to an auxiliary the message to display and the module name

```
\cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nf \__msg_expandable_error_module:nn
      {
        \exp_args:Nf \tl_to_str:n
          { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      }
      {#1}
  }
\cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
\cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
\cs_new:Npn \msg_expandable_error:nnn #1#2#3
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
\cs_new:Npn \msg_expandable_error:nn #1#2
  { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
\cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
\cs_generate_variant:Nn \msg_expandable_error:nnnnn  { nnfff }
\cs_generate_variant:Nn \msg_expandable_error:nnnn   { nnff }
\cs_generate_variant:Nn \msg_expandable_error:nnn    { nnf }
\cs_new:Npn \__msg_expandable_error_module:nn #1#2
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_none_delimit_by_q_stop:w
    \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
  }
```

(*End definition for* `\msg_expandable_error:nnnnnn` *and others. These functions are documented on page 235.*)

42.11 Additions to l3prop

```
⟨@@=prop⟩
```

\prop_count:N
\prop_count:c
__prop_count:nn

Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```
\cs_new:Npn \prop_count:N #1
  {
    \int_eval:n
      {
        0
        \prop_map_function:NN #1 \__prop_count:nn
      }
  }
\cs_new:Npn \__prop_count:nn #1#2 { + 1 }
\cs_generate_variant:Nn \prop_count:N { c }
```

(*End definition for* \prop_count:N *and* __prop_count:nn. *These functions are documented on page 235.*)

\prop_map_tokens:Nn
\prop_map_tokens:cn
__prop_map_tokens:nwwn

The mapping is very similar to \prop_map_function:NN. It grabs one key–value pair at a time, and stops when reaching the marker key \q_recursion_tail, which cannot appear in normal keys since those are strings. The odd construction \use:n {#1} allows #1 to contain any token without interfering with \prop_map_break:. Argument #2 of __prop_map_tokens:nwwn is \s__prop the first time, and is otherwise empty.

```
\cs_new:Npn \prop_map_tokens:Nn #1#2
  {
    \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
      \__prop_pair:wn \q_recursion_tail \s__prop { }
    \__prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \if_meaning:w \q_recursion_tail #3
      \exp_after:wN \prop_map_break:
    \fi:
    \use:n {#1} {#3} {#4}
    \__prop_map_tokens:nwwn {#1}
  }
\cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(*End definition for* \prop_map_tokens:Nn *and* __prop_map_tokens:nwwn. *These functions are documented on page 236.*)

\prop_rand_key_value:N
\prop_rand_key_value:c
__prop_rand:NN
__prop_rand_item:Nw

Contrarily to clist, seq and tl, there is no function to get an item of a prop given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because \int_rand:nn is always within bounds. At the end, leave either the key #3 or the value #4 in the input stream.

```
\cs_new:Npn \prop_rand_key_value:N { \__prop_rand:NN \__prop_rand:nNn }
\cs_new:Npn \__prop_rand:nNn #1#2#3 { \exp_not:n { {#1} {#3} } }
\cs_new:Npn \__prop_rand:NN #1#2
  {
    \prop_if_empty:NTF #2 { }
      {
        \exp_after:wN \__prop_rand_item:Nw \exp_after:wN #1
        \__int_value:w \int_rand:nn { 1 } { \prop_count:N #2 } #2
        \q_stop
      }
  }
\cs_new:Npn \__prop_rand_item:Nw #1#2 \s__prop \__prop_pair:wn #3 \s__prop #4
  {
    \int_compare:nNnF {#2} > 1
      { \use_i_delimit_by_q_stop:nw { #1 {#3} \exp_not:n {#4} } }
    \exp_after:wN \__prop_rand_item:Nw \exp_after:wN #1
    \__int_value:w \int_eval:n { #2 - 1 } \s__prop
  }
\cs_generate_variant:Nn \prop_rand_key_value:N { c }
```

(*End definition for* `\prop_rand_key_value:N`*,* `\__prop_rand:NN`*, and* `\__prop_rand_item:Nw`*. These functions are documented on page 236.*)

42.12 Additions to l3seq

```
⟨@@=seq⟩
```

`\seq_mapthread_function:NNN`
`\seq_mapthread_function:NcN`
`\seq_mapthread_function:cNN`
`\seq_mapthread_function:ccN`
`\__seq_mapthread_function:wNN`
`\__seq_mapthread_function:wNw`
`\__seq_mapthread_function:Nnnwnn`

The idea is to first expand both sequences, adding the usual `{ ? \__prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of `#2` and `#5`, which for items in both sequences are `\s__seq \__seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```
\cs_new:Npn \seq_mapthread_function:NNN #1#2#3
  { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
\cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
  {
    \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
      #1 { ? \__prg_break: } { }
    \__prg_break_point:
  }
\cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
  {
    \__seq_mapthread_function:Nnnwnn #2
      #1 { ? \__prg_break: } { }
    \q_stop
  }
\cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
  {
    \use_none:n #2
    \use_none:n #5
    #1 {#3} {#6}
    \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
  }
\cs_generate_variant:Nn \seq_mapthread_function:NNN {     Nc }
\cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }
```

(*End definition for* `\seq_mapthread_function:NNN` *and others. These functions are documented on page 236.*)

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`
`\__seq_set_filter:NNNn`

Similar to `\seq_map_inline:Nn`, without a `\__prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The __seq_wrap_item:n function inserts the relevant __seq_item:n without expansion in the input stream, hence in the x-expanding assignment.

```
\cs_new_protected:Npn \seq_set_filter:NNn
  { \__seq_set_filter:NNNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gset_filter:NNn
  { \__seq_set_filter:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End definition for* \seq_set_filter:NNn *,* \seq_gset_filter:NNn *, and* __seq_set_filter:NNNn*. These functions are documented on page 236.*)

\seq_set_map:NNn
\seq_gset_map:NNn
__seq_set_map:NNNn

Very similar to \seq_set_filter:NNn. We could actually merge the two within a single function, but it would have weird semantics.

```
\cs_new_protected:Npn \seq_set_map:NNn
  { \__seq_set_map:NNNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gset_map:NNn
  { \__seq_set_map:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End definition for* \seq_set_map:NNn *,* \seq_gset_map:NNn *, and* __seq_set_map:NNNn*. These functions are documented on page 236.*)

\seq_rand_item:N
\seq_rand_item:c

Importantly, \seq_item:Nn only evaluates its argument once.

```
\cs_new:Npn \seq_rand_item:N #1
  {
    \seq_if_empty:NF #1
      { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
  }
\cs_generate_variant:Nn \seq_rand_item:N { c }
```

(*End definition for* \seq_rand_item:N*. This function is documented on page 237.*)

42.13 Additions to l3skip

```
⟨@@=skip⟩
```

\skip_split_finite_else_action:nnNN

This macro is useful when performing error checking in certain circumstances. If the ⟨*skip*⟩ register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```
\cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
  {
    \skip_if_finite:nTF {#1}
      {
```

```
    #3 = \etex_gluestretch:D #1 \scan_stop:
    #4 = \etex_glueshrink:D  #1 \scan_stop:
   }
   {
    #3 = \c_zero_skip
    #4 = \c_zero_skip
    #2
   }
 }
```

(*End definition for* `\skip_split_finite_else_action:nnNN`. *This function is documented on page 237.*)

42.14 Additions to l3sys

```
⟨@@=sys⟩
```

\sys_if_rand_exist_p:
\sys_if_rand_exist:TF

Currently, randomness exists under pdfTEX and LuaTEX.

```
\cs_if_exist:NTF \pdftex_uniformdeviate:D
 {
  \prg_new_conditional:Npnn \sys_if_rand_exist: { p , T , F , TF }
   { \prg_return_true: }
 }
 {
  \prg_new_conditional:Npnn \sys_if_rand_exist: { p , T , F , TF }
   { \prg_return_false: }
 }
```

(*End definition for* `\sys_if_rand_exist:TF`. *This function is documented on page 237.*)

\sys_rand_seed:

Unpack the primitive.

```
\cs_new:Npn \sys_rand_seed: { \tex_the:D \pdftex_randomseed:D }
\cs_if_exist:NF \pdftex_randomseed:D
 { \cs_set:Npn \sys_rand_seed: { 0 } }
```

(*End definition for* `\sys_rand_seed:`. *This function is documented on page 237.*)

\sys_gset_rand_seed:n

The primitive always assigns the seed globally.

```
\__debug_patch_args:nNNpn { { (#1) } }
\cs_new_protected:Npn \sys_gset_rand_seed:n #1
 { \pdftex_setrandomseed:D \__int_eval:w #1 \__int_eval_end: }
```

(*End definition for* `\sys_gset_rand_seed:n`. *This function is documented on page 237.*)

\c_sys_shell_escape_int

Expose the engine's shell escape status to the user.

```
\int_const:Nn \c_sys_shell_escape_int
 {
  \sys_if_engine_luatex:TF
   {
    \luatex_directlua:D
     {
      tex.sprint((status.shell_escape~or~os.execute()) .. " ")
     }
   }
   {
    \pdftex_shellescape:D
   }
 }
```

(*End definition for* `\c_sys_shell_escape_int`. *This variable is documented on page 237.*)

`\sys_if_shell_p:`
`\sys_if_shell:TF`

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

```
\prg_new_conditional:Nnn \sys_if_shell: { p , T , F , TF }
  {
    \if_int_compare:w \c_sys_shell_escape_int = 0 ~
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\sys_if_shell:TF`. *This function is documented on page 238.*)

`\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF`

Performs a check for whether *unrestricted* shell escape is enabled.

```
\prg_new_conditional:Nnn \sys_if_shell_unrestricted: { p , T , F , TF }
  {
    \if_int_compare:w \c_sys_shell_escape_int = 1 ~
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\sys_if_shell_unrestricted:TF`. *This function is documented on page 238.*)

`\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF`

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

```
\prg_new_conditional:Nnn \sys_if_shell_restricted: { p , T , F , TF }
  {
    \if_int_compare:w \c_sys_shell_escape_int = 2 ~
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\sys_if_shell_unrestricted:TF`. *This function is documented on page 238.*)

`\c__sys_shell_stream_int`

This is not needed for LuaTEX: shell escape there isn't done using a TEX interface

```
\sys_if_engine_luatex:F
  { \int_const:Nn \c__sys_shell_stream_int { 18 } }
```

(*End definition for* `\c__sys_shell_stream_int`.)

`\sys_shell_now:n`

Execute commands through shell escape immediately.

```
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \sys_shell_now:n #1
      {
        \luatex_directlua:D
```

```
          {
            os.execute("
              \luatex_luaescapestring:D { \etex_detokenize:D {#1} }
            ")
          }
      }
  }
  {
    \cs_new_protected:Npn \sys_shell_now:n #1
      {
        \iow_now:Nn \c__sys_shell_stream_int { #1 }
      }
  }
\cs_generate_variant:Nn \sys_shell_now:n { x }
```

(*End definition for* `\sys_shell_now:n`. *This function is documented on page 238.*)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

```
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \sys_shell_shipout:n #1
      {
        \luatex_latelua:D
          {
            os.execute("
              \luatex_luaescapestring:D { \etex_detokenize:D {#1} }
            ")
          }
      }
  }
  {
    \cs_new_protected:Npn \sys_shell_shipout:n #1
      {
        \iow_shipout:Nn \c__sys_shell_stream_int { #1 }
      }
  }
\cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(*End definition for* `\sys_shell_shipout:n`. *This function is documented on page 238.*)

42.15 Additions to l3tl

```
⟨@@=tl⟩
```

`\tl_if_single_token_p:n`
`\tl_if_single_token:nTF`

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying `f`-expansion yields an empty result if and only if the token list is a single space.

```
\prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_if_empty_return:o { \use_none:n #1 } }
      {
        \tl_if_empty:nTF {#1}
          { \prg_return_false: }
          { \__tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
      }
  }
```

(*End definition for* `\tl_if_single_token:nTF`*. This function is documented on page 238.*)

`\tl_reverse_tokens:n`
`\__tl_reverse_group:nn`

The same as `\tl_reverse:n` but with recursion within brace groups.

```
\cs_new:Npn \tl_reverse_tokens:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \exp:w
        \__tl_act:NNNnn
          \__tl_reverse_normal:nN
          \__tl_reverse_group:nn
          \__tl_reverse_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \__tl_reverse_group:nn #1
  {
    \__tl_act_group_recurse:Nnn
      \__tl_act_reverse_output:n
      { \tl_reverse_tokens:n }
  }
```

`\__tl_act_group_recurse:Nnn`

In many applications of `\__tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, `#1` is the output function, `#2` is the transformation, which should expand in two steps, and `#3` is the group.

```
\cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
  {
    \exp_args:Nf #1
      { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
  }
```

(*End definition for* `\tl_reverse_tokens:n`*,* `\__tl_reverse_group:nn`*, and* `\__tl_act_group_recurse:Nnn`*. These functions are documented on page 238.*)

`\tl_count_tokens:n`
`\__tl_act_count_normal:nN`
`\__tl_act_count_group:nn`
`\__tl_act_count_space:n`

The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `\__tl_act_end:wn` (which is technically implemented as `\c_zero`). Somewhat a hack!

```
\cs_new:Npn \tl_count_tokens:n #1
  {
    \int_eval:n
      {
        \__tl_act:NNNnn
          \__tl_act_count_normal:nN
          \__tl_act_count_group:nn
          \__tl_act_count_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
\cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
\cs_new:Npn \__tl_act_count_group:nn #1 #2
  { 2 + \tl_count_tokens:n {#2} + }
```

(*End definition for* `\tl_count_tokens:n` *and others. These functions are documented on page 239.*)

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
__tl_set_from_file:NNnn
__tl_from_file_do:w

The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```
\cs_new_protected:Npn \tl_set_from_file:Nnn
  { \__tl_set_from_file:NNnn \tl_set:Nn }
\cs_new_protected:Npn \tl_gset_from_file:Nnn
  { \__tl_set_from_file:NNnn \tl_gset:Nn }
\cs_generate_variant:Nn \tl_set_from_file:Nnn  { c }
\cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
\cs_new_protected:Npn \__tl_set_from_file:NNnn #1#2#3#4
  {
    \file_get_full_name:nN {#4} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      { \__file_missing:n {#4} }
      {
        \group_begin:
          \exp_args:No \etex_everyeof:D
            { \c__tl_rescan_marker_tl \exp_not:N }
          #3 \scan_stop:
          \exp_after:wN \__tl_from_file_do:w
          \exp_after:wN \prg_do_nothing:
            \tex_input:D \l__file_full_name_str \scan_stop:
        \exp_args:NNNo \group_end:
        #1 #2 \l__tl_internal_a_tl
      }
  }
\exp_args:Nno \use:nn
  { \cs_new_protected:Npn \__tl_from_file_do:w #1 }
  { \c__tl_rescan_marker_tl }
  { \tl_set:No \l__tl_internal_a_tl {#1} }
```

(*End definition for* `\tl_set_from_file:Nnn` *and others. These functions are documented on page 242.*)

\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
__tl_set_from_file_x:NNnn

When reading a file and allowing expansion of the content, the set up only needs to prevent TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

```
\cs_new_protected:Npn \tl_set_from_file_x:Nnn
  { \__tl_set_from_file_x:NNnn \tl_set:Nn }
\cs_new_protected:Npn \tl_gset_from_file_x:Nnn
  { \__tl_set_from_file_x:NNnn \tl_gset:Nn }
\cs_generate_variant:Nn \tl_set_from_file_x:Nnn  { c }
\cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
\cs_new_protected:Npn \__tl_set_from_file_x:NNnn #1#2#3#4
```

```
  {
    \file_get_full_name:nN {#4} \l__file_full_name_str
    \str_if_empty:NTF \l__file_full_name_str
      { \__file_missing:n {#4} }
      {
        \group_begin:
          \etex_everyeof:D { \exp_not:N }
          #3 \scan_stop:
          \tl_set:Nx \l__tl_internal_a_tl
            { \tex_input:D \l__file_full_name_str \c_space_token }
        \exp_args:NNNo \group_end:
        #1 #2 \l__tl_internal_a_tl
      }
  }
```

(*End definition for* `\tl_set_from_file_x:Nnn`*,* `\tl_gset_from_file_x:Nnn`*, and* `\__tl_set_from_file_x:NNnn`*. These functions are documented on page 242.*)

42.15.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

\tl_if_head_eq_catcode:oNTF

Extra variants.

```
\cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }
```

(*End definition for* `\tl_if_head_eq_catcode:oNTF`*. This function is documented on page 45.*)

\tl_lower_case:n
\tl_upper_case:n
\tl_mixed_case:n
\tl_lower_case:nn
\tl_upper_case:nn
\tl_mixed_case:nn

The user level functions here are all wrappers around the internal functions for case changing.

```
\cs_new:Npn \tl_lower_case:n { \__tl_change_case:nnn { lower } { } }
\cs_new:Npn \tl_upper_case:n { \__tl_change_case:nnn { upper } { } }
\cs_new:Npn \tl_mixed_case:n { \__tl_change_case:nnn { mixed } { } }
\cs_new:Npn \tl_lower_case:nn { \__tl_change_case:nnn { lower } }
\cs_new:Npn \tl_upper_case:nn { \__tl_change_case:nnn { upper } }
\cs_new:Npn \tl_mixed_case:nn { \__tl_change_case:nnn { mixed } }
```

(*End definition for* `\tl_lower_case:n` *and others. These functions are documented on page 239.*)

__tl_change_case:nnn
__tl_change_case_aux:nnn
__tl_change_case_loop:wnn
__tl_change_case_output:nwn
__tl_change_case_output:Vwn
__tl_change_case_output:own
__tl_change_case_output:vwn
__tl_change_case_output:fwn
__tl_change_case_end:wn
__tl_change_case_group:nwnn
__tl_change_case_group_lower:nnnn
__tl_change_case_group_upper:nnnn
__tl_change_case_group_mixed:nnnn
__tl_change_case_space:wnn
__tl_change_case_N_type:Nwnn
__tl_change_case_N_type:NNNnnn
__tl_change_case_math:NNNnnn
__tl_change_case_math_loop:wNNnn
__tl_change_case_math:NwNNnn
__tl_change_case_math_group:nwNNnn
__tl_change_case_math_space:wNNnn
__tl_change_case_N_type:Nnnn
__tl_change_case_char_lower:Nnn
__tl_change_case_char_upper:Nnn

The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either processed as a single token, treated as a group or treated as a space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 \q_recursion_stop.

```
\cs_new:Npn \__tl_change_case:nnn #1#2#3
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \exp:w
        \__tl_change_case_aux:nnn {#1} {#2} {#3}
      }
  }
```

```
\cs_new:Npn \__tl_change_case_aux:nnn #1#2#3
  {
    \group_align_safe_begin:
    \__tl_change_case_loop:wnn
      #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
    \__tl_change_case_result:n { }
  }
\cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_change_case_N_type:Nwnn }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__tl_change_case_group:nwnn }
          { \__tl_change_case_space:wnn }
      }
    #1 \q_recursion_stop
  }
```

Earlier versions of the code where only `x`-type expandable rather than `f`-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for `f`-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to "load" and finalise the result. That is handled in the code below, which includes the necessary material to end the `\exp:w` expansion.

```
\cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
  { #2 \__tl_change_case_result:n { #3 #1 } }
\cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
\cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
  {
    \group_align_safe_end:
    \exp_end:
    #2
  }
```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the `N`-type grabber, after removing the current item from the input `\__tl_change_case_loop:wnn` is inserted in front of the remaining tokens.

```
\cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
  {
    \use:c { __tl_change_case_group_ #3 : nnnn } {#1} {#2} {#3} {#4}
  }
\cs_new:Npn \__tl_change_case_group_lower:nnnn #1#2#3#4
  {
    \__tl_change_case_output:own
      {
        \exp_after:wN
          {
            \exp:w
            \__tl_change_case_aux:nnn {#3} {#4} {#1}
          }
      }
    \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
  }
\cs_new_eq:NN \__tl_change_case_group_upper:nnnn
  \__tl_change_case_group_lower:nnnn
```

For the "mixed" case, a group is taken as forcing a switch to lower casing. That means we need a separate auxiliary. (Tracking whether we have found a first character inside a group and transferring the information out looks pretty horrible.)

```
\cs_new:Npn \__tl_change_case_group_mixed:nnnn #1#2#3#4
  {
    \__tl_change_case_output:own
      {
        \exp_after:wN
          {
            \exp:w
            \__tl_change_case_aux:nnn {#3} {#4} {#1}
          }
      }
    \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#4}
  }
\exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
  {
    \__tl_change_case_output:nwn { ~ }
    \__tl_change_case_loop:wnn
  }
```

For N-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other N-type tokens so is handled up front.

```
\cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
  {
    \quark_if_recursion_tail_stop_do:Nn #1
      { \__tl_change_case_end:wn }
    \exp_after:wN \__tl_change_case_N_type:NNNnnn
     \exp_after:wN #1 \l_tl_case_change_math_tl
     \q_recursion_tail ? \q_recursion_stop {#2}
  }
```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (#1) matches an open-math case (#2). If if does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each N-type token to see if it matches the close-math case required. If that is the situation then the "math loop" stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument #3 to __tl_change_case_math:NNNnnn. If no close-math token is found then the final clean-up is forced (*i.e.* there is no assumption of "well-behaved" input in terms of math mode).

```
\cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
  {
    \quark_if_recursion_tail_stop_do:Nn #2
     { \__tl_change_case_N_type:Nnnn #1 }
    \token_if_eq_meaning:NNTF #1 #2
      {
        \use_i_delimit_by_q_recursion_stop:nw
          {
            \__tl_change_case_math:NNNnnn
              #1 #3 \__tl_change_case_loop:wnn
          }
      }
      { \__tl_change_case_N_type:NNNnnn #1 }
  }
\cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
  {
    \__tl_change_case_output:nwn {#1}
    \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
  }
\cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_change_case_math:NwNNnn }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__tl_change_case_math_group:nwNNnn }
          { \__tl_change_case_math_space:wNNnn }
      }
    #1 \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
  {
    \token_if_eq_meaning:NNTF \q_recursion_tail #1
      { \__tl_change_case_end:wn }
      {
        \__tl_change_case_output:nwn {#1}
        \token_if_eq_meaning:NNTF #1 #3
          { #4 #2 \q_recursion_stop }
          { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
      }
  }
\cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
  {
    \__tl_change_case_output:nwn { {#1} }
    \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
  }
\exp_last_unbraced:NNo
  \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
  {
    \__tl_change_case_output:nwn { ~ }
    \__tl_change_case_math_loop:wNNnn
  }
```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `\__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have `w`-type arguments if they may do a look-ahead.

```
\cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
  {
    \token_if_cs:NTF #1
      { \__tl_change_case_cs_letterlike:Nn #1 {#3} }
      { \use:c { __tl_change_case_char_ #3 :Nnn } #1 {#3} {#4} }
    \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
  }
```

For character tokens there are some special cases to deal with then the majority of changes are covered by using the TEX data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```
\cs_new:Npn \__tl_change_case_char_lower:Nnn #1#2#3
  {
    \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
      { \use_ii:nn }
        #1
        {
          \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
            { \__tl_change_case_char:nN {#2} #1 }
        }
  }
\cs_new_eq:NN \__tl_change_case_char_upper:Nnn
  \__tl_change_case_char_lower:Nnn
```

For mixed case, the code is somewhat different: there is a need to look up both mixed and upper case chars and we have to cover the situation where there is a character to skip over.

```
\cs_new:Npn \__tl_change_case_char_mixed:Nnn #1#2#3
  {
    \__tl_change_case_mixed_switch:w
    \cs_if_exist_use:cF { __tl_change_case_mixed_ #3 :Nnw }
      {
        \cs_if_exist_use:cF { __tl_change_case_upper_ #3 :Nnw }
          { \use_ii:nn }
      }
        #1
        { \__tl_change_case_mixed_skip:N #1 }
  }
```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```
\cs_if_exist:NTF \utex_char:D
  {
    \cs_new:Npn \__tl_change_case_char:nN #1#2
      { \__tl_change_case_char_auxi:nN {#1} #2 }
  }
  {
    \cs_new:Npn \__tl_change_case_char:nN #1#2
      {
        \int_compare:nNnTF { `#2 } > { "80 }
          {
            \int_compare:nNnTF { `#2 } < { "E0 }
              { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
              {
                \int_compare:nNnTF { `#2 } < { "F0 }
                  { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }
                  { \__tl_change_case_char_UTFviii:nNNNNN {#1} #2 }
              }
          }
          { \__tl_change_case_char_auxi:nN {#1} #2 }
       }
  }
```

To allow for the special case of mixed case, we insert here a action-dependent auxiliary.

```
\cs_new:Npn \__tl_change_case_char_auxi:nN #1#2
  { \use:c { __tl_change_case_char_ #1 :N  } #2 }
\cs_new:Npn \__tl_change_case_char_lower:N #1
  {
    \__tl_change_case_output:fwn
      {
        \cs_if_exist_use:cF { c__unicode_lower_ \token_to_str:N #1 _tl }
          { \__tl_change_case_char_auxii:nN { lower } #1 }
      }
  }
\cs_new:Npn \__tl_change_case_char_upper:N #1
  {
    \__tl_change_case_output:fwn
      {
        \cs_if_exist_use:cF { c__unicode_upper_ \token_to_str:N #1 _tl }
          { \__tl_change_case_char_auxii:nN { upper } #1 }
      }
  }
\cs_new:Npn \__tl_change_case_char_mixed:N #1
  {
    \cs_if_exist:cTF { c__unicode_mixed_ \token_to_str:N #1 _tl }
      {
        \__tl_change_case_output:fwn
          { \tl_use:c { c__unicode_mixed_ \token_to_str:N #1 _tl } }
      }
      { \__tl_change_case_char_upper:N #1 }
  }
\cs_if_exist:NTF \utex_char:D
  {
    \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2
      {
        \int_compare:nNnTF { \use:c { __tl_lookup_ #1 :N } #2 } = { 0 }
          { \exp_stop_f: #2 }
          {
            \char_generate:nn
              { \use:c { __tl_lookup_ #1 :N } #2 }
              { \char_value_catcode:n { \use:c { __tl_lookup_ #1 :N } #2 } }
          }
      }
    \cs_new_protected:Npn \__tl_lookup_lower:N #1 { \tex_lccode:D `#1 }
    \cs_new_protected:Npn \__tl_lookup_upper:N #1 { \tex_uccode:D `#1 }
    \cs_new_eq:NN \__tl_lookup_mixed:N \__tl_lookup_upper:N
  }
  {
    \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2 { \exp_stop_f: #2 }
    \cs_new:Npn \__tl_change_case_char_UTFviii:nNNN #1#2#3#4
      { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
    \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
      { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
    \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
      { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
    \cs_new:Npn \__tl_change_case_char_UTFviii:nnN #1#2#3
      {
        \cs_if_exist:cTF { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
          {
            \__tl_change_case_output:vwn
              { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
          }
          { \__tl_change_case_output:nwn {#2} }
        #3
      }
  }
```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The split into two parts here allows us to insert the "switch" code for mixed casing.

```
\cs_new:Npn \__tl_change_case_cs_letterlike:Nn #1#2
  {
    \str_if_eq:nnTF {#2} { mixed }
      {
        \__tl_change_case_cs_letterlike:NnN #1 { upper }
          \__tl_change_case_mixed_switch:w
      }
      { \__tl_change_case_cs_letterlike:NnN #1 {#2} \prg_do_nothing: }
  }
\cs_new:Npn \__tl_change_case_cs_letterlike:NnN #1#2#3
  {
    \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
      {
        \__tl_change_case_output:vwn
          { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
        #3
      }
      {
        \cs_if_exist:cTF
          {
            c__tl_change_case_
            \str_if_eq:nnTF {#2} { lower } { upper } { lower }
            _ \token_to_str:N #1 _tl
          }
          {
            \__tl_change_case_output:nwn {#1}
            #3
          }
          {
            \exp_after:wN \__tl_change_case_cs_accents:NN
              \exp_after:wN #1 \l_tl_case_change_accents_tl
              \q_recursion_tail \q_recursion_stop
          }
      }
  }
\cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
  {
    \quark_if_recursion_tail_stop_do:Nn #2
      { \__tl_change_case_cs:N #1 }
    \str_if_eq:nnTF {#1} {#2}
      {
        \use_i_delimit_by_q_recursion_stop:nw
          { \__tl_change_case_output:nwn {#1} }
      }
      { \__tl_change_case_cs_accents:NN #1 }
  }
```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed: that comes *after* the loop function which is therefore rearranged. In a LaTeX 2$_\varepsilon$ context, `\protect` needs to be treated specially, to prevent expansion of the next token but output it without braces.

```
\cs_new:Npn \__tl_change_case_cs:N #1
  {
⟨*package⟩
    \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
⟨/package⟩
    \exp_after:wN \__tl_change_case_cs:NN
      \exp_after:wN #1 \l_tl_case_change_exclude_tl
      \q_recursion_tail \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_cs:NN #1#2
  {
    \quark_if_recursion_tail_stop_do:Nn #2
      {
        \__tl_change_case_cs_expand:Nnw #1
          { \__tl_change_case_output:nwn {#1} }
      }
    \str_if_eq:nnTF {#1} {#2}
      {
        \use_i_delimit_by_q_recursion_stop:nw
          { \__tl_change_case_cs:NNn #1 }
      }
      { \__tl_change_case_cs:NN #1 }
  }
\cs_new:Npn \__tl_change_case_cs:NNn #1#2#3
  {
    \__tl_change_case_output:nwn { #1 {#3} }
    #2
  }
⟨*package⟩
\cs_new:Npn \__tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
  { \__tl_change_case_output:nwn { \protect #3 } #2 }
⟨/package⟩
```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandability has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as `\bool_if:nTF` would choke if `#1` was `(`!

```
\cs_new:Npn \__tl_change_case_if_expandable:NTF #1
  {
    \token_if_expandable:NTF #1
      {
        \bool_lazy_any:nTF
          {
            { \token_if_eq_meaning_p:NN \q_recursion_tail #1 }
            { \token_if_protected_macro_p:N      #1 }
            { \token_if_protected_long_macro_p:N #1 }
          }
          { \use_ii:nn }
          { \use_i:nn }
      }
      { \use_ii:nn }
  }
\cs_new:Npn \__tl_change_case_cs_expand:Nnw #1#2
  {
    \__tl_change_case_if_expandable:NTF #1
      { \__tl_change_case_cs_expand:NN #1 }
      { #2 }
  }
\cs_new:Npn \__tl_change_case_cs_expand:NN #1#2
  { \exp_after:wN #2 #1 }
```

For mixed case, there is an additional list of exceptions to deal with: once that is sorted, we can move on back to the main loop.

```
\cs_new:Npn \__tl_change_case_mixed_skip:N #1
  {
    \exp_after:wN \__tl_change_case_mixed_skip:NN
      \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
      \q_recursion_tail \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_mixed_skip:NN #1#2
  {
    \quark_if_recursion_tail_stop_do:nn {#2}
      { \__tl_change_case_char:nN { mixed } #1 }
    \int_compare:nNnT { ‘#1 }  = { ‘#2 }
      {
        \use_i_delimit_by_q_recursion_stop:nw
          {
            \__tl_change_case_output:nwn {#1}
            \__tl_change_case_mixed_skip_tidy:Nwn
          }
      }
    \__tl_change_case_mixed_skip:NN #1
  }
\cs_new:Npn \__tl_change_case_mixed_skip_tidy:Nwn #1#2 \q_recursion_stop #3
  {
    \__tl_change_case_loop:wnn #2 \q_recursion_stop { mixed }
  }
```

Needed to switch from mixed to lower casing when we have found a first character in the former mode.

```
\cs_new:Npn \__tl_change_case_mixed_switch:w
  #1 \__tl_change_case_loop:wnn #2 \q_recursion_stop #3
  {
    #1
    \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower }
  }
```

(*End definition for* `\__tl_change_case:nnn` *and others.*)

__tl_change_case_lower_sigma:Nnw
__tl_change_case_lower_sigma:w
__tl_change_case_lower_sigma:Nw
__tl_change_case_upper_sigma:Nnw

If the current char is an upper case sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase.

```
\cs_new:Npn \__tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
  {
    \int_compare:nNnTF { `#1 } = { "03A3 }
      {
        \__tl_change_case_output:fwn
          { \__tl_change_case_lower_sigma:w #4 \q_recursion_stop }
      }
      {#2}
    #3 #4 \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_lower_sigma:w #1 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
      { \c__unicode_final_sigma_tl }
  }
\cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
  {
    \__tl_change_case_if_expandable:NTF #1
      {
        \exp_after:wN \__tl_change_case_lower_sigma:w #1
          #2 \q_recursion_stop
      }
      {
        \token_if_letter:NTF #1
          { \c__unicode_std_sigma_tl }
          { \c__unicode_final_sigma_tl }
      }
  }
```

Simply skip to the final step for upper casing.

```
\cs_new_eq:NN \__tl_change_case_upper_sigma:Nnw \use_ii:nn
```

(*End definition for* `\__tl_change_case_lower_sigma:Nnw` *and others.*)

__tl_change_case_lower_tr:Nnw
__tl_change_case_lower_tr_auxi:Nw
__tl_change_case_lower_tr_auxii:Nw
__tl_change_case_upper_tr:Nnw
__tl_change_case_lower_az:Nnw
__tl_change_case_upper_az:Nnw

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```
\cs_if_exist:NTF \utex_char:D
  {
    \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
      {
        \int_compare:nNnTF { `#1 } = { "0049 }
          { \__tl_change_case_lower_tr_auxi:Nw }
          {
            \int_compare:nNnTF { `#1 } = { "0130 }
              { \__tl_change_case_output:nwn { i } }
              {#2}
          }
      }
```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input, which is done by the `\use_i:nn` (it grabs `\__tl_change_case_loop:wn` and the dot-above char and discards the latter).

```
    \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
      {
        \tl_if_head_is_N_type:nTF {#2}
          { \__tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
          { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
        #1 #2 \q_recursion_stop
      }
    \cs_new:Npn \__tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
      {
        \__tl_change_case_if_expandable:NTF #1
          {
            \exp_after:wN \__tl_change_case_lower_tr_auxi:Nw #1
              #2 \q_recursion_stop
          }
          {
            \bool_lazy_or:nnTF
              { \token_if_cs_p:N #1 }
              { ! \int_compare_p:nNn { `#1 } = { "0307 } }
              { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
              {
                \__tl_change_case_output:nwn { i }
                \use_i:nn
              }
          }
      }
  }
```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is.

```
  {
    \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
      {
        \int_compare:nNnTF { ‘#1 } = { "0049 }
          { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
          {
            \int_compare:nNnTF { ‘#1 } = { 196 }
              { \__tl_change_case_lower_tr_auxi:Nw #1 {#2} }
              {#2}
          }
      }
    \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2#3#4
      {
        \int_compare:nNnTF { ‘#4 } = { 176 }
          {
            \__tl_change_case_output:nwn { i }
            #3
          }
          {
            #2
            #3 #4
          }
      }
  }
```

Upper casing is easier: just one exception with no context.

```
\cs_new:Npn \__tl_change_case_upper_tr:Nnw #1#2
  {
    \int_compare:nNnTF { ‘#1 } = { "0069 }
      { \__tl_change_case_output:Vwn \c__unicode_dotted_I_tl }
      {#2}
  }
```

Straight copies.

```
\cs_new_eq:NN \__tl_change_case_lower_az:Nnw \__tl_change_case_lower_tr:Nnw
\cs_new_eq:NN \__tl_change_case_upper_az:Nnw \__tl_change_case_upper_tr:Nnw
```

(*End definition for* `\__tl_change_case_lower_tr:Nnw` *and others.*)

__tl_change_case_lower_lt:Nnw
__tl_change_case_lower_lt:nNnw
__tl_change_case_lower_lt:nnw
__tl_change_case_lower_lt:Nw
__tl_change_case_lower_lt:NNw
__tl_change_case_upper_lt:Nnw
__tl_change_case_upper_lt:nnw
__tl_change_case_upper_lt:Nw
__tl_change_case_upper_lt:NNw

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: `\c__tl_accents_lt_tl` contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using `f`-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```
\cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
  {
    \exp_args:Nf \__tl_change_case_lower_lt:nNnw
      { \str_case:nVF #1 \c__unicode_accents_lt_tl \exp_stop_f: }
      #1
  }
\cs_new:Npn \__tl_change_case_lower_lt:nNnw #1#2
  {
    \tl_if_blank:nTF {#1}
      {
        \exp_args:Nf \__tl_change_case_lower_lt:nnw
          {
            \int_case:nnF {`#2}
              {
                { "0049 } i
                { "004A } j
                { "012E } \c__unicode_i_ogonek_tl
              }
              \exp_stop_f:
          }
      }
      {
        \__tl_change_case_output:nwn {#1}
        \use_none:n
      }
  }
\cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
  {
    \tl_if_blank:nTF {#1}
      {#2}
      {
        \__tl_change_case_output:nwn {#1}
        \__tl_change_case_lower_lt:Nw
      }
  }
```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```
\cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nT {#2}
      { \__tl_change_case_lower_lt:NNw }
    #1 #2 \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
  {
    \__tl_change_case_if_expandable:NTF #2
      {
        \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
          #3 \q_recursion_stop
      }
      {
        \bool_lazy_and:nnT
          { ! \token_if_cs_p:N #2 }
          {
            \bool_lazy_any_p:n
              {
                { \int_compare_p:nNn { `#2 } = { "0300 } }
                { \int_compare_p:nNn { `#2 } = { "0301 } }
                { \int_compare_p:nNn { `#2 } = { "0303 } }
              }
          }
          { \__tl_change_case_output:Vwn \c__unicode_dot_above_tl }
        #1 #2#3 \q_recursion_stop
      }
  }
```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```
\cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
  {
    \exp_args:Nf \__tl_change_case_upper_lt:nnw
      {
        \int_case:nnF {`#1}
          {
            { "0069 } I
            { "006A } J
            { "012F } \c__unicode_I_ogonek_tl
          }
          \exp_stop_f:
      }
  }
\cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
  {
    \tl_if_blank:nTF {#1}
      {#2}
      {
        \__tl_change_case_output:nwn {#1}
        \__tl_change_case_upper_lt:Nw
      }
  }
\cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nT {#2}
      { \__tl_change_case_upper_lt:NNw }
    #1 #2 \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_upper_lt:NNw #1#2#3 \q_recursion_stop
  {
    \__tl_change_case_if_expandable:NTF #2
      {
        \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
          #3 \q_recursion_stop
      }
      {
        \bool_lazy_and:nnTF
          { ! \token_if_cs_p:N #2 }
          { \int_compare_p:nNn { `#2 } = { "0307 } }
          { #1 }
          { #1 #2 }
        #3 \q_recursion_stop
      }
  }
```

(*End definition for* `\__tl_change_case_lower_lt:Nnw` *and others.*)

`\__tl_change_case_upper_de-alt:Nnw` A simple alternative version for German.

```
\cs_new:cpn { __tl_change_case_upper_de-alt:Nnw } #1#2
  {
    \int_compare:nNnTF { `#1 } = { 223 }
      { \__tl_change_case_output:Vwn \c__unicode_upper_Eszett_tl }
      {#2}
  }
```

(*End definition for* `\__tl_change_case_upper_de-alt:Nnw`*.*)

__unicode_codepoint_to_UTFviii:n
__unicode_codepoint_to_UTFviii_auxi:n
__unicode_codepoint_to_UTFviii_auxii:Nnn
__unicode_codepoint_to_UTFviii_auxiii:n

This code converts a codepoint into the correct UTF-8 representation. As there are a variable number of octets, the result starts with the numeral 1–4 to indicate the nature of the returned value. Note that this code covers the full range even though at this stage it is not required here. Also note that longer-term this is likely to need a public interface and/or moving to l3str (see experimental string conversions). In terms of the algorithm itself, see https://en.wikipedia.org/wiki/UTF-8 for the octet pattern.

```
\cs_new:Npn \__unicode_codepoint_to_UTFviii:n #1
  {
    \exp_args:Nf \__unicode_codepoint_to_UTFviii_auxi:n
      { \int_eval:n {#1} }
  }
\cs_new:Npn \__unicode_codepoint_to_UTFviii_auxi:n #1
  {
    \if_int_compare:w #1 > "80 ~
      \if_int_compare:w #1 < "800 ~
        2
        \__unicode_codepoint_to_UTFviii_auxii:Nnn C {#1} { 64 }
        \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
      \else:
        \if_int_compare:w #1 < "10000 ~
          3
          \__unicode_codepoint_to_UTFviii_auxii:Nnn E {#1} { 64 * 64 }
          \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
          \__unicode_codepoint_to_UTFviii_auxiii:n
            { \int_div_truncate:nn {#1} { 64 } }
        \else:
          4
          \__unicode_codepoint_to_UTFviii_auxii:Nnn F
            {#1} { 64 * 64 * 64 }
          \__unicode_codepoint_to_UTFviii_auxiii:n
            { \int_div_truncate:nn {#1} { 64 * 64 } }
          \__unicode_codepoint_to_UTFviii_auxiii:n
            { \int_div_truncate:nn {#1} { 64 } }
          \__unicode_codepoint_to_UTFviii_auxiii:n {#1}

        \fi:
      \fi:
    \else:
      1 {#1}
    \fi:
  }
\cs_new:Npn \__unicode_codepoint_to_UTFviii_auxii:Nnn #1#2#3
  { { \int_eval:n { "#10 + \int_div_truncate:nn {#2} {#3} } } }
\cs_new:Npn \__unicode_codepoint_to_UTFviii_auxiii:n #1
  { { \int_eval:n { \int_mod:nn {#1} { 64 } + 128 } } }
```

(*End definition for* `\__unicode_codepoint_to_UTFviii:n` *and others.*)

\c__unicode_std_sigma_tl
\c__unicode_final_sigma_tl
\c__unicode_accents_lt_tl
\c__unicode_dot_above_tl
\c__unicode_upper_Eszett_tl

The above needs various special token lists containg pre-formed characters. This set are only available in Unicode engines, with no-op definitions for 8-bit use.

```
\cs_if_exist:NTF \utex_char:D
  {
    \tl_const:Nx \c__unicode_std_sigma_tl    { \utex_char:D "03C3 ~ }
    \tl_const:Nx \c__unicode_final_sigma_tl  { \utex_char:D "03C2 ~ }
    \tl_const:Nx \c__unicode_accents_lt_tl
      {
        \utex_char:D "00CC ~
          { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0300 ~ }
        \utex_char:D "00CD ~
          { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0301 ~ }
        \utex_char:D "0128 ~
          { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0303 ~ }
      }
    \tl_const:Nx \c__unicode_dot_above_tl    { \utex_char:D "0307 ~ }
    \tl_const:Nx \c__unicode_upper_Eszett_tl { \utex_char:D "1E9E ~ }
  }
  {
      \tl_const:Nn \c__unicode_std_sigma_tl    { }
      \tl_const:Nn \c__unicode_final_sigma_tl  { }
      \tl_const:Nn \c__unicode_accents_lt_tl   { }
      \tl_const:Nn \c__unicode_dot_above_tl    { }
      \tl_const:Nn \c__unicode_upper_Eszett_tl { }
  }
```

(*End definition for* `\c__unicode_std_sigma_tl` *and others.*)

\c__unicode_dotless_i_tl
\c__unicode_dotted_I_tl
\c__unicode_i_ogonek_tl
\c__unicode_I_ogonek_tl

For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```
\group_begin:
  \cs_if_exist:NTF \utex_char:D
    {
      \cs_set_protected:Npn \__tl_tmp:w #1#2
        { \tl_const:Nx #1 { \utex_char:D "#2 ~ } }
    }
    {
      \cs_set_protected:Npn \__tl_tmp:w #1#2
        {
          \group_begin:
            \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
              {
                \tl_const:Nx #1
                  {
                    \exp_after:wN \exp_after:wN \exp_after:wN
                      \exp_not:N \__char_generate:nn {##2} { 13 }
                    \exp_after:wN \exp_after:wN \exp_after:wN
                      \exp_not:N \__char_generate:nn {##3} { 13 }
                  }
              }
            \tl_set:Nx \l__tl_internal_a_tl
              { \__unicode_codepoint_to_UTFviii:n {"#2} }
            \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
          \group_end:
        }
    }
  \__tl_tmp:w \c__unicode_dotless_i_tl { 0131 }
  \__tl_tmp:w \c__unicode_dotted_I_tl  { 0130 }
  \__tl_tmp:w \c__unicode_i_ogonek_tl  { 012F }
  \__tl_tmp:w \c__unicode_I_ogonek_tl  { 012E }
\group_end:
```

(*End definition for* `\c__unicode_dotless_i_tl` *and others.*)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```
\group_begin:
  \bool_lazy_or:nnT
    { \sys_if_engine_pdftex_p: }
    { \sys_if_engine_uptex_p: }
    {
      \cs_set_protected:Npn \__tl_loop:nn #1#2
        {
          \quark_if_recursion_tail_stop:n {#1}
          \tl_set:Nx \l__tl_internal_a_tl
            {
              \__unicode_codepoint_to_UTFviii:n {"#1}
              \__unicode_codepoint_to_UTFviii:n {"#2}
            }
          \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
          \__tl_loop:nn
        }
      \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6
        {
          \tl_const:cx
            {
              c__unicode_lower_
              \char_generate:nn {#2} { 12 }
              \char_generate:nn {#3} { 12 }
              _tl
            }
            {
              \exp_after:wN \exp_after:wN \exp_after:wN
                \exp_not:N \__char_generate:nn {#5} { 13 }
              \exp_after:wN \exp_after:wN \exp_after:wN
                \exp_not:N \__char_generate:nn {#6} { 13 }
            }
          \tl_const:cx
            {
              c__unicode_upper_
              \char_generate:nn {#5} { 12 }
              \char_generate:nn {#6} { 12 }
              _tl
            }
            {
```

```
        \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_not:N \__char_generate:nn {#2} { 13 }
        \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_not:N \__char_generate:nn {#3} { 13 }
      }
  }
\__tl_loop:nn
  { 00C0 } { 00E0 }
  { 00C2 } { 00E2 }
  { 00C3 } { 00E3 }
  { 00C4 } { 00E4 }
  { 00C5 } { 00E5 }
  { 00C6 } { 00E6 }
  { 00C7 } { 00E7 }
  { 00C8 } { 00E8 }
  { 00C9 } { 00E9 }
  { 00CA } { 00EA }
  { 00CB } { 00EB }
  { 00CC } { 00EC }
  { 00CD } { 00ED }
  { 00CE } { 00EE }
  { 00CF } { 00EF }
  { 00D0 } { 00F0 }
  { 00D1 } { 00F1 }
  { 00D2 } { 00F2 }
  { 00D3 } { 00F3 }
  { 00D4 } { 00F4 }
  { 00D5 } { 00F5 }
  { 00D6 } { 00F6 }
  { 00D8 } { 00F8 }
  { 00D9 } { 00F9 }
  { 00DA } { 00FA }
  { 00DB } { 00FB }
  { 00DC } { 00FC }
  { 00DD } { 00FD }
  { 00DE } { 00FE }
  { 0100 } { 0101 }
  { 0102 } { 0103 }
  { 0104 } { 0105 }
  { 0106 } { 0107 }
  { 0108 } { 0109 }
  { 010A } { 010B }
  { 010C } { 010D }
  { 010E } { 010F }
  { 0110 } { 0111 }
  { 0112 } { 0113 }
  { 0114 } { 0115 }
  { 0116 } { 0117 }
  { 0118 } { 0119 }
  { 011A } { 011B }
  { 011C } { 011D }
  { 011E } { 011F }
  { 0120 } { 0121 }
  { 0122 } { 0123 }
```

```
        { 0124 } { 0125 }
        { 0128 } { 0129 }
        { 012A } { 012B }
        { 012C } { 012D }
        { 012E } { 012F }
        { 0132 } { 0133 }
        { 0134 } { 0135 }
        { 0136 } { 0137 }
        { 0139 } { 013A }
        { 013B } { 013C }
        { 013E } { 013F }
        { 0141 } { 0142 }
        { 0143 } { 0144 }
        { 0145 } { 0146 }
        { 0147 } { 0148 }
        { 014A } { 014B }
        { 014C } { 014D }
        { 014E } { 014F }
        { 0150 } { 0151 }
        { 0152 } { 0153 }
        { 0154 } { 0155 }
        { 0156 } { 0157 }
        { 0158 } { 0159 }
        { 015A } { 015B }
        { 015C } { 015D }
        { 015E } { 015F }
        { 0160 } { 0161 }
        { 0162 } { 0163 }
        { 0164 } { 0165 }
        { 0168 } { 0169 }
        { 016A } { 016B }
        { 016C } { 016D }
        { 016E } { 016F }
        { 0170 } { 0171 }
        { 0172 } { 0173 }
        { 0174 } { 0175 }
        { 0176 } { 0177 }
        { 0178 } { 00FF }
        { 0179 } { 017A }
        { 017B } { 017C }
        { 017D } { 017E }
        { 01CD } { 01CE }
        { 01CF } { 01D0 }
        { 01D1 } { 01D2 }
        { 01D3 } { 01D4 }
        { 01E2 } { 01E3 }
        { 01E6 } { 01E7 }
        { 01E8 } { 01E9 }
        { 01EA } { 01EB }
        { 01F4 } { 01F5 }
        { 0218 } { 0219 }
        { 021A } { 021B }
        \q_recursion_tail ?
        \q_recursion_stop
```

```
      \cs_set_protected:Npn \__tl_tmp:w #1#2#3
        {
          \group_begin:
            \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
              {
                \tl_const:cx
                  {
                    c__unicode_ #3 _
                    \char_generate:nn {##2} { 12 }
                    \char_generate:nn {##3} { 12 }
                    _tl
                  }
                    {#2}
              }
            \tl_set:Nx \l__tl_internal_a_tl
              { \__unicode_codepoint_to_UTFviii:n { "#1 } }
            \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
          \group_end:
        }
      \__tl_tmp:w { 00DF } { SS } { upper }
      \__tl_tmp:w { 00DF } { Ss } { mixed }
      \__tl_tmp:w { 0131 } { I }  { upper }
    }
  \group_end:
```

The (fixed) look-up mappings for letter-like control sequences.

```
\group_begin:
  \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
    {
      \quark_if_recursion_tail_stop:N #1
      \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl } { #2 }
      \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl } { #1 }
      \__tl_change_case_setup:NN
    }
  \__tl_change_case_setup:NN
  \AA \aa
  \AE \ae
  \DH \dh
  \DJ \dj
  \IJ \ij
  \L  \l
  \NG \ng
  \O  \o
  \OE \oe
  \SS \ss
  \TH \th
  \q_recursion_tail ?
  \q_recursion_stop
  \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
  \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
\group_end:
```

\l_tl_case_change_accents_tl A list of accents to leave alone.

```
\tl_new:N \l_tl_case_change_accents_tl
```

```
\tl_set:Nn \l_tl_case_change_accents_tl
  { \" \' \. \^ \` \~ \c \H \k \r \t \u \v }
```

(*End definition for \l_tl_case_change_accents_tl. This variable is documented on page 241.*)

__tl_change_case_mixed_nl:Nnw
__tl_change_case_mixed_nl:Nw
__tl_change_case_mixed_nl:NNw

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```
\cs_new:Npn \__tl_change_case_mixed_nl:Nnw #1
  {
    \bool_lazy_or:nnTF
      { \int_compare_p:nNn { `#1 } = { `i } }
      { \int_compare_p:nNn { `#1 } = { `I } }
      {
        \__tl_change_case_output:nwn { I }
        \__tl_change_case_mixed_nl:Nw
      }
  }
\cs_new:Npn \__tl_change_case_mixed_nl:Nw #1#2 \q_recursion_stop
  {
    \tl_if_head_is_N_type:nT {#2}
      { \__tl_change_case_mixed_nl:NNw }
    #1 #2 \q_recursion_stop
  }
\cs_new:Npn \__tl_change_case_mixed_nl:NNw #1#2#3 \q_recursion_stop
  {
    \__tl_change_case_if_expandable:NTF #2
      {
        \exp_after:wN \__tl_change_case_mixed_nl:Nw \exp_after:wN #1 #2
          #3 \q_recursion_stop
      }
      {
        \bool_lazy_and:nnTF
          { ! ( \token_if_cs_p:N #2 ) }
          {
            \bool_lazy_or_p:nn
              { \int_compare_p:nNn { `#2 } = { `j } }
              { \int_compare_p:nNn { `#2 } = { `J } }
          }
          {
            \__tl_change_case_output:nwn { J }
            #1
          }
          { #1 #2 }
        #3 \q_recursion_stop
      }
  }
```

(*End definition for __tl_change_case_mixed_nl:Nnw, __tl_change_case_mixed_nl:Nw, and __tl_change_case_mixed_nl:NNw.*)

\l_tl_case_change_math_tl

The list of token pairs which are treated as math mode and so not case changed.

```
\tl_new:N \l_tl_case_change_math_tl
⟨*package⟩
\tl_set:Nn \l_tl_case_change_math_tl
  { $ $ \( \) }
```

```
⟨/package⟩
```

(*End definition for* `\l_tl_case_change_math_tl`*. This variable is documented on page 240.*)

`\l_tl_case_change_exclude_tl`

The list of commands for which an argument is not case changed.

```
\tl_new:N \l_tl_case_change_exclude_tl
⟨*package⟩
\tl_set:Nn \l_tl_case_change_exclude_tl
  { \cite \ensuremath \label \ref }
⟨/package⟩
```

(*End definition for* `\l_tl_case_change_exclude_tl`*. This variable is documented on page 240.*)

`\l_tl_mixed_case_ignore_tl`

Characters to skip over when finding the first letter in a word to be mixed cased.

```
\tl_new:N \l_tl_mixed_case_ignore_tl
\tl_set:Nx \l_tl_mixed_case_ignore_tl
  {
    ( % )
    [ % ]
    \cs_to_str:N \{ % \}
    ‘
    -
  }
```

(*End definition for* `\l_tl_mixed_case_ignore_tl`*. This variable is documented on page 241.*)

42.15.2 Other additions to l3tl

`\tl_rand_item:n`
`\tl_rand_item:N`
`\tl_rand_item:c`

Importantly `\tl_item:nn` only evaluates its argument once.

```
\cs_new:Npn \tl_rand_item:n #1
  {
    \tl_if_blank:nF {#1}
      { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
  }
\cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
\cs_generate_variant:Nn \tl_rand_item:N { c }
```

(*End definition for* `\tl_rand_item:n` *and* `\tl_rand_item:N`*. These functions are documented on page 242.*)

Some preliminary code is needed for the `\tl_range:nnn` family of functions.

`\tl_range:Nnn`
`\tl_range:cnn`
`\tl_range:nnn`
`\tl_range_braced:Nnn`
`\tl_range_braced:cnn`
`\tl_range_braced:nnn`
`\tl_range_unbraced:Nnn`
`\tl_range_unbraced:cnn`
`\tl_range_unbraced:nnn`
`\__tl_range:Nnnn`
`\__tl_range:nnnNn`
`\__tl_range:nnNn`
`\__tl_range_skip:w`
`\__tl_range_braced:w`
`\__tl_range_collect_braced:w`
`\__tl_range_unbraced:w`
`\__tl_range_collect_unbraced:w`
`\__tl_range:w`
`\__tl_range_skip_spaces:n`
`\__tl_range_collect:nn`
`\__tl_range_collect:ff`
`\__tl_range_collect_space:nw`
`\__tl_range_collect_N:nN`

To avoid checking for the end of the token list at every step, start by counting the number l of items and "normalizing" the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, `\__tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function among `\__tl_range:w`, `\__tl_range_braced:w`, `\__tl_range_unbraced:w`, and the token list itself. If nothing should be kept, leave `{}`: this stops the `f`-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `\__tl_range_skip:w` to delete `#1` items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `\__tl_range_braced:w` sets up `\__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `\__tl_range_collect:nn` with a first argument of the form `{`

{⟨*collected*⟩} ⟨*tokens*⟩ }, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the ⟨*collected*⟩ tokens. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```
\cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
\cs_generate_variant:Nn \tl_range:Nnn { c }
\cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
\cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
\cs_generate_variant:Nn \tl_range_braced:Nnn { c }
\cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
\cs_new:Npn \tl_range_unbraced:Nnn { \exp_args:No \tl_range_unbraced:nnn }
\cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
\cs_new:Npn \tl_range_unbraced:nnn { \__tl_range:Nnnn \__tl_range_unbraced:w }
\cs_new:Npn \__tl_range:Nnnn #1#2#3#4
  {
    \tl_head:f
      {
        \exp_args:Nf \__tl_range:nnnNn
          { \tl_count:n {#2} } {#3} {#4} #1 {#2}
      }
  }
\cs_new:Npn \__tl_range:nnnNn #1#2#3
  {
    \exp_args:Nff \__tl_range:nnNn
      {
        \exp_args:Nf \__tl_range_normalize:nn
          { \int_eval:n { #2 - 1 } } {#1}
      }
      {
        \exp_args:Nf \__tl_range_normalize:nn
          { \int_eval:n {#3} } {#1}
      }
  }
\cs_new:Npn \__tl_range:nnNn #1#2#3#4
  {
    \if_int_compare:w #2 > #1 \exp_stop_f: \else:
      \exp_after:wN { \exp_after:wN }
    \fi:
    \exp_after:wN #3
    \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
    \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
  }
\cs_new:Npn \__tl_range_skip:w #1 ; #2
  {
    \if_int_compare:w #1 > 0 \exp_stop_f:
      \exp_after:wN \__tl_range_skip:w
      \__int_value:w \__int_eval:w #1 - 1 \exp_after:wN ;
    \else:
      \exp_after:wN \exp_end:
    \fi:
  }
```

```
\cs_new:Npn \__tl_range_braced:w #1 ; #2
  { \__tl_range_collect_braced:w #1 ; { } #2 }
\cs_new:Npn \__tl_range_unbraced:w #1 ; #2
  { \__tl_range_collect_unbraced:w #1 ; { } #2 }
\cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
  {
    \if_int_compare:w #1 > 1 \exp_stop_f:
      \exp_after:wN \__tl_range_collect_braced:w
      \__int_value:w \__int_eval:w #1 - 1 \exp_after:wN ;
    \fi:
    { #2 {#3} }
  }
\cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
  {
    \if_int_compare:w #1 > 1 \exp_stop_f:
      \exp_after:wN \__tl_range_collect_unbraced:w
      \__int_value:w \__int_eval:w #1 - 1 \exp_after:wN ;
    \fi:
    { #2 #3 }
  }
\cs_new:Npn \__tl_range:w #1 ; #2
  {
    \exp_args:Nf \__tl_range_collect:nn
      { \__tl_range_skip_spaces:n {#2} } {#1}
  }
\cs_new:Npn \__tl_range_skip_spaces:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
      { { } #1 }
  }
\cs_new:Npn \__tl_range_collect:nn #1#2
  {
    \int_compare:nNnTF {#2} = 0
      {#1}
      {
        \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
          {
            \exp_args:Nf \__tl_range_collect:nn
              { \__tl_range_collect_space:nw #1 }
              {#2}
          }
          {
            \__tl_range_collect:ff
              {
                \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
                  { \__tl_range_collect_N:nN }
                  { \__tl_range_collect_group:nn }
                #1
              }
              { \int_eval:n { #2 - 1 } }
          }
      }
  }
```

```
\cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
\cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
\cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
\cs_generate_variant:Nn \__tl_range_collect:nn { ff }
```

(*End definition for* `\tl_range:Nnn` *and others. These functions are documented on page* ??.)

__tl_range_normalize:nn

This function converts an ⟨*index*⟩ argument into an explicit position in the token list (a result of 0 denoting "out of bounds"). Expects two explicit integer arguments: the ⟨*index*⟩ `#1` and the string count `#2`. If `#1` is negative, replace it by `#1` + `#2` + 1, then limit to the range [0, `#2`].

```
\cs_new:Npn \__tl_range_normalize:nn #1#2
  {
    \int_eval:n
      {
        \if_int_compare:w #1 < 0 \exp_stop_f:
          \if_int_compare:w #1 < -#2 \exp_stop_f:
            0
          \else:
            #1 + #2 + 1
          \fi:
        \else:
          \if_int_compare:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

(*End definition for* `\__tl_range_normalize:nn`.)

42.16 Additions to l3token

\c_catcode_active_space_tl

While `\__char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```
\group_begin:
  \char_set_catcode_active:N *
  \char_set_lccode:nn { `* } { `\ }
  \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
\group_end:
```

(*End definition for* `\c_catcode_active_space_tl`. *This variable is documented on page* 244.)

```
⟨@@=peek⟩
```

\peek_N_type:*TF*
__peek_execute_branches_N_type:
__peek_N_type:w
__peek_N_type_aux:nnw

All tokens are `N`-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-`N`-type tokens (explicit or implicit), thus we call `\__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\use_none_delimit_by_q_stop:w` cleans up, and we call `\__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `\__peek_true:w` or `\__peek_false:w` as appropriate. Here, there is no ⟨*search token*⟩, so we feed a dummy `\scan_stop:` to the `\__peek_token_generic:NNTF` function.

```
\group_begin:
  \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
    {
      \cs_new_protected:Npn \__peek_execute_branches_N_type:
        {
          \if_int_odd:w
              \if_catcode:w \exp_not:N \l_peek_token {   0 \exp_stop_f: \fi:
              \if_catcode:w \exp_not:N \l_peek_token }   0 \exp_stop_f: \fi:
              \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
              1 \exp_stop_f:
            \exp_after:wN \__peek_N_type:w
              \token_to_meaning:N \l_peek_token
              \q_mark \__peek_N_type_aux:nnw
              #1 \q_mark \use_none_delimit_by_q_stop:w
              \q_stop
            \exp_after:wN \__peek_true:w
          \else:
            \exp_after:wN \__peek_false:w
          \fi:
        }
      \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
        { ##3 {##1} {##2} }
    }
  \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
\group_end:
\cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
  {
    \fi:
    \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
      { \__peek_true:w }
      { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
  }
\cs_new_protected:Npn \peek_N_type:TF
  { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected:Npn \peek_N_type:T
  { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected:Npn \peek_N_type:F
  { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }
```

(*End definition for `\peek_N_type:TF` and others. These functions are documented on page 244.*)

```
⟨/initex | package⟩
```

43 l3luatex implementation

```
⟨*initex | package⟩
```

43.1 Breaking out to Lua

```
⟨*tex⟩
```

\lua_now_x:n
\lua_now:n
\lua_shipout_x:n
\lua_shipout:n
\lua_escape_x:n
\lua_escape:n

Wrappers around the primitives. As with engines other than LuaTEX these have to be macros, we give them the same status in all cases. When LuaTEX is not in use, simply give an error message/

```
\cs_new:Npn \lua_now_x:n #1 { \luatex_directlua:D {#1} }
\cs_new:Npn \lua_now:n #1   { \lua_now_x:n { \exp_not:n {#1} } }
\cs_new_protected:Npn \lua_shipout_x:n #1 { \luatex_latelua:D {#1} }
\cs_new_protected:Npn \lua_shipout:n #1
  { \lua_shipout_x:n { \exp_not:n {#1} } }
\cs_new:Npn \lua_escape_x:n #1 { \luatex_luaescapestring:D {#1} }
\cs_new:Npn \lua_escape:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }
\sys_if_engine_luatex:F
  {
    \clist_map_inline:nn
      { \lua_now_x:n , \lua_now:n , \lua_escape_x:n , \lua_escape:n }
      {
        \cs_set:Npn #1 ##1
          {
            \__msg_kernel_expandable_error:nnn
              { kernel } { luatex-required } { #1 }
          }
      }
    \clist_map_inline:nn
      { \lua_shipout_x:n , \lua_shipout:n }
      {
        \cs_set_protected:Npn #1 ##1
          {
            \__msg_kernel_error:nnn
              { kernel } { luatex-required } { #1 }
          }
      }
  }
```

(*End definition for* `\lua_now_x:n` *and others. These functions are documented on page 245.*)

43.2 Messages

```
\__msg_kernel_new:nnnn { kernel } { luatex-required }
  { LuaTeX~engine~not~in~use!~Ignoring~#1. }
  {
    The~feature~you~are~using~is~only~available~
    with~the~LuaTeX~engine.~LaTeX3~ignored~'#1'.
  }
⟨/tex⟩
```

43.3 Lua functions for internal use

```
⟨*lua⟩
```

Most of the emulation of pdfTEX here is based heavily on Heiko Oberdiek's pdftexcmds package.

l3kernel Create a table for the kernel's own use.

```
l3kernel = l3kernel or { }
```

(*End definition for* l3kernel.)

Local copies of global tables.

```
local io      = io
local kpse    = kpse
local lfs     = lfs
local math    = math
local md5     = md5
local os      = os
local string  = string
local tex     = tex
local unicode = unicode
```

Local copies of standard functions.

```
local abs        = math.abs
local byte       = string.byte
local floor      = math.floor
local format     = string.format
local gsub       = string.gsub
local kpse_find  = kpse.find_file
local lfs_attr   = lfs.attributes
local md5_sum    = md5.sum
local open       = io.open
local os_date    = os.date
local setcatcode = tex.setcatcode
local str_format = string.format
local sprint     = tex.sprint
local write      = tex.write
local utf8_char  = unicode.utf8.char
```

escapehex An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdftexcmds but is not currently required here.

```
local function escapehex(str)
  write((gsub(str, ".",
    function (ch) return format("%02X", byte(ch)) end)))
end
```

(*End definition for* escapehex.)

l3kernel.charcat Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see l3bootstrap) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

```
local charcat_table = l3kernel.charcat_table or 1
local function charcat(charcode, catcode)
  setcatcode(charcat_table, charcode, catcode)
  sprint(charcat_table, utf8_char(charcode))
end
l3kernel.charcat = charcat
```

(*End definition for* `l3kernel.charcat`.)

`l3kernel.filemdfivesum` Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see pdftexcmds and how it handles strings that have passed through LuaTEX).

```
local function filemdfivesum(name)
  local file =  kpse_find(name, "tex", true)
  if file then
    local f = open(file, "r")
    if f then
      local data = f:read("*a")
      escapehex(md5_sum(data))
      f:close()
    end
  end
end
l3kernel.filemdfivesum = filemdfivesum
```

(*End definition for* `l3kernel.filemdfivesum`.)

`l3kernel.filemoddate` See procedure `makepdftime` in `utils.c` of pdfTEX.

```
local function filemoddate(name)
  local file =  kpse_find(name, "tex", true)
  if file then
    local date = lfs_attr(file, "modification")
    if date then
      local d = os_date("*t", date)
      if d.sec >= 60 then
        d.sec = 59
      end
      local u = os_date("!*t", date)
      local off = 60 * (d.hour - u.hour) + d.min - u.min
      if d.year ~= u.year then
        if d.year > u.year then
          off = off + 1440
        else
          off = off - 1440
        end
      elseif d.yday ~= u.yday then
        if d.yday > u.yday then
          off = off + 1440
        else
          off = off - 1440
        end
      end
      local timezone
      if off == 0 then
        timezone = "Z"
      else
        local hours = floor(off / 60)
        local mins  = abs(off - hours * 60)
        timezone = str_format("%+03d", hours)
          .. "'" .. str_format("%02d", mins) .. "'"
      end
      write("D:"
        .. str_format("%04d", d.year)
        .. str_format("%02d", d.month)
        .. str_format("%02d", d.day)
        .. str_format("%02d", d.hour)
        .. str_format("%02d", d.min)
        .. str_format("%02d", d.sec)
        .. timezone)
    end
  end
end
l3kernel.filemoddate = filemoddate
```

(*End definition for* `l3kernel.filemoddate`.)

`l3kernel.filesize` A simple disk lookup.

```
local function filesize(name)
  local file =  kpse_find(name, "tex", true)
  if file then
    local size = lfs_attr(file, "size")
    if size then
      write(size)
    end
  end
end
l3kernel.filesize = filesize
```

(*End definition for* `l3kernel.filesize`.)

`l3kernel.strcmp` String comparison which gives the same results as pdfTEX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```
local function strcmp(A, B)
  if A == B then
    write("0")
  elseif A < B then
    write("-1")
  else
    write("1")
  end
end
l3kernel.strcmp = strcmp
```

(*End definition for* `l3kernel.strcmp`.)

43.4 Generic Lua and font support

```
⟨*initex⟩
```

A small amount of generic code is used by almost all LuaTEX material so needs to be loaded by the format.

```
attribute_count_name = "g__alloc_attribute_int"
bytecode_count_name  = "g__alloc_bytecode_int"
chunkname_count_name = "g__alloc_chunkname_int"
whatsit_count_name   = "g__alloc_whatsit_int"
require("ltluatex")
```

With the above available the font loader code used by plain TeX and LaTeX 2$_\varepsilon$ when used with LuaTeX can be loaded here. This is thus being treated more-or-less as part of the engine itself.

```
require("luaotfload-main")
local _void = luaotfload.main()
⟨/initex⟩
⟨/lua⟩
⟨/initex | package⟩
```

44 l3drivers Implementation

```
⟨*initex | package⟩
⟨@@=driver⟩
```

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

```
⟨*package⟩
\ProvidesExplFile
⟨*dvipdfmx⟩
  {l3dvidpfmx.def}{2017/03/18}{}
  {L3 Experimental driver: dvipdfmx}
⟨/dvipdfmx⟩
⟨*dvips⟩
  {l3dvips.def}{2017/03/18}{}
  {L3 Experimental driver: dvips}
⟨/dvips⟩
⟨*dvisvgm⟩
  {l3dvisvgm.def}{2017/03/18}{}
  {L3 Experimental driver: dvisvgm}
⟨/dvisvgm⟩
⟨*pdfmode⟩
  {l3pdfmode.def}{2017/03/18}{}
  {L3 Experimental driver: PDF mode}
⟨/pdfmode⟩
⟨*xdvipdfmx⟩
  {l3xdvidpfmx.def}{2017/03/18}{}
  {L3 Experimental driver: xdvipdfmx}
⟨/xdvipdfmx⟩
⟨/package⟩
```

The order of the driver code here is such that we get somewhat logical outcomes in terms of code sharing whilst keeping things readable. (Trying to mix all of the code by concept is almost unmanageable.) The key parts which are shared are

- Color support is either `dvips`-like or `pdfmode`-like.
- `pdfmode` and `(x)dvipdfmx` share drawing routines.
- `xdvipdfmx` is largely the same as `dvipdfmx` so takes most of the same code.

44.1 Color support

Whilst (x)dvipdfmx does have its own approach to color specials, it is easier to use dvips-like ones for all cases except direct PDF output. As such the color code is collected here in two blocks.

44.1.1 dvips-style

```
⟨*dvisvgm | dvipdfmx | dvips | xdvipdfmx⟩
```

__driver_color_pickup: Allow for $\LaTeX\ 2_{\varepsilon}$.

```
⟨*package⟩
\AtBeginDocument
  {
    \@ifpackageloaded { color }
      {
        \cs_new_protected:Npn \__driver_color_pickup:
          { \tl_set:Nx \l__color_current_tl { \current@color } }
      }
      { \cs_new_protected:Npn \__driver_color_pickup: { } }
  }
⟨/package⟩
```

(*End definition for* __driver_color_pickup:.)

__driver_color_ensure_current:
__driver_color_reset: Directly set the color using the specials: no optimisation here.

```
\cs_new_protected:Npn \__driver_color_ensure_current:
  {
⟨*package⟩
    \__driver_color_pickup:
⟨/package⟩
    \tex_special:D { color~push~\l__color_current_tl }
  }
\cs_new_protected:Npn \__driver_color_reset:
  { \tex_special:D { color~pop } }
```

(*End definition for* __driver_color_ensure_current: *and* __driver_color_reset:.)

```
⟨/dvisvgm | dvipdfmx | dvips | xdvipdfmx⟩
```

44.1.2 pdfmode

```
⟨*pdfmode⟩
```

__driver_color_pickup:
__driver_color_pickup_aux:w The current color in driver-dependent format: pick up the package-mode data if available. We end up converting back and forward in this route as we store our color data in dvips format. The \current@color needs to be x-expanded before __driver_color_pickup_aux:w breaks it apart, because for instance xcolor sets it to be instructions to generate a colour

```
⟨*package⟩
\AtBeginDocument
  {
    \@ifpackageloaded { color }
      {
        \cs_new_protected:Npn \__driver_color_pickup:
          {
            \exp_last_unbraced:Nx \__driver_color_pickup_aux:w
              { \current@color } ~ 0 ~ 0 ~ 0 \q_stop
          }
        \cs_new:Npn \__driver_color_pickup_aux:w #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 \q_stop
          {
            \tl_set:Nx \l__color_current_tl
              {
                \str_if_eq:nnTF {#2} { g }
                  { gray ~ #1 }
                  {
                    \str_if_eq:nnTF {#4} { rg }
                      { rgb ~ #1 ~ #2 ~ #3 }
                      {
                        \str_if_eq:nnTF {#5} { k }
                          { cmyk ~ #1 ~ #2 ~ #3 ~ #4 }
                          { gray ~ #1 }
                      }
                  }
              }
          }
      }
      { \cs_new_protected:Npn \__driver_color_pickup: { } }
  }
⟨/package⟩
```

(*End definition for* `\__driver_color_pickup:` *and* `\__driver_color_pickup_aux:w`.)

`\l__driver_color_stack_int`

pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```
\int_new:N \l__driver_color_stack_int
```

(*End definition for* `\l__driver_color_stack_int`.)

`\__driver_color_ensure_current:`
`\__driver_color_convert:w`
`\__driver_color_convert_gray:w`
`\__driver_color_convert_cmyk:w`
`\__driver_color_convert_rgb:w`
`\__driver_color_reset:`

There is a dedicated primitive/primitive interface for setting colors. As with scoping, this approach is not suitable for cached operations. Since we are using the `dvips` format to store color, there is a bit of work to correctly place it in the output.

```
\cs_new_protected:Npx \__driver_color_ensure_current:
  {
⟨*package⟩
    \exp_not:N \__driver_color_pickup:
⟨/package⟩
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D colorstack }
      { \pdftex_pdfcolorstack:D }
        \exp_not:N \l__driver_color_stack_int push
          {
            \exp_not:N \exp_after:wN
            \exp_not:N \__driver_color_convert:w
            \exp_not:N \l__color_current_tl
            \c_space_tl 0 ~ 0 ~ 0
            \exp_not:N \q_stop
          }
  }
\cs_new:Npn \__driver_color_convert:w #1 ~
```

```
  { \use:c { __driver_color_convert_ #1 :w } }
\cs_new:Npn \__driver_color_convert_gray:w #1 ~ #2 \q_stop
  { #1 ~ g ~ #1 ~ G }
\cs_new:Npn \__driver_color_convert_cmyk:w #1 ~ #2 ~ #3 ~ #4 ~#5 \q_stop
  { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
\cs_new:Npn \__driver_color_convert_rgb:w #1 ~ #2 ~ #3 ~ #4 \q_stop
  { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG }
\cs_new_protected:Npx \__driver_color_reset:
  {
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D colorstack }
      { \pdftex_pdfcolorstack:D }
        \exp_not:N \l__driver_color_stack_int pop \scan_stop:
  }
```

(*End definition for* `\__driver_color_ensure_current:` *and others.*)

```
⟨/pdfmode⟩
```

44.2 dvips driver

```
⟨*dvips⟩
```

44.2.1 Basics

`\__driver_literal:n`

In the case of dvips there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position "up front" and to move back to it at the end of the process.

```
\cs_new_protected:Npn \__driver_literal:n #1
  {
    \tex_special:D
      {
        ps:
          currentpoint~
          currentpoint~translate~
          #1 ~
          neg~exch~neg~exch~translate
      }
  }
```

(*End definition for* `\__driver_literal:n`.)

`\__driver_scope_begin:`
`\__driver_scope_end:`

Scope saving/restoring is done directly with no need to worry about the transformation matrix. General scoping is only for the graphics stack so the lower-cost gsave/grestore pair are used.

```
\cs_new_protected:Npn \__driver_scope_begin:
  { \tex_special:D { ps:gsave } }
\cs_new_protected:Npn \__driver_scope_end:
  { \tex_special:D { ps:grestore } }
```

(*End definition for* `\__driver_scope_begin:` *and* `\__driver_scope_end:`.)

44.3 Driver-specific auxiliaries

__driver_absolute_lengths:n

The dvips driver scales all absolute dimensions based on the output resolution selected and any TEX magnification. Thus for any operation involving absolute lengths there is a correction to make. This is based on normalscale from special.pro but using the stack rather than a definition to save the current matrix.

```
\cs_new:Npn \__driver_absolute_lengths:n #1
  {
     matrix~currentmatrix~
     Resolution~72~div~VResolution~72~div~scale~
     DVImag~dup~scale~
     #1 ~
     setmatrix
  }
```

(*End definition for* __driver_absolute_lengths:n.)

44.3.1 Box operations

__driver_box_use_clip:N

Much the same idea as for the PDF mode version but with a slightly different syntax for creating the clip path. To avoid any scaling issues we need the absolute length auxiliary here.

```
\cs_new_protected:Npn \__driver_box_use_clip:N #1
  {
    \__driver_scope_begin:
    \__driver_literal:n
      {
        \__driver_absolute_lengths:n
          {
            0 ~
            \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
            \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
            \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
            rectclip
          }
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
    \skip_horizontal:n { \box_wd:N #1 }
  }
```

(*End definition for* __driver_box_use_clip:N.)

__driver_box_use_rotate:Nn

Rotating using dvips does not require that the box dimensions are altered and has a very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is a quick test.

```
\cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
  {
    \__driver_scope_begin:
    \__driver_literal:n
      {
        \fp_compare:nNnTF {#2} = \c_zero_fp
          { 0 }
          { \fp_eval:n { round ( -#2 , 5 ) } } ~
        rotate
      }
   \box_use:N #1
   \__driver_scope_end:
  }
% \end{macro}
%
% \begin{macro}{\__driver_box_use_scale:Nnn}
%   The \texttt{dvips} driver once again has a dedicated operation we can
%   use here.
%    \begin{macrocode}
\cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
  {
    \__driver_scope_begin:
    \__driver_literal:n
      {
        \fp_eval:n { round ( #2 , 5 ) } ~
        \fp_eval:n { round ( #3 , 5 ) } ~
        scale
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
  }
```

(*End definition for* `\__driver_box_use_rotate:Nn`.)

44.4 Images

`\__driver_image_getbb_eps:n`

Simply use the generic function.

```
\cs_new_eq:NN \__driver_image_getbb_eps:n \__image_read_bb:n
```

(*End definition for* `\__driver_image_getbb_eps:n`.)

`\__driver_image_include_eps:n`

The special syntax is relatively clear here: remember we need PostScript sizes here.

```
\cs_new_protected:Npn \__driver_image_include_eps:n #1
  {
    \tex_special:D { PSfile = #1 }
  }
```

(*End definition for* `\__driver_image_include_eps:n`.)

44.5 Drawing

`\__driver_draw_literal:n`
`\__driver_draw_literal:x`

Literals with no positioning (using `ps:` each one is positioned but cut off from everything else, so no good for the stepwise approach needed here).

```
\cs_new_protected:Npn \__driver_draw_literal:n #1
  { \tex_special:D { ps:: ~ #1 } }
\cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* `\__driver_draw_literal:n`.)

__driver_draw_begin:
__driver_draw_end:

The `ps::[begin]` special here deals with positioning but allows us to continue on to a matching `ps::[end]`: contrast with `ps:`, which positions but where we can't split material between separate calls. The `@beginspecial`/`@endspecial` pair are from `special.pro` and correct the scale and y-axis direction. The reference point at the start of the box is saved (as `l3x`/`l3y`) as it is needed when inserting various items.

```
\cs_new_protected:Npn \__driver_draw_begin:
  {
    \tex_special:D { ps::[begin] }
    \tex_special:D { ps::~save }
    \tex_special:D { ps::~/l3x~currentpoint~/l3y~exch~def~def }
    \tex_special:D { ps::~@beginspecial }
  }
\cs_new_protected:Npn \__driver_draw_end:
  {
    \tex_special:D { ps::~@endspecial }
    \tex_special:D { ps::~restore }
    \tex_special:D { ps::[end] }
  }
```

(*End definition for* `\__driver_draw_begin:` *and* `\__driver_draw_end:`.)

__driver_draw_scope_begin:
__driver_draw_scope_end:

Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

```
\cs_new_protected:Npn \__driver_draw_scope_begin:
  { \__driver_draw_literal:n { save } }
\cs_new_protected:Npn \__driver_draw_scope_end:
  { \__driver_draw_literal:n { restore } }
```

(*End definition for* `\__driver_draw_scope_begin:` *and* `\__driver_draw_scope_end:`.)

__driver_draw_moveto:nn
__driver_draw_lineto:nn
__driver_draw_rectangle:nnnn
__driver_draw_curveto:nnnnnn

Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to `bp`. Notice that `x`-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```
\cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ moveto }
  }
\cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ lineto }
  }
\cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
  {
     \__driver_draw_literal:x
       {
         \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
         moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~closepath
       }
  }
\cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
  {
    \__driver_draw_literal:x
      {
        \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
        \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
        \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
        curveto
      }
 }
```

(*End definition for* `\__driver_draw_moveto:nn` *and others.*)

`\__driver_draw_evenodd_rule:`
`\__driver_draw_nonzero_rule:`
`\g__driver_draw_eor_bool`

The even-odd rule here can be implemented as a simply switch.

```
\cs_new_protected:Npn \__driver_draw_evenodd_rule:
  { \bool_gset_true:N \g__driver_draw_eor_bool }
\cs_new_protected:Npn \__driver_draw_nonzero_rule:
  { \bool_gset_false:N \g__driver_draw_eor_bool }
\bool_new:N \g__driver_draw_eor_bool
```

(*End definition for* `\__driver_draw_evenodd_rule:` , `\__driver_draw_nonzero_rule:` , *and* `\g__driver_draw_eor_bool`.)

`\__driver_draw_closepath:`
`\__driver_draw_stroke:`
`\__driver_draw_closestroke:`
`\__driver_draw_fill:`
`\__driver_draw_fillstroke:`
`\__driver_draw_clip:`
`\__driver_draw_discardpath:`
`\g__driver_draw_clip_bool`

Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is also desirable to have the `clip` keyword after a stroke or fill. To achieve those outcomes, there is some work to do. For color, if a stroke or fill color is defined it is used for the relevant operation, with a graphic scope inserted as required. That does mean that once such a color is set all further uses inside the same scope have to use scoping: see also the color set up functions. For clipping, the required ordering is achieved using a TeX switch. All of the operations end with a new path instruction as they do not terminate (again in contrast to PDF).

```
\cs_new_protected:Npn \__driver_draw_closepath:
  { \__driver_draw_literal:n { closepath } }
\cs_new_protected:Npn \__driver_draw_stroke:
  {
    \__driver_draw_literal:n { currentdict~/l3sc~known~{gsave~l3sc}~if }
    \__driver_draw_literal:n { stroke }
    \__driver_draw_literal:n { currentdict~/l3sc~known~{grestore}~if }
    \bool_if:NT \g__driver_draw_clip_bool
      {
        \__driver_draw_literal:x
          {
            \bool_if:NT \g__driver_draw_eor_bool { eo }
            clip
          }
      }
    \__driver_draw_literal:n { newpath }
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
\cs_new_protected:Npn \__driver_draw_closestroke:
  {
    \__driver_draw_closepath:
    \__driver_draw_stroke:
  }
\cs_new_protected:Npn \__driver_draw_fill:
  {
    \__driver_draw_literal:n { currentdict~/l3fc~known~{gsave~l3fc}~if }
    \__driver_draw_literal:x
      {
        \bool_if:NT \g__driver_draw_eor_bool { eo }
        fill
      }
    \__driver_draw_literal:n { currentdict~/l3fc~known~{grestore}~if }
    \bool_if:NT \g__driver_draw_clip_bool
      {
        \__driver_draw_literal:x
          {
            \bool_if:NT \g__driver_draw_eor_bool { eo }
            clip
          }
      }
    \__driver_draw_literal:n { newpath }
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
\cs_new_protected:Npn \__driver_draw_fillstroke:
  {
    \__driver_draw_literal:n { currentdict~/l3fc~known~{gsave~l3fc}~if }
    \__driver_draw_literal:x
      {
        \bool_if:NT \g__driver_draw_eor_bool { eo }
        fill
      }
    \__driver_draw_literal:n { currentdict~/l3fc~known~{grestore}~if }
    \__driver_draw_literal:n { currentdict~/l3sc~known~{gsave~l3sc}~if }
    \__driver_draw_literal:n { stroke }
    \__driver_draw_literal:n { currentdict~/l3sc~known~{grestore}~if }
    \bool_if:NT \g__driver_draw_clip_bool
      {
        \__driver_draw_literal:x
          {
            \bool_if:NT \g__driver_draw_eor_bool { eo }
            clip
          }
      }
    \__driver_draw_literal:n { newpath }
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
\cs_new_protected:Npn \__driver_draw_clip:
  { \bool_gset_true:N \g__driver_draw_clip_bool }
\bool_new:N \g__driver_draw_clip_bool
\cs_new_protected:Npn \__driver_draw_discardpath:
  {
    \bool_if:NT \g__driver_draw_clip_bool
      {
        \__driver_draw_literal:x
          {
            \bool_if:NT \g__driver_draw_eor_bool { eo }
            clip
          }
      }
    \__driver_draw_literal:n { newpath }
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
```

(*End definition for* `\__driver_draw_closepath:` *and others.*)

`\__driver_draw_dash:nn`
`\__driver_draw_dash:n`
`\__driver_draw_linewidth:n`
`\__driver_draw_miterlimit:n`
`\__driver_draw_cap_butt:`
`\__driver_draw_cap_round:`
`\__driver_draw_cap_rectangle:`
`\__driver_draw_join_miter:`
`\__driver_draw_join_round:`
`\__driver_draw_join_bevel:`

Converting paths to output is again a case of mapping directly to PostScript operations.

```
\cs_new_protected:Npn \__driver_draw_dash:nn #1#2
  {
    \__driver_draw_literal:x
      {
        [ ~
          \clist_map_function:nN {#1} \__driver_draw_dash:n
        ] ~
        \dim_to_decimal_in_bp:n {#2} ~ setdash
      }
  }
\cs_new:Npn \__driver_draw_dash:n #1
  { \dim_to_decimal_in_bp:n {#1} ~ }
\cs_new_protected:Npn \__driver_draw_linewidth:n #1
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
  }
\cs_new_protected:Npn \__driver_draw_miterlimit:n #1
  { \__driver_draw_literal:x { \fp_eval:n {#1} ~ setmiterlimit } }
\cs_new_protected:Npn \__driver_draw_cap_butt:
  { \__driver_draw_literal:n { 0 ~ setlinecap } }
\cs_new_protected:Npn \__driver_draw_cap_round:
  { \__driver_draw_literal:n { 1 ~ setlinecap } }
\cs_new_protected:Npn \__driver_draw_cap_rectangle:
  { \__driver_draw_literal:n { 2 ~ setlinecap } }
\cs_new_protected:Npn \__driver_draw_join_miter:
  { \__driver_draw_literal:n { 0 ~ setlinejoin } }
\cs_new_protected:Npn \__driver_draw_join_round:
  { \__driver_draw_literal:n { 1 ~ setlinejoin } }
\cs_new_protected:Npn \__driver_draw_join_bevel:
  { \__driver_draw_literal:n { 2 ~ setlinejoin } }
```

(*End definition for* `\__driver_draw_dash:nn` *and others.*)

`\__driver_draw_color_reset:`
`\__driver_draw_color_cmyk:nnnn`
`\__driver_draw_color_cmyk_fill:nnnn`
`\__driver_draw_color_cmyk_stroke:nnnn`
`\__driver_draw_color_gray:n`
`\__driver_draw_color_gray_fill:n`
`\__driver_draw_color_gray_stroke:n`
`\__driver_draw_color_rgb:nnn`
`\__driver_draw_color_rgb_fill:nnn`
`\__driver_draw_color_rgb_stroke:nnn`

To allow color to be defined for strokes and fills separately and to respect scoping, the data needs to be stored at the PostScript level. We cannot undefine (local) fill/stroke colors once set up but we can set them blank to improve performance slightly.

```
\cs_new_protected:Npn \__driver_draw_color_reset:
  {
    \__driver_draw_literal:n { currentdic~/l3fc~known~{ /l3fc~ { } ~def }~if }
    \__driver_draw_literal:n { currentdic~/l3sc~known~{ /l3sc~ { } ~def }~if }
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
  {
    \__driver_draw_literal:x
      {
        \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
        \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
        setcmykcolor ~
      }
    \__driver_draw_color_reset:
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
  {
    \__driver_draw_literal:x
      {
        /l3fc ~
          {
            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
            \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
            setcmykcolor
          } ~
        def
      }
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
  {
    \__driver_draw_literal:x
      {
        /l3sc ~
          {
            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
            \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
            setcmykcolor
          } ~
        def
      }
  }
\cs_new_protected:Npn \__driver_draw_color_gray:n #1
  {
    \__driver_draw_literal:x { fp_eval:n {#1} ~ setgray  }
    \__driver_draw_color_reset:
  }
\cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
  { \__driver_draw_literal:x { /l3fc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
\cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
  { \__driver_draw_literal:x { /l3sc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
\cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
  {
    \__driver_draw_literal:x
      {
        \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
        setrgbcolor
      }
    \__driver_draw_color_reset:
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
  {
    \__driver_draw_literal:x
      {
        /l3fc ~
          {
            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
            setrgbcolor
          } ~
        def
      }
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
  {
    \__driver_draw_literal:x
      {
        /l3sc ~
          {
            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
            setrgbcolor
          } ~
        def
      }
  }
```

(*End definition for* `\__driver_draw_color_reset:` *and others.*)

`\__driver_draw_transformcm:nnnnnn`

The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```
\cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
  {
    \__driver_draw_literal:x
      {
        [
          \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
          \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
          \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
        ] ~
        concat
      }
  }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn`.)

`\__driver_draw_hbox:Nnnnnnn`

Inside a picture `@beginspecial`/`@endspecial` are active, which is normally a good thing but means that the position and scaling would be off if the box was inserted directly. Instead, we need to reverse the effect of the (normally desirable) shift/scaling within the box. That requires knowing where the reference point for the drawing is: saved as `l3x`/`l3y` at the start of the picture. Transformation here is relative to the drawing origin so has to be done purely in driver code not using TEX offsets.

```
\cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
  {
    \__driver_scope_begin:
    \tex_special:D { ps::[end] }
    \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
    \tex_special:D { ps::~72~Resolution~div~72~VResolution~div~neg~scale }
    \tex_special:D { ps::~magscale~{1~DVImag~div~dup~scale}~if }
    \tex_special:D { ps::~l3x~neg~l3y~neg~translate }
    \box_set_wd:Nn #1 { 0pt }
    \box_set_ht:Nn #1 { 0pt }
    \box_set_dp:Nn #1 { 0pt }
    \box_use:N #1
    \tex_special:D { ps::[begin] }
    \__driver_scope_end:
  }
```

(*End definition for* `\__driver_draw_hbox:Nnnnnnn`.)

```
⟨/dvips⟩
```

44.6 pdfmode driver

```
⟨*pdfmode⟩
```

The direct PDF driver covers both pdfTEX and LuaTEX. The latter renames/restructures the driver primitives but this can be handled at one level of abstraction. As such, we avoid using two separate drivers for this material at the cost of some x-type definitions to get everything expanded up-front.

44.6.1 Basics

`\__driver_literal:n`

This is equivalent to `\special{pdf:}` but the engine can track it. Without the `direct` keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (`BT` ... `ET` block).

```
\cs_new_protected:Npx \__driver_literal:n #1
  {
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D literal }
      { \pdftex_pdfliteral:D }
        {#1}
  }
```

(*End definition for* `\__driver_literal:n`.)

`\__driver_scope_begin:`
`\__driver_scope_end:`

Higher-level interfaces for saving and restoring the graphic state.

```
\cs_new_protected:Npx \__driver_scope_begin:
  {
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D save \scan_stop: }
      { \pdftex_pdfsave:D }
  }
\cs_new_protected:Npx \__driver_scope_end:
  {
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D restore \scan_stop: }
      { \pdftex_pdfrestore:D }
  }
```

(*End definition for* `\__driver_scope_begin:` *and* `\__driver_scope_end:`.)

`\__driver_matrix:n` Here the appropriate function is set up to insert an affine matrix into the PDF. With pdfTEX and LuaTEX in direct PDF output mode there is a primitive for this, which only needs the rotation/scaling/skew part.

```
\cs_new_protected:Npx \__driver_matrix:n #1
  {
    \cs_if_exist:NTF \luatex_pdfextension:D
      { \luatex_pdfextension:D setmatrix }
      { \pdftex_pdfsetmatrix:D }
        {#1}
  }
```

(*End definition for* `\__driver_matrix:n`.)

44.6.2 Box operations

`\__driver_box_use_clip:N` The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The “real” width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here shares as much code as possible and uses the same conversions (and so same rounding errors) in all cases.

```
\cs_new_protected:Npn \__driver_box_use_clip:N #1
  {
    \__driver_scope_begin:
    \__driver_literal:n
      {
        0~
        \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
        \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
        \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
        re~W~n
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
    \skip_horizontal:n { \box_wd:N #1 }
  }
```

(*End definition for* `\__driver_box_use_clip:N`.)

`\__driver_box_use_rotate:Nn`
`\l__driver_cos_fp`
`\l__driver_sin_fp`

Rotations are set using an affine transformation matrix which therefore requires sine/cosine values not the angle itself. We store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that `-0` is not written to the output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```
\cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
  {
    \__driver_scope_begin:
    \box_set_wd:Nn #1 { 0pt }
    \fp_set:Nn \l__driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
    \fp_compare:nNnT \l__driver_cos_fp = \c_zero_fp
      { \fp_zero:N \l__driver_cos_fp }
    \fp_set:Nn \l__driver_sin_fp { round ( sind ( #2 ) , 5 ) }
    \__driver_matrix:n
```

```
      {
        \fp_use:N \l__driver_cos_fp \c_space_tl
        \fp_compare:nNnTF \l__driver_sin_fp = \c_zero_fp
          { 0~0 }
          {
            \fp_use:N \l__driver_sin_fp
            \c_space_tl
            \fp_eval:n { -\l__driver_sin_fp }
          }
        \c_space_tl
        \fp_use:N \l__driver_cos_fp
      }
   \box_use:N #1
   \__driver_scope_end:
  }
\fp_new:N \l__driver_cos_fp
\fp_new:N \l__driver_sin_fp
```

(*End definition for* __driver_box_use_rotate:Nn , \l__driver_cos_fp , *and* \l__driver_sin_fp.)

__driver_box_use_scale:Nnn

The same idea as for rotation but without the complexity of signs and cosines.

```
\cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
  {
    \__driver_scope_begin:
    \__driver_matrix:n
      {
        \fp_eval:n { round ( #2 , 5 ) } ~
        0~0~
        \fp_eval:n { round ( #3 , 5 ) }
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
  }
```

(*End definition for* __driver_box_use_scale:Nnn.)

44.7 Images

\l__driver_image_attr_tl

In PDF mode, additional attributes of an image (such as page number) are needed both to obtain the bounding box and when inserting the image: this occurs as the image dictionary approach means they are read as part of the bounding box operation. As such, it is easier to track additional attributes using a dedicated tl rather than build up the same data twice.

```
\tl_new:N \l__driver_image_attr_tl
```

(*End definition for* \l__driver_image_attr_tl.)

__driver_image_getbb_jpg:n
__driver_image_getbb_pdf:n
__driver_image_getbb_png:n
__driver_image_getbb_auxi:n
__driver_image_getbb_auxii:n

Getting the bounding box here requires us to box up the image and measure it. To deal with the difference in feature support in bitmap and vector images but keeping the common parts, there is a little work to do in terms of auxiliaries. The key here is to notice that we need two forms of the attributes: a "short" set to allow us to track for caching, and the full form to pass to the primitive.

```
\cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
  {
    \int_zero:N \l__image_page_int
    \tl_clear:N \l__image_pagebox_tl
    \tl_set:Nx \l__driver_image_attr_tl
      {
        \tl_if_empty:NF \l__image_decode_tl
          { :D \l__image_decode_tl }
        \bool_if:NT \l__image_interpolate_bool
          { :I }
      }
    \tl_clear:N \l__driver_image_attr_tl
    \__driver_image_getbb_auxi:n {#1}
  }
\cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
\cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
  {
    \tl_clear:N \l__image_decode_tl
    \bool_set_false:N \l__image_interpolate_bool
    \tl_set:Nx \l__driver_image_attr_tl
      {
        : \l__image_pagebox_tl
        \int_compare:nNnT \l__image_page_int > 1
          { :P \int_use:N \l__image_page_int }
      }
    \__driver_image_getbb_auxi:n {#1}
  }
\cs_new_protected:Npn \__driver_image_getbb_auxi:n #1
  {
    \dim_zero:N \l__image_llx_dim
    \dim_zero:N \l__image_lly_dim
    \dim_if_exist:cTF { c__image_ #1 \l__driver_image_attr_tl _urx_dim }
      {
        \dim_set_eq:Nc \l__image_urx_dim
          { c__image_ #1 \l__driver_image_attr_tl _urx_dim }
        \dim_set_eq:Nc \l__image_ury_dim
          { c__image_ #1 \l__driver_image_attr_tl _ury_dim }
      }
      { \__driver_image_getbb_auxii:n {#1} }
  }
%    \begin{macrocode}
%   Measuring the image is done by boxing up: for PDF images we could
%   use |\pdftex_pdfximagebbox:D|, but if doesn't work for other types.
%   As the box always starts at $(0,0)$ there is no need to worry about
%   the lower-left position.
%    \begin{macrocode}
\cs_new_protected:Npn \__driver_image_getbb_auxii:n #1
  {
    \tex_immediate:D \pdftex_pdfximage:D
      \bool_lazy_or:nnT
        { \l__image_interpolate_bool }
        { ! \tl_if_empty_p:N \l__image_decode_tl }
        {
          attr ~
            {
              \tl_if_empty:NF \l__image_decode_tl
```

```
                { /Decode~[ \l__image_decode_tl ] }
              \bool_if:NT \l__image_interpolate_bool
                { /Interpolate~true }
            }
        }
      \int_compare:nNnT \l__image_page_int > 0
        { page ~ \int_use:N \l__image_page_int }
      \tl_if_empty:NF \l__image_pagebox_tl
        { \l__image_pagebox_tl }
      {#1}
    \hbox_set:Nn \l__image_tmp_box
      { \pdftex_pdfrefximage:D \pdftex_pdflastximage:D }
    \dim_set:Nn \l__image_urx_dim { \box_wd:N \l__image_tmp_box }
    \dim_set:Nn \l__image_ury_dim { \box_ht:N \l__image_tmp_box }
    \int_const:cn { c__image_ #1 \l__driver_image_attr_tl _int }
      { \tex_the:D \pdftex_pdflastximage:D }
    \dim_const:cn { c__image_ #1 \l__driver_image_attr_tl _urx_dim }
      { \l__image_urx_dim }
    \dim_const:cn { c__image_ #1 \l__driver_image_attr_tl _ury_dim }
      { \l__image_ury_dim }
  }
```

(*End definition for* `\__driver_image_getbb_jpg:n` *and others.*)

`\__driver_image_include_jpg:n`
`\__driver_image_include_pdf:n`
`\__driver_image_include_png:n`

Images are already loaded for the measurement part of the code, so inclusion is straightforward, with only any attributes to worry about. The latter carry through from determination of the bounding box.

```
\cs_new_protected:Npn \__driver_image_include_jpg:n #1
  {
    \pdftex_pdfrefximage:D
      \int_use:c { c__image_ #1 \l__driver_image_attr_tl _int }
  }
\cs_new_eq:NN \__driver_image_include_pdf:n \__driver_image_include_jpg:n
\cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n
```

(*End definition for* `\__driver_image_include_jpg:n`*,* `\__driver_image_include_pdf:n`*, and* `\__driver_image_include_png:n`*.*)

```
⟨/pdfmode⟩
```

44.8 dvipdfmx driver

```
⟨*dvipdfmx | xdvipdfmx⟩
```

The `dvipdfmx` shares code with the PDF mode one (using the common section to this file) but also with `xdvipdfmx`. The latter is close to identical to `dvipdfmx` and so all of the code here is extracted for both drivers, with some `clean up` for `xdvipdfmx` as required.

44.8.1 Basics

`\__driver_literal:n`

Equivalent to `pdf:content` but favored as the link to the pdfTEX primitive approach is clearer. Some higher-level operations use `\tex_special:D` directly: see the later comments on where this is useful.

```
\cs_new_protected:Npn \__driver_literal:n #1
  { \tex_special:D { pdf:literal~ #1 } }
```

(*End definition for* `\__driver_literal:n`.)

`\__driver_scope_begin:`
`\__driver_scope_end:`

Scoping is done using the driver-specific specials.

```
\cs_new_protected:Npn \__driver_scope_begin:
  { \tex_special:D { x:gsave } }
\cs_new_protected:Npn \__driver_scope_end:
  { \tex_special:D { x:grestore } }
```

(*End definition for* `\__driver_scope_begin:` *and* `\__driver_scope_end:`.)

44.8.2 Box operations

`\__driver_box_use_clip:N`

The code here is idential to that for `pdfmode`: unlike rotation and scaling, there is no higher-level support in the driver for clipping.

```
\cs_new_protected:Npn \__driver_box_use_clip:N #1
  {
    \__driver_scope_begin:
    \__driver_literal:n
      {
        0~
        \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
        \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
        \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
        re~W~n
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
    \skip_horizontal:n { \box_wd:N #1 }
  }
```

(*End definition for* `\__driver_box_use_clip:N`.)

`\__driver_box_use_rotate:Nn`

Rotating in `(x)`dvipdmfx can be implemented using either PDF or driver-specific code. The former approach however is not "aware" of the content of boxes: this means that any embedded links would not be adjusted by the rotation. As such, the driver-native approach is prefered: the code therefore is similar (though not identical) to the `dvips` version (notice the rotation angle here is positive). As for `dvips`, zero rotation is written as `0` not `-0`.

```
\cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
  {
    \__driver_scope_begin:
    \tex_special:D
      {
        x:rotate~
        \fp_compare:nNnTF {#2} = \c_zero_fp
          { 0 }
          { \fp_eval:n { round ( #2 , 5 ) } }
      }
    \box_use:N #1
    \__driver_scope_end:
  }
```

(*End definition for* `\__driver_box_use_rotate:Nn`.)

__driver_box_use_scale:Nnn

Much the same idea for scaling: use the higher-level driver operation to allow for box content.

```
\cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
  {
    \__driver_scope_begin:
    \tex_special:D
      {
        x:scale~
        \fp_eval:n { round ( #2 , 5 ) } ~
        \fp_eval:n { round ( #3 , 5 ) }
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
  }
```

(*End definition for* __driver_box_use_scale:Nnn.)

44.9 Images

__driver_image_getbb_eps:n
__driver_image_getbb_jpg:n
__driver_image_getbb_pdf:n
__driver_image_getbb_png:n

Simply use the generic functions: only for dvipdfmx in the extraction cases.

```
\cs_new_eq:NN \__driver_image_getbb_eps:n \__image_read_bb:n
⟨*dvipdfmx⟩
\cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
  {
    \int_zero:N \l__image_page_int
    \tl_clear:N \l__image_pagebox_tl
    \__image_extract_bb:n {#1}
  }
\cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
\cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
  {
    \tl_clear:N \l__image_decode_tl
    \bool_set_false:N \l__image_interpolate_bool
    \__image_extract_bb:n {#1}
  }
⟨/dvipdfmx⟩
```

(*End definition for* __driver_image_getbb_eps:n *and others.*)

\g__driver_image_int

Used to track the object number associated with each image.

```
\int_new:N \g__driver_image_int
```

(*End definition for* \g__driver_image_int.)

__driver_image_include_eps:n
__driver_image_include_jpg:n
__driver_image_include_pdf:n
__driver_image_include_png:n
__driver_image_include_auxi:nn
__driver_image_include_auxii:nnn
__driver_image_include_auxii:xnn
__driver_image_include_auxiii:nn

The special syntax depends on the file type. There is a difference in how PDF images are best handled between dvipdfmx and xdvipdfmx: for the latter it is better to use the primitive route. The relevant code for that is included later in this file.

```
\cs_new_protected:Npn \__driver_image_include_eps:n #1
  {
    \tex_special:D { PSfile = #1 }
  }
\cs_new_protected:Npn \__driver_image_include_jpg:n #1
  { \__driver_image_include_auxi:nn {#1} { image } }
\cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n
```

```
⟨*dvipdfmx⟩
\cs_new_protected:Npn \__driver_image_include_pdf:n #1
  { \__driver_image_include_auxi:nn {#1} { epdf } }
⟨/dvipdfmx⟩
```

Image inclusion is set up to use the fact that each image is stored in the PDF as an XObject. This means that we can include repeated images only once and refer to them. To allow that, track the nature of each image: much the same as for the direct PDF mode case.

```
\cs_new_protected:Npn \__driver_image_include_auxi:nn #1#2
  {
    \__driver_image_include_auxii:xnn
      {
        \tl_if_empty:NF \l__image_pagebox_tl
          { : \l__image_pagebox_tl }
        \int_compare:nNnT \l__image_page_int > 1
          { :P \int_use:N \l__image_page_int }
        \tl_if_empty:NF \l__image_decode_tl
          { :D \l__image_decode_tl }
        \bool_if:NT \l__image_interpolate_bool
           { :I }
      }
      {#1} {#2}
  }
\cs_new_protected:Npn \__driver_image_include_auxii:nnn #1#2#3
  {
    \int_if_exist:cTF { c__image_ #2#1 _int }
      {
        \tex_special:D
          { pdf:usexobj~@image \int_use:c { c__image_ #2#1 _int } }
      }
      { \__driver_image_include_auxiii:nn {#2} {#1} {#3} }
  }
\cs_generate_variant:Nn \__driver_image_include_auxii:nnn { x }
```

Inclusion using the specials is relatively straight-forward, but there is one wrinkle. To get the pagebox correct for PDF images in all cases, it is necessary to provide both that information and the bbox argument: odd things happen otherwise!

```
\cs_new_protected:Npn \__driver_image_include_auxiii:nnn #1#2#3
  {
    \int_gincr:N \g__driver_image_int
    \int_const:cn { c__image_ #1#2 _int } { \g__driver_image_int }
    \tex_special:D
      {
        pdf:#3~
        @image \int_use:c { c__image_ #1#2 _int }
        \int_compare:nNnT \l__image_page_int > 1
          { page ~ \int_use:N \l__image_page_int \c_space_tl }
        \tl_if_empty:NF \l__image_pagebox_tl
          {
            pagebox ~ \l__image_pagebox_tl \c_space_tl
            bbox ~
              \dim_to_decimal_in_bp:n \l__image_llx_dim \c_space_tl
              \dim_to_decimal_in_bp:n \l__image_lly_dim \c_space_tl
              \dim_to_decimal_in_bp:n \l__image_urx_dim \c_space_tl
              \dim_to_decimal_in_bp:n \l__image_ury_dim \c_space_tl
          }
        (#1)
        \bool_lazy_or:nnT
          { \l__image_interpolate_bool }
          { ! \tl_if_empty_p:N \l__image_decode_tl }
          {
            <<
              \tl_if_empty:NF \l__image_decode_tl
                { /Decode~[ \l__image_decode_tl ] }
              \bool_if:NT \l__image_interpolate_bool
                { /Interpolate~true> }
            >>
          }
      }
  }
```

(*End definition for* `\__driver_image_include_eps:n` *and others.*)

⟨/dvipdfmx | xdvipdfmx⟩

44.10 xdvipdfmx driver

⟨*xdvipdfmx⟩

44.11 Images

`\__driver_image_getbb_jpg:n`
`\__driver_image_getbb_pdf:n`
`\__driver_image_getbb_png:n`
`\__driver_image_getbb_auxi:nN`
`\__driver_image_getbb_auxii:nnN`
`\__driver_image_getbb_auxii:VnN`
`\__driver_image_getbb_auxiii:nNnn`
`\__driver_image_getbb_auxiv:nnNnn`
`\__driver_image_getbb_auxiv:VnNnn`
`\__driver_image_getbb_auxv:nNnn`
`\__driver_image_getbb_auxv:nNnn`
`\__driver_image_getbb_pagebox:w`

For xdvipdfmx, there are two primitives that allow us to obtain the bounding box without needing extractbb. The only complexity is passing the various minor variations to a common core process. The XƎTEX primitive omits the text box from the page box specification, so there is also some “trimming” to do here.

```
\cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
  {
    \int_zero:N \l__image_page_int
    \tl_clear:N \l__image_pagebox_tl
    \__driver_image_getbb_auxi:nN {#1} \xetex_picfile:D
  }
\cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
\cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
  {
    \tl_clear:N \l__image_decode_tl
    \bool_set_false:N \l__image_interpolate_bool
    \__driver_image_getbb_auxi:nN {#1} \xetex_pdffile:D
  }
\cs_new_protected:Npn \__driver_image_getbb_auxi:nN #1#2
  {
    \int_compare:nNnTF \l__image_page_int > 1
      { \__driver_image_getbb_auxii:VnN \l__image_page_int {#1} #2  }
      { \__driver_image_getbb_auxiii:nNnn {#1} #2 }
  }
\cs_new_protected:Npn \__driver_image_getbb_auxii:nnN #1#2#3
  { \__driver_image_getbb_aux:nNnn {#2} #3 { :P #1 } { page #1 } }
\cs_generate_variant:Nn \__driver_image_getbb_auxii:nnN { V }
\cs_new_protected:Npn \__driver_image_getbb_auxiii:nNnn #1#2#3#4
```

```
  {
    \tl_if_empty:NTF \l__image_pagebox_tl
      { \__driver_image_getbb_auxiv:VnNnn \l__image_pagebox_tl }
      { \__driver_image_getbb_auxv:nNnn }
      {#1} #2 {#3} {#4}
  }
\cs_new_protected:Npn \__driver_image_getbb_auxiv:nnNnn #1#2#3#4#5
  {
    \use:x
      {
        \__driver_image_getbb_auxv:nNnn {#2} #3 { : #1 #4 }
          { #5 ~ \__driver_image_getbb_pagebox:w #1 }
      }
  }
\cs_generate_variant:Nn \__driver_image_getbb_auxiv:nnNnn { V }
\cs_new_protected:Npn \__driver_image_getbb_auxv:nNnn #1#2#3#4
  {
    \dim_zero:N \l__image_llx_dim
    \dim_zero:N \l__image_lly_dim
    \dim_if_exist:cTF { c__image_ #1#3 _urx_dim }
      {
        \dim_set_eq:Nc \l__image_urx_dim { c__image_ #1#3 _urx_dim }
        \dim_set_eq:Nc \l__image_ury_dim { c__image_ #1#3 _ury_dim }
      }
      { \__driver_image_getbb_auxvi:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new_protected:Npn \__driver_image_getbb_auxvi:nNnn #1#2#3#4
  {
    \hbox_set:Nn \l__image_tmp_box { #2 #1 ~ #4 }
    \dim_set:Nn \l__image_utx_dim { \box_wd:N \l__image_tmp_box }
    \dim_set:Nn \l__image_ury_dim { \box_ht:N \l__image_tmp_box }
    \dim_const:cn { c__image_ #1#3 _urx_dim }
      { \l__image_urx_dim }
    \dim_const:cn { c__image_ #1#3 _ury_dim }
      { \l__image_ury_dim }
  }
\cs_new:Npn \__driver_image_getbb_pagebox:w #1 box {#1}
```

(*End definition for* `\__driver_image_getbb_jpg:n` *and others.*)

`\__driver_image_include_pdf:n` For PDF images, properly supporting the `pagebox` concept in XeTeX is best done using the `\xetex_pdffile:D` primitive. The syntax here is the same as for the image measurement part, although we know at this stage that there must be some valid setting for `\l__image_pagebox_tl`.

```
\cs_new_protected:Npn \__driver_image_include_pdf:n #1
  {
    \xetex_pdffile:D "#1" ~
      \int_compare:nNnT \l__image_page_int > 0
        { page~ \int_use:N \l__image_page_int }
      \__driver_image_getbb_auxiv:VnNnn \l__image_pagebox_tl
  }
```

(*End definition for* `\__driver_image_include_pdf:n`.)

⟨/xdvipdfmx⟩

44.12 Drawing commands: `pdfmode` and `(x)dvipdfmx`

Both `pdfmode` and `(x)dvipdfmx` directly produce PDF output and understand a shared set of specials for drawing commands.

```
⟨*dvipdfmx | pdfmode | xdvipdfmx⟩
```

44.13 Drawing

__driver_draw_literal:n
__driver_draw_literal:x

Pass data through using a dedicated interface.

```
\cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
\cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* __driver_draw_literal:n.)

__driver_draw_begin:
__driver_draw_end:

No special requirements here, so simply set up a drawing scope.

```
\cs_new_protected:Npn \__driver_draw_begin:
  { \__driver_draw_scope_begin: }
\cs_new_protected:Npn \__driver_draw_end:
  { \__driver_draw_scope_end: }
```

(*End definition for* __driver_draw_begin: *and* __driver_draw_end:.)

__driver_draw_scope_begin:
__driver_draw_scope_end:

In contrast to a general scope, a drawing scope is always done using the PDF operators so is the same for all relevant drivers.

```
\cs_new_protected:Npn \__driver_draw_scope_begin:
  { \__driver_draw_literal:n { q } }
\cs_new_protected:Npn \__driver_draw_scope_end:
  { \__driver_draw_literal:n { Q } }
```

(*End definition for* __driver_draw_scope_begin: *and* __driver_draw_scope_end:.)

__driver_draw_moveto:nn
__driver_draw_lineto:nn
__driver_draw_curveto:nnnnnn
__driver_draw_rectangle:nnnn

Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to `bp`. Notice that `x`-type expansion is included here to ensure that any variable values are forced to literals before any possible caching.

```
\cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ m }
  }
\cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ l }
  }
\cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
  {
    \__driver_draw_literal:x
      {
        \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
        \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
        \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
        c
      }
 }
```

```
\cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
  {
     \__driver_draw_literal:x
      {
        \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
        \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
        re
      }
  }
```

(*End definition for* `\__driver_draw_moveto:nn` *and others.*)

`\__driver_draw_evenodd_rule:`
`\__driver_draw_nonzero_rule:`
`\g__driver_draw_eor_bool`

The even-odd rule here can be implemented as a simply switch.

```
\cs_new_protected:Npn \__driver_draw_evenodd_rule:
  { \bool_gset_true:N \g__driver_draw_eor_bool }
\cs_new_protected:Npn \__driver_draw_nonzero_rule:
  { \bool_gset_false:N \g__driver_draw_eor_bool }
\bool_new:N \g__driver_draw_eor_bool
```

(*End definition for* `\__driver_draw_evenodd_rule:`*,* `\__driver_draw_nonzero_rule:`*, and* `\g__driver_draw_eor_bool`*.*)

`\__driver_draw_closepath:`
`\__driver_draw_stroke:`
`\__driver_draw_closestroke:`
`\__driver_draw_fill:`
`\__driver_draw_fillstroke:`
`\__driver_draw_clip:`
`\__driver_draw_discardpath:`

Converting paths to output is again a case of mapping directly to PDF operations.

```
\cs_new_protected:Npn \__driver_draw_closepath:
  { \__driver_draw_literal:n { h } }
\cs_new_protected:Npn \__driver_draw_stroke:
  { \__driver_draw_literal:n { S } }
\cs_new_protected:Npn \__driver_draw_closestroke:
  { \__driver_draw_literal:n { s } }
\cs_new_protected:Npn \__driver_draw_fill:
  {
    \__driver_draw_literal:x
      { f \bool_if:NT \g__driver_draw_eor_bool * }
  }
\cs_new_protected:Npn \__driver_draw_fillstroke:
  {
    \__driver_draw_literal:x
      { B \bool_if:NT \g__driver_draw_eor_bool * }
  }
\cs_new_protected:Npn \__driver_draw_clip:
  {
    \__driver_draw_literal:x
      { W \bool_if:NT \g__driver_draw_eor_bool * }
  }
\cs_new_protected:Npn \__driver_draw_discardpath:
  { \__driver_draw_literal:n { n } }
```

(*End definition for* `\__driver_draw_closepath:` *and others.*)

`\__driver_draw_dash:nn`
`\__driver_draw_dash:n`
`\__driver_draw_linewidth:n`
`\__driver_draw_miterlimit:n`
`\__driver_draw_cap_butt:`
`\__driver_draw_cap_round:`
`\__driver_draw_cap_rectangle:`
`\__driver_draw_join_miter:`
`\__driver_draw_join_round:`
`\__driver_draw_join_bevel:`

Converting paths to output is again a case of mapping directly to PDF operations.

```
\cs_new_protected:Npn \__driver_draw_dash:nn #1#2
  {
    \__driver_draw_literal:x
      {
        [ ~
```

```
          \clist_map_function:nN {#1} \__driver_draw_dash:n
        ] ~
        \dim_to_decimal_in_bp:n {#2} ~ d
      }
  }
\cs_new:Npn \__driver_draw_dash:n #1
  { \dim_to_decimal_in_bp:n {#1} ~ }
\cs_new_protected:Npn \__driver_draw_linewidth:n #1
  {
    \__driver_draw_literal:x
      { \dim_to_decimal_in_bp:n {#1} ~ w }
  }
\cs_new_protected:Npn \__driver_draw_miterlimit:n #1
  { \__driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
\cs_new_protected:Npn \__driver_draw_cap_butt:
  { \__driver_draw_literal:n { 0 ~ J } }
\cs_new_protected:Npn \__driver_draw_cap_round:
  { \__driver_draw_literal:n { 1 ~ J } }
\cs_new_protected:Npn \__driver_draw_cap_rectangle:
  { \__driver_draw_literal:n { 2 ~ J } }
\cs_new_protected:Npn \__driver_draw_join_miter:
  { \__driver_draw_literal:n { 0 ~ j } }
\cs_new_protected:Npn \__driver_draw_join_round:
  { \__driver_draw_literal:n { 1 ~ j } }
\cs_new_protected:Npn \__driver_draw_join_bevel:
  { \__driver_draw_literal:n { 2 ~ j } }
```

(*End definition for* `\__driver_draw_dash:nn` *and others.*)

__driver_draw_color_cmyk:nnnn
__driver_draw_color_cmyk_fill:nnnn
__driver_draw_color_cmyk_stroke:nnnn
__driver_draw_color_cmyk_aux:nnnn
__driver_draw_color_gray:n
__driver_draw_color_gray_fill:n
__driver_draw_color_gray_stroke:n
__driver_draw_color_gray_aux:n
__driver_draw_color_rgb:nnn
__driver_draw_color_rgb_fill:nnn
__driver_draw_color_rgb_stroke:nnn
__driver_draw_color_rgb_aux:nnn

Yet more fast conversion, all using the FPU to allow for expressions in numerical input.

```
\cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
  {
    \use:x
      {
        \__driver_draw_color_cmyk_aux:nnnn
          { \fp_eval:n {#1} }
          { \fp_eval:n {#2} }
          { \fp_eval:n {#3} }
          { \fp_eval:n {#4} }
      }
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk_aux:nnnn #1#2#3#4
  {
    \__driver_draw_literal:n
      { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
  {
    \__driver_draw_literal:x
      {
        \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
        \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
        k
      }
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
  {
    \__driver_draw_literal:x
      {
        \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
        \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
        K
      }
  }
\cs_new_protected:Npn \__driver_draw_color_gray:n #1
  {
    \use:x
      { \__driver_draw_color_gray_aux:n { \fp_eval:n {#1} } }
  }
\cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
  {
    \__driver_draw_literal:n { #1 ~ g ~ #1 ~ G }
  }
\cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
  { \__driver_draw_literal:x { \fp_eval:n {#1} ~ g } }
\cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
  { \__driver_draw_literal:x { \fp_eval:n {#1} ~ G } }
\cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
  {
    \use:x
      {
        \__driver_draw_color_rgb_aux:nnn
          { \fp_eval:n {#1} }
          { \fp_eval:n {#2} }
          { \fp_eval:n {#3} }
      }
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_aux:nnn #1#2#3
  {
    \__driver_draw_literal:n
      { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG }
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
  {
    \__driver_draw_literal:x
      { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
  {
    \__driver_draw_literal:x
      { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }
  }
```

(*End definition for* `\__driver_draw_color_cmyk:nnnn` *and others.*)

`\__driver_draw_transformcm:nnnnnn` The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```
\cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
```

```
  {
    \__driver_draw_literal:x
      {
        \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
        \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
        \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
        cm
      }
  }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn`.)

`\__driver_draw_hbox:Nnnnnnn`
`\l__driver_tmp_box`

Inserting a TEX box transformed to the requested position and using the current matrix is done using a mixture of TEX and low-level manipulation. The offset can be handled by TEX, so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the `draw` version.

```
\cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
  {
    \hbox_set:Nn \l__driver_tmp_box
      {
        \tex_kern:D \__dim_eval:w #6 \__dim_eval_end:
        \__driver_scope_begin:
        \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5}
          { 0pt } { 0pt }
        \box_move_up:nn {#7} { \box_use:N #1 }
        \__driver_scope_end:
      }
    \box_set_wd:Nn \l__driver_tmp_box { 0pt }
    \box_set_ht:Nn \l__driver_tmp_box { 0pt }
    \box_set_dp:Nn \l__driver_tmp_box { 0pt }
    \box_use:N \l__driver_tmp_box
  }
\box_new:N \l__driver_tmp_box
```

(*End definition for* `\__driver_draw_hbox:Nnnnnnn` *and* `\l__driver_tmp_box`.)

⟨/dvipdfmx | pdfmode | xdvipdfmx⟩

44.14 dvisvgm driver

⟨*dvisvgm⟩

44.14.1 Basics

`\__driver_literal:n`

Unlike the other drivers, the requirements for making SVG files mean that we can't conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```
\cs_new_protected:Npn \__driver_literal:n #1
  { \tex_special:D { dvisvgm:raw~ #1 { ?nl } } }
```

(*End definition for* `\__driver_literal:n`.)

__driver_scope_begin:
__driver_scope_end:

A scope in SVG terms is slightly different to the other drivers as operations have to be “tied” to these not simply inside them.

```
\cs_new_protected:Npn \__driver_scope_begin:
  { \__driver_literal:n { <g> } }
\cs_new_protected:Npn \__driver_scope_end:
  { \__driver_literal:n { </g> } }
```

(*End definition for* __driver_scope_begin: *and* __driver_scope_end:.)

44.15 Driver-specific auxiliaries

__driver_scope_begin:n

In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful than `\__driver_scope_begin:` as a result. No assumptions are made about the nature of the scoped operation(s).

```
\cs_new_protected:Npn \__driver_scope_begin:n #1
  { \__driver_literal:n { <g~ #1 > } }
```

(*End definition for* __driver_scope_begin:n.)

44.15.1 Box operations

__driver_box_use_clip:N
\g__driver_clip_path_int

Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `l3cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the TeX box and keep the reference point the same!

```
\cs_new_protected:Npn \__driver_box_use_clip:N #1
  {
    \int_gincr:N \g__driver_clip_path_int
    \__driver_literal:n
      { < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " > }
    \__driver_literal:n
      {
        <
          path ~ d =
            "
              M ~ 0 ~
                  \dim_to_decimal:n { -\box_dp:N #1 } ~
              L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
                  \dim_to_decimal:n { -\box_dp:N #1 } ~
              L ~ \dim_to_decimal:n { \box_wd:N #1 }  ~
                  \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
              L ~ 0 ~
                  \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
              Z
            "
        />
      }
    \__driver_literal:n
      { < /clipPath > }
```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the TEX box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the TEX box.

```
    \__driver_scope_begin:n
      {
        transform =
          "
            translate ( { ?x } , { ?y } ) ~
            scale ( 1 , -1 )
          "
      }
    \__driver_scope_begin:n
      {
        clip-path = "url ( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int ) "
      }
    \__driver_scope_begin:n
      {
        transform =
          "
            scale ( -1 , 1 ) ~
            translate ( { ?x } , { ?y } ) ~
            scale ( -1 , -1 )
          "
      }
    \box_use:N #1
    \__driver_scope_end:
    \__driver_scope_end:
    \__driver_scope_end:
%    \skip_horizontal:n { \box_wd:N #1 }
  }
\int_new:N \g__driver_clip_path_int
```

(*End definition for* `\__driver_box_use_clip:N` *and* `\g__driver_clip_path_int`.)

`\__driver_box_use_rotate:Nn` Rotation has a dedicated operation which includes a centre-of-rotation optional pair. That can be picked up from the driver syntax, so there is no need to worry about the transformation matrix.

```
\cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
  {
    \__driver_scope_begin:n
      {
        transform =
          "
            rotate
            ( \fp_eval:n { round ( -#2 , 5 ) } , ~ { ?x } , ~ { ?y } )
          "
      }
    \box_use:N #1
    \__driver_scope_end:
  }
```

(*End definition for* `\__driver_box_use_rotate:Nn`.)

__driver_box_use_scale:Nnn

In contrast to rotation, we have to account for the current position in this case. That is done using a couple of translations in addition to the scaling (which is therefore done backward with a flip).

```
\cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
  {
    \__driver_scope_begin:n
      {
        transform =
          "
            translate ( { ?x } , { ?y } ) ~
            scale
              (
                \fp_eval:n { round ( -#2 , 5 ) } ,
                \fp_eval:n { round ( -#3 , 5 ) }
              ) ~
            translate ( { ?x } , { ?y } ) ~
            scale ( -1 )
          "
      }
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_scope_end:
  }
```

(*End definition for* __driver_box_use_scale:Nnn*.*)

44.16 Images

__driver_image_getbb_png:n
__driver_image_getbb_jpg:n

These can be included by extracting the bounding box data.

```
\cs_new_eq:NN \__driver_image_getbb_png:n \__image_extract_bb:n
\cs_new_eq:NN \__driver_image_getbb_jpg:n \__image_extract_bb:n
```

(*End definition for* __driver_image_getbb_png:n *and* __driver_image_getbb_jpg:n*.*)

__driver_image_include_png:n
__driver_image_include_jpg:n
__driver_image_include_bitmap_quote:w

The driver here has built-in support for basic image inclusion (see `dvisvgm.def` for a more complex approach, needed if clipping, *etc.*, is covered at the image driver level). The only issue is that `#1` must be quote-corrected. The `dvisvgm:img` operation quotes the file name, but if it is already quoted (contains spaces) then we have an issue: we simply strip off any quotes as a result.

```
\cs_new_protected:Npn \__driver_image_include_png:n #1
  {
     \tex_special:D
       {
         dvisvgm:img~
         \dim_to_decimal:n { \l__image_ury_dim } ~
         \dim_to_decimal:n { \l__image_ury_dim } ~
         \__driver_image_include_bitmap_quote:w #1 " " \q_stop
       }
  }
\cs_new_eq:NN \__driver_image_include_jpg:n \__driver_image_include_png:n
\cs_new:Npn \__driver_image_include_bitmap_quote:w #1 " #2 " #3 \q_stop { #1#2 }
```

(*End definition for* __driver_image_include_png:n *,* __driver_image_include_jpg:n *, and* __driver_image_include_bitmap_quote:w*.*)

44.17 Drawing

__driver_draw_literal:n
__driver_draw_literal:x

The same as the more general literal call.

```
\cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
\cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* __driver_draw_literal:n.)

__driver_draw_begin:
__driver_draw_end:

A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```
\cs_new_protected:Npn \__driver_draw_begin:
  {
    \__driver_draw_scope_begin:
    \__driver_draw_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
  }
\cs_new_protected:Npn \__driver_draw_end:
  { \__driver_draw_scope_end: }
```

(*End definition for* __driver_draw_begin: *and* __driver_draw_end:.)

__driver_draw_scope_begin:
__driver_draw_scope_end:
__driver_draw_scope:n
__driver_draw_scope:x
\g__driver_draw_scope_int
\l__driver_draw_scope_int

Several settings that with other drivers are "stand alone" have to be given as part of a scope in SVG. As a result, there is a need to provide a mechanism to automatically close these extra scopes. That is done using a dedicated function and a pair of tracking variables. Within each graphics scope we use a global variable to do the work, with a group used to save the value between scopes. The result is that no direct action is needed when creating a scope.

```
\cs_new_protected:Npn \__driver_draw_scope_begin:
  {
    \int_set_eq:NN
      \l__driver_draw_scope_int
      \g__driver_draw_scope_int
    \group_begin:
      \int_gzero:N \g__driver_draw_scope_int
  }
\cs_new_protected:Npn \__driver_draw_scope_end:
  {
      \prg_replicate:nn
        { \g__driver_draw_scope_int }
        { \__driver_draw_literal:n { </g> } }
    \group_end:
    \int_gset_eq:NN
      \g__driver_draw_scope_int
      \l__driver_draw_scope_int
  }
\cs_new_protected:Npn \__driver_draw_scope:n #1
  {
    \__driver_draw_literal:n { <g~ #1 > }
    \int_gincr:N \g__driver_draw_scope_int
  }
\cs_generate_variant:Nn \__driver_draw_scope:n { x }
\int_new:N \g__driver_draw_scope_int
\int_new:N \l__driver_draw_scope_int
```

(*End definition for* __driver_draw_scope_begin: *and others.*)

__driver_draw_moveto:nn
__driver_draw_lineto:nn
__driver_draw_rectangle:nnnn
__driver_draw_curveto:nnnnnn
__driver_draw_add_to_path:n
\g__driver_draw_path_tl

Once again, some work is needed to get path constructs correct. Rather then write the values as they are given, the entire path needs to be collected up before being output in one go. For that we use a dedicated storage routine, which adds spaces as required. Since paths should be fully expanded there is no need to worry about the internal x-type expansion.

```
\cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
  {
    \__driver_draw_add_to_path:n
      { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
  }
\cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
  {
    \__driver_draw_add_to_path:n
      { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
  }
\cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
  {
    \__driver_draw_add_to_path:n
      {
        M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
        h ~ \dim_to_decimal:n {#3} ~
        v ~ \dim_to_decimal:n {#4} ~
        h ~ \dim_to_decimal:n { -#3 } ~
        Z
      }
  }
\cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
  {
    \__driver_draw_add_to_path:n
      {
        C ~
        \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
        \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
        \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
      }
  }
\cs_new_protected:Npn \__driver_draw_add_to_path:n #1
  {
    \tl_gset:Nx \g__driver_draw_path_tl
      {
        \g__driver_draw_path_tl
        \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
        #1
      }
  }
\tl_new:N \g__driver_draw_path_tl
```

(*End definition for* __driver_draw_moveto:nn *and others.*)

__driver_draw_evenodd_rule:
__driver_draw_nonzero_rule:

The fill rules here have to be handled as scopes.

```
\cs_new_protected:Npn \__driver_draw_evenodd_rule:
  { \__driver_draw_scope:n { fill-rule="evenodd" } }
\cs_new_protected:Npn \__driver_draw_nonzero_rule:
  { \__driver_draw_scope:n { fill-rule="nonzero" } }
```

(*End definition for* __driver_draw_evenodd_rule: *and* __driver_draw_nonzero_rule:.)

__driver_draw_path:n
__driver_draw_closepath:
__driver_draw_stroke:
__driver_draw_closestroke:
__driver_draw_fill:
__driver_draw_fillstroke:
__driver_draw_clip:
__driver_draw_discardpath:
\g__driver_draw_clip_bool
\g__driver_draw_path_int

Setting fill and stroke effects and doing clipping all has to be done using scopes. This means setting up the various requirements in a shared auxiliary which deals with the bits and pieces. Clipping paths are reused for path drawing: not essential but avoids constructing them twice. Discarding a path needs a separate function as it's not quite the same.

```
\cs_new_protected:Npn \__driver_draw_closepath:
  { \__driver_draw_add_to_path:n { Z } }
\cs_new_protected:Npn \__driver_draw_path:n #1
  {
    \bool_if:NTF \g__driver_draw_clip_bool
      {
        \int_gincr:N \g__driver_clip_path_int
        \__driver_draw_literal:x
          {
            < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " >
              { ?nl }
            <path~d=" \g__driver_draw_path_tl "/> { ?nl }
            < /clipPath > { ? nl }
            <
              use~xlink:href =
                "\c_hash_str l3path \int_use:N \g__driver_path_int " ~
                #1
            />
          }
        \__driver_draw_scope:x
          {
            clip-path =
              "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
          }
      }
      {
        \__driver_draw_literal:x
          { <path ~ d=" \g__driver_draw_path_tl " ~ #1 /> }
      }
    \tl_gclear:N \g__driver_draw_path_tl
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
\int_new:N \g__driver_path_int
\cs_new_protected:Npn \__driver_draw_stroke:
  { \__driver_draw_path:n { style="fill:none" } }
\cs_new_protected:Npn \__driver_draw_closestroke:
  {
    \__driver_draw_closepath:
    \__driver_draw_stroke:
  }
\cs_new_protected:Npn \__driver_draw_fill:
  { \__driver_draw_path:n { style="stroke:none" } }
\cs_new_protected:Npn \__driver_draw_fillstroke:
  { \__driver_draw_path:n { } }
\cs_new_protected:Npn \__driver_draw_clip:
  { \bool_gset_true:N \g__driver_draw_clip_bool }
\bool_new:N \g__driver_draw_clip_bool
```

```
\cs_new_protected:Npn \__driver_draw_discardpath:
  {
    \bool_if:NT \g__driver_draw_clip_bool
      {
        \int_gincr:N \g__driver_clip_path_int
        \__driver_draw_literal:x
          {
            < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " >
              { ?nl }
            <path~d=" \g__driver_draw_path_tl "/> { ?nl }
            < /clipPath >
          }
        \__driver_draw_scope:x
          {
            clip-path =
              "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
          }
      }
    \tl_gclear:N \g__driver_draw_path_tl
    \bool_gset_false:N \g__driver_draw_clip_bool
  }
```

(*End definition for* `\__driver_draw_path:n` *and others.*)

__driver_draw_dash:nn
__driver_draw_dash:n
__driver_draw_dash_aux:nn
__driver_draw_linewidth:n
__driver_draw_miterlimit:n
__driver_draw_cap_butt:
__driver_draw_cap_round:
__driver_draw_cap_rectangle:
__driver_draw_join_miter:
__driver_draw_join_round:
__driver_draw_join_bevel:

All of these ideas are properties of scopes in SVG. The only slight complexity is converting the dash array properly (doing any required maths).

```
\cs_new_protected:Npn \__driver_draw_dash:nn #1#2
  {
    \use:x
      {
        \__driver_draw_dash_aux:nn
          { \clist_map_function:nn {#1} \__driver_draw_dash:n }
          { \dim_to_decimal:n {#2} }
      }
  }
\cs_new:Npn \__driver_draw_dash:n #1
  { , \dim_to_decimal_in_bp:n {#1} }
\cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
  {
    \__driver_draw_scope:x
      {
        stroke-dasharray =
          "
            \tl_if_empty:oTF { \use_none:n #1 }
              { none }
              { \use_none:n #1 }
          " ~
          stroke-offset=" #2 "
      }
  }
\cs_new_protected:Npn \__driver_draw_linewidth:n #1
  { \__driver_draw_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
\cs_new_protected:Npn \__driver_draw_miterlimit:n #1
  { \__driver_draw_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
```

```
\cs_new_protected:Npn \__driver_draw_cap_butt:
  { \__driver_draw_scope:n { stroke-linecap="butt" } }
\cs_new_protected:Npn \__driver_draw_cap_round:
  { \__driver_draw_scope:n { stroke-linecap="round" } }
\cs_new_protected:Npn \__driver_draw_cap_rectangle:
  { \__driver_draw_scope:n { stroke-linecap="square" } }
\cs_new_protected:Npn \__driver_draw_join_miter:
  { \__driver_draw_scope:n { stroke-linejoin="miter" } }
\cs_new_protected:Npn \__driver_draw_join_round:
  { \__driver_draw_scope:n { stroke-linejoin="round" } }
\cs_new_protected:Npn \__driver_draw_join_bevel:
  { \__driver_draw_scope:n { stroke-linejoin="bevel" } }
```

(*End definition for* `\__driver_draw_dash:nn` *and others.*)

__driver_draw_color_cmyk:nnnn
__driver_draw_color_cmyk_fill:nnnn
__driver_draw_color_cmyk_stroke:nnnn
__driver_draw_color_gray:n
__driver_draw_color_gray_fill:n
__driver_draw_color_gray_stroke:n
__driver_draw_color_rgb:nnn
__driver_draw_color_rgb_fill:nnn
__driver_draw_color_rgb_stroke:nnn

SVG only works with RGB colors, so there is some conversion to do. The values also need to be given as percentages, which means a little more maths.

```
\cs_new_protected:Npn \__driver_draw_color_cmyk_aux:NNnnnnn #1#2#3#4#5#6
  {
    \use:x
      {
        \__driver_draw_color_rgb_auxii:nnn
          { \fp_eval:n { -100 * ( (#3) * ( 1 - (#6) ) - 1 ) } }
          { \fp_eval:n { -100 * ( (#4) * ( 1 - (#6) ) + #6 - 1 ) } }
          { \fp_eval:n { -100 * ( (#5) * ( 1 - (#6) ) + #6 - 1 ) } }
      }
      #1 #2
  }
\cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn
  { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn
  { \__driver_draw_color_cmyk_aux:NNnnnnn \c_false_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn
  { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_false_bool }
\cs_new_protected:Npn \__driver_draw_color_gray_aux:NNn #1#2#3
  {
    \use:x
      {
        \__driver_draw_color_gray_aux:nNN
          { \fp_eval:n { 100 * (#3)} }
      }
        #1 #2
  }
\cs_new_protected:Npn \__driver_draw_color_gray_aux:nNN #1
  { \__driver_draw_color_rgb_auxii:nnnNN {#1} {#1} {#1} }
\cs_generate_variant:Nn \__driver_draw_color_gray_aux:nNN { x }
\cs_new_protected:Npn \__driver_draw_color_gray:n
  { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_gray_fill:n
  { \__driver_draw_color_gray_aux:NNn \c_false_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_gray_stroke:n
  { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_false_bool }
\cs_new_protected:Npn \__driver_draw_color_rgb_auxi:NNnnn #1#2#3#4#5
  {
    \use:x
      {
        \__driver_draw_color_rgb_auxii:nnnNN
          { \fp_eval:n { 100 * (#3) } }
          { \fp_eval:n { 100 * (#4) } }
          { \fp_eval:n { 100 * (#5) } }
      }
        #1 #2
  }
\cs_new_protected:Npn \__driver_draw_color_rgb_auxii:nnnNN #1#2#3#4#5
  {
    \__driver_draw_scope:x
      {
        \bool_if:NT #4
          {
            fill =
             "
               rgb
                 (
                   #1 \c_percent_str ,
                   #2 \c_percent_str ,
                   #3 \c_percent_str
                 )
             "
             \bool_if:NT #5 { ~ }
          }
        \bool_if:NT #5
          {
            stroke =
             "
               rgb
                 (
                   #1 \c_percent_str ,
                   #2 \c_percent_str ,
                   #3 \c_percent_str
                 )
             "
          }
      }
  }
\cs_new_protected:Npn \__driver_draw_color_rgb:nnn
  { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn
  { \__driver_draw_color_rgb_auxi:NNnnn \c_false_bool \c_true_bool }
\cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn
  { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_false_bool }
```

(*End definition for* `\__driver_draw_color_cmyk:nnnn` *and others.*)

__driver_draw_transformcm:nnnnnn

The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```
\cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
  {
    \__driver_draw_scope:x
```

```
      {
       transform =
         "
           matrix
             (
               \fp_eval:n {#1} , \fp_eval:n {#2} ,
               \fp_eval:n {#3} , \fp_eval:n {#4} ,
               \dim_to_decimal:n {#5} , \dim_to_decimal:n {#6}
             )
         "
      }
  }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn`.)

`\__driver_draw_hbox:Nnnnnnn`

No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```
\cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
  {
    \__driver_scope_begin:
    \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
    \__driver_literal:n
      {
        < g~
            stroke="none"~
            transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
        >
      }
    \box_set_wd:Nn #1 { 0pt }
    \box_set_ht:Nn #1 { 0pt }
    \box_set_dp:Nn #1 { 0pt }
    \box_use:N #1
    \__driver_literal:n { </g> }
    \__driver_scope_end:
  }
```

(*End definition for* `\__driver_draw_hbox:Nnnnnnn`.)

⟨/dvisvgm⟩

⟨/initex | package⟩