

The L^AT_EX3 Sources

The L^AT_EX3 Project*

January 10, 2011

Abstract

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

The T_EX and ε -T_EX 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 L^AT_EX3 programming language.

At present, the `expl3` modules are designed to be loaded on top of L^AT_EX 2 ε . In time, a L^AT_EX3 format will be produced based on this code. This allows the code to be used in L^AT_EX 2 ε packages *now* while a stand-alone L^AT_EX3 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.

*Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.0.1	Terminological inexactitude	3
2	Documentation conventions	3
II	The <code>l3names</code> package: A systematic naming scheme for <code>TeX</code>	5
3	Setting up the <code>LaTeX3</code> programming language	5
4	Using the modules	5
III	The <code>l3basics</code> package: Basic Definitions	6
5	Predicates and conditionals	7
5.1	Primitive conditionals	8
5.2	Non-primitive conditionals	10
6	Control sequences	12
7	Selecting and discarding tokens from the input stream	12
7.1	Extending the interface	14
7.2	Selecting tokens from delimited arguments	14
8	That which belongs in other modules but needs to be defined earlier	15
9	Defining functions	16
9.1	Defining new functions using primitive parameter text	17
9.2	Defining new functions using the signature	18
9.3	Defining functions using primitive parameter text	19
9.4	Defining functions using the signature (no checks)	20

9.5	Undefining functions	22
9.6	Copying function definitions	22
9.7	Internal functions	23
10	The innards of a function	23
11	Grouping and scanning	24
12	Checking the engine	25
IV	The l3expan package: Controlling Expansion of Function Arguments	25
13	Brief overview	25
14	Defining new variants	26
14.1	Methods for defining variants	26
15	Introducing the variants	27
16	Manipulating the first argument	28
17	Manipulating two arguments	29
18	Manipulating three arguments	30
19	Preventing expansion	31
20	Unbraced expansion	32
V	The l3prg package: Program control structures	32
21	Conditionals and logical operations	32
22	Defining a set of conditional functions	33
23	The boolean data type	34

24 Boolean expressions	36
25 Case switches	37
26 Generic loops	38
27 Choosing modes	38
28 Alignment safe grouping and scanning	39
29 Producing n copies	39
30 Sorting	40
30.1 Variable type and scope	41
30.2 Mapping to variables	41
VI The l3quark package: “Quarks”	42
31 Functions	42
32 Recursion	43
33 Constants	44
VII The l3token package: A token of my appreciation...	45
34 Character tokens	45
35 Generic tokens	48
35.1 Useless code: because we can!	52
36 Peeking ahead at the next token	52
VIII The l3int package: Integers/counters	54

37 Integer values	54
37.1 Integer expressions	54
37.2 Integer variables	55
37.3 Comparing integer expressions	58
37.4 Formatting integers	60
37.5 Converting from other formats	61
37.6 Low-level conversion functions	62
38 Variables and constants	63
38.1 Internal functions	65
 IX The l3skip package: Dimension and skip registers	 66
39 Skip registers	66
39.1 Functions	66
39.2 Formatting a skip register value	68
39.3 Variable and constants	68
40 Dim registers	69
40.1 Functions	69
40.2 Variable and constants	72
41 Muskip	73
 X The l3tl package: Token Lists	 73
42 Functions	74
43 Predicates and conditionals	78
44 Working with the contents of token lists	79
45 Variables and constants	80
46 Searching for and replacing tokens	81

47 Heads or tails?	82
XI The l3toks package: Token Registers	84
48 Allocation and use	84
49 Adding to the contents of token registers	86
50 Predicates and conditionals	87
51 Variable and constants	88
XII The l3seq package: Sequences	88
52 Functions for creating/initialising sequences	89
53 Adding data to sequences	90
54 Working with sequences	91
55 Predicates and conditionals	92
56 Internal functions	93
57 Functions for ‘Sequence Stacks’	93
XIII The l3clist package: Comma separated lists	94
58 Functions for creating/initialising comma-lists	95
59 Putting data in	96
60 Getting data out	97
61 Mapping functions	97
62 Predicates and conditionals	99

63 Higher level functions	99
64 Functions for ‘comma-list stacks’	100
65 Internal functions	101
 XIV The <code>l3prop</code> package: Property Lists	 102
66 Functions	102
67 Predicates and conditionals	104
68 Internal functions	105
 XV The <code>l3io</code> package: Low-level file i/o	 106
69 Opening and closing streams	106
69.1 Writing to files	107
69.2 Reading from files	109
70 Internal functions	109
71 Variables and constants	110
 XVI The <code>l3msg</code> package: Communicating with the user	 110
72 Creating new messages	111
73 Message classes	111
74 Redirecting messages	113
75 Support functions for output	114
76 Low-level functions	114
77 Kernel-specific functions	115

78 Variables and constants	116
XVII The l3box package: Boxes	117
79 Generic functions	118
80 Horizontal mode	121
81 Vertical mode	122
XVIII The l3xref package: Cross references	124
XIX The l3keyval package: Key-value parsing	124
82 Features of l3keyval	125
83 Functions for keyval processing	125
84 Internal functions	126
85 Variables and constants	127
XX The l3keys package: Key-value support	127
86 Creating keys	129
87 Sub-dividing keys	132
87.1 Multiple choices	133
88 Setting keys	134
88.1 Examining keys: internal representation	135
89 Internal functions	135
90 Variables and constants	137

XXI	The <code>l3file</code> package: File Loading	138
91	Loading files	138
XXII	The <code>l3fp</code> package: Floating point arithmetic	139
92	Floating point numbers	139
92.1	Constants	140
92.2	Floating-point variables	140
92.3	Conversion to other formats	142
92.4	Rounding floating point values	143
92.5	Tests on floating-point values	144
92.6	Unary operations	144
92.7	Arithmetic operations	145
92.8	Power operations	146
92.9	Exponential and logarithm functions	146
92.10	Trigonometric functions	147
92.11	Notes on the floating point unit	148
XXIII	The <code>l3luatex</code> package: Lua_{TeX}-specific functions	148
93	Breaking out to Lua	148
94	Category code tables	149
XXIV	Implementation	150
95	<code>l3names</code> implementation	151
95.1	Internal functions	151
95.2	Bootstrap code	151
95.3	Requirements	153
95.4	Catcode assignments	154

95.5	Setting up primitive names	154
95.6	Reassignment of primitives	155
95.7	<code>expl3</code> code switches	166
95.8	Package loading	168
95.9	Finishing up	171
95.10	Showing memory usage	173
96	<code>l3basics</code> implementation	174
96.1	Renaming some \TeX primitives (again)	174
96.2	Defining functions	176
96.3	Selecting tokens	177
96.4	Gobbling tokens from input	178
96.5	Expansion control from <code>l3expan</code>	178
96.6	Conditional processing and definitions	178
96.7	Dissecting a control sequence	183
96.8	Exist or free	185
96.9	Defining and checking (new) functions	187
96.10	More new definitions	190
96.11	Copying definitions	192
96.12	Undefining functions	193
96.13	Diagnostic wrapper functions	193
96.14	Engine specific definitions	193
96.15	Scratch functions	194
96.16	Defining functions from a given number of arguments	194
96.17	Using the signature to define functions	196
97	<code>l3expan</code> implementation	198
97.1	Internal functions and variables	198
97.2	Module code	199
97.3	General expansion	200
97.4	Hand-tuned definitions	203
97.5	Definitions with the ‘general’ technique	204

97.6 Preventing expansion	205
97.7 Defining function variants	206
97.8 Last-unbraced versions	209
97.9 Items held from earlier	209
98 l3prg implementation	210
98.1 Variables	210
98.2 Module code	210
98.3 Choosing modes	211
98.4 Producing n copies	212
98.5 Booleans	216
98.6 Parsing boolean expressions	217
98.7 Case switch	224
98.8 Sorting	226
98.9 Variable type and scope	228
98.10 Mapping to variables	229
99 l3quark implementation	231
100 l3token implementation	234
100.1 Documentation of internal functions	234
100.2 Module code	234
100.3 Character tokens	234
100.4 Generic tokens	237
100.5 Peeking ahead at the next token	245
101 l3int implementation	251
101.1 Internal functions and variables	251
101.2 Module loading and primitives definitions	251
101.3 Allocation and setting	252
101.4 Scanning and conversion	260
101.5 Defining constants	269
101.6 Backwards compatibility	271

1023skip implementation	272
102.1Skip registers	273
102.2Dimen registers	276
102.3Muskip	281
1033tl implementation	282
103.1Functions	282
103.2Variables and constants	287
103.3Predicates and conditionals	288
103.4Working with the contents of token lists	290
103.5Checking for and replacing tokens	295
103.6Heads or tails?	297
1043toks implementation	301
104.1Allocation and use	302
104.2Adding to token registers' contents	304
104.3Predicates and conditionals	305
104.4Variables and constants	306
1053seq implementation	307
105.1Allocating and initialisation	307
105.2Predicates and conditionals	308
105.3Getting data out	309
105.4Putting data in	310
105.5Mapping	311
105.6Manipulation	312
105.7Sequence stacks	312
1063clist implementation	313
106.1Allocation and initialisation	313
106.2Predicates and conditionals	314
106.3Retrieving data	315
106.4Storing data	316
106.5Mapping	317
106.6Higher level functions	318
106.7Stack operations	319

107	prop implementation	320
107.1	Functions	321
107.2	Predicates and conditionals	324
107.3	Mapping functions	324
108	io implementation	326
108.1	Variables and constants	326
108.2	Stream management	327
108.3	Immediate writing	332
108.4	Deferred writing	333
109	Special characters for writing	333
109.1	Reading input	334
110	msg implementation	334
110.1	Variables and constants	335
110.2	Output helper functions	336
110.3	Generic functions	337
110.4	General functions	339
110.5	Redirection functions	343
110.6	Kernel-specific functions	343
111	box implementation	346
111.1	Generic boxes	346
111.2	Vertical boxes	349
111.3	Horizontal boxes	350
112	xref implementation	352
112.1	Internal functions and variables	352
112.2	Module code	352
113	xref test file	355

1143	keyval implementation	356
114.1	Module code	357
114.1.1	Variables and constants	365
114.1.2	Internal functions	366
114.1.3	Properties	373
114.1.4	Messages	376
1153	file implementation	377
116	Implementation	381
116.1	Constants	381
116.2	Variables	382
116.3	Parsing numbers	384
116.4	Internal utilities	388
116.5	Operations for fp variables	389
116.6	Transferring to other types	394
116.7	Rounding numbers	400
116.8	Unary functions	403
116.9	Basic arithmetic	405
116.10	Arithmetic for internal use	414
116.11	Trigonometric functions	420
116.12	Exponent and logarithm functions	433
116.13	Tests for special values	456
116.14	Floating-point conditionals	457
116.15	Messages	461
117	Implementation	462
117.1	Category code tables	463

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 though exactly as given. Usually, if you use a single token for an `n` argument, all will be well.
- c** This means *cname*, and indicates that the argument will be turned into a `cname` 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 `cname` 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*: the plain $\text{\TeX}\ \backslash\text{edef}$.
- f The **f** specifier stands for *full expansion*, and in contrast to *x* stops at the first non-expandable item without trying to execute it.
- 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 $\text{\TeX}\ \textit{parameters}$. 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 module¹ 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.

dim ‘Rigid’ lengths.

int Integer-valued count register.

¹The 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.

num A ‘fake’ integer type using only macros. Useful for setting up allocation routines.

prop Property list.

skip ‘Rubber’ lengths.

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

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

t1 Token list variables: placeholder for a token list.

toks Token register.

1.0.1 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 mayn’t 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 \TeX book 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:

<code>\ExplSyntaxOn</code>
<code>\ExplSyntaxOff</code>

`\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:

<code>\seq_new:N</code>
<code>\seq_new:c</code>

`\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.

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

<code>\cs_to_str:N *</code>

`\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>`.

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:

<code>\xetex_if_engine:TF *</code>

`\xetex_if_engine:TF <true code> <>false code>`

The underlining and italic of `TF` indicates that `\xetex_if_engine:T`, `\xetex_if_engine:F` and `\xetex_if_engine: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 $\text{\LaTeX} 2_{\epsilon}$ or plain \TeX . In these cases, the text will include an extra ‘ **\TeX hackers note**’ section:

`\token_to_str:N *` `\token_to_str:N` $\langle token \rangle$

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or $\text{\LaTeX} 2_{\epsilon}$ programmer. In this case, it would point out that this function is the \TeX primitive `\string`.

Part II

The `l3names` package

A systematic naming scheme for \TeX

3 Setting up the $\text{\LaTeX} 3$ programming language

This module is at the core of the $\text{\LaTeX} 3$ programming language. It performs the following tasks:

- defines new names for all \TeX primitives;
- defines catcode regimes for programming;
- provides settings for when the code is used in a format;
- provides tools for when the code is used as a package within a $\text{\LaTeX} 2_{\epsilon}$ context.

4 Using the modules

The modules documented in `source3` are designed to be used on top of $\text{\LaTeX} 2_{\epsilon}$ and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the $\text{\LaTeX} 3$ format, but work in this area is incomplete and not included in this documentation.

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

<code>\ExplSyntaxOn</code>	
<code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn <code> \ExplSyntaxOff</code>

Issues a catcode regime where spaces are ignored and colon and underscore are letters. A space character may be input with `~` instead.

<code>\ExplSyntaxNamesOn</code>	
<code>\ExplSyntaxNamesOff</code>	<code>\ExplSyntaxNamesOn <code> \ExplSyntaxNamesOff</code>

Issues a catcode regime where colon and underscore are letters, but spaces remain the same.

<code>\ProvidesExplPackage</code>	<code>\RequirePackage{expl3}</code>
<code>\ProvidesExplClass</code>	<code>\ProvidesExplPackage {<package>}</code>
<code>\ProvidesExplFile</code>	<code>{<date>} {<version>} {<description>}</code>

The package `l3names` (this module) provides `\ProvidesExplPackage` which is a wrapper for `\ProvidesPackage` and sets up the $\text{\LaTeX}3$ catcode settings for programming automatically. Similar for the relationship between `\ProvidesExplClass` and `\ProvidesClass`. Spaces are not ignored in the arguments of these commands.

<code>\GetIdInfo</code>	
<code>\filename</code>	
<code>\filenameext</code>	
<code>\filedate</code>	
<code>\fileversion</code>	
<code>\filetimestamp</code>	
<code>\fileauthor</code>	
<code>\filedescription</code>	<code>\RequirePackage{l3names}</code>
	<code>\GetIdInfo \$Id: < cvs or svn info field> \$ {<description>}</code>

Extracts all information from a CVS or SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\filename` for the part of the file name leading up to the period, `\filenameext` for the extension, `\filedate` for date, `\fileversion` for version, `\filetimestamp` for the time and `\fileauthor` for the author.

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 alike are loaded with usual \LaTeX catcodes and the $\text{\LaTeX}3$ catcode 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{\filename}{\filedate}{\fileversion}{\filedescription}
```

Part III

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.

5 Predicates and conditionals

L^AT_EX3 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 in the $\langle true\ arg \rangle$ or the $\langle false\ arg \rangle$. These arguments are denoted with T and F repectively. An example would be

```
\cs_if_free:cTF{abc} {\langle true code \rangle} {\langle false code \rangle}
```

a function that will turn 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 carry 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 will usually be 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 will always provide all three versions.

Important to note is that these branching conditionals with $\langle true\ code \rangle$ and/or $\langle false\ code \rangle$ 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 will be 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 function `\if_predicate:w` or in 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

```
\if_predicate:w \cs_if_free_p:N \l_tmpz_tl <true code> \else:
<false code> \fi:
```

or in expressions utilizing the boolean logic parser:

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {(true code)} {(false code)}
```

Like their branching cousins, predicate functions ensure that all underlying primitive `\else:` or `\fi:` have been removed before returning the boolean true or false values.²

For each predicate defined, a ‘predicate conditional’ will also exist 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`. 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.

5.1 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 will often contain a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_num: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 will prefix primitive conditionals with `\if_`.

<code>\if_true:</code>	★	
<code>\if_false:</code>	★	
<code>\or:</code>	★	
<code>\else:</code>	★	
<code>\fi:</code>	★	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\reverse_if:N</code>	★	<code>\if_false: <true code> \else: <false code> \fi:</code>
		<code>\reverse_if:N <primitive conditional></code>

`\if_true:` always executes `<true code>`, while `\if_false:` always executes `<false code>`.

²If defined using the interface provided.

`\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional. `\or:` is used in case switches, see `l3intexpr` for more.

T_EXhackers note: These are equivalent to their corresponding T_EX primitive conditionals; `\reverse_if:N` is ε -T_EX's `\unless`.

<code>\if_meaning:w</code> *	<code>\if_meaning:w</code> $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
------------------------------	---

`\if_meaning:w` executes $\langle true\ code \rangle$ when $\langle arg_1 \rangle$ and $\langle arg_2 \rangle$ are the same, otherwise it executes $\langle false\ code \rangle$. $\langle arg_1 \rangle$ and $\langle arg_2 \rangle$ could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

T_EXhackers note: This is T_EX's `\ifx`.

<code>\if:w</code> *	<code>\if:w</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_charcode:w</code> *	
<code>\if_catcode:w</code> *	

`\if_catcode:w` $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\langle true\ code \rangle$ `\else:` $\langle false\ code \rangle$ `\fi:`

These conditionals will 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`.

<code>\if_predicate:w</code> *	<code>\if_predicate:w</code> $\langle predicate \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
--------------------------------	---

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 $\langle predicate \rangle$ but to make the coding clearer this should be done through `\if_bool:N`.)

<code>\if_bool:N</code> *	<code>\if_bool:N</code> $\langle boolean \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
---------------------------	--

This function takes a boolean variable and branches according to the result.

<code>\if_cs_exist:N</code> *	<code>\if_cs_exist:N</code> $\langle cs \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_cs_exist:w</code> *	

`\if_cs_exist:w` $\langle tokens \rangle$ `\cs_end:` $\langle true\ code \rangle$ `\else:` $\langle false\ code \rangle$ `\fi:`

Check if $\langle cs \rangle$ appears in the hash table or if the control sequence that can be formed from $\langle tokens \rangle$ 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.

<code>\if_mode_horizontal:</code>	<code>*</code>
<code>\if_mode_vertical:</code>	<code>*</code>
<code>\if_mode_math:</code>	<code>*</code>
<code>\if_mode_inner:</code>	<code>*</code>

`\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.

5.2 Non-primitive conditionals

<code>\cs_if_eq_name_p:NN</code>

`\cs_if_eq_name_p:NN <cs1> <cs2>`

Returns ‘true’ if *<cs₁>* and *<cs₂>* are textually the same, i.e. have the same name, otherwise it returns ‘false’.

<code>\cs_if_eq_p:NN</code>	<code>*</code>
<code>\cs_if_eq_p:cN</code>	<code>*</code>
<code>\cs_if_eq_p:Nc</code>	<code>*</code>
<code>\cs_if_eq_p:cc</code>	<code>*</code>
<code>\cs_if_eq:NNTF</code>	<code>*</code>
<code>\cs_if_eq:cNTF</code>	<code>*</code>
<code>\cs_if_eq:NcTF</code>	<code>*</code>
<code>\cs_if_eq:ccTF</code>	<code>*</code>

`\cs_if_eq_p:NNTF <cs1> <cs2>`
`\cs_if_eq:NNTF <cs1> <cs2> {\<true code>} {\<false code>}`

These functions check if *<cs₁>* and *<cs₂>* have same meaning.

<code>\cs_if_free_p:N</code>	<code>*</code>
<code>\cs_if_free_p:c</code>	<code>*</code>
<code>\cs_if_free:NTF</code>	<code>*</code>
<code>\cs_if_free:cTF</code>	<code>*</code>

`\cs_if_free_p:N <cs>`
`\cs_if_free:NTF <cs> {\<true code>} {\<false code>}`

Returns ‘true’ if *<cs>* is either undefined or equal to `\tex_relax:D` (the function that is assigned to newly created control sequences by T_EX when `\cs:w ... \cs_end:` is used). In addition to this, ‘true’ is only returned if *<cs>* does not have a signature equal to D, i.e., ‘do not use’ functions are not free to be redefined.

<code>\cs_if_exist_p:N</code>	<code>*</code>
<code>\cs_if_exist_p:c</code>	<code>*</code>
<code>\cs_if_exist:NTF</code>	<code>*</code>
<code>\cs_if_exist:cTF</code>	<code>*</code>

`\cs_if_exist_p:N <cs>`
`\cs_if_exist:NTF <cs> {\<true code>} {\<false code>}`

These functions check if *<cs>* exists, i.e., if *<cs>* is present in the hash table and is not the primitive `\tex_relax:D`.

<code>\cs_if_do_not_use_p:N</code>	<code>*</code>
------------------------------------	----------------

`\cs_if_do_not_use_p:N <cs>`

These functions check if $\langle cs \rangle$ has the arg spec D for ‘do not use’. There are no TF-type conditionals for this function as it is only used internally and not expected to be widely used. (For now, anyway.)

<code>\chk_if_free_cs:N</code> <code>\chk_if_free_cs:c</code>	<code>\chk_if_free_cs:N</code> $\langle cs \rangle$
--	---

This function checks that $\langle cs \rangle$ is $\langle free \rangle$ according to the criteria for `\cs_if_free_p:N` above. If not, an error is generated.

<code>\chk_if_exist_cs:N</code> <code>\chk_if_exist_cs:c</code>	<code>\chk_if_exist_cs:N</code> $\langle cs \rangle$
--	--

This function checks that $\langle cs \rangle$ is defined. If it is not an error is generated.

<code>\str_if_eq_p:nn</code>	<code>*</code>
<code>\str_if_eq_p:Vn</code>	<code>*</code>
<code>\str_if_eq_p:on</code>	<code>*</code>
<code>\str_if_eq_p:no</code>	<code>*</code>
<code>\str_if_eq_p:nV</code>	<code>*</code>
<code>\str_if_eq_p:VV</code>	<code>*</code>
<code>\str_if_eq_p:xx</code>	<code>*</code>
<code>\str_if_eq:nnTF</code>	<code>*</code>
<code>\str_if_eq:VnTF</code>	<code>*</code>
<code>\str_if_eq:onTF</code>	<code>*</code>
<code>\str_if_eq:noTF</code>	<code>*</code>
<code>\str_if_eq:nVTF</code>	<code>*</code>
<code>\str_if_eq:VVTF</code>	<code>*</code>
<code>\str_if_eq:xxTF</code>	<code>*</code>

`\str_if_eq_p:nn` $\{\langle tl_1 \rangle\}$ $\{\langle tl_2 \rangle\}$
`\str_if_eq:nnTF` $\{\langle tl_1 \rangle\}$ $\{\langle tl_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Compares the two $\langle token\ lists \rangle$ 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:xx { abc } { \tl_to_str:n { abc } }`

is logically **true**. The branching versions then leave either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version. All versions of these functions are fully expandable (including those involving an x-type expansion).

<code>\c_true_bool</code> <code>\c_false_bool</code>

Constants that represent ‘true’ or ‘false’, respectively. Used to implement predicates.

6 Control sequences

<code>\cs:w</code>	<code>*</code>
<code>\cs_end:</code>	<code>*</code>

`\cs:w <tokens> \cs_end:`

This is the \TeX internal way of generating a control sequence from some token list. $\langle tokens \rangle$ get expanded and must ultimately result in a sequence of characters.

\TeX hackers note: These functions are the primitives `\csname` and `\endcsname`. `\cs:w` is considered weird because it expands tokens until it reaches `\cs_end:`.

<code>\cs_show:N</code>	<code>*</code>
<code>\cs_show:c</code>	<code>*</code>

`\cs_show:N <cs>`
`\cs_show:c {\<arg>}`

This function shows in the console output the *meaning* of the control sequence $\langle cs \rangle$ or that created by $\langle arg \rangle$.

\TeX hackers note: This is \TeX 's `\show` and associated `csname` version of it.

<code>\cs_meaning:N</code>	<code>*</code>
<code>\cs_meaning:c</code>	<code>*</code>

`\cs_meaning:N <cs>`
`\cs_meaning:c {\<arg>}`

This function expands to the *meaning* of the control sequence $\langle cs \rangle$ or that created by $\langle arg \rangle$.

\TeX hackers note: This is \TeX 's `\meaning` and associated `csname` version of it.

7 Selecting and discarding tokens from the input stream

The conditional processing cannot be implemented without being able to gobble and select which tokens to use from the input stream.

<code>\use:n</code>	<code>*</code>
<code>\use:nn</code>	<code>*</code>
<code>\use:nnn</code>	<code>*</code>
<code>\use:nnnn</code>	<code>*</code>

`\use:n {\<arg>}`

Functions that returns all of their arguments to the input stream after removing the surrounding braces around each argument.

T_EXhackers note: `\use:n` is L^AT_EX 2_ε's `\@firstofone/\@iden`.

<code>\use:c</code> ★	<code>\use:c</code> $\{\langle cs \rangle\}$
-----------------------	--

Function that returns to the input stream the control sequence created from its argument. Requires two expansions before a control sequence is returned.

T_EXhackers note: `\use:c` is L^AT_EX 2_ε's `\@nameuse`.

<code>\use:x</code>	<code>\use:x</code> $\{\langle expandable\ tokens \rangle\}$
---------------------	--

Function that fully expands its argument before passing it to the input stream. Contents of the argument must be fully expandable.

T_EXhackers note: LuaT_EX provides `\expanded` which performs this operation in an expandable manner, but we cannot assume this behaviour on all platforms yet.

<code>\use_none:n</code>	★	
<code>\use_none:nn</code>	★	
<code>\use_none:nnn</code>	★	
<code>\use_none:nnnn</code>	★	
<code>\use_none:nnnnn</code>	★	
<code>\use_none:nnnnnn</code>	★	
<code>\use_none:nnnnnnn</code>	★	
<code>\use_none:nnnnnnnn</code>	★	<code>\use_none:n</code> $\{\langle arg_1 \rangle\}$
<code>\use_none:nnnnnnnnn</code>	★	<code>\use_none:nn</code> $\{\langle arg_1 \rangle\}$ $\{\langle arg_2 \rangle\}$

These functions gobble the tokens or brace groups from the input stream.

T_EXhackers note: `\use_none:n`, `\use_none:nn`, `\use_none:nnnn` are L^AT_EX 2_ε's `\@gobble`, `\@gobbletwo`, and `\@gobblefour`.

<code>\use_i:nn</code> ★	
<code>\use_ii:nn</code> ★	<code>\use_i:nn</code> $\{\langle code_1 \rangle\}$ $\{\langle code_2 \rangle\}$

Functions that execute the first or second argument respectively, after removing the surrounding braces. Primarily used to implement conditionals.

T_EXhackers note: These are L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`, respectively.

<code>\use_i:nnn</code>	<code>*</code>
<code>\use_ii:nnn</code>	<code>*</code>
<code>\use_iii:nnn</code>	<code>*</code>

`\use_i:nnn {<arg1>} {<arg2>} {<arg3>}`

Functions that pick up one of three arguments and execute them after removing the surrounding braces.

TeXhackers note: L^AT_EX 2_ε has only `\@thirdofthree`.

<code>\use_i:nnnn</code>	<code>*</code>
<code>\use_ii:nnnn</code>	<code>*</code>
<code>\use_iii:nnnn</code>	<code>*</code>
<code>\use_iv:nnnn</code>	<code>*</code>

`\use_i:nnnn {<arg1>} {<arg2>} {<arg3>} {<arg4>}`

Functions that pick up one of four arguments and execute them after removing the surrounding braces.

7.1 Extending the interface

<code>\use_i_ii:nnn</code>	<code>*</code>
----------------------------	----------------

`\use_i_ii:nnn {<arg1>} {<arg2>} {<arg3>}`

This function used in the expansion module reads three arguments and returns (without braces) the first and second argument while discarding the third argument.

If you wish to select multiple arguments while discarding others, use a syntax like this. Its definition is

```
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
```

7.2 Selecting tokens from delimited arguments

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

<code>\use_none_delimit_by_q_nil:w</code>	<code>*</code>
<code>\use_none_delimit_by_q_stop:w</code>	<code>*</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	<code>*</code>

`\use_none_delimit_by_q_nil:w <balanced text> \q_nil`

Gobbles *<balanced text>*. Useful in gobbling the remainder in a list structure or terminating recursion.

<code>\use_i_delimit_by_q_nil:nw</code>	<code>*</code>	
<code>\use_i_delimit_by_q_stop:nw</code>	<code>*</code>	
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>*</code>	<code>\use_i_delimit_by_q_nil:nw {\langle arg \rangle} \langle balanced text \rangle \q_</code>

Gobbles $\langle balanced\ text \rangle$ and executes $\langle arg \rangle$ afterwards. This can also be used to get the first item in a token list.

<code>\use_i_after_fi:nw</code>	<code>*</code>	<code>\use_i_after_fi:nw {\langle arg \rangle} \fi:</code>
<code>\use_i_after_else:nw</code>	<code>*</code>	<code>\use_i_after_else:nw {\langle arg \rangle} \else: \langle balanced text \rangle \fi:</code>
<code>\use_i_after_or:nw</code>	<code>*</code>	<code>\use_i_after_or:nw {\langle arg \rangle} \or: \langle balanced text \rangle \fi:</code>
<code>\use_i_after_orelse:nw</code>	<code>*</code>	<code>\use_i_after_orelse:nw {\langle arg \rangle} \or:/\else: \langle balanced text \rangle \fi:</code>

Executes $\langle arg \rangle$ after executing closing out `\fi:`. `\use_i_after_orelse:nw` can be used anywhere where `\use_i_after_else:nw` or `\use_i_after_or:nw` are used.

8 That which belongs in other modules but needs to be defined earlier

<code>\exp_after:wN</code>	<code>*</code>	<code>\exp_after:wN \langle token_1 \rangle \langle token_2 \rangle</code>
----------------------------	----------------	--

Expands $\langle token_2 \rangle$ once and then continues processing from $\langle token_1 \rangle$.

T_EXhackers note: This is T_EX's `\expandafter`.

<code>\exp_not:N</code>	<code>*</code>	<code>\exp_not:N \langle token \rangle</code>
<code>\exp_not:n</code>	<code>*</code>	<code>\exp_not:n {\langle tokens \rangle}</code>

In an expanding context, this function prevents $\langle token \rangle$ or $\langle tokens \rangle$ from expanding.

T_EXhackers note: These are T_EX's `\noexpand` and ε -T_EX's `\unexpanded`, respectively.

<code>\prg_do_nothing:</code>	<code>*</code>	This is as close as we get to a null operation or no-op.
-------------------------------	----------------	--

T_EXhackers note: Definition as in L^AT_EX's `\empty` but not used for the same thing.

<code>\iow_log:x</code>		
<code>\iow_term:x</code>		
<code>\iow_shipout_x:Nn</code>		<code>\iow_log:x {\langle message \rangle}</code>
		<code>\iow_shipout_x:Nn \langle write_stream \rangle {\langle message \rangle}</code>

Writes $\langle message \rangle$ to either to log or the terminal.

<code>\msg_kernel_bug:x</code>	<code>\msg_kernel_bug:x {\message}</code>
--------------------------------	---

Internal function for calling errors in our code.

<code>\cs_record_meaning:N</code>	Placeholder for a function to be defined by <code>l3chk</code> .
-----------------------------------	--

<code>\c_minus_one</code> <code>\c_zero</code> <code>\c_sixteen</code>	Numeric constants.
--	--------------------

9 Defining functions

There are two types of function definitions in L^AT_EX3: versions that check if the function name is still unused, and versions that simply make the definition. The latter are used for internal scratch functions that get new meanings all over the place.

For each type there is an additional choice to be made: Does the function to be defined contain delimited arguments? The answer in 99% of the cases is no. For this type the programmer will know the number of arguments and in most cases use the argument signature to signal this, e.g., `\foo_bar:nnn` presumably takes three arguments. We therefore also provide functions that automatically detect how many arguments are required and construct the parameter text on the fly.

A definition of a new function can be done locally and globally. Currently nearly all function definitions are done locally on top level, in other words they are global but don't show it. Therefore I think it may be better to remove the local variants in the future and declare all checked function definitions global.

T_EXhackers note: While T_EX makes all definition functions directly available to the user L^AT_EX3 hides them very carefully to avoid the problems with definitions that are overwritten accidentally. Many functions that are in T_EX a combination of prefixes and definition functions are provided as individual functions.

A slew of functions are defined in the following sections for defining new functions.

Here's a quick summary to get an idea of what's available:

`\cs_(g)(new/set)(_protected)(_nopar):(N/c)(p)(n/x)`

That stands for, respectively, the following variations:

g Global or local;

new/set Define a new function or re-define an existing one;

protected Prevent expansion of the function in **x** arguments;

nopar Restrict the argument(s) from containing `\par`;

N/c Either a control sequence or a ‘csname’;

p Either the a primitive T_EX argument or the number of arguments is detected from the argument signature, i.e., `\foo:nnn` is assumed to have three arguments `#1#2#3`;

n/x Either an unexpanded or an expanded definition.

That adds up to 128 variations (!). However, the system is very logical and only a handful will usually be required often.

9.1 Defining new functions using primitive parameter text

<code>\cs_new:Npn</code>
<code>\cs_new:Npx</code>
<code>\cs_new:cpn</code>
<code>\cs_new:cpx</code>

`\cs_new:Npn <cs> <parms> {<code>}`

Defines a function that may contain `\par` tokens in the argument(s) when called. This is not allowed for normal functions.

<code>\cs_new_nopar:Npn</code>
<code>\cs_new_nopar:Npx</code>
<code>\cs_new_nopar:cpn</code>
<code>\cs_new_nopar:cpx</code>

`\cs_new_nopar:Npn <cs> <parms> {<code>}`

Defines a new function, making sure that `<cs>` is unused so far. `<parms>` may consist of arbitrary parameter specification in T_EX syntax. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. `<code>` is either passed literally or may be subject to expansion (under the `x` variants).

<code>\cs_new_protected:Npn</code>
<code>\cs_new_protected:Npx</code>
<code>\cs_new_protected:cpn</code>
<code>\cs_new_protected:cpx</code>

`\cs_new_protected:Npn <cs> <parms> {<code>}`

Defines a function that is both robust and may contain `\par` tokens in the argument(s) when called.

<code>\cs_new_protected_nopar:Npn</code>
<code>\cs_new_protected_nopar:Npx</code>
<code>\cs_new_protected_nopar:cpn</code>
<code>\cs_new_protected_nopar:cpx</code>

`\cs_new_protected_nopar:Npn <cs> <parms> {<code>}`

Defines a function that does not expand when inside an `x` type expansion.

9.2 Defining new functions using the signature

<code>\cs_new:Nn</code>
<code>\cs_new:Nx</code>
<code>\cs_new:cn</code>
<code>\cs_new:cx</code>

`\cs_new:Nn <cs> {<code>}`

Defines a new function, making sure that `<cs>` is unused so far. The parameter text is automatically detected from the length of the function signature. If `<cs>` is missing a colon in its name, an error is raised. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. `<code>` is either passed literally or may be subject to expansion (under the `x` variants).

T_EXhackers note: Internally, these use T_EX's `\long`. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

<code>\cs_new_nopar:Nn</code>
<code>\cs_new_nopar:Nx</code>
<code>\cs_new_nopar:cn</code>
<code>\cs_new_nopar:cx</code>

`\cs_new_nopar:Nn <cs> {<code>}`

Version of the above in which `\par` is not allowed to appear within the argument(s) of the defined functions.

<code>\cs_new_protected:Nn</code>
<code>\cs_new_protected:Nx</code>
<code>\cs_new_protected:cn</code>
<code>\cs_new_protected:cx</code>

`\cs_new_protected:Nn <cs> {<code>}`

Defines a function that is both robust and may contain `\par` tokens in the argument(s) when called.

<code>\cs_new_protected_nopar:Nn</code>
<code>\cs_new_protected_nopar:Nx</code>
<code>\cs_new_protected_nopar:cn</code>
<code>\cs_new_protected_nopar:cx</code>

`\cs_new_protected_nopar:Nn <cs> {<code>}`

Defines a function that does not expand when inside an `x` type expansion. `\par` is not allowed in the argument(s) of the defined function.

9.3 Defining functions using primitive parameter text

Besides the function definitions that check whether or not their argument is an unused function we need function definitions that overwrite currently used definitions. The following functions are provided for this purpose.

<code>\cs_set:Npn</code>
<code>\cs_set:Npx</code>
<code>\cs_set:cpn</code>
<code>\cs_set:cpx</code>

`\cs_set:Npn <cs> <parms> {<code>}`

Like `\cs_set_nopar:Npn` but allows `\par` tokens in the arguments of the function being defined.

T_EXhackers note: These are equivalent to T_EX’s `\long\def` and so on. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn’t have been expected.

<code>\cs_gset:Npn</code>
<code>\cs_gset:Npx</code>
<code>\cs_gset:cpn</code>
<code>\cs_gset:cpx</code>

`\cs_gset:Npn <cs> <parms> {<code>}`

Global variant of `\cs_set:Npn`.

<code>\cs_set_nopar:Npn</code>
<code>\cs_set_nopar:Npx</code>
<code>\cs_set_nopar:cpn</code>
<code>\cs_set_nopar:cpx</code>

`\cs_set_nopar:Npn <cs> <parms> {<code>}`

Like `\cs_new_nopar:Npn` etc. but does not check the `<cs>` name.

T_EXhackers note: `\cs_set_nopar:Npn` is the L^AT_EX3 name for T_EX’s `\def` and `\cs_set_nopar:Npx` corresponds to the primitive `\edef`. The `\cs_set_nopar:cpn` function was known in L^AT_EX2 as `\@namedef`. `\cs_set_nopar:cpx` has no equivalent.

<code>\cs_gset_nopar:Npn</code>
<code>\cs_gset_nopar:Npx</code>
<code>\cs_gset_nopar:cpn</code>
<code>\cs_gset_nopar:cpx</code>

`\cs_gset_nopar:Npn <cs> <parms> {<code>}`

Like `\cs_set_nopar:Npn` but defines the `<cs>` globally.

T_EXhackers note: `\cs_gset_nopar:Npn` and `\cs_gset_nopar:Npx` are T_EX’s `\gdef` and `\xdef`.

<code>\cs_set_protected:Npn</code>
<code>\cs_set_protected:Npx</code>
<code>\cs_set_protected:cpn</code>
<code>\cs_set_protected:cpx</code>

`\cs_set_protected:Npn <cs> <parms> {<code>}`

Naturally robust macro that won't expand in an `x` type argument. These varieties allow `\par` tokens in the arguments of the function being defined.

<code>\cs_gset_protected:Npn</code>
<code>\cs_gset_protected:Npx</code>
<code>\cs_gset_protected:cpn</code>
<code>\cs_gset_protected:cpx</code>

`\cs_gset_protected:Npn <cs> <parms> {<code>}`

Global versions of the above functions.

<code>\cs_set_protected_nopar:Npn</code>
<code>\cs_set_protected_nopar:Npx</code>
<code>\cs_set_protected_nopar:cpn</code>
<code>\cs_set_protected_nopar:cpx</code>

`\cs_set_protected_nopar:Npn <cs> <parms> {<code>}`

Naturally robust macro that won't expand in an `x` type argument. If you want for some reason to expand it inside an `x` type expansion, prefix it with `\exp_after:wN \prg_do_nothing:`.

<code>\cs_gset_protected_nopar:Npn</code>
<code>\cs_gset_protected_nopar:Npx</code>
<code>\cs_gset_protected_nopar:cpn</code>
<code>\cs_gset_protected_nopar:cpx</code>

`\cs_gset_protected_nopar:Npn <cs> <parms> {<code>}`

Global versions of the above functions.

9.4 Defining functions using the signature (no checks)

As above but now detecting the parameter text from inspecting the signature.

<code>\cs_set:Nn</code>
<code>\cs_set:Nx</code>
<code>\cs_set:cn</code>
<code>\cs_set:cx</code>

`\cs_set:Nn <cs> {<code>}`

Like `\cs_set_nopar:Nn` but allows `\par` tokens in the arguments of the function being defined.

<code>\cs_gset:Nn</code>
<code>\cs_gset:Nx</code>
<code>\cs_gset:cn</code>
<code>\cs_gset:cx</code>

`\cs_gset:Nn <cs> {<code>}`

Global variant of `\cs_set:Nn`.

<code>\cs_set_nopar:Nn</code>
<code>\cs_set_nopar:Nx</code>
<code>\cs_set_nopar:cn</code>
<code>\cs_set_nopar:cx</code>

`\cs_set_nopar:Nn <cs> {<code>}`

Like `\cs_new_nopar:Nn` etc. but does not check the `<cs>` name.

<code>\cs_gset_nopar:Nn</code>
<code>\cs_gset_nopar:Nx</code>
<code>\cs_gset_nopar:cn</code>
<code>\cs_gset_nopar:cx</code>

`\cs_gset_nopar:Nn <cs> {<code>}`

Like `\cs_set_nopar:Nn` but defines the `<cs>` globally.

<code>\cs_set_protected:Nn</code>
<code>\cs_set_protected:cn</code>
<code>\cs_set_protected:Nx</code>
<code>\cs_set_protected:cx</code>

`\cs_set_protected:Nn <cs> {<code>}`

Naturally robust macro that won't expand in an `x` type argument. These varieties also allow `\par` tokens in the arguments of the function being defined.

<code>\cs_gset_protected:Nn</code>
<code>\cs_gset_protected:cn</code>
<code>\cs_gset_protected:Nx</code>
<code>\cs_gset_protected:cx</code>

`\cs_gset_protected:Nn <cs> {<code>}`

Global versions of the above functions.

<code>\cs_set_protected_nopar:Nn</code>
<code>\cs_set_protected_nopar:cn</code>
<code>\cs_set_protected_nopar:Nx</code>
<code>\cs_set_protected_nopar:cx</code>

`\cs_set_protected_nopar:Nn <cs> {<code>}`

Naturally robust macro that won't expand in an `x` type argument. This also comes as a `long` version. If you for some reason want to expand it inside an `x` type expansion, prefix it with `\exp_after:wN \prg_do_nothing:.`

<code>\cs_gset_protected_nopar:Nn</code> <code>\cs_gset_protected_nopar:cn</code> <code>\cs_gset_protected_nopar:Nx</code> <code>\cs_gset_protected_nopar:cx</code>	<code>\cs_gset_protected_nopar:Nn <cs> {<code>}</code>
--	--

Global versions of the above functions.

9.5 Undefined functions

<code>\cs_undefine:N</code> <code>\cs_undefine:c</code> <code>\cs_gundefine:N</code> <code>\cs_gundefine:c</code>	<code>\cs_gundefine:N <cs></code>
--	---

Undefines the control sequence locally or globally. In a global context, this is useful for reclaiming a small amount of memory but shouldn't often be needed for this purpose. In a local context, this can be useful if you need to clear a definition before applying a short-term modification to something.

9.6 Copying function definitions

<code>\cs_new_eq:NN</code> <code>\cs_new_eq:cN</code> <code>\cs_new_eq:Nc</code> <code>\cs_new_eq:cc</code>	<code>\cs_new_eq:NN <cs₁> <cs₂></code>
--	--

Gives the function $\langle cs_1 \rangle$ locally or globally the current meaning of $\langle cs_2 \rangle$. If $\langle cs_1 \rangle$ already exists then an error is called.

<code>\cs_set_eq:NN</code> <code>\cs_set_eq:cN</code> <code>\cs_set_eq:Nc</code> <code>\cs_set_eq:cc</code> <code>\cs_gset_eq:NN</code> <code>\cs_gset_eq:cN</code> <code>\cs_gset_eq:Nc</code> <code>\cs_gset_eq:cc</code>	<code>\cs_set_eq:cN <cs₁> <cs₂></code>
--	--

Gives the function $\langle cs_1 \rangle$ the current meaning of $\langle cs_2 \rangle$. Again, we may always do this globally.

<code>\cs_set_eq:NwN</code>	<code>\cs_set_eq:NwN <cs₁> <cs₂></code> <code>\cs_set_eq:NwN <cs₁> = <cs₂></code>
-----------------------------	--

These functions assign the meaning of $\langle cs_2 \rangle$ locally or globally to the function $\langle cs_1 \rangle$.

Because the \TeX primitive operation is being used which may have an equal sign and (a certain number of) spaces between $\langle cs_1 \rangle$ and $\langle cs_2 \rangle$ the name contains a `w`. (Not happy about this convention!).

\TeX hackers note: `\cs_set_eq:NwN` is the \LaTeX 3 name for \TeX 's `\let`.

9.7 Internal functions

<code>\pref_global:D</code> <code>\pref_long:D</code> <code>\pref_protected:D</code>	<code>\pref_global:D \cs_set_nopar:Npn</code>
--	---

Prefix functions that can be used in front of some definition functions (namely ...). The result of prefixing a function definition with `\pref_global:D` makes the definition global, `\pref_long:D` change the argument scanning mechanism so that it allows `\par` tokens in the argument of the prefixed function, and `\pref_protected:D` makes the definition robust in `\writes` etc.

None of these internal functions should be used by a programmer since the necessary combinations are all available as separate function, e.g., `\cs_set:Npn` is internally implemented as `\pref_long:D \cs_set_nopar:Npn`.

\TeX hackers note: These prefixes are the primitives `\global`, `\long`, and `\protected`. The `\outer` prefix isn't used at all within \LaTeX 3 because ... (it causes more hassle than it's worth? It's never proved useful in any meaningful way?)

10 The innards of a function

<code>\cs_to_str:N *</code>	<code>\cs_to_str:N \langle cs \rangle</code>
-----------------------------	--

This function returns the name of $\langle cs \rangle$ as a sequence of letters with the escape character removed.

<code>\token_to_str:N *</code> <code>\token_to_str:c *</code>	<code>\token_to_str:N \langle arg \rangle</code>
--	--

This function return the name of $\langle arg \rangle$ as a sequence of letters including the escape character.

\TeX hackers note: This is \TeX 's `\string`.

<code>\token_to_meaning:N *</code>

`\token_to_meaning:N <arg>`

This function returns the type and definition of `<arg>` as a sequence of letters.

T_EXhackers note: This is T_EX's `\meaning`.

<code>\cs_get_function_name:N *</code>
<code>\cs_get_function_signature:N *</code>

`\cs_get_function_name:N \<fn>:<args>`

The **name** variant strips off the leading escape character and the trailing argument specification (including the colon) to return `<fn>`. The **signature** variants does the same but returns the signature `<args>` instead.

<code>\cs_split_function:NN *</code>

`\cs_split_function:NN \<fn>:<args> <post process>`

Strips off the leading escape character, splits off the signature without the colon, informs whether or not a colon was present and then prefixes these results with `<post process>`, i.e., `<post process>{\<name>}{\<signature>}{\<true>/\<false>}`. For example, `\cs_get_function_name:N` is nothing more than `\cs_split_function:NN \<fn>:<args> \use_i:nnn`.

<code>\cs_get_arg_count_from_signature:N *</code>

`\cs_get_arg_count_from_signature:N \<fn>:<args>`

Returns the number of chars in `<args>`, signifying the number of arguments that the function uses.

Other functions regarding arbitrary tokens can be found in the `l3token` module.

11 Grouping and scanning

<code>\scan_stop:</code>

`\scan_stop:`

This function stops T_EX's scanning ahead when ending a number.

T_EXhackers note: This is the T_EX primitive `\relax` renamed.

<code>\group_begin:</code>
<code>\group_end:</code>

`\group_begin: <...> \group_end:`

Encloses `<...>` inside a group.

T_EXhackers note: These are the T_EX primitives `\begingroup` and `\endgroup` renamed.

<code>\group_execute_after:N</code>

`\group_execute_after:N` $\langle token \rangle$

Adds $\langle token \rangle$ to the list of tokens to be inserted after the current group ends (through an explicit or implicit `\group_end:`).

T_EXhackers note: This is T_EX's `\aftergroup`.

12 Checking the engine

<code>\xetex_if_engine:TF</code> \star
--

`\xetex_if_engine:TF` $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

This function detects if we're running a XeT_EX-based format.

<code>\luatex_if_engine:TF</code> \star

`\luatex_if_engine:TF` $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

This function detects if we're running a LuaT_EX-based format.

<code>\c_xetex_is_engine_bool</code> <code>\c_luatex_is_engine_bool</code>

 Boolean variables used for the above functions.

Part IV

The l3expan package

Controlling Expansion of Function Arguments

13 Brief overview

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 L^AT_EX3 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.

14 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 the third argument gets expanded once. If `\seq_gpush:No` wouldn't be defined the example above 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_new_nopar:Npn\seq_gpush:No{\exp_args:NNo\seq_gpush:Nn}
```

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. 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.

14.1 Methods for defining variants

<code>\cs_generate_variant:Nn</code>	<code>\cs_generate_variant:Nn</code> <i><parent control sequence></i> { <i><variant argument specifier></i> }
--------------------------------------	--

The *<parent control sequence>* is first separated into the *<base name>* and *<original>* argument specifier. The *<variant>* is then used to modify this by replacing the beginning of the *<original>* with the *<variant>*. Thus the *<variant>* must be no longer than the *<original>* argument specifier. This new specifier is used to create a modified function which will expand its arguments as required. So for example


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

will create a new function `\foo:cn` which will expand its first argument into a control sequence name and pass the result to `\foo:Nn`. Similarly

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

would generate the functions `\foo:NV` and `\foo:cV` in the same way. `\cs_generate_variant:Nn` can only be applied if the *parent control sequence* is already defined. If the *parent control sequence* is protected then the new sequence will also be protected. The variants are generated globally.

Internal functions

```
\cs_generate_internal_variant:n \cs_generate_internal_variant:n {<args>}
```

Defines the appropriate `\exp_args:N<args>` function, if necessary, to perform the expansion control specified by `<args>`.

15 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster than others. Therefore 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 very fast.
- In general `n`, `x`, and `o` (if not in the last position) will need special processing which is not fast and not expandable, i.e., functions of this type may not work correctly in arguments that are itself subject to `x` expansion. Therefore it is best to use the “expandable” functions (i.e., those that contain only `c`, `N`, `o` or `f` in the last position) whenever possible.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `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. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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. Let's pretend we want to set `\aaa` equal to the control sequence stemming from turning `b \l_tmpa_tl b` into a control sequence. Furthermore we want to store the execution of it in a `<toks>` register. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straight forward approach is

```
\toks_set:No \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

Unfortunately this only puts `\exp_args:Nnc \cs_set_eq:NN \aaa {b \l_tmpa_tl b}` into `\l_tmpa_toks` and not `\cs_set_eq:NwN \aaa = \blurb` as we probably wanted. Using `\toks_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\toks_set:Nf \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

which puts the desired result in `\l_tmpa_toks`. It requires `\toks_set:Nf` to be defined as

```
\cs_set_nopar:Npn \toks_set:Nf {\exp_args:Nnf \toks_set:Nn}
```

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!

16 Manipulating the first argument

```
\exp_args:No ★ \exp_args:No <funct> <arg1> <arg2> ...
```

The first argument of `<funct>` (i.e., `<arg1>`) is expanded once, the result is surrounded by braces and passed to `<funct>`. `<funct>` may have more than one argument—all others are passed unchanged.

<code>\exp_args:Nc</code> ★ <code>\exp_args:cc</code> ★
--

`\exp_args:Nc` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The first argument of $\langle funct \rangle$ (i.e., $\langle arg_1 \rangle$) is expanded until only characters remain. (An internal error occurs if something else is the result of this expansion.) Then the result is turned into a control sequence and passed to $\langle funct \rangle$ as the first argument. $\langle funct \rangle$ may have more than one argument—all others are passed unchanged.

In the `:cc` variant, the $\langle funct \rangle$ control sequence itself is constructed (with the same process as described above) before $\langle arg_1 \rangle$ is turned into a control sequence and passed as its argument.

<code>\exp_args:NV</code> ★

`\exp_args:NV` $\langle funct \rangle$ $\langle register \rangle$

The first argument of $\langle funct \rangle$ (i.e., $\langle register \rangle$) is expanded to its value. By value we mean a number stored in an `int` or `num` register, the length value of a `dim`, `skip` or `muskip` register, the contents of a `toks` register or the unexpanded contents of a `tl var.` register. The value is passed onto $\langle funct \rangle$ in braces.

<code>\exp_args:Nv</code> ★

`\exp_args:Nv` $\langle funct \rangle$ $\{\langle register \rangle\}$

Like the `V` type except the register is given by a list of characters from which a control sequence name is generated.

<code>\exp_args:Nx</code>

`\exp_args:Nx` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The first argument of $\langle funct \rangle$ (i.e., $\langle arg_1 \rangle$) is fully expanded until only unexpandable tokens remain, the result is surrounded by braces and passed to $\langle funct \rangle$. $\langle funct \rangle$ may have more than one argument—all others are passed unchanged. As mentioned before, this type of function is relatively slow.

<code>\exp_args:Nf</code> ★

`\exp_args:Nf` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The first argument of $\langle funct \rangle$ (i.e., $\langle arg_1 \rangle$) undergoes full expansion until the first unexpandable token is encountered, the result is surrounded by braces and passed to $\langle funct \rangle$. $\langle funct \rangle$ may have more than one argument—all others are passed unchanged. Beware of its special behavior as explained above.

17 Manipulating two arguments

<code>\exp_args:NNx</code> <code>\exp_args:Nnx</code> <code>\exp_args:Ncx</code> <code>\exp_args:Nox</code> <code>\exp_args:Nxo</code> <code>\exp_args:Nxx</code>
--

`\exp_args:Nnx` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The above functions all manipulate the first two arguments of $\langle funct \rangle$. They are all slow

and non-expandable.

<code>\exp_args:NNo</code>	<code>*</code>
<code>\exp_args:NNc</code>	<code>*</code>
<code>\exp_args:NNv</code>	<code>*</code>
<code>\exp_args:NNV</code>	<code>*</code>
<code>\exp_args:NNf</code>	<code>*</code>
<code>\exp_args:Nno</code>	<code>*</code>
<code>\exp_args:NnV</code>	<code>*</code>
<code>\exp_args:Nnf</code>	<code>*</code>
<code>\exp_args:Noo</code>	<code>*</code>
<code>\exp_args:Noc</code>	<code>*</code>
<code>\exp_args:Nco</code>	<code>*</code>
<code>\exp_args:Ncf</code>	<code>*</code>
<code>\exp_args:Ncc</code>	<code>*</code>
<code>\exp_args:Nff</code>	<code>*</code>
<code>\exp_args:Nfo</code>	<code>*</code>
<code>\exp_args:NVV</code>	<code>*</code>

`\exp_args:NNo <funct> <arg_1> <arg_2> ...`

These are the fast and expandable functions for the first two arguments.

18 Manipulating three arguments

So far not all possible functions are provided and even the selection below may be reduced in the future as far as the non-expandable functions are concerned.

<code>\exp_args:NNnx</code>
<code>\exp_args:NNox</code>
<code>\exp_args:Nnnx</code>
<code>\exp_args:Nnox</code>
<code>\exp_args:Noox</code>
<code>\exp_args:Ncnx</code>
<code>\exp_args:Nccx</code>

`\exp_args:Nnnx <funct> <arg_1> <arg_2> <arg_3> ...`

All the above functions are non-expandable.

<code>\exp_args:NNNo</code>	<code>*</code>
<code>\exp_args:NNNV</code>	<code>*</code>
<code>\exp_args:NNoo</code>	<code>*</code>
<code>\exp_args:NNno</code>	<code>*</code>
<code>\exp_args:Nnno</code>	<code>*</code>
<code>\exp_args:Nnnc</code>	<code>*</code>
<code>\exp_args:Nooo</code>	<code>*</code>
<code>\exp_args:Nccc</code>	<code>*</code>
<code>\exp_args:NcNc</code>	<code>*</code>
<code>\exp_args:NcNo</code>	<code>*</code>
<code>\exp_args:Ncco</code>	<code>*</code>

`\exp_args:NNoo` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ $\langle arg_3 \rangle$...

These are the fast and expandable functions for the first three arguments.

19 Preventing expansion

<code>\exp_not:N</code>
<code>\exp_not:c</code>
<code>\exp_not:n</code>

`\exp_not:N` $\langle token \rangle$
`\exp_not:n` $\{ \langle token list \rangle \}$

This function will prohibit the expansion of $\langle token \rangle$ in situation where $\langle token \rangle$ would otherwise be replaced by its definition, e.g., inside an argument that is handled by the `x` convention.

T_EXhackers note: `\exp_not:N` is the primitive `\noexpand` renamed and `\exp_not:n` is the ε -T_EX primitive `\unexpanded`.

<code>\exp_not:o</code>
<code>\exp_not:f</code>

`\exp_not:o` $\{ \langle token list \rangle \}$

Same as `\exp_not:n` except $\langle token list \rangle$ is expanded once for the `o` type and for the `f` type the token list is expanded until an unexpandable token is found, and the result of these expansions is then prohibited from being expanded further.

<code>\exp_not:V</code>
<code>\exp_not:v</code>

`\exp_not:V` $\langle register \rangle$
`\exp_not:v` $\{ \langle token list \rangle \}$

The value of $\langle register \rangle$ is retrieved and then passed on to `\exp_not:n` which will prohibit further expansion. The `v` type first creates a control sequence from $\langle token list \rangle$ but is otherwise identical to `V`.

<code>\exp_stop_f:</code>

$\langle f expansion \rangle$... `\exp_stop_f:`

This function stops an `f` type expansion. An example use is one such as

```

\tl_set:Nf \l_tmpa_tl {
  \if_case:w \l_tmpa_int
    \or: \use_i_after_orelse:nw {\exp_stop_f: \textbullet}
    \or: \use_i_after_orelse:nw {\exp_stop_f: \textendash}
    \else: \use_i_after_fi:nw {\exp_stop_f: else-item}
  \fi:
}

```

This ensures the expansion is stopped right after finishing the conditional but without expanding `\textbullet` etc.

TeXhackers note: This function is a space token but it is better to distinguish this expansion stopping token from a desired space token when writing code.

20 Unbraced expansion

<code>\exp_last_unbraced:Nf</code>
<code>\exp_last_unbraced:NV</code>
<code>\exp_last_unbraced:No</code>
<code>\exp_last_unbraced:Nv</code>
<code>\exp_last_unbraced:NcV</code>
<code>\exp_last_unbraced:NNV</code>
<code>\exp_last_unbraced:NNo</code>
<code>\exp_last_unbraced:NNNV</code>
<code>\exp_last_unbraced:NNNo</code>

`\exp_last_unbraced:NV <token> <variable name>`

There are a small number of occasions where the last argument in an expansion run must be expanded unbraced. These functions should only be used inside functions, *not* for creating variants.

Part V

The l3prg package

Program control structures

21 Conditionals and logical operations

Conditional processing in L^AT_EX3 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 typical states returned are $\langle true \rangle$ and $\langle false \rangle$ but other states are possible, say an $\langle error \rangle$ state for erroneous input, e.g., text as input in a function comparing integers.

L^AT_EX3 has two primary forms of conditional flow processing based on these states. One type is predicate functions that turn the returned state into a boolean $\langle true \rangle$ or $\langle false \rangle$. 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 $\langle true \rangle$ or $\langle false \rangle$ values to be used in testing with `\if_predicate:w` or in functions to be described below. The other type 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 $\langle true \rangle$ or $\langle false \rangle$ depending on the result. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure

22 Defining a set of conditional functions

<code>\prg_return_true:</code> <code>\prg_return_false:</code>

These functions exit conditional processing when used in conjunction with the generating functions listed below.

<code>\prg_set_conditional:Nnn</code> <code>\prg_set_conditional:Npnn</code> <code>\prg_new_conditional:Nnn</code> <code>\prg_new_conditional:Npnn</code> <code>\prg_set_protected_conditional:Nnn</code> <code>\prg_set_protected_conditional:Npnn</code> <code>\prg_new_protected_conditional:Nnn</code> <code>\prg_new_protected_conditional:Npnn</code> <code>\prg_set_eq_conditional:NNn</code> <code>\prg_new_eq_conditional:NNn</code>	<code>\prg_set_conditional:Nnn <test> <conds> <code></code> <code>\prg_set_conditional:Npnn <test> <param> <conds> <code></code>
--	---

This defines a conditional *base function* which upon evaluation using `\prg_return_true:` and `\prg_return_false:` to finish branches, returns a state. Currently the states are either $\langle true \rangle$ or $\langle false \rangle$ although this can change as more states may be introduced, say an $\langle error \rangle$ state. $\langle conds \rangle$ is a comma separated list possibly consisting of `p` for denoting a predicate function returning the boolean $\langle true \rangle$ or $\langle false \rangle$ values and `TF`, `T` and `F` for the functions that act on the tokens following in the input stream. The `:Nnn` form implicitly determines the number of arguments from the function being defined whereas the `:Npnn` form expects a primitive parameter text.

An example can easily clarify matters here:

```
\prg_set_conditional:Nnn \foo_if_bar:NN {p,TF,T} {
```

```

\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`, `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the `<conds>` 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 an error if that branch is executed.

23 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 are expandable and expect the input to also be fully expandable (which will generally mean being constructed from predicate functions, possibly nested).

<code>\bool_new:N</code> <code>\bool_new:c</code>	<code>\bool_new:N <bool></code>
--	---------------------------------------

Define a new boolean variable. The initial value is `<false>`. A boolean is actually just either `\c_true_bool` or `\c_false_bool`.

<code>\bool_set_true:N</code>
<code>\bool_set_true:c</code>
<code>\bool_set_false:N</code>
<code>\bool_set_false:c</code>
<code>\bool_gset_true:N</code>
<code>\bool_gset_true:c</code>
<code>\bool_gset_false:N</code>
<code>\bool_gset_false:c</code>

`\bool_gset_false:N <bool>`

Set $\langle bool \rangle$ either $\langle true \rangle$ or $\langle false \rangle$. We can also do this globally.

<code>\bool_set_eq:NN</code>
<code>\bool_set_eq:Nc</code>
<code>\bool_set_eq:cN</code>
<code>\bool_set_eq:cc</code>
<code>\bool_gset_eq:NN</code>
<code>\bool_gset_eq:Nc</code>
<code>\bool_gset_eq:cN</code>
<code>\bool_gset_eq:cc</code>

`\bool_set_eq:NN <bool1> <bool2>`

Set $\langle bool_1 \rangle$ equal to the value of $\langle bool_2 \rangle$.

<code>\bool_if_p:N *</code>
<code>\bool_if:NTF *</code>
<code>\bool_if_p:c *</code>
<code>\bool_if:cTF *</code>

`\bool_if:NNTF <bool> {<true>} {<false>}`
`\bool_if_p:N <bool>`

Test the truth value of $\langle bool \rangle$ and execute the $\langle true \rangle$ or $\langle false \rangle$ code. `\bool_if_p:N` is a predicate function for use in `\if_predicate:w` tests or `\bool_if:nTF`-type functions described below.

<code>\bool_while_do:Nn</code>
<code>\bool_while_do:cn</code>
<code>\bool_until_do:Nn</code>
<code>\bool_until_do:cn</code>
<code>\bool_do_while:Nn</code>
<code>\bool_do_while:cn</code>
<code>\bool_do_until:Nn</code>
<code>\bool_do_until:cn</code>

`\bool_while_do:Nn <bool> {<code>}`
`\bool_until_do:Nn <bool> {<code>}`

The ‘while’ versions execute $\langle code \rangle$ as long as the boolean is true and the ‘until’ versions execute $\langle code \rangle$ as long as the boolean is false. The `while_do` functions execute the body after testing the boolean and the `do_while` functions executes the body first and then tests the boolean.

24 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean\ expressions \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators $\&\&$, $||$ and $!$. 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} ||
  \int_compare_p:n {1=\error} % is skipped
) &&
!(\int_compare_p:n {2=4})
```

is a valid boolean expression. Note that minimal evaluation is carried out whenever possible so that whenever a truth value cannot be changed anymore, the remaining tests within the current group are skipped.

<pre>\bool_if_p:n * \bool_if:nTF *</pre>	<pre>\bool_if:nTF {\boolean expression} {\true} {\false}</pre>
--	--

The functions evaluate the truth value of $\langle boolean\ expression \rangle$ where each predicate is separated by $\&\&$ or $||$ denoting logical ‘And’ and ‘Or’ functions. (and) denote grouping of sub-expressions while $!$ is used to as a prefix to either negate a single expression or a group. Hence

```
\bool_if_p:n{
  \int_compare_p:n {1=1} &&
  (
    \int_compare_p:n {2=3} ||
    \int_compare_p:n {4=4} ||
    \int_compare_p:n {1=\error} % is skipped
  ) &&
  !(\int_compare_p:n {2=4})
}
```

from above returns $\langle true \rangle$.

Logical operators take higher precedence the later in the predicate they appear. “ $\langle x \rangle || \langle y \rangle \&\& \langle z \rangle$ ” is interpreted as the equivalent of “ $\langle x \rangle$ OR [$\langle y \rangle$ AND $\langle z \rangle$]” (but now we have grouping you shouldn’t write this sort of thing, anyway).

`\bool_not_p:n *` `\bool_not_p:n {<boolean expression>}`

Longhand for writing `!(<boolean expression>)` within a boolean expression. Might not stick around.

`\bool_xor_p:nn *` `\bool_xor_p:nn {<boolean expression>} {<boolean expression>}`

Implements an ‘exclusive or’ operation between two boolean expressions. There is no infix operation for this.

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn` `\bool_set:Nn <bool> {<boolean expression>}`

Sets `<bool>` to the logical outcome of evaluating `<boolean expression>`.

25 Case switches

`\prg_case_int:nnn {<integer expr>} {`
`{<integer expr1>} {<code1>}`
`{<integer expr2>} {<code2>}`
`...`
`{<integer exprn>} {<coden>}`
`\prg_case_int:nnn *} {<else case>}`

This function evaluates the first `<integer expr>` and then compares it to the values found in the list. Thus the expression

```
\prg_case_int:nnn{2*5}{
  {5}{Small} {4+6}{Medium} {-2*10}{Negative}
}{Other}
```

evaluates first the term to look for and then tries to find this value in the list of values. If the value is found, the code on its right is executed after removing the remainder of the list. If the value is not found, the `<else case>` is executed. The example above will return “Medium”.

The function is expandable and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

`\prg_case_int:nnn {<dim expr>} {`
`{<dim expr1>} {<code1>}`
`{<dim expr2>} {<code2>}`
`...`
`{<dim exprn>} {<coden>}`
`\prg_case_dim:nnn *} {<else case>}`

This function works just like `\prg_case_int:nnn` except it works for `<dim>` registers.

```

\prg_case_str:nnn {\string} {
  {\string_1} {\code_1}
  {\string_2} {\code_2}
  ...
  {\string_n} {\code_n}
\prg_case_str:nnn * } {\else case}

```

This function works just like `\prg_case_int:nnn` except it compares strings. Each string is evaluated fully using `x` style expansion.

The function is expandable³ and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

```

\prg_case_tl:Nnn <tl var.> {
  <tl var._1> {\code_1} <tl var._2> {\code_2} ... <tl var._n>
  {\code_n}
\prg_case_tl:Nnn * } {\else case}

```

This function works just like `\prg_case_int:nnn` except it compares token list variables.

The function is expandable⁴ and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

26 Generic loops

```

\bool_while_do:nn
\bool_until_do:nn
\bool_do_while:nn
\bool_do_until:nn
\bool_while_do:nn {\boolean expression} {\code}
\bool_until_do:nn {\boolean expression} {\code}

```

The ‘while’ versions execute the code as long as `\boolean expression` is true and the ‘until’ versions execute `\code` as long as `\boolean expression` is false. The `\while_do` functions execute the body after testing the boolean and the `\do_while` functions executes the body first and then tests the boolean.

27 Choosing modes

```

\mode_if_vertical_p: *
\mode_if_vertical:TF * \mode_if_vertical:TF {\true code} {\false code}

```

Determines if `TEX` is in vertical mode or not and executes either `\true code` or `\false code` accordingly.

³Provided you use pdfTeX v1.30 or later

⁴Provided you use pdfTeX v1.30 or later

<code>\mode_if_horizontal_p: *</code> <code>\mode_if_horizontal:TF *</code>	<code>\mode_if_horizontal:TF {\langle true code \rangle} {\langle false code \rangle}</code>
--	--

Determines if T_EX is in horizontal mode or not and executes either $\langle true code \rangle$ or $\langle false code \rangle$ accordingly.

<code>\mode_if_inner_p: *</code> <code>\mode_if_inner:TF *</code>	<code>\mode_if_inner:TF {\langle true code \rangle} {\langle false code \rangle}</code>
--	---

Determines if T_EX is in inner mode or not and executes either $\langle true code \rangle$ or $\langle false code \rangle$ accordingly.

<code>\mode_if_math_p: *</code> <code>\mode_if_math:TF *</code>	<code>\mode_if_math:TF {\langle true code \rangle} {\langle false code \rangle}</code>
--	--

Determines if T_EX is in math mode or not and executes either $\langle true code \rangle$ or $\langle false code \rangle$ accordingly.

T_EXhackers note: This version will choose the right branch even at the beginning of an alignment cell.

28 Alignment safe grouping and scanning

<code>\scan_align_safe_stop:</code>	<code>\scan_align_safe_stop:</code>
-------------------------------------	-------------------------------------

This function gets T_EX on the right track inside an alignment cell but without destroying any kerning.

<code>\group_align_safe_begin:</code> <code>\group_align_safe_end:</code>	<code>\group_align_safe_begin: \langle ... \rangle \group_align_safe_end:</code>
--	--

Encloses $\langle ... \rangle$ inside a group but is safe inside an alignment cell. See the implementation of `\peek_token_generic:NNTF` for an application.

29 Producing n copies

There are often several different requirements for producing multiple copies of something. Sometimes one might want to produce a number of identical copies of a sequence of tokens whereas at other times the goal is to simulate a for loop as known from most real programming languages.

<code>\prg_replicate:nn *</code>

`\prg_replicate:nn {<number>} {<arg>}`
Creates *<number>* copies of *<arg>*. Note that it is expandable.

<code>\prg_stepwise_function:nnnN *</code>
--

`\prg_stepwise_function:nnnN {<start>} {<step>} {<end>} {<function>}`

This function performs *<action>* once for each step starting at *<start>* and ending once *<end>* is passed. *<function>* is placed directly in front of a brace group holding the current number so it should usually be a function taking one argument.

<code>\prg_stepwise_inline:nnnn</code>
--

`\prg_stepwise_inline:nnnn {<start>} {<step>} {<end>} {<action>}`

Same as `\prg_stepwise_function:nnnN` except here *<action>* is performed each time with `##1` as a placeholder for the number currently being tested. This function is not expandable and it is nestable.

<code>\prg_stepwise_variable:nnnNn</code>

`\prg_stepwise_variable:nnnn {<start>} {<step>} {<end>} {<temp-var>} {<action>}`

Same as `\prg_stepwise_inline:nnnn` except here the current value is stored in *<temp-var>* and the programmer can use it in *<action>*. This function is not expandable.

30 Sorting

<code>\prg_quicksort:n</code>

`\prg_quicksort:n { {<item1>} {<item2>} ... {<itemn>} }`
Performs a Quicksort on the token list. The comparisons are performed by the function `\prg_quicksort_compare:nnTF` which is up to the programmer to define. When the sorting process is over, all items are given as argument to the function `\prg_quicksort_function:n` which the programmer also controls.

<code>\prg_quicksort_function:n</code> <code>\prg_quicksort_compare:nnTF</code>	<code>\prg_quicksort_function:n {<element>}</code> <code>\prg_quicksort_compare:nnTF {<element₁>} {<element₂>}</code>
--	--

The two functions the programmer must define before calling `\prg_quicksort:n`. As an example we could define

```
\cs_set_nopar:Npn\prg_quicksort_function:n #1{{#1}}
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2#3#4 {\int_compare:nNnTF{#1}>{#2}}
```

Then the function call

```
\prg_quicksort:n {876234520}
```

would return {0}{2}{2}{3}{4}{5}{6}{7}{8}. An alternative example where one sorts a list of words, `\prg_quicksort_compare:nnTF` could be defined as

```
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2 {
  \int_compare:nNnTF{\tl_compare:nn{#1}{#2}}>\c_zero }
```

30.1 Variable type and scope

```
\prg_variable_get_scope:N * \prg_variable_get_scope:N <variable>
```

Returns the scope (g for global, blank otherwise) for the $\langle variable \rangle$.

```
\prg_variable_get_type:N * \prg_variable_get_type:N <variable>
```

Returns the type of $\langle variable \rangle$ (tl, int, etc.)

30.2 Mapping to variables

```
\prg_new_map_functions:Nn \prg_new_map_functions:Nn <token> {\<name>}
```

Creates a family of mapping functions which can be applied to a token list, dividing the list up at each occurrence of the $\langle token \rangle$. The functions defined will be

- $\langle name \rangle_map_function:NN$
- $\langle name \rangle_map_function:nN$
- $\langle name \rangle_map_inline:Nn$
- $\langle name \rangle_map_inline:nn$
- $\langle name \rangle_map_break:$

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map...` family defined by L^AT_EX3 itself for mapping to comma-separated lists. An error will be raised if the $\langle name \rangle$ has already been used to generate a family of mapping functions. All of the definitions are created globally.

`\prg_set_map_functions:Nn` `\prg_set_map_functions:Nn <token> {<name>}`

Creates a family of mapping functions which can be applied to a token list, dividing the list up at each occurrence of the `<token>`. The functions defined will be

- `\<name>_map_function:NN`
- `\<name>_map_function:nN`
- `\<name>_map_inline:Nn`
- `\<name>_map_inline:nn`
- `\<name>_map_break:`

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map_...` family defined by L^AT_EX3 itself for mapping to comma-separated lists. Any existing definitions for the `<name>` will be overwritten. All of the definitions are created globally.

Part VI

The l3quark package

“Quarks”

A special type of constants in L^AT_EX3 are ‘quarks’. These 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 is weird functions (for example as the stop token (i.e., `\q_stop`). They also permit the following ingenious trick: when you pick up a token in a temporary, and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary to the quark using `\if_meaning:w`. 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.

By convention all constants of type quark start out with `\q_`.

The documentation needs some updating.

31 Functions

`\quark_new:N` `\quark_new:N <quark>`

Defines `<quark>` to be a new constant of type quark.

<code>\quark_if_no_value_p:n *</code> <code>\quark_if_no_value:nTF *</code> <code>\quark_if_no_value_p:N *</code> <code>\quark_if_no_value:NTF *</code>	<code>\quark_if_no_value:nTF {<token list>} {<true code>} {<>false code>}</code> <code>\quark_if_no_value:NTF <tl var.> {<true code>} {<>false code>}</code>
--	---

This tests whether or not *<token list>* contains only the quark `\q_no_value`.

If *<token list>* to be tested is stored in a token list variable use `\quark_if_no_value:NTF`, or `\quark_if_no_value:NF` or check the value directly with `\if_meaning:w`. All those cases are faster then `\quark_if_no_value:nTF` so should be preferred.⁵

T_EXhackers note: But be aware of the fact that `\if_meaning:w` can result in an overflow of T_EX's parameter stack since it leaves the corresponding `\fi:` on the input until the whole replacement text is processed. It is therefore better in recursions to use `\quark_if_no_value:NTF` as it will remove the conditional prior to processing the T or F case and so allows tail-recursion.

<code>\quark_if_nil_p:N *</code> <code>\quark_if_nil:NTF *</code>	<code>\quark_if_nil:NTF <token> {<true code>} {<>false code>}</code>
--	---

This tests whether or not *<token>* is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

<code>\quark_if_nil_p:n *</code> <code>\quark_if_nil_p:V *</code> <code>\quark_if_nil_p:o *</code> <code>\quark_if_nil:nTF *</code> <code>\quark_if_nil:VTF *</code> <code>\quark_if_nil:oTF *</code>	<code>\quark_if_nil:nTF {<tokens>} {<true code>} {<>false code>}</code>
--	--

This tests whether or not *<tokens>* is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

32 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below.

<code>\q_recursion_tail</code>	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.
--------------------------------	---

⁵Clarify semantic of the “n” case ... i think it is not implement according to what we originally intended /FMi

`\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.

<code>\quark_if_recursion_tail_stop:N *</code> <code>\quark_if_recursion_tail_stop:n *</code> <code>\quark_if_recursion_tail_stop:o *</code>	<code>\quark_if_recursion_tail_stop:n {⟨list element⟩}</code> <code>\quark_if_recursion_tail_stop:N ⟨list element⟩</code>
--	--

This tests whether or not $\langle list\ element \rangle$ is equal to `\q_recursion_tail` and then exits, i.e., it gobbles the remainder of the list up to and including `\q_recursion_stop` which *must* be present.

If $\langle list\ element \rangle$ is not under your complete control it is advisable to use the `n`. If you wish to use the `N` form you *must* ensure it is really a single token such as if you have

`\tl_set:Nn \l_tmpa_tl { ⟨list element⟩ }`

<code>\quark_if_recursion_tail_stop_do:Nn *</code> <code>\quark_if_recursion_tail_stop_do:nn *</code> <code>\quark_if_recursion_tail_stop_do:on *</code>	<code>\quark_if_recursion_tail_stop_do:nn</code> <code>{⟨list element⟩} {⟨post action⟩}</code> <code>\quark_if_recursion_tail_stop_do:Nn</code> <code>⟨list element⟩ {⟨post action⟩}</code>
--	--

Same as `\quark_if_recursion_tail_stop:N` except here the second argument is executed after the recursion has been terminated.

33 Constants

`\q_no_value` The canonical ‘missing value quark’ that is returned by certain functions to denote that a requested value is not found in the data structure.

`\q_stop` This constant is used as a marker in parameter text. This allows a scanning function to find the end of some input string.

`\q_nil` This constant represent the nil pointer in pointer structures.

`\q_error` Delimits the end of the computation for purposes of error recovery.

`\q_mark` Used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

Part VII

The l3token package

A token of my appreciation...

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 T_EX, 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 will have two primary function categories: `\token` for anything that deals with tokens and `\peek` for looking ahead in the token stream.

Most of the time we will be using the term ‘token’ but most of the time the function we’re describing can equally well be used on a control sequence as such one is one token as well.

We shall refer to list of tokens as `tlists` and such lists represented by a single control sequence is a ‘token list variable’ `tl var`. Functions for these two types are found in the `l3tl` module.

34 Character tokens

<code>\char_set_catcode:nn</code>	
<code>\char_set_catcode:w</code>	
<code>\char_value_catcode:n</code>	
<code>\char_value_catcode:w</code>	
<code>\char_show_value_catcode:n</code>	
<code>\char_show_value_catcode:w</code>	

`\char_set_catcode:nn {<char number>} {<number>}`
`\char_set_catcode:w <char> = <number>`
`\char_value_catcode:n {<char number>}`
`\char_show_value_catcode:n {<char number>}`

`\char_set_catcode:nn` sets the category code of a character, `\char_value_catcode:n` returns its value for use in integer tests and `\char_show_value_catcode:n` pausing the typesetting and prints the value on the terminal and in the log file. The `:w` form should be avoided. (Will: should we then just not mention it?)

`\char_set_catcode` is more usefully abstracted below.

T_EXhackers note: `\char_set_catcode:w` is the T_EX primitive `\catcode` renamed.

<code>\char_make_escape:n</code>	
<code>\char_make_begin_group:n</code>	
<code>\char_make_end_group:n</code>	
<code>\char_make_math_shift:n</code>	
<code>\char_make_alignment:n</code>	
<code>\char_make_end_line:n</code>	
<code>\char_make_parameter:n</code>	
<code>\char_make_math_superscript:n</code>	
<code>\char_make_math_subscript:n</code>	
<code>\char_make_ignore:n</code>	
<code>\char_make_space:n</code>	
<code>\char_make_letter:n</code>	
<code>\char_make_other:n</code>	
<code>\char_make_active:n</code>	
<code>\char_make_comment:n</code>	<code>\char_make_letter:n {⟨character number⟩}</code>
<code>\char_make_invalid:n</code>	<code>\char_make_letter:n {64}</code>
	<code>\char_make_letter:n {'\@}</code>

Sets the catcode of the character referred to by its *⟨character number⟩*.

<code>\char_make_escape:N</code>	
<code>\char_make_begin_group:N</code>	
<code>\char_make_end_group:N</code>	
<code>\char_make_math_shift:N</code>	
<code>\char_make_alignment:N</code>	
<code>\char_make_end_line:N</code>	
<code>\char_make_parameter:N</code>	
<code>\char_make_math_superscript:N</code>	
<code>\char_make_math_subscript:N</code>	
<code>\char_make_ignore:N</code>	
<code>\char_make_space:N</code>	
<code>\char_make_letter:N</code>	
<code>\char_make_other:N</code>	
<code>\char_make_active:N</code>	
<code>\char_make_comment:N</code>	<code>\char_make_letter:N {⟨character⟩}</code>
<code>\char_make_invalid:N</code>	<code>\char_make_letter:N @</code>
	<code>\char_make_letter:N \%</code>

Sets the catcode of the *⟨character⟩*, which may have to be escaped.

T_EXhackers note: `\char_make_other:N` is L^AT_EX 2_ε's `\@makeother`.

<pre> \char_set_lccode:nn \char_set_lccode:w \char_value_lccode:n \char_value_lccode:w \char_show_value_lccode:n \char_show_value_lccode:w </pre>	<pre> \char_set_lccode:nn {<char>} {<number>} \char_set_lccode:w <char> = <number> \char_value_lccode:n {<char>} \char_show_value_lccode:n {<char>} </pre>
---	--

Set the lower case representation of $\langle char \rangle$ for when $\langle char \rangle$ is being converted in $\backslash\mathrm{tl_to_lowercase:n}$. As above, the $\mathrm{:w}$ form is only for people who really, really know what they are doing.

TeXhackers note: $\backslash\mathrm{char_set_lccode:w}$ is the TeX primitive $\backslash\mathrm{lccode}$ renamed.

<pre> \char_set_uccode:nn \char_set_uccode:w \char_value_uccode:n \char_value_uccode:w \char_show_value_uccode:n \char_show_value_uccode:w </pre>	<pre> \char_set_uccode:nn {<char>} {<number>} \char_set_uccode:w <char> = <number> \char_value_uccode:n {<char>} \char_show_value_uccode:n {<char>} </pre>
---	--

Set the uppercase representation of $\langle char \rangle$ for when $\langle char \rangle$ is being converted in $\backslash\mathrm{tl_to_uppercase:n}$. As above, the $\mathrm{:w}$ form is only for people who really, really know what they are doing.

TeXhackers note: $\backslash\mathrm{char_set_uccode:w}$ is the TeX primitive $\backslash\mathrm{uccode}$ renamed.

<pre> \char_set_sfcode:nn \char_set_sfcode:w \char_value_sfcode:n \char_value_sfcode:w \char_show_value_sfcode:n \char_show_value_sfcode:w </pre>	<pre> \char_set_sfcode:nn {<char>} {<number>} \char_set_sfcode:w <char> = <number> \char_value_sfcode:n {<char>} \char_show_value_sfcode:n {<char>} </pre>
---	--

Set the space factor for $\langle char \rangle$.

TeXhackers note: $\backslash\mathrm{char_set_sfcode:w}$ is the TeX primitive $\backslash\mathrm{sfcode}$ renamed.

<code>\char_set_mathcode:nn</code>	
<code>\char_set_mathcode:w</code>	
<code>\char_gset_mathcode:nn</code>	
<code>\char_gset_mathcode:w</code>	
<code>\char_value_mathcode:n</code>	<code>\char_set_mathcode:nn {<char>} {<number>}</code>
<code>\char_value_mathcode:w</code>	<code>\char_set_mathcode:w <char> = <number></code>
<code>\char_show_value_mathcode:n</code>	<code>\char_value_mathcode:n {<char>}</code>
<code>\char_show_value_mathcode:w</code>	<code>\char_show_value_mathcode:n {<char>}</code>

Set the math code for $\langle char \rangle$.

TeXhackers note: `\char_set_mathcode:w` is the TeX primitive `\mathcode` renamed.

35 Generic tokens

<code>\token_new:Nn</code>	<code>\token_new:Nn <token₁> {<token₂>}</code>
----------------------------	--

Defines $\langle token_1 \rangle$ to globally be a snapshot of $\langle token_2 \rangle$. This will be an implicit representation of $\langle token_2 \rangle$.

<code>\c_group_begin_token</code>
<code>\c_group_end_token</code>
<code>\c_math_shift_token</code>
<code>\c_alignment_tab_token</code>
<code>\c_parameter_token</code>
<code>\c_math_superscript_token</code>
<code>\c_math_subscript_token</code>
<code>\c_space_token</code>
<code>\c_letter_token</code>
<code>\c_other_char_token</code>
<code>\c_active_char_token</code>

Some useful constants. They have category codes 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, and 13 respectively. They are all implicit tokens.

<code>\token_if_group_begin_p:N *</code>	
<code>\token_if_group_begin:NTF *</code>	<code>\token_if_group_begin:NTF <token> {<true>} {<false>}</code>

Check if $\langle token \rangle$ is a begin group token.

<code>\token_if_group_end_p:N *</code>	
<code>\token_if_group_end:NTF *</code>	<code>\token_if_group_end:NTF <token> {<true>} {<false>}</code>

Check if $\langle token \rangle$ is an end group token.

$\backslash token_if_math_shift_p:N \star$ $\backslash token_if_math_shift:N\overline{TF} \star$	$\backslash token_if_math_shift:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a math shift token.

$\backslash token_if_alignment_tab_p:N \star$ $\backslash token_if_alignment_tab:N\overline{TF} \star$	$\backslash token_if_alignment_tab:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

Check if $\langle token \rangle$ is an alignment tab token.

$\backslash token_if_parameter_p:N \star$ $\backslash token_if_parameter:N\overline{TF} \star$	$\backslash token_if_parameter:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a parameter token.

$\backslash token_if_math_superscript_p:N \star$ $\backslash token_if_math_superscript:N\overline{TF} \star$	$\backslash token_if_math_superscript:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a math superscript token.

$\backslash token_if_math_subscript_p:N \star$ $\backslash token_if_math_subscript:N\overline{TF} \star$	$\backslash token_if_math_subscript:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a math subscript token.

$\backslash token_if_space_p:N \star$ $\backslash token_if_space:N\overline{TF} \star$	$\backslash token_if_space:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a space token.

$\backslash token_if_letter_p:N \star$ $\backslash token_if_letter:N\overline{TF} \star$	$\backslash token_if_letter:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

Check if $\langle token \rangle$ is a letter token.

$\backslash token_if_other_char_p:N \star$ $\backslash token_if_other_char:N\overline{TF} \star$	$\backslash token_if_other_char:N\overline{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is an other char token.

<code>\token_if_active_char_p:N *</code> <code>\token_if_active_char:NTF *</code>	<code>\token_if_active_char:NTF <token> {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is an active char token.

<code>\token_if_eq_meaning_p:NN *</code> <code>\token_if_eq_meaning:NNTF *</code>	<code>\token_if_eq_meaning:NNTF <token₁> <token₂> {\true} {\false}</code>
--	---

Check if the meaning of two tokens are identical.

<code>\token_if_eq_catcode_p:NN *</code> <code>\token_if_eq_catcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF <token₁> <token₂> {\true} {\false}</code>
--	---

Check if the category codes of two tokens are equal. If both tokens are control sequences the test will be true.

<code>\token_if_eq_charcode_p:NN *</code> <code>\token_if_eq_charcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF <token₁> <token₂> {\true} {\false}</code>
--	---

Check if the character codes of two tokens are equal. If both tokens are control sequences the test will be true.

<code>\token_if_macro_p:N *</code> <code>\token_if_macro:NTF *</code>	<code>\token_if_macro:NTF <token> {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is a macro.

<code>\token_if_cs_p:N *</code> <code>\token_if_cs:NTF *</code>	<code>\token_if_cs:NTF <token> {\true} {\false}</code>
--	--

Check if $\langle token \rangle$ is a control sequence or not. This can be useful for situations where the next token in the input stream is being looked at and you want to determine what should be done to it.

<code>\token_if_expandable_p:N *</code> <code>\token_if_expandable:NTF *</code>	<code>\token_if_expandable:NTF <token> {\true} {\false}</code>
--	--

Check if $\langle token \rangle$ is expandable or not. Note that $\langle token \rangle$ can very well be an active character.

The next set of functions here are for picking apart control sequences. Sometimes it is useful to know if a control sequence has arguments and if so, how many. Similarly its status with respect to `\long` or `\protected` is good to have. Finally it can be very useful

to know if a control sequence is of a certain type: Is this $\langle toks \rangle$ register we’re trying to to something with really a $\langle toks \rangle$ register at all?

$\backslash token_if_long_macro_p:N \star$ $\backslash token_if_long_macro:N\mathit{TF} \star$	$\backslash token_if_long_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is a “long” macro.

$\backslash token_if_protected_macro_p:N \star$ $\backslash token_if_protected_macro:N\mathit{TF} \star$	$\backslash token_if_long_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is a “protected” macro. This test does *not* return $\langle true \rangle$ if the macro is also “long”, see below.

$\backslash token_if_protected_long_macro_p:N \star$ $\backslash token_if_protected_long_macro:N\mathit{TF} \star$	$\backslash token_if_protected_long_macro:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

Check if $\langle token \rangle$ is a “protected long” macro.

$\backslash token_if_chardef_p:N \star$ $\backslash token_if_chardef:N\mathit{TF} \star$	$\backslash token_if_chardef:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is defined to be a chardef.

$\backslash token_if_mathchardef_p:N \star$ $\backslash token_if_mathchardef:N\mathit{TF} \star$	$\backslash token_if_mathchardef:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is defined to be a mathchardef.

$\backslash token_if_int_register_p:N \star$ $\backslash token_if_int_register:N\mathit{TF} \star$	$\backslash token_if_int_register:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is defined to be an integer register.

$\backslash token_if_dim_register_p:N \star$ $\backslash token_if_dim_register:N\mathit{TF} \star$	$\backslash token_if_dim_register:N\mathit{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Check if $\langle token \rangle$ is defined to be a dimension register.

<code>\token_if_skip_register_p:N *</code> <code>\token_if_skip_register:NTF *</code>	<code>\token_if_skip_register:NTF <token> {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is defined to be a skip register.

<code>\token_if_toks_register_p:N *</code> <code>\token_if_toks_register:NTF *</code>	<code>\token_if_toks_register:NTF <token> {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is defined to be a toks register.

<code>\token_get_prefix_spec:N *</code> <code>\token_get_arg_spec:N *</code> <code>\token_get_replacement_spec:N *</code>	<code>\token_get_arg_spec:N <token></code>
---	--

If token is a macro with definition `\cs_set:Npn\next #1#2{x'#1--#2'y}`, the `prefix` function will return the string `\long`, the `arg` function returns the string `#1#2` and the `replacement` function returns the string `x'#1--#2'y`. If $\langle token \rangle$ isn't a macro, these functions return the `\scan_stop:` token.

If the `arg_spec` contains the string `->`, then the `spec` function will produce incorrect results.

35.1 Useless code: because we can!

<code>\token_if_primitive_p:N *</code> <code>\token_if_primitive:NTF *</code>	<code>\token_if_primitive:NTF <token> {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is a primitive. Probably not a very useful function.

36 Peeking ahead at the next token

<code>\l_peek_token</code> <code>\g_peek_token</code> <code>\l_peek_search_token</code>

Some useful variables. Initially they are set to ?.

<code>\peek_after:NN</code> <code>\peek_gafter:NN</code>	<code>\peek_after:NN <function><token></code>
---	---

Assign $\langle token \rangle$ to `\l_peek_token` and then run $\langle function \rangle$ which should perform some

sort of test on this token. Leaves $\langle token \rangle$ in the input stream. $\backslash peek_gafter:NN$ does this globally to the token $\backslash g_peek_token$.

T_EXhackers note: This is the primitive $\backslash futurelet$ turned into a function.

$\backslash peek_meaning:N\overline{TF}$ $\backslash peek_meaning_ignore_spaces:N\overline{TF}$ $\backslash peek_meaning_remove:N\overline{TF}$ $\backslash peek_meaning_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_meaning:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

$\backslash peek_meaning:N\overline{TF}$ checks (by using $\backslash if_meaning:w$) if $\langle token \rangle$ equals the next token in the input stream and executes either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ accordingly. $\backslash peek_meaning_remove:N\overline{TF}$ does the same but additionally removes the token if found. The $ignore_spaces$ versions skips blank spaces before making the decision.

T_EXhackers note: This is equivalent to L^AT_EX 2_ε's $\backslash @ifnextchar$.

$\backslash peek_charcode:N\overline{TF}$ $\backslash peek_charcode_ignore_spaces:N\overline{TF}$ $\backslash peek_charcode_remove:N\overline{TF}$ $\backslash peek_charcode_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_charcode:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Same as for the $\backslash peek_meaning:N\overline{TF}$ functions above but these use $\backslash if_charcode:w$ to compare the tokens.

$\backslash peek_catcode:N\overline{TF}$ $\backslash peek_catcode_ignore_spaces:N\overline{TF}$ $\backslash peek_catcode_remove:N\overline{TF}$ $\backslash peek_catcode_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_catcode:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

Same as for the $\backslash peek_meaning:N\overline{TF}$ functions above but these use $\backslash if_catcode:w$ to compare the tokens.

$\backslash peek_token_generic:N\overline{N\overline{TF}}$ $\backslash peek_token_remove_generic:N\overline{N\overline{TF}}$	$\backslash peek_token_generic:N\overline{N\overline{TF}} \langle token \rangle \langle function \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

$\backslash peek_token_generic:N\overline{N\overline{TF}}$ looks ahead and checks if the next token in the input stream is equal to $\langle token \rangle$. It uses $\langle function \rangle$ to make that decision. $\backslash peek_token_remove_generic:N\overline{N\overline{TF}}$ does the same thing but additionally removes $\langle token \rangle$ from the input stream if it is found. This also works if $\langle token \rangle$ is either $\backslash c_group_begin_token$ or $\backslash c_group_end_token$.

<code>\peek_execute_branches_meaning:</code> <code>\peek_execute_branches_charcode:</code> <code>\peek_execute_branches_catcode:</code>	<code>\peek_execute_branches_meaning:</code>
---	--

These functions compare the token we are searching for with the token found (after optional ignoring of specific tokens). They come in the usual three versions when T_EX is comparing tokens: meaning, character code, and category code.

Part VIII

The l3int package

Integers/counters

37 Integer values

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* (`'int expr'`).

37.1 Integer expressions

<code>\int_eval:n *</code>	<code>\int_eval:n {⟨integer expression⟩}</code>
----------------------------	---

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 $\{\langle integer\ expression\rangle\}$ may contain the operators $+$, $-$, $*$ and $/$, along with parenthesis $($ and $)$. After two expansions, `\int_eval:n` yields a $\langle integer\ donation\rangle$ which is left in the input stream. This is *not* an $\langle internal\ integer\rangle$, and therefore requires suitable termination if used in a T_EX-style integer assignment.

<code>\int_abs:n *</code>	<code>\int_abs:n {\langle integer\ expression\rangle}</code>
---------------------------	--

Evaluates the $\langle integer\ expression\rangle$ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_div_round:nn *</code>	<code>\int_div_round:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
----------------------------------	--

Evaluates the two $\langle integer\ expressions\rangle$ as described earlier, then calculates the result of dividing the first value by the second, rounding any remainder. Note that division using $/$ is identical to this function. The result is left in the input stream as a $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_div_truncate:nn *</code>	<code>\int_div_truncate:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
-------------------------------------	---

Evaluates the two $\langle integer\ expressions\rangle$ as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using $/$ rounds the result. The result is left in the input stream as a $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_max:nn *</code>	<code>\int_max:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
<code>\int_min:nn *</code>	<code>\int_min:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>

Evaluates the $\langle integer\ expressions\rangle$ as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_mod:nn *</code>	<code>\int_mod:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
----------------------------	--

Evaluates the two $\langle integer\ expressions\rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is left in the input stream as an $\langle integer\ denotation\rangle$ after two expansions.

37.2 Integer variables

<code>\int_new:N</code>	<code>\int_new:N \langle integer\rangle</code>
<code>\int_new:c</code>	<code>\int_new:N \langle integer\rangle</code>

Creates a new $\langle integer\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer\rangle$ will initially be equal to 0.

<code>\int_set_eq:NN</code>
<code>\int_set_eq:cN</code>
<code>\int_set_eq:Nc</code>
<code>\int_set_eq:cc</code>

`\int_set_eq:NN` $\langle integer1 \rangle$ $\langle integer2 \rangle$

Sets the content of $\langle integer1 \rangle$ equal to that of $\langle integer2 \rangle$. This assignment is restricted to the current TeX group level.

<code>\int_gset_eq:NN</code>
<code>\int_gset_eq:cN</code>
<code>\int_gset_eq:Nc</code>
<code>\int_gset_eq:cc</code>

`\int_gset_eq:NN` $\langle integer1 \rangle$ $\langle integer2 \rangle$

Sets the content of $\langle integer1 \rangle$ equal to that of $\langle integer2 \rangle$. This assignment is global and so is not limited by the current TeX group level.

<code>\int_add:Nn</code>
<code>\int_add:cn</code>

`\int_add:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is local.

<code>\int_gadd:Nn</code>
<code>\int_gadd:cn</code>

`\int_gadd:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is global.

<code>\int_decr:N</code>
<code>\int_decr:c</code>

`\int_decr:N` $\langle integer \rangle$

Decreases the value stored in $\langle integer \rangle$ by 1 within the scope of the current TeX group.

<code>\int_gdecr:N</code>
<code>\int_gdecr:c</code>

`\int_gdecr:N` $\langle integer \rangle$

Decreases the value stored in $\langle integer \rangle$ by 1 globally (*i.e.* not limited by the current group level).

<code>\int_incr:N</code>
<code>\int_incr:c</code>

`\int_incr:N` $\langle integer \rangle$

Increases the value stored in $\langle integer \rangle$ by 1 within the scope of the current TeX group.

<code>\int_gincr:N</code>
<code>\int_gincr:c</code>

`\int_gincr:N` $\langle integer \rangle$

Increases the value stored in $\langle integer \rangle$ by 1 globally (*i.e.* not limited by the current group level).

<code>\int_set:Nn</code>
<code>\int_set:cn</code>

`\int_set:Nn <integer> {<integer expression>}`

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`). This assignment is restricted to the current TeX group.

<code>\int_gset:Nn</code>
<code>\int_gset:cn</code>

`\int_gset:Nn <integer> {<integer expression>}`

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`). This assignment is global and is not limited to the current TeX group level.

<code>\int_sub:Nn</code>
<code>\int_sub:cn</code>

`\int_sub:Nn <integer> {<integer expression>}`

Subtracts the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is local.

<code>\int_gsub:Nn</code>
<code>\int_gsub:cn</code>

`\int_gsub:Nn <integer> {<integer expression>}`

Subtracts the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is global.

<code>\int_zero:N</code>
<code>\int_zero:c</code>

`\int_zero:N <integer>`

Sets $\langle integer \rangle$ to 0 within the scope of the current TeX group.

<code>\int_gzero:N</code>
<code>\int_gzero:c</code>

`\int_gzero:N <integer>`

Sets $\langle integer \rangle$ to 0 globally, *i.e.* not restricted by the current TeX group level.

<code>\int_show:N</code>
<code>\int_show:c</code>

`\int_show:N <integer>`

Displays the value of the $\langle integer \rangle$ on the terminal.

<code>\int_use:N *</code>
<code>\int_use:c *</code>

`\int_use:N <integer>`

Recovers the content of a $\langle integer \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle integer \rangle$ is required (such as in the first and third arguments of `\int_compare:nNnTF`).

37.3 Comparing integer expressions

	<code>\int_compare_p:nNn</code>	<code>{⟨integer expression 1⟩} ⟨relation⟩ {⟨integer expression 2⟩}</code>
<code>\int_compare_p:nNn *</code>	<code>\int_compare:nNnTF</code>	<code>{⟨integer expression 1⟩} ⟨relation⟩ {⟨integer expression 2⟩}</code>
<code>\int_compare:nNnTF *</code>		<code>{⟨true code⟩} {⟨false code⟩}</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	<

The branching versions then leave either *⟨true code⟩* or *⟨false code⟩* in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

	<code>\int_compare_p:n</code>	<code>{ ⟨integer expression 1⟩ ⟨relation⟩ ⟨integer expression 2⟩ }</code>
<code>\int_compare_p:n *</code>	<code>\int_compare:nTF</code>	<code>{ ⟨integer expression 1⟩ ⟨relation⟩ ⟨integer expression 2⟩ }</code>
<code>\int_compare:nTF *</code>		<code>{⟨true code⟩} {⟨false code⟩}</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	= or ==
Greater than or equal to	=>
Greater than	>
Less than or equal to	=<
Less than	<
Not equal	!=

The branching versions then leave either *⟨true code⟩* or *⟨false code⟩* in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\int_if_even_p:n *</code>	<code>\int_if_odd_p:n {⟨integer expression⟩}</code>
<code>\int_if_even:nTF *</code>	<code>\int_if_odd:nTF {⟨integer expression⟩}</code>
<code>\int_if_odd_p:n *</code>	
<code>\int_if_odd:nTF *</code>	<code>{⟨true code⟩} {⟨false code⟩}</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. The branching versions then leave

either $\langle \text{true code} \rangle$ or $\langle \text{false code} \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\int_do_while:nNnn *</code>	<code>\int_do_while:nNnn</code> $\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle \{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$
-----------------------------------	--

Evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nNnTF`, and then places the $\langle \text{code} \rangle$ in the input stream if the $\langle \text{relation} \rangle$ is **true**. After the $\langle \text{code} \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **false**.

<code>\int_do_until:nNnn *</code>	<code>\int_do_until:nNnn</code> $\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle \{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$
-----------------------------------	--

Evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nNnTF`, and then places the $\langle \text{code} \rangle$ in the input stream if the $\langle \text{relation} \rangle$ is **false**. After the $\langle \text{code} \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **true**.

<code>\int_until_do:nNnn *</code>	<code>\int_until_do:nNnn</code> $\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle \{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$
-----------------------------------	--

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nNnTF`. If the test is **false** then the $\langle \text{code} \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle \text{relation} \rangle$ is **true**.

<code>\int_while_do:nNnn *</code>	<code>\int_while_do:nNnn</code> $\{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$	$\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle$
-----------------------------------	---	--

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nNnTF`. If the test is **true** then the $\langle \text{code} \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle \text{relation} \rangle$ is **false**.

<code>\int_do_while:nn *</code>	<code>\int_do_while:nn</code> $\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle \{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$
---------------------------------	--

Evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nTF`, and then places the $\langle \text{code} \rangle$ in the input stream if the $\langle \text{relation} \rangle$ is **true**. After the $\langle \text{code} \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **false**.

<code>\int_do_until:nn *</code>	<code>\int_do_until:nn</code> $\{ \langle \text{intexpr}_1 \rangle \} \langle \text{relation} \rangle \{ \langle \text{intexpr}_2 \rangle \} \{ \langle \text{code} \rangle \}$
---------------------------------	--

Evaluates the relationship between the two $\langle \text{integer expressions} \rangle$ as described for `\int_compare:nTF`, and then places the $\langle \text{code} \rangle$ in the input stream if the $\langle \text{relation} \rangle$ is **false**.

After the $\langle code \rangle$ has been processed by \TeX the test will be repeated, and a loop will occur until the test is **true**.

$\backslash\text{int_until_do:nn}$ *	$\backslash\text{int_until_do:nn}$ { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } { $\langle code \rangle$ }
--	---

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for $\backslash\text{int_compare:nTF}$. If the test is **false** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **true**.

$\backslash\text{int_while_do:nn}$ *	$\backslash\text{int_while_do:nn}$ { $\langle intexpr1 \rangle$ $\langle relation \rangle$ $\langle intexpr2 \rangle$ } { $\langle code \rangle$ }
--	--

Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for $\backslash\text{int_compare:nTF}$. If the test is **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **false**.

37.4 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

$\backslash\text{int_to_arabic:n}$ *	$\backslash\text{int_to_arabic:n}$ { $\langle integer\ expression \rangle$ }
--	--

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

$\backslash\text{int_to_alph:n}$ *	$\backslash\text{int_to_alph:n}$ { $\langle integer\ expression \rangle$ }
$\backslash\text{int_to_Alph:n}$ *	

Evaluates the $\langle integer\ expression \rangle$ 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. Thus

$\backslash\text{int_to_alph:n}$ { 1 }

places **a** in the input stream,

$\backslash\text{int_to_alph:n}$ { 26 }

is represented as **z** and

$\backslash\text{int_to_alph:n}$ { 27 }

is converted to ‘aa’. For conversions using other alphabets, use `\int_convert_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.

`\int_to_binary:n *` `\int_to_binary:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the binary representation of the result in the input stream.

`\int_to_hexadecimal:n *` `\int_to_hexadecimal:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the hexadecimal (base 16) representation of the result in the input stream. Upper case letters are used for digits beyond 9.

`\int_to_octal:n *` `\int_to_octal:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the octal (base 8) representation of the result in the input stream.

`\int_to_roman:n *`
`\int_to_Roman:n *` `\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 numerals are letters with category code 11 (letter).

`\int_to_symbol:n *` `\int_to_symbol:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the symbol representation of the result in the input stream. The list of symbols used is equivalent to L^AT_EX 2_ε’s `\@fnsymbol` set.

37.5 Converting from other formats

`\int_from_alph:n *` `\int_from_alph:n {⟨letters⟩}`

Converts the *⟨letters⟩* into the integer (base 10) representation and leaves this in the input stream. The *⟨letters⟩* are treated using the English alphabet only, with ‘a’ equal to 1 through to ‘z’ equal to 26. Either lower or upper case letters may be used. This is the inverse function of `\int_to_alph:n`.

`\int_from_binary:n *` `\int_from_binary:n {⟨binary number⟩}`

Converts the $\langle binary\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream.

`\int_from_hexadecimal:n *` `\int_from_binary:n { $\langle hexadecimal\ number \rangle$ }`

Converts the $\langle hexadecimal\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the $\langle hexadecimal\ number \rangle$ by upper or lower case letters.

`\int_from_octal:n *` `\int_from_octal:n { $\langle octal\ number \rangle$ }`

Converts the $\langle octal\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream.

`\int_from_roman:n *` `\int_from_roman:n { $\langle roman\ numeral \rangle$ }`

Converts the $\langle roman\ numeral \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman\ numeral \rangle$ may be in upper or lower case; if the numeral is not valid then the resulting value will be -1 .

37.6 Low-level conversion functions

As well as the higher-level functions already documented, there are a series of lower-level functions which can be used to carry out generic conversions. These are used to create the higher-level versions documented above.

`\int_convert_from_base_ten:nn *` `\int_convert_from_base_ten:nn { $\langle integer\ expression \rangle$ }`
`{ $\langle base \rangle$ }`

Calculates the value of the $\langle integer\ expression \rangle$ and converts it into the appropriate representation in the $\langle base \rangle$; the later may be given as an integer expression. For bases greater than 10 the higher ‘digits’ are represented by the upper case letters from the English alphabet (with normal category codes). The maximum $\langle base \rangle$ value is 36.

`\int_convert_to_base_ten:nn *` `\int_convert_to_base_ten:nn { $\langle number \rangle$ }`
`{ $\langle base \rangle$ }`

Converts the $\langle number \rangle$ in $\langle base \rangle$ into the appropriate value in base 10. The $\langle number \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base \rangle$ value is 36.

`\int_convert_to_symbols:nnn *` `\int_convert_to_symbols:nnn`
`{ $\langle integer\ expression \rangle$ }` `{ $\langle total\ symbols \rangle$ }`
 `$\langle value\ to\ symbol\ mapping \rangle$`

This is the low-level function for conversion of an *integer expression* into a symbolic form (which will often be 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_convert_to_sybols: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 }
  }
}
```

38 Variables and constants

<code>\l_tmpa_int</code>
<code>\l_tmpb_int</code>
<code>\l_tmpc_int</code>
<code>\g_tmpa_int</code>
<code>\g_tmpb_int</code>

Scratch register for immediate use. They are not used by conditionals or predicate functions.

`\int_const:Nn`

`\int_const:cn`

`\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>* will be set globally to the *<integer expression>*.

`\c_max_int`

The maximum value that can be stored as an integer.

`\c_minus_one`

`\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_hundred_one`

`\c_twohundred_fifty_five`

`\c_twohundred_fifty_six`

`\c_thousand`

`\c_ten_thousand`

`\c_ten_thousand_one`

`\c_ten_thousand_two`

`\c_ten_thousand_three`

`\c_ten_thousand_four`

`\c_twenty_thousand`

Integer values used with primitive tests and assignments: self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_register_int`

Maximum number of registers.

38.1 Internal functions

<code>\int_to_roman:w *</code>

`\int_to_roman:w` $\langle integer \rangle$ $\langle space \rangle$ or $\langle non-expandable\ token \rangle$

Converts $\langle integer \rangle$ to it lowercase roman representation. Note that it produces a string of letters with catcode 12.

T_EXhackers note: This is the T_EX primitive `\romannumeral` renamed.

<code>\int_roman_lcuc_mapping:Nnn</code>
<code>\int_to_roman_lcuc:NN</code>

`\int_roman_lcuc_mapping:Nnn` $\langle roman_char \rangle$ $\{\langle licr \rangle\}$
 $\{\langle LICR \rangle\}$
`\int_to_roman_lcuc:NN` $\langle roman_char \rangle$ $\langle char \rangle$

`\int_roman_lcuc_mapping:Nnn` specifies how the roman numeral $\langle roman_char \rangle$ (i, v, x, l, c, d, or m) should be interpreted when converting the number. $\langle licr \rangle$ is the lower case and $\langle LICR \rangle$ is the uppercase mapping. `\int_to_roman_lcuc:NN` is a recursive function converting the roman numerals.

<code>\int_convert_number_with_rule:nnN</code>
<code>\int_symbol_math_conversion_rule:n</code>
<code>\int_symbol_text_conversion_rule:n</code>

`\int_convert_number_with_rule:nnN` $\{\langle int_1 \rangle\}$ $\{\langle int_2 \rangle\}$
 $\langle function \rangle$

`\int_convert_number_with_rule:nnN` converts $\langle int_1 \rangle$ into letters, symbols, whatever as defined by $\langle function \rangle$. $\langle int_2 \rangle$ denotes the base number for the conversion.

<code>\if_num:w *</code>
<code>\if_int_compare:w *</code>

`\if_num:w` $\langle number_1 \rangle$ $\langle rel \rangle$ $\langle number_2 \rangle$ $\langle true \rangle$ `\else:` $\langle false \rangle$
`\fi:`

Compare two integers using $\langle rel \rangle$, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

<code>\if_case:w *</code>
<code>\or: *</code>

`\if_case:w` $\langle number \rangle$ $\langle case_0 \rangle$ `\or:` $\langle case_1 \rangle$ `\or:` ... `\else:`
 $\langle default \rangle$ `\fi:`

Selects a case to execute based on the value of $\langle number \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle number \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle number \rangle$ is 1, etc. The $\langle number \rangle$ may be a literal, a constant or an integer expression (e.g. using `\int_eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\int_value:w *</code>	<code>\int_value:w <integer></code> <code>\int_value:w <tokens> <optional space></code>
-----------------------------	--

Expands `<tokens>` until an `<integer>` is formed. One space may be gobbled in the process.

T_EXhackers note: This is the T_EX primitive `\number`.

<code>\int_eval:w *</code> <code>\int_eval_end:</code>	<code>\int_eval:w <int expr> \int_eval_end:</code>
---	--

Evaluates `<integer expression>` as described for `\int_eval:n`. The evaluation stops when an unexpandable token with category code other than 12 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.

T_EXhackers note: This is the ε -T_EX primitive `\numexpr`.

<code>\if_int_odd:w *</code>	<code>\if_int_odd:w <tokens> <true> \else: <false> \fi:</code> <code>\if_int_odd:w <number> <true> \else: <false> \fi:</code>
------------------------------	--

Expands `<tokens>` until a non-numeric tokens is found, and tests whether the resulting `<number>` is odd. If so, `<true code>` is executed. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifodd`.

Part IX

The l3skip package

Dimension and skip registers

L^AT_EX3 knows about two types of length registers for internal use: rubber lengths (`skips`) and rigid lengths (`dims`).

39 Skip registers

39.1 Functions

<code>\skip_new:N</code> <code>\skip_new:c</code>	<code>\skip_new:N <skip></code>
--	---------------------------------------

Defines `<skip>` to be a new variable of type `skip`.

T_EXhackers note: `\skip_new:N` is the equivalent to plain T_EX's `\newskip`.

<code>\skip_zero:N</code>
<code>\skip_zero:c</code>
<code>\skip_gzero:N</code>
<code>\skip_gzero:c</code>

`\skip_zero:N <skip>`

Locally or globally reset `<skip>` to zero. For global variables the global versions should be used.

<code>\skip_set:Nn</code>
<code>\skip_set:cn</code>
<code>\skip_gset:Nn</code>
<code>\skip_gset:cn</code>

`\skip_set:Nn <skip> {<skip value>}`

These functions will set the `<skip>` register to the `<length>` value.

<code>\skip_add:Nn</code>
<code>\skip_add:cn</code>
<code>\skip_gadd:Nn</code>
<code>\skip_gadd:cn</code>

`\skip_add:Nn <skip> {<length>}`

These functions will add to the `<skip>` register the value `<length>`. If the second argument is a `<skip>` register too, the surrounding braces can be left out.

<code>\skip_sub:Nn</code>
<code>\skip_gsub:Nn</code>

`\skip_gsub:Nn <skip> {<length>}`

These functions will subtract from the `<skip>` register the value `<length>`. If the second argument is a `<skip>` register too, the surrounding braces can be left out.

<code>\skip_use:N</code>
<code>\skip_use:c</code>

`\skip_use:N <skip>`

This function returns the length value kept in `<skip>` in a way suitable for further processing.

T_EXhackers note: The function `\skip_use:N` could be implemented directly as the T_EX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\skip_show:N</code>
<code>\skip_show:c</code>

`\skip_show:N <skip>`

This function pauses the compilation and displays the length value kept in `<skip>` in the console output and log file.

T_EXhackers note: The function `\skip_show:N` could be implemented directly as the T_EX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\skip_horizontal:N</code>	
<code>\skip_horizontal:c</code>	
<code>\skip_horizontal:n</code>	
<code>\skip_vertical:N</code>	
<code>\skip_vertical:c</code>	<code>\skip_horizontal:N</code> $\langle skip \rangle$
<code>\skip_vertical:n</code>	<code>\skip_horizontal:n</code> $\{\langle length \rangle\}$

The **hor** functions insert $\langle skip \rangle$ or $\langle length \rangle$ with the T_EX primitive `\hskip`. The **vertical** variants do the same with `\vskip`. The **n** versions evaluate $\langle length \rangle$ with `\skip_eval:n`.

<code>\skip_if_infinite_glue_p:n</code>	
<code>\skip_if_infinite_glue:nTF</code>	<code>\skip_if_infinite_glue:nTF</code> $\{\langle skip \rangle\}$ $\{\langle true \rangle\}$ $\{\langle false \rangle\}$

Checks if $\langle skip \rangle$ contains infinite stretch or shrink components and executes either $\langle true \rangle$ or $\langle false \rangle$. Also works on input like `3pt plus .5in`.

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN</code> $\{\langle skip \rangle\}$ $\{\langle action \rangle\}$ $\langle dimen_1 \rangle$ $\langle dimen_2 \rangle$
--	---

Checks if $\langle skip \rangle$ contains finite glue. If it does then it assigns $\langle dimen_1 \rangle$ the stretch component and $\langle dimen_2 \rangle$ the shrink component. If it contains infinite glue set $\langle dimen_1 \rangle$ and $\langle dimen_2 \rangle$ to zero and execute **#2** which is usually an error or warning message of some sort.

<code>\skip_eval:n</code> ★	<code>\skip_eval:n</code> $\{\langle skip expression \rangle\}$
-----------------------------	---

Evaluates the $\langle skip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ 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 $\langle glue denotation \rangle$ after two expansions. This will be expressed in points (**pt**), and will require suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal glue \rangle$.

39.2 Formatting a skip register value

39.3 Variable and constants

<code>\c_max_skip</code>	Constant that denotes the maximum value which can be stored in a $\langle skip \rangle$ register.
--------------------------	---

<code>\c_zero_skip</code>	Constants denoting a zero skip.
---------------------------	---------------------------------

<code>\l_tmpa_skip</code> <code>\l_tmpb_skip</code> <code>\l_tmpc_skip</code> <code>\g_tmpa_skip</code> <code>\g_tmpb_skip</code>	Scratch register for immediate use.
---	-------------------------------------

40 Dim registers

40.1 Functions

<code>\dim_new:N</code> <code>\dim_new:c</code>	<code>\dim_new:N</code> $\langle dim \rangle$
--	---

Defines $\langle dim \rangle$ to be a new variable of type `dim`.

T_EXhackers note: `\dim_new:N` is the equivalent to plain T_EX's `\newdimen`.

<code>\dim_zero:N</code> <code>\dim_zero:c</code> <code>\dim_gzero:N</code> <code>\dim_gzero:c</code>	<code>\dim_zero:N</code> $\langle dim \rangle$
--	--

Locally or globally reset $\langle dim \rangle$ to zero. For global variables the global versions should be used.

<code>\dim_set:Nn</code> <code>\dim_set:Nc</code> <code>\dim_set:cn</code> <code>\dim_gset:Nn</code> <code>\dim_gset:Nc</code> <code>\dim_gset:cn</code> <code>\dim_gset:cc</code>	<code>\dim_set:Nn</code> $\langle dim \rangle$ $\{ \langle dim \text{ value} \rangle \}$
--	--

These functions will set the $\langle dim \rangle$ register to the $\langle dim \text{ value} \rangle$ value.

<code>\dim_set_max:Nn</code> <code>\dim_set_max:cn</code>	<code>\dim_set_max:Nn</code> $\langle dimension \rangle$ $\{ \langle dimension \text{ expression} \rangle \}$
--	---

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension \text{ expression} \rangle$, and sets the $\langle dimension \rangle$ to the larger of these two value. This assignment is local to the current T_EX group.

<code>\dim_gset_max:Nn</code>
<code>\dim_gset_max:cn</code>

`\dim_gset_max:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the larger of these two value. This assignment is global.

<code>\dim_set_min:Nn</code>
<code>\dim_set_min:cn</code>

`\dim_set_min:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the smaller of these two value. This assignment is local to the current \TeX group.

<code>\dim_gset_min:Nn</code>
<code>\dim_gset_min:cn</code>

`\dim_gset_min:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the smaller of these two value. This assignment is global.

<code>\dim_add:Nn</code>
<code>\dim_add:Nc</code>
<code>\dim_add:cn</code>
<code>\dim_gadd:Nn</code>
<code>\dim_gadd:cn</code>

`\dim_add:Nn` $\langle dim \rangle$ $\{ \langle length \rangle \}$

These functions will add to the $\langle dim \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle dim \rangle$ register too, the surrounding braces can be left out.

<code>\dim_sub:Nn</code>
<code>\dim_sub:Nc</code>
<code>\dim_sub:cn</code>
<code>\dim_gsub:Nn</code>
<code>\dim_gsub:cn</code>

`\dim_gsub:Nn` $\langle dim \rangle$ $\{ \langle length \rangle \}$

These functions will subtract from the $\langle dim \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle dim \rangle$ register too, the surrounding braces can be left out.

<code>\dim_use:N</code>
<code>\dim_use:c</code>

`\dim_use:N` $\langle dim \rangle$

This function returns the length value kept in $\langle dim \rangle$ in a way suitable for further processing.

\TeX hackers note: The function `\dim_use:N` could be implemented directly as the \TeX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

```
\dim_show:N
\dim_show:c
```

 $\backslash\mathrm{dim_show:N}\ \langle dim\rangle$

TeXhackers note: The function `\dim_show:N` could be implemented directly as the TeX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

\dim_eval:n ★

$$\backslash\mathrm{dim_eval:n} \{\langle dimension\ expression\rangle\}$$

\if_dim:w

$$\text{\if_dim:w } \langle \textit{dimen}_1 \rangle \langle \textit{rel} \rangle \langle \textit{dimen}_2 \rangle \langle \textit{true} \rangle \text{\else: } \langle \textit{false} \rangle \text{\fi:}$$

T_FXhackers note: This is the T_FX primitive `\ifdim`.

$$\begin{aligned} \backslash \dim_compare_p:n & \star \\ \backslash \dim_compare:n\textit{TF} & \star \end{aligned}$$

$$\begin{array}{l} \backslash \text{dim_compare_p:n} \{ \langle \langle \text{dim expr.}_1 \rangle \langle \text{rel} \rangle \langle \text{dim expr.}_2 \rangle \} \\ \backslash \text{dim_compare:nTF} \{ \langle \langle \text{dim expr.}_1 \rangle \langle \text{rel} \rangle \langle \text{dim expr.}_2 \rangle \} \\ \quad \langle \text{true code} \rangle \langle \text{false code} \rangle \end{array}$$

Less than	<	Less than or equal	<=
Greater than	>	Greater than or equal	>=
Equal	== or =	Not equal	!=

Based on the result of the comparison either the *⟨true code⟩* or *⟨false code⟩* is executed. Both dimension expressions are evaluated fully in the process. Note the syntax, which allows natural input in the style of

```
\dim_compare_p:n {2.54cm != \l_tmpb_int}
```

A single equals sign is available as comparator (in addition to those familiar to C users) as standard TeX practice is to compare values using =.

<code>\dim_compare:nNnTF *</code>	<code>\dim_compare:nNnTF</code>	<code>{\dim expr}</code>	<code>\rel</code>	<code>{\dim expr}</code>
<code>\dim_compare_p:nNn *</code>		<code>{\true}</code>		<code>{\false}</code>

These functions test two dimension expressions against each other. They are both evaluated by `\dim_eval:n`. Note that if both expressions are normal dimension variables as in

```
\dim_compare:nNnTF \l_temp_dim < \c_zero_skip {negative}{non-negative}
```

you can safely omit the braces.

These functions are faster than the `n` variants described above but do not support an extended set of relational operators.

T_EXhackers note: This is the T_EX primitive `\ifdim` turned into a function.

<code>\dim_while_do:nNnn</code>	<code>\dim_while_do:nNnn</code>	<code>\dim expr</code>	<code>\rel</code>	<code>\dim expr</code>	<code>\code</code>
<code>\dim_until_do:nNnn</code>					
<code>\dim_do_while:nNnn</code>					
<code>\dim_do_until:nNnn</code>					

`\dim_while_do:nNnn` tests the dimension expressions and if true performs `\code` repeatedly while the test remains true. `\dim_do_while:nNnn` is similar but executes the body first and then performs the check, thus ensuring that the body is executed at least once. The ‘until’ versions are similar but continue the loop as long as the test is false.

40.2 Variable and constants

<code>\c_max_dim</code>	Constant that denotes the maximum value which can be stored in a <code>\dim</code> register.
-------------------------	--

<code>\c_zero_dim</code>	Set of constants denoting useful values.
--------------------------	--

<code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code> <code>\l_tmpc_dim</code> <code>\l_tmpd_dim</code> <code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code>	Scratch register for immediate use.
--	-------------------------------------

41 Muskips

<code>\muskip_new:N</code>

`\muskip_new:N $\langle muskip \rangle$`

T_EXhackers note: Defines $\langle muskip \rangle$ to be a new variable of type `muskip`. `\muskip_new:N` is the equivalent to plain T_EX's `\newmuskip`.

<code>\muskip_set:Nn</code> <code>\muskip_gset:Nn</code>

`\muskip_set:Nn $\langle muskip \rangle$ { $\langle muskip value \rangle$ }`

These functions will set the $\langle muskip \rangle$ register to the $\langle length \rangle$ value.

<code>\muskip_add:Nn</code> <code>\muskip_gadd:Nn</code>

`\muskip_add:Nn $\langle muskip \rangle$ { $\langle length \rangle$ }`

These functions will add to the $\langle muskip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle muskip \rangle$ register too, the surrounding braces can be left out.

<code>\muskip_sub:Nn</code> <code>\muskip_gsub:Nn</code>

`\muskip_gsub:Nn $\langle muskip \rangle$ { $\langle length \rangle$ }`

These functions will subtract from the $\langle muskip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle muskip \rangle$ register too, the surrounding braces can be left out.

<code>\muskip_use:N</code>

`\muskip_use:N $\langle muskip \rangle$`

This function returns the length value kept in $\langle muskip \rangle$ in a way suitable for further processing.

T_EXhackers note: See note for `\dim_use:N`.

<code>\muskip_show:N</code>

`\muskip_show:N $\langle muskip \rangle$`

This function pauses the compilation and displays the length value kept in $\langle muskip \rangle$ in the console output and log file.

Part X

The l3tl package

Token Lists

L^AT_EX3 stores token lists in variables also called ‘token lists’. Variables of this type get the suffix `tl` and functions of this type have the prefix `tl`. To use a token list variable you simply call the corresponding variable.

Often you find yourself with not a token list variable but an arbitrary token list which has to undergo certain tests. We will *also* prefix these functions with `tl`. While token list variables are always single tokens, token lists are always surrounded by braces.

42 Functions

<code>\tl_new:N</code>
<code>\tl_new:c</code>
<code>\tl_new:Nn</code>
<code>\tl_new:cn</code>
<code>\tl_new:Nx</code>

`\tl_new:Nn <tl var.> {<initial token list>}`

Defines `<tl var.>` globally to be a new variable to store a token list. `<initial token list>` is the initial value of `<tl var.>`. This makes it possible to assign values to a constant token list variable.

The form `\tl_new:N` initializes the token list variable with an empty value.

<code>\tl_const:Nn</code>

`\tl_const:Nn <tl var.> {<token list>}`

Defines `<tl var.>` as a global constant expanding to `<token list>`. The name of the constant must be free when the constant is created.

<code>\tl_use:N</code>
<code>\tl_use:c</code>

`\tl_use:N <tl var.>`

Function that inserts the `<tl var.>` into the processing stream. Instead of `\tl_use:N` simply placing the `<tl var.>` into the input stream is also supported. `\tl_use:c` will complain if the `<tl var.>` hasn’t been declared previously!

<code>\tl_show:N</code>
<code>\tl_show:c</code>
<code>\tl_show:n</code>

`\tl_show:N <tl var.>`
`\tl_show:n {<token list>}`

Function that pauses the compilation and displays the `<tl var.>` or `<token list>` on the console output and in the log file.

<code>\tl_set:Nn</code>
<code>\tl_set:Nc</code>
<code>\tl_set:NV</code>
<code>\tl_set:No</code>
<code>\tl_set:Nv</code>
<code>\tl_set:Nf</code>
<code>\tl_set:Nx</code>
<code>\tl_set:cn</code>
<code>\tl_set:co</code>
<code>\tl_set:cV</code>
<code>\tl_set:cx</code>
<code>\tl_gset:Nn</code>
<code>\tl_gset:Nc</code>
<code>\tl_gset:No</code>
<code>\tl_gset:NV</code>
<code>\tl_gset:Nv</code>
<code>\tl_gset:Nx</code>
<code>\tl_gset:cn</code>
<code>\tl_gset:cx</code>

`\tl_set:Nn <tl var.> {<token list>}`

Defines $\langle tl\ var.\rangle$ to hold the token list $\langle token\ list\rangle$. Global variants of this command assign the value globally the other variants expand the $\langle token\ list\rangle$ up to a certain level before the assignment or interpret the $\langle token\ list\rangle$ as a character list and form a control sequence out of it.

<code>\tl_clear:N</code>
<code>\tl_clear:c</code>
<code>\tl_gclear:N</code>
<code>\tl_gclear:c</code>

`\tl_clear:N <tl var.>`

The $\langle tl\ var.\rangle$ is locally or globally cleared. The `c` variants will generate a control sequence name which is then interpreted as $\langle tl\ var.\rangle$ before clearing.

<code>\tl_clear_new:N</code>
<code>\tl_clear_new:c</code>
<code>\tl_gclear_new:N</code>
<code>\tl_gclear_new:c</code>

`\tl_clear_new:N <tl var.>`

These functions check if $\langle tl\ var.\rangle$ exists. If it does it will be cleared; if it doesn't it will be allocated.

<code>\tl_put_left:Nn</code>
<code>\tl_put_left:NV</code>
<code>\tl_put_left:No</code>
<code>\tl_put_left:Nx</code>
<code>\tl_put_left:cn</code>
<code>\tl_put_left:cV</code>
<code>\tl_put_left:co</code>

`\tl_put_left:Nn <tl var.> {<token list>}`

These functions will append *<token list>* to the left of *<tl var.>*. *<token list>* might be subject to expansion before assignment.

<code>\tl_put_right:Nn</code>
<code>\tl_put_right:NV</code>
<code>\tl_put_right:No</code>
<code>\tl_put_right:Nx</code>
<code>\tl_put_right:cn</code>
<code>\tl_put_right:cV</code>
<code>\tl_put_right:co</code>

`\tl_put_right:Nn <tl var.> {<token list>}`

These functions append *<token list>* to the right of *<tl var.>*.

<code>\tl_gput_left:Nn</code>
<code>\tl_gput_left:No</code>
<code>\tl_gput_left:NV</code>
<code>\tl_gput_left:Nx</code>
<code>\tl_gput_left:cn</code>
<code>\tl_gput_left:co</code>
<code>\tl_gput_left:cV</code>

`\tl_gput_left:Nn <tl var.> {<token list>}`

These functions will append *<token list>* globally to the left of *<tl var.>*.

<code>\tl_gput_right:Nn</code>
<code>\tl_gput_right:No</code>
<code>\tl_gput_right:NV</code>
<code>\tl_gput_right:Nx</code>
<code>\tl_gput_right:cn</code>
<code>\tl_gput_right:co</code>
<code>\tl_gput_right:cV</code>

`\tl_gput_right:Nn <tl var.> {<token list>}`

These functions will globally append *<token list>* to the right of *<tl var.>*.

A word of warning is appropriate here: Token list variables are implemented as macros and as such currently inherit some of the peculiarities of how T_EX handles #s in the argument of macros. In particular, the following actions are legal

```
\tl_set:Nn \l_tmpa_tl{##1}
\tl_put_right:Nn \l_tmpa_tl{##2}
\tl_set:No \l_tmpb_tl{\l_tmpa_tl ##3}
```

x type expansions where macros being expanded contain #s do not work and will not work until there is an `\expanded` primitive in the engine. If you want them to work you must double #s another level.

<code>\tl_set_eq:NN</code>
<code>\tl_set_eq:Nc</code>
<code>\tl_set_eq:cN</code>
<code>\tl_set_eq:cc</code>
<code>\tl_gset_eq:NN</code>
<code>\tl_gset_eq:Nc</code>
<code>\tl_gset_eq:cN</code>
<code>\tl_gset_eq:cc</code>

`\tl_set_eq:NN <tl var. 1> <tl var. 2>`

Fast form for `\tl_set:No <tl var. 1> {\<tl var. 2>}`
when `<tl var. 2>` is known to be a variable of type `tl`.

<code>\tl_to_str:N</code>
<code>\tl_to_str:c</code>

`\tl_to_str:N <tl var. >`

This function returns the token list kept in `<tl var. >` as a string list with all characters catcoded to ‘other’.

<code>\tl_to_str:n</code>

`\tl_to_str:n {\<token list>}`

This function turns its argument into a string where all characters have catcode ‘other’.

TeXhackers note: This is the ε -TeX primitive `\detokenize`.

<code>\tl_rescan:nn</code>

`\tl_rescan:nn {\<catcode setup>} {\<token list>}`

Returns the result of re-tokenising `<token list>` with the catcode setup (and whatever other redefinitions) specified. This is useful because the catcodes of characters are ‘frozen’ when first tokenised; this allows their meaning to be changed even after they’ve been read as an argument. Also see `\tl_set_rescan:Nnn` below.

TeXhackers note: This is a wrapper around ε -TeX’s `\scantokens`.

<code>\tl_set_rescan:Nnn</code>
<code>\tl_set_rescan:Nno</code>
<code>\tl_set_rescan:Nnx</code>
<code>\tl_gset_rescan:Nnn</code>
<code>\tl_gset_rescan:Nno</code>
<code>\tl_gset_rescan:Nnx</code>

`\tl_set_rescan:Nnn <tl var. > {\<catcode setup>} {\<token list>}`

Sets `<tl var. >` to the result of re-tokenising `<token list>` with the catcode setup (and whatever other redefinitions) specified.

T_EXhackers note: This is a wrapper around ε -T_EX’s `\scantokens`.

43 Predicates and conditionals

<code>\tl_if_empty_p:N *</code>
<code>\tl_if_empty_p:c *</code>

`\tl_if_empty_p:N <tl var.>`

This predicate returns ‘true’ if $\langle tl\ var.\rangle$ is ‘empty’ i.e., doesn’t contain any tokens.

<code>\tl_if_empty:NTF *</code>
<code>\tl_if_empty:cTF *</code>

`\tl_if_empty:NTF <tl var.> {\langle true code\rangle} {\langle false code\rangle}`

Execute $\langle true\ code\rangle$ if $\langle tl\ var.\rangle$ is empty and $\langle false\ code\rangle$ if it contains any tokens.

<code>\tl_if_eq_p:NN *</code>
<code>\tl_if_eq_p:cN *</code>
<code>\tl_if_eq_p:Nc *</code>
<code>\tl_if_eq_p:cc *</code>

`\tl_if_eq_p:NN <tl var.1> <tl var.2>`

Predicate function which returns ‘true’ if the two token list variables are identical and ‘false’ otherwise.

<code>\tl_if_eq:NNTF *</code>
<code>\tl_if_eq:cNTF *</code>
<code>\tl_if_eq:NcTF *</code>
<code>\tl_if_eq:ccTF *</code>

`\tl_if_eq:NNTF <tl var.1> <tl var.2> {\langle true code\rangle} {\langle false code\rangle}`

Execute $\langle true\ code\rangle$ if $\langle tl\ var.₁\rangle$ holds the same token list as $\langle tl\ var.₂\rangle$ and $\langle false\ code\rangle$ otherwise.

<code>\tl_if_empty_p:n *</code>
<code>\tl_if_empty_p:V *</code>
<code>\tl_if_empty_p:o *</code>
<code>\tl_if_empty:nTF</code>
<code>\tl_if_empty:VTF</code>
<code>\tl_if_empty:oTF</code>

`\tl_if_empty:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}`

Execute $\langle true\ code\rangle$ if $\langle token\ list\rangle$ doesn’t contain any tokens and $\langle false\ code\rangle$ otherwise.

<code>\tl_if_eq:nnTF</code>

`\tl_if_eq:nnTF <token list1> {\langle token list2\rangle} {\langle true code\rangle} {\langle false code\rangle}`

Tests if $\langle token\ list1\rangle$ and $\langle token\ list2\rangle$ both in respect of character codes and category codes. Either the $\langle true\ code\rangle$ or $\langle false\ code\rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

<code>\tl_if_blank_p:n *</code>
<code>\tl_if_blank:nTF *</code>
<code>\tl_if_blank_p:V *</code>
<code>\tl_if_blank_p:o *</code>
<code>\tl_if_blank:VTF *</code>
<code>\tl_if_blank:oTF *</code>

`\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}`

Execute *<true code>* if *<token list>* is blank meaning that it is either empty or contains only blank spaces.

<code>\tl_if_single_p:n *</code>
<code>\tl_if_single:nTF *</code>
<code>\tl_if_single_p:N *</code>
<code>\tl_if_single:NTF *</code>

`\tl_if_single:NTF {<tl var.>} {<true code>} {<false code>}`

`\tl_if_single:nTF {<token list>} {<true code>} {<false code>}`

Conditional returning true if the token list or the contents of the *tl var.* consists of a single token only.

Note that an input of ‘space’⁶ returns *<true>* from this function.

<code>\tl_to_lowercase:n</code>
<code>\tl_to_uppercase:n</code>

`\tl_to_lowercase:n {<token list>}`

`\tl_to_lowercase:n` converts all tokens in *<token list>* to their lower case representation. Similar for `\tl_to_uppercase:n`.

T_EXhackers note: These are the T_EX primitives `\lowercase` and `\uppercase` renamed.

44 Working with the contents of token lists

<code>\tl_map_function:nN *</code>
<code>\tl_map_function:NN</code>
<code>\tl_map_function:cN</code>

`\tl_map_function:nN {<token list>} <function>`

`\tl_map_function:NN <tl var.> <function>`

Runs through all elements in a *<token list>* from left to right and places *<function>* in front of each element. As this function will also pick up elements in brace groups, the element is returned with braces and hence *<function>* should be a function with a `:n` suffix even though it may very well only deal with a single token.

This function uses a purely expandable loop function and will stay so as long as *<function>* is expandable too.

⁶But remember any number of consecutive spaces are read as a single space by T_EX.

<code>\tl_map_inline:nn</code>
<code>\tl_map_inline:Nn</code>
<code>\tl_map_inline:cn</code>

<code>\tl_map_inline:nn</code> $\{\langle token\ list\rangle\}$ $\{\langle inline\ function\rangle\}$
<code>\tl_map_inline:Nn</code> $\langle tl\ var.\rangle$ $\{\langle inline\ function\rangle\}$

Allows a syntax like `\tl_map_inline:nn` $\{\langle token\ list\rangle\}$ $\{\backslash token_to_str:N\ \#\#1\}$. This renders it non-expandable though. Remember to double the #s for each level.

<code>\tl_map_variable:nNn</code>
<code>\tl_map_variable:NNn</code>
<code>\tl_map_variable:cNn</code>

<code>\tl_map_variable:nNn</code> $\{\langle token\ list\rangle\}$ $\langle temp\rangle$ $\{\langle action\rangle\}$
<code>\tl_map_variable:NNn</code> $\langle tl\ var.\rangle$ $\langle temp\rangle$ $\{\langle action\rangle\}$

Assigns $\langle temp\rangle$ to each element on $\langle token\ list\rangle$ and executes $\langle action\rangle$. As there is an assignment in this process it is not expandable.

T_EXhackers note: This is the L^AT_EX2 function `\@tfor` but with a more sane syntax. Also it works by tail recursion and so is faster as lists grow longer.

<code>\tl_map_break:</code>

`\tl_map_break:`

For breaking out of a loop. Must not be nested inside a primitive `\if` structure.

<code>\tl_reverse:n</code>
<code>\tl_reverse:V</code>
<code>\tl_reverse:o</code>
<code>\tl_reverse:N</code>

<code>\tl_reverse:n</code> $\{\langle token_1\rangle\langle token_2\rangle\ldots\langle token_n\rangle\}$
<code>\tl_reverse:N</code> $\langle tl\ var.\rangle$

Reverse the token list (or the token list in the $\langle tl\ var.\rangle$) to result in $\langle token_n\rangle\ldots\langle token_2\rangle\langle token_1\rangle$. Note that spaces in this token list are gobbled in the process.

Note also that braces are lost in the process of reversing a $\langle tl\ var.\rangle$. That is, `\tl_set:Nn \l_tmpa_tl {a{bcd}e} \tl_reverse:N \l_tmpa_tl` will result in `ebcda`. This behaviour is probably more of a bug than a feature.

<code>\tl_elt_count:n *</code>
<code>\tl_elt_count:V *</code>
<code>\tl_elt_count:o *</code>
<code>\tl_elt_count:N *</code>

<code>\tl_elt_count:n</code> $\{\langle token\ list\rangle\}$
<code>\tl_elt_count:N</code> $\langle tl\ var.\rangle$

Returns the number of elements in the token list. Brace groups encountered count as one element. Note that spaces in this token list are gobbled in the process.

45 Variables and constants

<code>\c_job_name_tl</code>

Constant that gets the ‘job name’ assigned when T_EX starts.

T_EXhackers note: This is the new name for the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

<code>\c_empty_tl</code>	Constant that is always empty.
--------------------------	--------------------------------

T_EXhackers note: This was named `\@empty` in L^AT_EX2 and `\empty` in plain T_EX.

<code>\c_space_tl</code>	A space token contained in a token list (compare this with <code>\char_space_token</code>). For use where an explicit space is required.
--------------------------	---

<code>\l_tmpa_tl</code> <code>\l_tmpb_tl</code> <code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code>	Scratch register for immediate use. They are not used by conditionals or predicate functions. However, it is important to note that you should never rely on such scratch variables unless you fully control the code used between setting them and retrieving their value. Calling code from other modules, or worse allowing arbitrary user input to interfere might result in them not containing what you expect. In that is the case you better define your own scratch variables that are tight to your code by giving them suitable names.
--	---

<code>\l_tl_replace_tl</code>	Internal register used in the replace functions.
-------------------------------	--

<code>\l_kernel_testa_tl</code> <code>\l_kernel_testb_tl</code>	Registers used for conditional processing if the engine doesn't support arbitrary string comparison. Not for use outside the kernel code!
--	---

<code>\l_kernel_tmpa_tl</code> <code>\l_kernel_tmpb_tl</code>	Scratch registers reserved for other places in kernel code. Not for use outside the kernel code!
--	--

<code>\g_tl_inline_level_int</code>	Internal register used in the inline map functions.
-------------------------------------	---

46 Searching for and replacing tokens

<code>\tl_if_in:NnTF</code> <code>\tl_if_in:cnTF</code> <code>\tl_if_in:nnTF</code> <code>\tl_if_in:VnTF</code> <code>\tl_if_in:onTF</code>	<code>\tl_if_in:NnTF</code> <i><tl var.> {<item>} {<true code>} {<false code>}</i>
---	--

Function that tests if $\langle item \rangle$ is in $\langle tl\ var. \rangle$. Depending on the result either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ is executed. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

$\backslash tl_replace_in:Nnn$ $\backslash tl_replace_in:cnn$ $\backslash tl_greplace_in:Nnn$ $\backslash tl_greplace_in:cnn$	$\backslash tl_replace_in:Nnn\ \langle tl\ var. \rangle\ \{\langle item_1 \rangle\}\ \{\langle item_2 \rangle\}$
--	--

Replaces the leftmost occurrence of $\langle item_1 \rangle$ in $\langle tl\ var. \rangle$ with $\langle item_2 \rangle$ if present, otherwise the $\langle tl\ var. \rangle$ is left untouched. Note that $\langle item_1 \rangle$ cannot contain brace groups nor $\#_6$ tokens, and $\langle item_2 \rangle$ cannot contain $\#_6$ tokens.

$\backslash tl_replace_all_in:Nnn$ $\backslash tl_replace_all_in:cnn$ $\backslash tl_greplace_all_in:Nnn$ $\backslash tl_greplace_all_in:cnn$	$\backslash tl_replace_all_in:Nnn\ \langle tl\ var. \rangle\ \{\langle item_1 \rangle\}\ \{\langle item_2 \rangle\}$
--	---

Replaces *all* occurrences of $\langle item_1 \rangle$ in $\langle tl\ var. \rangle$ with $\langle item_2 \rangle$. Note that $\langle item_1 \rangle$ cannot contain brace groups nor $\#_6$ tokens, and $\langle item_2 \rangle$ cannot contain $\#_6$ tokens.

$\backslash tl_remove_in:Nn$ $\backslash tl_remove_in:cn$ $\backslash tl_gremove_in:Nn$ $\backslash tl_gremove_in:cn$	$\backslash tl_remove_in:Nn\ \langle tl\ var. \rangle\ \{\langle item \rangle\}$
--	--

Removes the leftmost occurrence of $\langle item \rangle$ from $\langle tl\ var. \rangle$ if present. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

$\backslash tl_remove_all_in:Nn$ $\backslash tl_remove_all_in:cn$ $\backslash tl_gremove_all_in:Nn$ $\backslash tl_gremove_all_in:cn$	$\backslash tl_remove_all_in:Nn\ \langle tl\ var. \rangle\ \{\langle item \rangle\}$
--	---

Removes *all* occurrences of $\langle item \rangle$ from $\langle tl\ var. \rangle$. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

47 Heads or tails?

Here are some functions for grabbing either the head or tail of a list and perform some tests on it.

<code>\tl_head:n</code>	<code>*</code>	
<code>\tl_head:V</code>	<code>*</code>	
<code>\tl_head:v</code>	<code>*</code>	
<code>\tl_tail:n</code>	<code>*</code>	
<code>\tl_tail:V</code>	<code>*</code>	
<code>\tl_tail:v</code>	<code>*</code>	
<code>\tl_tail:f</code>	<code>*</code>	
<code>\tl_head_i:n</code>	<code>*</code>	
<code>\tl_head_iii:n</code>	<code>*</code>	
<code>\tl_head_iii:f</code>	<code>*</code>	
<code>\tl_head:w</code>	<code>*</code>	
<code>\tl_tail:w</code>	<code>*</code>	
<code>\tl_head_i:w</code>	<code>*</code>	<code>\tl_head:n { <token₁><token₂>...<token_k> }</code>
<code>\tl_head_iii:w</code>	<code>*</code>	<code>\tl_tail:n { <token₁><token₂>...<token_k> }</code>
		<code>\tl_head:w <token₁><token₂>...<token_k> \q_stop</code>

These functions return either the head or the tail from a list of tokens, thus in the above example `\tl_head:n` would return `<token1>` and `\tl_tail:n` would return `<token2>...<tokenk>`. `\tl_head_iii:n` returns the first three tokens. The `:w` versions require some care as they expect the token list to be delimited by `\q_stop`.

T_EXhackers note: These are the Lisp functions `car` and `cdr` but with L^AT_EX3 names.

<code>\tl_if_head_eq_meaning_p:nN</code>	<code>*</code>	<code>\tl_if_head_eq_meaning:nNTF {<token list>} <token></code>
<code>\tl_if_head_eq_meaning:nNTF</code>	<code>*</code>	<code>{<true>} {<false>}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_meaning:w`.

<code>\tl_if_head_eq_charcode_p:nN</code>	<code>*</code>	
<code>\tl_if_head_eq_charcode_p:fN</code>	<code>*</code>	
<code>\tl_if_head_eq_charcode:nNTF</code>	<code>*</code>	<code>\tl_if_head_eq_charcode:nNTF {<token list>} <token></code>
<code>\tl_if_head_eq_charcode:fNTF</code>	<code>*</code>	<code>{<true>} {<false>}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_charcode:w` but it prevents expansion of them. If you want them to expand, you can use an `f` type expansion first (define `\tl_if_head_eq_charcode:fNTF` or similar).

<code>\tl_if_head_eq_catcode_p:nN</code>	<code>*</code>	<code>\tl_if_head_eq_catcode:nNTF {<token list>} <token></code>
<code>\tl_if_head_eq_catcode:nNTF</code>	<code>*</code>	<code>{<true>} {<false>}</code>

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. This version uses `\if_catcode:w` for the test but is otherwise identical to the `charcode` version.

Part XI

The l3toks package

Token Registers

There is a second form beside token list variables in which L^AT_EX3 stores token lists, namely the internal T_EX token registers. Functions dealing with these registers got the prefix `\toks_`. Unlike token list variables we have an accessing function as one can see below.

The main difference between $\langle toks \rangle$ (token registers) and $\langle tl var. \rangle$ (token list variable) is their behavior regarding expansion. While $\langle tl vars \rangle$ expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denoted by \mathbf{x}) $\langle toks \rangle$'s expand always only up to one level, i.e., passing their contents without further expansion.

There are fewer restrictions on the contents of a token register over a token list variable. So while $\langle token list \rangle$ is used to describe the contents of both of these, bear in mind that slightly different lists of tokens are allowed in each case. The best (only?) example is that a $\langle toks \rangle$ can contain the `#` character (i.e., characters of catcode 6), whereas a $\langle tl var. \rangle$ will require its input to be sanitised before that is possible.

If you're not sure which to use between a $\langle tl var. \rangle$ or a $\langle toks \rangle$, consider what data you're trying to hold. If you're dealing with function parameters involving `#`, or building some sort of data structure then you probably want a $\langle toks \rangle$ (e.g., `l3prop` uses $\langle toks \rangle$ to store its property lists).

If you're storing ad-hoc data for later use (possibly from direct user input) then usually a $\langle tl var. \rangle$ will be what you want.

48 Allocation and use

<code>\toks_new:N</code>
<code>\toks_new:c</code>

`\toks_new:N \langle toks \rangle`

Defines $\langle toks \rangle$ to be a new token list register.

T_EXhackers note: This is the L^AT_EX3 allocation for what was called `\newtoks` in plain T_EX.

<code>\toks_use:N</code>
<code>\toks_use:c</code>

`\toks_use:N \langle toks \rangle`

Accesses the contents of $\langle toks \rangle$. Contrary to token list variables $\langle toks \rangle$ can't be access simply by calling them directly.

T_EXhackers note: Something like `\the <toks>`.

```
\toks_set:Nn
\toks_set:NV
\toks_set:Nv
\toks_set:No
\toks_set:Nx
\toks_set:Nf
\toks_set:cn
\toks_set:co
\toks_set:cV
\toks_set:cv
\toks_set:cx
\toks_set:cf
```

```
\toks_set:Nn <toks> {\<token list>}
```

Defines `<toks>` to hold the token list `<token list>`.

T_EXhackers note: `\toks_set:Nn` could have been specified in plain T_EX by `<toks> = {\<token list>}` but all other functions have no counterpart in plain T_EX.

```
\toks_gset:Nn
\toks_gset:NV
\toks_gset:No
\toks_gset:Nx
\toks_gset:cn
\toks_gset:cV
\toks_gset:co
\toks_gset:cx
```

```
\toks_gset:Nn <toks> {\<token list>}
```

Defines `<toks>` to globally hold the token list `<token list>`.

```
\toks_set_eq:NN
\toks_set_eq:cN
\toks_set_eq:Nc
\toks_set_eq:cc
```

```
\toks_set_eq:NN <toks12
```

Set `<toks1 to the value of <toks2. Don't try to use \toks_set:Nn for this purpose if the second argument is also a token register.`

```
\toks_gset_eq:NN
\toks_gset_eq:cN
\toks_gset_eq:Nc
\toks_gset_eq:cc
```

```
\toks_gset_eq:NN <toks12
```

The `<toks1 globally set to the value of <toks2. Don't try to use \toks_gset:Nn for this purpose if the second argument is also a token register.`

<code>\toks_clear:N</code>
<code>\toks_clear:c</code>
<code>\toks_gclear:N</code>
<code>\toks_gclear:c</code>

`\toks_clear:N <toks>`

The $\langle toks \rangle$ is locally or globally cleared.

<code>\toks_use_clear:N</code>
<code>\toks_use_clear:c</code>
<code>\toks_use_gclear:N</code>
<code>\toks_use_gclear:c</code>

`\toks_use_clear:N <toks>`

Accesses the contents of $\langle toks \rangle$ and clears (locally or globally) it afterwards. Actually the clearing operation is done in a way that does not prohibit the access of the following tokens in the input stream with functions stored in the token register. In other words this function is not exactly the same as calling `\toks_use:N <toks> \toks_clear:N <toks>` in sequence.

<code>\toks_show:N</code>
<code>\toks_show:c</code>

`\toks_show:N <toks>`

Displays the contents of $\langle toks \rangle$ in the terminal output and log file. # signs in the $\langle toks \rangle$ will be shown doubled.

TpXhackers note: Something like `\showthe <toks>`.

49 Adding to the contents of token registers

<code>\toks_put_left:Nn</code>
<code>\toks_put_left:NV</code>
<code>\toks_put_left:No</code>
<code>\toks_put_left:Nx</code>
<code>\toks_put_left:cn</code>
<code>\toks_put_left:cV</code>
<code>\toks_put_left:co</code>

`\toks_put_left:Nn <toks> {<token list>}`

These functions will append $\langle token list \rangle$ to the left of $\langle toks \rangle$. Assignment is done locally. If possible append to the right since this operation is faster.

<code>\toks_gput_left:Nn</code>
<code>\toks_gput_left:NV</code>
<code>\toks_gput_left:No</code>
<code>\toks_gput_left:Nx</code>
<code>\toks_gput_left:cn</code>
<code>\toks_gput_left:cV</code>
<code>\toks_gput_left:co</code>

`\toks_gput_left:Nn <toks> {<token list>}`

These functions will append *<token list>* to the left of *<toks>*. Assignment is done globally. If possible append to the right since this operation is faster.

<code>\toks_put_right:Nn</code>
<code>\toks_put_right:NV</code>
<code>\toks_put_right:No</code>
<code>\toks_put_right:Nx</code>
<code>\toks_put_right:cV</code>
<code>\toks_put_right:cn</code>
<code>\toks_put_right:co</code>

`\toks_put_right:Nn <toks> {<token list>}`

These functions will append *<token list>* to the right of *<toks>*. Assignment is done locally.

<code>\toks_put_right:Nf</code>

`\toks_put_right:Nf <toks> {<token list>}`

Variant of the above. `:Nf` is used by `template.dtx` and will perhaps be moved to that package.

<code>\toks_gput_right:Nn</code>
<code>\toks_gput_right:NV</code>
<code>\toks_gput_right:No</code>
<code>\toks_gput_right:Nx</code>
<code>\toks_gput_right:cn</code>
<code>\toks_gput_right:cV</code>
<code>\toks_gput_right:co</code>

`\toks_gput_right:Nn <toks> {<token list>}`

These functions will append *<token list>* to the right of *<toks>*. Assignment is done globally.

50 Predicates and conditionals

<code>\toks_if_empty_p:N *</code>
<code>\toks_if_empty:NTF *</code>
<code>\toks_if_empty_p:c *</code>
<code>\toks_if_empty:cTF *</code>

`\toks_if_empty:Nf <toks> {<true code>} {<false code>}`

Expandable test for whether *<toks>* is empty.

<code>\toks_if_eq:NNTF</code>	<code>*</code>
<code>\toks_if_eq:NcTF</code>	<code>*</code>
<code>\toks_if_eq:cNNTF</code>	<code>*</code>
<code>\toks_if_eq:ccTF</code>	<code>*</code>
<code>\toks_if_eq_p:NN</code>	<code>*</code>
<code>\toks_if_eq_p:cN</code>	<code>*</code>
<code>\toks_if_eq_p:Nc</code>	<code>*</code>
<code>\toks_if_eq_p:cc</code>	<code>*</code>

`\toks_if_eq:NNTF <toks1> <toks2> {<true code>} {<false code>}`

Expandably tests if $\langle toks_1 \rangle$ and $\langle toks_2 \rangle$ are equal.

51 Variable and constants

<code>\c_empty_toks</code>

Constant that is always empty.

<code>\l_tmpa_toks</code>
<code>\l_tmpb_toks</code>
<code>\l_tmpc_toks</code>
<code>\g_tmpa_toks</code>
<code>\g_tmpb_toks</code>
<code>\g_tmpc_toks</code>

Scratch register for immediate use. They are not used by conditionals or predicate functions.

<code>\l_tl_replace_toks</code>

A placeholder for contents of functions replacing contents of strings.

Part XII

The l3seq package

Sequences

L^AT_EX3 implements a data type called ‘sequences’. These are special token lists that can be accessed via special function on the ‘left’. Appending tokens is possible at both ends. Appended token lists can be accessed only as a union. The token lists that form the individual items of a sequence might contain any tokens except two internal functions that are used to structure sequences (see section internal functions below). It is also possible to map functions on such sequences so that they are executed for every item on the sequence.

All functions that return items from a sequence in some $\langle tl\ var.\rangle$ assume that the $\langle tl\ var.\rangle$ is local. See remarks below if you need a global returned value.

The defined functions are not orthogonal in the sense that every possible variation possible is actually available. If you need a new variant use the expansion functions described in the package `l3expan` to build it.

Adding items to the left of a sequence can currently be done with either something like `\seq_put_left:Nn` or with a “stack” function like `\seq_push:Nn` which has the same effect. Maybe one should therefore remove the “left” functions totally.

52 Functions for creating/initialising sequences

<code>\seq_new:N</code> <code>\seq_new:c</code>	<code>\seq_new:N</code> $\langle sequence \rangle$
--	--

Defines $\langle sequence \rangle$ to be a variable of type `seq`.

<code>\seq_clear:N</code> <code>\seq_clear:c</code> <code>\seq_gclear:N</code> <code>\seq_gclear:c</code>	<code>\seq_clear:N</code> $\langle sequence \rangle$
--	--

These functions locally or globally clear $\langle sequence \rangle$.

<code>\seq_clear_new:N</code> <code>\seq_clear_new:c</code> <code>\seq_gclear_new:N</code> <code>\seq_gclear_new:c</code>	<code>\seq_clear_new:N</code> $\langle sequence \rangle$
--	--

These functions locally or globally clear $\langle sequence \rangle$ if it exists or otherwise allocates it.

<code>\seq_set_eq:NN</code> <code>\seq_set_eq:cN</code> <code>\seq_set_eq:Nc</code> <code>\seq_set_eq:cc</code>	<code>\seq_set_eq:NN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$
--	---

Function that locally makes $\langle seq_1 \rangle$ identical to $\langle seq_2 \rangle$.

<code>\seq_gset_eq:NN</code> <code>\seq_gset_eq:cN</code> <code>\seq_gset_eq:Nc</code> <code>\seq_gset_eq:cc</code>	<code>\seq_gset_eq:NN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$
--	--

Function that globally makes $\langle seq_1 \rangle$ identical to $\langle seq_2 \rangle$.

<code>\seq_concat:NNN</code>	
<code>\seq_concat:ccc</code>	
<code>\seq_gconcat:NNN</code>	<code>\seq_concat:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle seq_3 \rangle$
<code>\seq_gconcat:ccc</code>	<code>\seq_gconcat:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle seq_3 \rangle$

Function that concatenates $\langle seq_2 \rangle$ and $\langle seq_3 \rangle$ and locally or globally assigns the result to $\langle seq_1 \rangle$.

53 Adding data to sequences

<code>\seq_put_left:Nn</code>	
<code>\seq_put_left:NV</code>	
<code>\seq_put_left:No</code>	
<code>\seq_put_left:Nx</code>	
<code>\seq_put_left:cn</code>	
<code>\seq_put_left:cV</code>	
<code>\seq_put_left:co</code>	<code>\seq_put_left:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Locally appends $\langle token list \rangle$ as a single item to the left of $\langle sequence \rangle$. $\langle token list \rangle$ might get expanded before appending according to the variant.

<code>\seq_put_right:Nn</code>	
<code>\seq_put_right:NV</code>	
<code>\seq_put_right:No</code>	
<code>\seq_put_right:Nx</code>	
<code>\seq_put_right:cn</code>	
<code>\seq_put_right:cV</code>	
<code>\seq_put_right:co</code>	<code>\seq_put_right:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Locally appends $\langle token list \rangle$ as a single item to the right of $\langle sequence \rangle$. $\langle token list \rangle$ might get expanded before appending according to the variant.

<code>\seq_gput_left:Nn</code>	
<code>\seq_gput_left:NV</code>	
<code>\seq_gput_left:No</code>	
<code>\seq_gput_left:Nx</code>	
<code>\seq_gput_left:cn</code>	
<code>\seq_gput_left:cV</code>	
<code>\seq_gput_left:co</code>	<code>\seq_gput_left:Nn</code> $\langle sequence \rangle$ $\langle token list \rangle$

Globally appends $\langle token list \rangle$ as a single item to the left of $\langle sequence \rangle$.

<code>\seq_gput_right:Nn</code>
<code>\seq_gput_right:NV</code>
<code>\seq_gput_right:No</code>
<code>\seq_gput_right:Nx</code>
<code>\seq_gput_right:cn</code>
<code>\seq_gput_right:cV</code>
<code>\seq_gput_right:co</code>

`\seq_gput_right:Nn` $\langle sequence \rangle$ $\langle token list \rangle$

Globally appends $\langle token list \rangle$ as a single item to the right of $\langle sequence \rangle$.

54 Working with sequences

<code>\seq_get:NN</code>
<code>\seq_get:cN</code>

`\seq_get:NN` $\langle sequence \rangle$ $\langle tl var. \rangle$

Functions that locally assign the left-most item of $\langle sequence \rangle$ to the token list variable $\langle tl var. \rangle$. Item is not removed from $\langle sequence \rangle$! If you need a global return value you need to code something like this:

```
\seq_get:NN  $\langle sequence \rangle$  \l_tmpa_tl
\tl_gset_eq:NN  $\langle global tl var. \rangle$  \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

<code>\seq_map_variable:NNn</code>
<code>\seq_map_variable:cNn</code>

`\seq_map_variable:NNn` $\langle sequence \rangle$ $\langle tl var. \rangle$ { $\langle code using tl var. \rangle$ }

Every element in $\langle sequence \rangle$ is assigned to $\langle tl var. \rangle$ and then $\langle code using tl var. \rangle$ is executed. The operation is not expandable which means that it can't be used within write operations etc. However, this function can be nested which the others can't.

<code>\seq_map_function:NN</code>
<code>\seq_map_function:cN</code>

`\seq_map_function:NN` $\langle sequence \rangle$ $\langle function \rangle$

This function applies $\langle function \rangle$ (which must be a function with one argument) to every item of $\langle sequence \rangle$. $\langle function \rangle$ is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

In the current implementation the next functions are more efficient and should be preferred.

<code>\seq_map_inline:Nn</code> <code>\seq_map_inline:cn</code>
--

`\seq_map_inline:Nn` $\langle sequence \rangle$ $\{\langle inline function \rangle\}$

Applies $\langle inline function \rangle$ (which should be the direct coding for a function with one argument (i.e. use #1 as the place holder for this argument)) to every item of $\langle sequence \rangle$. $\langle inline function \rangle$ is not executed within a sub-group so that side effects can be achieved locally. The operation is not expandable which means that it can't be used within write operations etc.

<code>\seq_map_break:</code> <code>\seq_map_break:n</code>

These functions are used to break out of a mapping function at the point of execution. (Please do not put ‘\q_stop’ inside a $\langle seq \rangle$ that uses these functions.)

<code>\seq_show:N</code> <code>\seq_show:c</code>
--

`\seq_show:N` $\langle sequence \rangle$

Function that pauses the compilation and displays $\langle seq \rangle$ in the terminal output and in the log file. (Usually used for diagnostic purposes.)

<code>\seq_display:N</code> <code>\seq_display:c</code>
--

`\seq_display:N` $\langle sequence \rangle$

As with `\seq_show:N` but pretty prints the output one line per element.

<code>\seq_remove_duplicates:N</code> <code>\seq_gremove_duplicates:N</code>

`\seq_gremove_duplicates:N` $\langle seq \rangle$

Function that removes any duplicate entries in $\langle seq \rangle$.

55 Predicates and conditionals

<code>\seq_if_empty_p:N *</code> <code>\seq_if_empty_p:c *</code>
--

`\seq_if_empty_p:N` $\langle sequence \rangle$

This predicate returns ‘true’ if $\langle sequence \rangle$ is ‘empty’ i.e., doesn’t contain any items. Note that this is ‘false’ even if the $\langle sequence \rangle$ only contains a single empty item.

<code>\seq_if_empty:NTF</code> <code>\seq_if_empty:cTF</code>
--

`\seq_if_empty:N` $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Set of conditionals that test whether or not a particular $\langle sequence \rangle$ is empty and if so executes either $\langle true code \rangle$ or $\langle false code \rangle$.

<code>\seq_if_in:NnTF</code>
<code>\seq_if_in:NVTF</code>
<code>\seq_if_in:cnTF</code>
<code>\seq_if_in:cVTF</code>
<code>\seq_if_in:coTF</code>
<code>\seq_if_in:cxTF</code>

`\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}`

Functions that test if $\langle item \rangle$ is in $\langle sequence \rangle$. Depending on the result either $\langle true code \rangle$ or $\langle false code \rangle$ is executed.

56 Internal functions

<code>\seq_if_empty_err:N</code>

`\seq_if_empty_err:N <sequence>`

Signals an L^AT_EX3 error if $\langle sequence \rangle$ is empty.

<code>\seq_pop_aux:nnNN</code>

`\seq_pop_aux:nnNN <assign1> <assign2> <sequence> <tl var.>`

Function that assigns the left-most item of $\langle sequence \rangle$ to $\langle tl var. \rangle$ using $\langle assign_1 \rangle$ and assigns the tail to $\langle sequence \rangle$ using $\langle assign_2 \rangle$. This function could be used to implement a global return function.

<code>\seq_get_aux:w</code>
<code>\seq_pop_aux:w</code>
<code>\seq_put_aux:Nnn</code>
<code>\seq_put_aux:w</code>

Functions used to implement put and get operations. They are not for meant for direct use.

<code>\seq_elt:w</code>
<code>\seq_elt_end:</code>

Functions (usually used as constants) that separates items within a sequence. They might get special meaning during mapping operations and are not supposed to show up as tokens within an item appended to a sequence.

57 Functions for ‘Sequence Stacks’

Special sequences in L^AT_EX3 are ‘stacks’ with their usual operations of ‘push’, ‘pop’, and ‘top’. They are internally implemented as sequences and share some of the functions (like `\seq_new:N` etc.)

```

\seq_push:Nn
\seq_push:NV
\seq_push:No
\seq_push:cn
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:No
\seq_gpush:Nv
\seq_gpush:cn

```

```
\seq_push:Nn <stack> {<token list>}
```

Locally or globally pushes *<token list>* as a single item onto the *<stack>*.

```

\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN

```

```
\seq_pop:NN <stack> <tl var.>
```

Functions that assign the top item of *<stack>* to *<tl var.>* and removes it from *<stack>*!

```

\seq_top:NN
\seq_top:cN

```

```
\seq_top:NN <stack> <tl var.>
```

Functions that locally assign the top item of *<stack>* to the *<tl var.>*. Item is *not* removed from *<stack>*!

Part XIII

The l3clist package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the sequence. This gives an ordered list which can then be utilised with the `\clist_map_function:NN` function. Comma lists cannot contain empty items, thus

```

\clist_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { }
\clist_if_empty:NTF \l_my_clist { true } { false }

```

will leave `true` in the input stream.

58 Functions for creating/initialising comma-lists

<code>\clist_new:N</code>
<code>\clist_new:c</code>

`\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>* will initially contain no entries.

<code>\clist_set_eq:NN</code>
<code>\clist_set_eq:cN</code>
<code>\clist_set_eq:Nc</code>
<code>\clist_set_eq:cc</code>

`\clist_set_eq:NN <comma list1> <comma list2>`

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is restricted to the current \TeX group level.

<code>\clist_gset_eq:NN</code>
<code>\clist_gset_eq:cN</code>
<code>\clist_gset_eq:Nc</code>
<code>\clist_gset_eq:cc</code>

`\clist_gset_eq:NN <comma list1> <comma list2>`

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is global and so is not limited by the current \TeX group level.

<code>\clist_clear:N</code>
<code>\clist_clear:c</code>

`\clist_clear:N <comma list>`

Clears all entries from the *<comma list>* within the scope of the current \TeX group.

<code>\clist_gclear:N</code>
<code>\clist_gclear:c</code>

`\clist_gclear:N <comma list>`

Clears all entries from the *<comma list>* globally.

<code>\clist_clear_new:N</code>
<code>\clist_clear_new:c</code>
<code>\clist_gclear_new:N</code>
<code>\clist_gclear_new:c</code>

`\clist_clear_new:N <comma-list>`

These functions locally or globally clear *<comma-list>* if it exists or otherwise allocates it.

59 Putting data in

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<entry>}</code>
<code>\clist_put_left:NV</code>	
<code>\clist_put_left:No</code>	
<code>\clist_put_left:Nx</code>	
<code>\clist_put_left:cn</code>	
<code>\clist_put_left:cV</code>	
<code>\clist_put_left:co</code>	

Adds $\langle entry \rangle$ onto the left of the $\langle comma list \rangle$. The assignment is restricted to the current \TeX group.

<code>\clist_gput_left:Nn</code>	<code>\clist_gput_left:Nn <comma list> {\<entry>}</code>
<code>\clist_gput_left:NV</code>	
<code>\clist_gput_left:No</code>	
<code>\clist_gput_left:Nx</code>	
<code>\clist_gput_left:cn</code>	
<code>\clist_gput_left:cV</code>	
<code>\clist_gput_left:co</code>	

Adds $\langle entry \rangle$ onto the left of the $\langle comma list \rangle$. The assignment is global.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<entry>}</code>
<code>\clist_put_right:NV</code>	
<code>\clist_put_right:No</code>	
<code>\clist_put_right:Nx</code>	
<code>\clist_put_right:cn</code>	
<code>\clist_put_right:cV</code>	
<code>\clist_put_right:co</code>	

Adds $\langle entry \rangle$ onto the right of the $\langle comma list \rangle$. The assignment is restricted to the current \TeX group.

<code>\clist_gput_right:Nn</code>	<code>\clist_gput_right:Nn <comma list> {\<entry>}</code>
<code>\clist_gput_right:NV</code>	
<code>\clist_gput_right:No</code>	
<code>\clist_gput_right:Nx</code>	
<code>\clist_gput_right:cn</code>	
<code>\clist_gput_right:cV</code>	
<code>\clist_gput_right:co</code>	

Adds $\langle entry \rangle$ onto the right of the $\langle comma list \rangle$. The assignment is global.

60 Getting data out

<code>\clist_use:N *</code>
<code>\clist_use:c *</code>

`\clist_use:N <comma list>`

Recovers the content of a *<comma list>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. This function is intended mainly for use when a *<comma list>* is being saved to an auxiliary file.

<code>\clist_show:N</code>
<code>\clist_show:c</code>

`\clist_show:N <clist>`

Function that pauses the compilation and displays *<clist>* in the terminal output and in the log file. (Usually used for diagnostic purposes.)

<code>\clist_display:N</code>
<code>\clist_display:c</code>

`\clist_display:N <comma list>`

Displays the value of the *<comma list>* on the terminal.

<code>\clist_get:NN</code>
<code>\clist_get:cN</code>

`\clist_get:NN <comma-list> <tl var.>`

Functions that locally assign the left-most item of *<comma-list>* to the token list variable *<tl var.>*. Item is not removed from *<comma-list>*! If you need a global return value you need to code something like this:

```
\clist_get:NN <comma-list> \l_tmpa_tl
\tl_gset_eq:NN <global tl var.> \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

61 Mapping functions

We provide three types of mapping functions, each with their own strengths. The `\clist_map_function:NN` is expandable whereas `\clist_map_inline:Nn` type uses `##1` as a placeholder for the current item in *<clist>*. Finally we have the `\clist_map_variable:NNn` type which uses a user-defined variable as placeholder. Both the `_inline` and `_variable` versions are nestable.

<code>\clist_map_function:NN</code>	<code>*</code>
<code>\clist_map_function:Nc</code>	<code>*</code>
<code>\clist_map_function:cN</code>	<code>*</code>
<code>\clist_map_function:cc</code>	<code>*</code>
<code>\clist_map_function:nN</code>	<code>*</code>
<code>\clist_map_function:nc</code>	<code>*</code>

`\clist_map_function:NN <comma list> <function>`

Applies $\langle function \rangle$ to every $\langle entry \rangle$ stored in the $\langle comma list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle entries \rangle$ in the $\langle comma list \rangle$ are supplied to the $\langle function \rangle$ reading from the left to the right. These function may be nested.

<code>\clist_map_inline:Nn</code>
<code>\clist_map_inline:cn</code>
<code>\clist_map_inline:nn</code>

`\clist_map_inline:Nn <comma list> {\langle inline function \rangle}`

Applies $\langle inline function \rangle$ to every $\langle entry \rangle$ stored within the $\langle comma list \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle entry \rangle$ as #1. One inline mapping can be nested inside another. The $\langle entries \rangle$ in the $\langle comma list \rangle$ are supplied to the $\langle function \rangle$ reading from the left to the right.

<code>\clist_map_variable:NNn</code>
<code>\clist_map_variable:cNn</code>
<code>\clist_map_variable:nNn</code>

`\clist_map_variable:NNn <comma-list> <temp-var> {\langle action \rangle}`

Assigns $\langle temp-var \rangle$ to each element in $\langle clist \rangle$ and then executes $\langle action \rangle$ which should contain $\langle temp-var \rangle$. As the operation performs an assignment, it is not expandable.

T_EXhackers note: These functions resemble the L^AT_EX 2_ε function `\@for` but does not borrow the somewhat strange syntax.

<code>\clist_map_break: *</code>

`\clist_map_break:`

Used to terminate a `\clist_map...` function before all entries in the $\langle comma list \rangle$ have been processed. This will normally take 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 will lead low level T_EX errors.

62 Predicates and conditionals

<code>\clist_if_empty_p:N *</code>	
<code>\clist_if_empty_p:c *</code>	
<code>\clist_if_empty:N\underline{TF} *</code>	<code>\clist_if_empty_p:N \langleclist\rangle</code>
<code>\clist_if_empty:c\underline{TF} *</code>	<code>\clist_if_empty:N\underline{TF} \langleclist\rangle {\langletrue code\rangle} {\langlefalse code\rangle}</code>

Tests if the \langle comma list \rangle is empty (containing no items). The branching versions then leave either \langle true code \rangle or \langle false code \rangle in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_eq_p:NN *</code>	
<code>\clist_if_eq_p:Nc *</code>	
<code>\clist_if_eq_p:cN *</code>	
<code>\clist_if_eq_p:cc *</code>	
<code>\clist_if_eq:NN\underline{TF} *</code>	
<code>\clist_if_eq:Nc\underline{TF} *</code>	<code>\clist_if_eq_p:NN {\langleclist$_1$$\rangle$} {$\langle$clist$_2$$\rangle$}</code>
<code>\clist_if_eq:cN\underline{TF} *</code>	<code>\clist_if_eq:NN\underline{TF} {\langleclist$_1$$\rangle$} {$\langle$clist$_2$$\rangle$} {$\langle$true code$\rangle$}</code>
<code>\clist_if_eq:cc\underline{TF} *</code>	<code>{\langlefalse code\rangle}</code>

Compares the content of two \langle comma lists \rangle and is logically **true** if the two contain the same list of entries in the same order. The branching versions then leave either \langle true code \rangle or \langle false code \rangle in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_in:Nn\underline{TF}</code>	
<code>\clist_if_in:NV\underline{TF}</code>	
<code>\clist_if_in:No\underline{TF}</code>	
<code>\clist_if_in:cn\underline{TF}</code>	
<code>\clist_if_in:cV\underline{TF}</code>	<code>\clist_if_in:Nn\underline{TF} \langleclist\rangle {\langleentry\rangle} {\langletrue code\rangle}</code>
<code>\clist_if_in:co\underline{TF}</code>	<code>{\langlefalse code\rangle}</code>

Tests if the \langle entry \rangle is present in the \langle comma list \rangle as a discrete entry. The \langle entry \rangle cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply). Either the \langle true code \rangle or \langle false code \rangle in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

63 Higher level functions

<code>\clist_concat:NNN</code>	
<code>\clist_concat:ccc</code>	<code>\clist_concat:NNN \langleclist1\rangle \langleclist2\rangle \langleclist3\rangle</code>

Concatenates the content of \langle comma list2 \rangle and \langle comma list3 \rangle together and saves the

result in $\langle comma list1 \rangle$. $\langle comma list2 \rangle$ will be placed at the left side of the new comma list. This operation is local to the current \TeX group and will remove any existing content in $\langle comma list1 \rangle$.

$\backslash\text{clist_gconcat:NNN}$
$\backslash\text{clist_gconcat:ccc}$

 $\backslash\text{clist_gconcat:NNN}$ $\langle clist1 \rangle$ $\langle clist2 \rangle$ $\langle clist3 \rangle$

Concatenates the content of $\langle comma list2 \rangle$ and $\langle comma list3 \rangle$ together and saves the result in $\langle comma list1 \rangle$. $\langle comma list2 \rangle$ will be placed at the left side of the new comma list. This operation is global and will remove any existing content in $\langle comma list1 \rangle$.

$\backslash\text{clist_remove_duplicates:N}$
--

 $\backslash\text{clist_remove_duplicates:N}$ $\langle comma list \rangle$

Removes duplicate entries from the $\langle comma list \rangle$, leaving left most entry in the $\langle comma list \rangle$. The removal is local to the current \TeX group.

$\backslash\text{clist_gremove_duplicates:N}$

 $\backslash\text{clist_gremove_duplicates:N}$ $\langle comma list \rangle$

Removes duplicate entries from the $\langle comma list \rangle$, leaving left most entry in the $\langle comma list \rangle$. The removal is applied globally.

$\backslash\text{clist_remove_element:Nn}$
--

 $\backslash\text{clist_remove_element:Nn}$ $\langle comma list \rangle$ $\{\langle entry \rangle\}$

Removes each occurrence of $\langle entry \rangle$ from the $\langle comma list \rangle$, where $\langle entry \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (assuming normal \TeX category codes). The removal is local to the current \TeX group.

$\backslash\text{clist_gremove_element:Nn}$

 $\backslash\text{clist_gremove_element:Nn}$ $\langle comma list \rangle$ $\{\langle entry \rangle\}$

Removes each occurrence of $\langle entry \rangle$ from the $\langle comma list \rangle$, where $\langle entry \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (assuming normal \TeX category codes). The removal applied globally.

64 Functions for ‘comma-list stacks’

Special comma-lists in $\text{\LaTeX}3$ are ‘stacks’ with their usual operations of ‘push’, ‘pop’, and ‘top’. They are internally implemented as comma-lists and share some of the functions (like $\backslash\text{clist_new:N}$ etc.)

<code>\clist_push:Nn</code>
<code>\clist_push:NV</code>
<code>\clist_push:No</code>
<code>\clist_push:cn</code>
<code>\clist_gpush:Nn</code>
<code>\clist_gpush:NV</code>
<code>\clist_gpush:No</code>
<code>\clist_gpush:cn</code>

`\clist_push:Nn <stack> {<token list>}`

Locally or globally pushes $\langle token\ list \rangle$ as a single item onto the $\langle stack \rangle$. $\langle token\ list \rangle$ might get expanded before the operation.

<code>\clist_pop:NN</code>
<code>\clist_pop:cN</code>
<code>\clist_gpop:NN</code>
<code>\clist_gpop:cN</code>

`\clist_pop:NN <stack> <tl var.>`

Functions that assign the top item of $\langle stack \rangle$ to the token list variable $\langle tl\ var. \rangle$ and removes it from $\langle stack \rangle$!

<code>\clist_top:NN</code>
<code>\clist_top:cN</code>

`\clist_top:NN <stack> <tl var.>`

Functions that locally assign the top item of $\langle stack \rangle$ to the token list variable $\langle tl\ var. \rangle$. Item is not removed from $\langle stack \rangle$!

65 Internal functions

<code>\clist_if_empty_err:N</code>

`\clist_if_empty_err:N <comma-list>`

Signals an L^AT_EX3 error if $\langle comma-list \rangle$ is empty.

<code>\clist_pop_aux:nnNN</code>

`\clist_pop_aux:nnNN <assign1> <assign2> <comma-list> <tl var.>`

Function that assigns the left-most item of $\langle comma-list \rangle$ to $\langle tl\ var. \rangle$ using $\langle assign_1 \rangle$ and assigns the tail to $\langle comma-list \rangle$ using $\langle assign_2 \rangle$. This function could be used to implement a global return function.

<code>\clist_get_aux:w</code>
<code>\clist_pop_aux:w</code>
<code>\clist_pop_auxi:w</code>
<code>\clist_put_aux:NNnnNn</code>

Functions used to implement put and get operations. They are not for meant for direct use.

Part XIV

The l3prop package

Property Lists

L^AT_EX3 implements a data structure called a ‘property list’ which allows arbitrary information to be stored and accessed using keywords rather than numerical indexing.

A property list might contain a set of keys such as `name`, `age`, and `ID`, which each have individual values that can be saved and retrieved.

66 Functions

<code>\prop_new:N</code>
<code>\prop_new:c</code>

`\prop_new:N <prop>`
Defines `<prop>` to be a variable of type `<prop>`.

<code>\prop_clear:N</code>
<code>\prop_clear:c</code>
<code>\prop_gclear:N</code>
<code>\prop_gclear:c</code>

`\prop_clear:N <prop>`
These functions locally or globally clear `<prop>`.

<code>\prop_put:Nnn</code>
<code>\prop_put:Nno</code>
<code>\prop_put:NnV</code>
<code>\prop_put:NVn</code>
<code>\prop_put:NVV</code>
<code>\prop_put:Nnx</code>
<code>\prop_put:cnn</code>
<code>\prop_put:cnx</code>
<code>\prop_gput:Nnn</code>
<code>\prop_gput:NVn</code>
<code>\prop_gput:Nno</code>
<code>\prop_gput:NnV</code>
<code>\prop_gput:Nnx</code>
<code>\prop_gput:cnn</code>
<code>\prop_gput:ccx</code>

`\prop_put:Nnn <prop> {<key>} {<token list>}`

Locally or globally associates `<token list>` with `<key>` in the `<prop>` `<prop>`. If `<key>` has already a meaning within `<prop>` this value is overwritten.

The $\langle key \rangle$ must not contain unescaped # tokens but the $\langle token list \rangle$ may.

<code>\prop_gput_if_new:Nnn</code>

`\prop_gput_if_new:Nnn <prop> {<key>} {<token list>}`

Globally associates $\langle token list \rangle$ with $\langle key \rangle$ in the $\langle prop \rangle$ $\langle prop \rangle$ but only if $\langle key \rangle$ has so far no meaning within $\langle prop \rangle$. Silently ignored if $\langle key \rangle$ is already set in the $\langle prop \rangle$.

<code>\prop_get:NnN</code>
<code>\prop_get:NVN</code>
<code>\prop_get:cnN</code>
<code>\prop_get:cVN</code>
<code>\prop_gget:NnN</code>
<code>\prop_gget:NVN</code>
<code>\prop_gget:cnN</code>
<code>\prop_gget:cVN</code>

`\prop_get:NnN <prop> {<key>} <tl var.>`

If $\langle info \rangle$ is the information associated with $\langle key \rangle$ in the $\langle prop \rangle$ $\langle prop \rangle$ then the token list variable $\langle tl var. \rangle$ gets $\langle info \rangle$ assigned. Otherwise its value is the special quark `\q_no_value`. The assignment is done either locally or globally.

<code>\prop_set_eq:NN</code>
<code>\prop_set_eq:cN</code>
<code>\prop_set_eq:Nc</code>
<code>\prop_set_eq:cc</code>
<code>\prop_gset_eq:NN</code>
<code>\prop_gset_eq:cN</code>
<code>\prop_gset_eq:Nc</code>
<code>\prop_gset_eq:cc</code>

`\prop_set_eq:NN <prop1> <prop2>`

A fast assignment of $\langle prop \rangle$ s.

<code>\prop_get_gdel:NnN</code>

`\prop_get_gdel:NnN <prop> {<key>} <tl var.>`

Like `\prop_get:NnN` but additionally removes $\langle key \rangle$ (and its $\langle info \rangle$) from $\langle prop \rangle$.

<code>\prop_del:Nn</code>
<code>\prop_del:NV</code>
<code>\prop_gdel:Nn</code>
<code>\prop_gdel:NV</code>

`\prop_del:Nn <prop> {<key>}`

Locally or globally deletes $\langle key \rangle$ and its $\langle info \rangle$ from $\langle prop \rangle$ if found. Otherwise does nothing.

<code>\prop_map_function:NN *</code>
<code>\prop_map_function:cN *</code>
<code>\prop_map_function:Nc *</code>
<code>\prop_map_function:cc *</code>

`\prop_map_function:NN <prop> <function>`

Maps $\langle function \rangle$ which should be a function with two arguments ($\langle key \rangle$ and $\langle info \rangle$) over every $\langle key \rangle$ $\langle info \rangle$ pair of $\langle prop \rangle$. Property lists do not have any intrinsic “order” when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

<code>\prop_map_inline:Nn</code>
<code>\prop_map_inline:cn</code>

`\prop_map_inline:Nn $\langle prop \rangle$ {\mathit{inline function}}`

Just like `\prop_map_function:NN` but with the function of two arguments supplied as inline code. Within $\langle inline function \rangle$ refer to the arguments via #1 ($\langle key \rangle$) and #2 ($\langle info \rangle$). Nestable. Property lists do not have any intrinsic “order” when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

<code>\prop_map_break:</code>

`\prop_map_inline:Nn $\langle prop \rangle$ {
... \mathit{break test}:T {\prop_map_break:} }`

For breaking out of a loop. To be used inside TF-type functions as shown in the example above.

<code>\prop_show:N</code>
<code>\prop_show:c</code>

`\prop_show:N $\langle prop \rangle$`

Pauses the compilation and shows $\langle prop \rangle$ on the terminal output and in the log file.

<code>\prop_display:N</code>
<code>\prop_display:c</code>

`\prop_display:N $\langle prop \rangle$`

As with `\prop_show:N` but pretty prints the output one line per property pair.

67 Predicates and conditionals

<code>\prop_if_empty_p:N</code>
<code>\prop_if_empty_p:c</code>

`\prop_if_empty_p:N $\langle prop \rangle$ {\mathit{true code}} {\mathit{false code}}`

Predicates to test whether or not a particular $\langle prop \rangle$ is empty.

<code>\prop_if_empty:N\TF *</code>
<code>\prop_if_empty:c\TF *</code>

`\prop_if_empty:N\TF $\langle prop \rangle$ {\mathit{true code}} {\mathit{false code}}`

Set of conditionals that test whether or not a particular $\langle prop \rangle$ is empty.

```

\prop_if_eq_p:NN *
\prop_if_eq_p:cN *
\prop_if_eq_p:Nc *
\prop_if_eq_p:cc *
\prop_if_eq:NN $\overline{TF}$  *
\prop_if_eq:cN $\overline{TF}$  *
\prop_if_eq:Nc $\overline{TF}$  *
\prop_if_eq:cc $\overline{TF}$  *

```

`\prop_if_eq:NMF $\langle prop_1 \rangle$ $\langle prop_2 \rangle$ { $\langle false code \rangle$ }`

Execute $\langle false code \rangle$ if $\langle prop_1 \rangle$ doesn't hold the same token list as $\langle prop_2 \rangle$. Only expandable for new versions of pdfTeX.

```

\prop_if_in:Nn $\overline{TF}$ 
\prop_if_in:NV $\overline{TF}$ 
\prop_if_in:No $\overline{TF}$ 
\prop_if_in:cn $\overline{TF}$ 
\prop_if_in:cc $\overline{TF}$ 

```

`\prop_if_in:NnTF $\langle prop \rangle$ { $\langle key \rangle$ } { $\langle true code \rangle$ } { $\langle false code \rangle$ }`

Tests if $\langle key \rangle$ is used in $\langle prop \rangle$ and then either executes $\langle true code \rangle$ or $\langle false code \rangle$.

68 Internal functions

`\q_prop`

Quark used to delimit property lists internally.

```

\prop_put_aux:w
\prop_put_if_new_aux:w

```

Internal functions implementing the put operations.

```

\prop_get_aux:w
\prop_gget_aux:w
\prop_get_del_aux:w
\prop_del_aux:w

```

Internal functions implementing the get and delete operations.

`\prop_if_in_aux:w`

Internal function implementing the key test operation.

`\prop_map_function_aux:w`

Internal function implementing the map operations.

`\g_prop_inline_level_int`

Integer used in internal name for function used inside `\prop_map_inline:NN`.

`\prop_split_aux:Nnn`

`\prop_split_aux:Nnn $\langle prop \rangle$ $\langle key \rangle$ $\langle cmd \rangle$`

Internal function that invokes $\langle cmd \rangle$ with 3 arguments: 1st is the beginning of $\langle prop \rangle$

before $\langle key \rangle$, 2nd is the value associated with $\langle key \rangle$, 3rd is the rest of $\langle prop \rangle$ after $\langle key \rangle$. If there is no key $\langle key \rangle$ in $\langle prop \rangle$, then the 2 arg is `\q_no_value` and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens $\langle key \rangle$ `\q_no_value` at the end.

This function is used to implement various get operations.

Part XV

The l3io package

Low-level file i/o

Reading and writing from file streams is handled in L^AT_EX3 using functions with prefixes `\iow_...` (file reading) and `\ior_...` (file writing). Many of the basic functions are very similar, with reading and writing using the same syntax and function concepts. As a result, the reading and writing functions are documented together where this makes sense.

As T_EX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in L^AT_EX3. 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.

Reading from or writing to a file requires a $\langle stream \rangle$ to be used. This is a c_sname which refers to the file being processed, and is independent of the name of the file (except of course that the file name is needed when the file is opened).

69 Opening and closing streams

<code>\iow_new:N</code> <code>\iow_new:c</code> <code>\ior_new:N</code> <code>\ior_new:c</code>	<code>\iow_new:N</code> $\langle stream \rangle$
--	--

Reserves the name $\langle stream \rangle$ for use in accessing a file stream. This operation does not open a raw T_EX stream, which is handled internally using a pool and is should not be accessed directly by the programmer.

<code>\iow_open:Nn</code> <code>\iow_open:cn</code> <code>\ior_open:Nn</code> <code>\ior_open:cn</code>	<code>\iow_open:Nn</code> $\langle stream \rangle$ $\{ \langle file name \rangle \}$ <code>\iow_open:cn</code> $\langle stream \rangle$ $\{ \langle file name \rangle \}$
--	--

Opens $\langle file name \rangle$ for writing (`\iow_...`) or reading (`\ior_...`) using $\langle stream \rangle$ as the

csname by which the file is accessed. If $\langle stream \rangle$ was already open (for either writing or reading) it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately after issuing an `open` instruction. The $\langle stream \rangle$ will remain allocated to $\langle file name \rangle$ until a `close` instruction is given or at the end of the \TeX run.

Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive). As the total number of writing streams is limited, it may well be best to save material to be written to an intermediate storage format (for example a token list or toks), and to write the material in one ‘shot’ from this variable. In this way the file stream is only required for a limited time.

<code>\iow_close:N</code>	
<code>\iow_close:c</code>	
<code>\ior_close:N</code>	<code>\iow_close:N $\langle stream \rangle$</code>
<code>\ior_close:c</code>	<code>\ior_close:N $\langle stream \rangle$</code>

Closes $\langle stream \rangle$, freeing up one of the underlying \TeX streams for reuse. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers (the resources here are limited). The name of the $\langle stream \rangle$ will be freed at this stage, to ensure that any further attempts to write to it result in an error.

<code>\iow_open_streams:</code>	
<code>\iow_open_streams:</code>	<code>\iow_open_streams:</code>

Displays a list of the file names associated with each open stream: intended for tracking down problems.

69.1 Writing to files

<code>\iow_now:Nx</code>	
<code>\iow_now:Nn</code>	<code>\iow_now:Nx $\langle stream \rangle$ $\{\langle tokens \rangle\}$</code>

`\iow_now:Nx` immediately writes the expansion of $\langle tokens \rangle$ to the output $\langle stream \rangle$. If the $\langle stream \rangle$ is not open output goes to the terminal. The variant `\iow_now:Nn` writes out $\langle tokens \rangle$ without any further expansion.

\TeX hackers note: These are the equivalent of \TeX ’s `\immediate\write` with and without expansion control.

<code>\iow_log:n</code>	
<code>\iow_log:x</code>	
<code>\iow_term:n</code>	
<code>\iow_term:x</code>	<code>\iow_log:x $\{\langle tokens \rangle\}$</code>

These are dedicated functions which write to the log (transcript) file and the terminal,

respectively. They are equivalent to using `\iow_now:N(n/x)` to the streams `\c_iow_log_stream` and `\c_iow_term_stream`. The writing takes place immediately.

<code>\iow_now_buffer_safe:Nn</code>	<code>\iow_now_buffer_safe:Nn <stream> {<tokens>}</code>
<code>\iow_now_buffer_safe:Nx</code>	

Immediately write `<tokens>` expanded to `<stream>`, with every space converted into a newline. This means that the file can be read back without the danger that very long lines overflow T_EX's buffer.

<code>\iow_now_when_avail:Nn</code>	<code>\iow_now_when_avail:Nn <stream> {<tokens>}</code>
<code>\iow_now_when_avail:cn</code>	
<code>\iow_now_when_avail:Nx</code>	
<code>\iow_now_when_avail:cx</code>	

If `<stream>` is open, writes the `<tokens>` to the `<stream>` in the same manner as `\iow_now:N(n/x)`. If the `<stream>` is not open, the `<tokens>` are simply thrown away.

<code>\iow_shipout:Nx</code>	<code>\iow_shipout:Nx <stream> {<tokens>}</code>
<code>\iow_shipout:Nn</code>	

Write `<tokens>` to `<stream>` at the point at which the current page is finished. The `<tokens>` are either written unexpanded (`\iow_shipout:Nn`) or expanded only at the point that the function is used (`\iow_shipout:Nx`), *i.e.* no expansion takes place when writing to the file.

<code>\iow_shipout_x:Nx</code>	<code>\iow_shipout_x:Nx <stream> {<tokens>}</code>
<code>\iow_shipout_x:Nn</code>	

Write `<tokens>` to `<stream>` at the point at which the current page is finished. The `<tokens>` are expanded at the time of writing in addition to any expansion at the time of use of the function. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

T_EXhackers note: These are the equivalent of T_EX's `\write` with and without expansion control at point of use.

<code>\iow_newline: *</code>	<code>\iow_newline:</code>
------------------------------	----------------------------

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 a `\iow_now:Nn` call).

<code>\iow_char:N *</code>	<code>\iow_char:N \<char></code>
<code>\iow_char:N *</code>	<code>\iow_char:N \%</code>

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_stream { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in a `\iow_now:Nn` call).

69.2 Reading from files

<code>\ior_to:NN</code> <code>\ior_gto:NN</code>	<code>\ior_to:NN</code> $\langle stream \rangle$ $\langle token\ list\ variable \rangle$
---	--

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input stream $\langle stream \rangle$ and places the result locally or globally into the $\langle token\ list\ variable \rangle$. If $\langle stream \rangle$ is not open, input is requested from the terminal.

<code>\ior_if_eof_p:N *</code> <code>\ior_if_eof:NTF *</code>	<code>\ior_if_eof:NTF</code> $\langle stream \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
--	--

Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will also return a `true` value if the $\langle stream \rangle$ is not open or the $\langle file\ name \rangle$ associated with a $\langle stream \rangle$ does not exist at all.

70 Internal functions

<code>\iow_raw_new:N</code> <code>\iow_raw_new:c</code> <code>\ior_raw_new:N</code> <code>\ior_raw_new:c</code>	<code>\iow_raw_new:N</code> $\langle stream \rangle$
--	--

Creates a new low-level $\langle stream \rangle$ for use in subsequent functions. As allocations are made using a pool *do not use this function!*

T_EXhackers note: This is L^AT_EX 2_ε's `\newwrite`.

<code>\if_eof:w *</code>	<code>\if_eof:w</code> $\langle stream \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
--------------------------	--

Tests if the end of $\langle stream \rangle$ has been reached during a reading operation.

T_EXhackers note: This is the primitive `\ifeof`.

71 Variables and constants

<code>\c_io_streams_tl</code>	A list of the positions available for stream allocation (numbers 0 to 15).
-------------------------------	--

<code>\c_iow_term_stream</code> <code>\c_ior_term_stream</code> <code>\c_iow_log_stream</code> <code>\c_ior_log_stream</code>	Fixed stream numbers for accessing to the log and the terminal. The reading and writing values are the same but are provided so that the meaning is clear.
--	--

<code>\g_iow_streams_prop</code> <code>\g_ior_streams_prop</code>	Allocation records for streams, linking the stream number to the current name being used for that stream.
--	---

<code>\g_iow_tmp_stream</code> <code>\g_ior_tmp_stream</code>	Used when creating new streams at the \TeX level.
--	--

<code>\l_iow_stream_int</code> <code>\l_ior_stream_int</code>	Number of stream currently being allocated.
--	---

Part XVI

The `l3msg` package

Communicating with the user

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 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.

72 Creating new messages

All messages have to be created before they can be used. Inside the message text, spaces are *not* ignored. A space where T_EX would normally gobble one can be created using `\` , and a new line with `\\`. New lines may have ‘continuation’ text added by the output system.

<pre>\msg_new:nnnn \msg_new:nnn \msg_set:nnnn \msg_set:nnn</pre>	<pre>\msg_new:nnnn {\module} {\message} {\text} {\more text}</pre>
--	--

Creates new $\langle message \rangle$ for $\langle module \rangle$ to produce $\langle text \rangle$ initially and $\langle more text \rangle$ if requested by the user. $\langle text \rangle$ and $\langle more text \rangle$ can use up to four macro parameters (**#1** to **#4**), which are supplied by the message system. At the point where $\langle message \rangle$ is printed, the material supplied for **#1** to **#4** will be subject to an x-type expansion.

An error will be raised by the **new** functions if the message already exists: the **set** functions do not carry any checking. For messages defined using `\msg_new:nnn` or `\msg_set:nnn` L^AT_EX3 will supply a standard $\langle more text \rangle$ at the point the message is used, if this is required.

73 Message classes

Creating message output requires the message to be given a class.

<pre>\msg_class_new:nn \msg_class_set:nn</pre>	<pre>\msg_class_new:nn {\class} {\code}</pre>
--	---

Creates new $\langle class \rangle$ to output a message, using $\langle code \rangle$ to process the message text. The $\langle class \rangle$ should be a text value, while the $\langle code \rangle$ may be any arbitrary material.

The module defines several common message classes. The following describes the standard behaviour of each class if no redirection of the class or message is active. In all cases, the message may be issued supplying 0 to 4 arguments. The code will ensure that there are no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

```

\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn

```

```

\msg_fatal:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Issues $\langle module \rangle$ error message $\langle name \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run will halt.

```

\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn

```

```

\msg_error:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Issues $\langle module \rangle$ error message $\langle name \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageError`.

```

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

```

```

\msg_warning:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Prints $\langle module \rangle$ message $\langle name \rangle$ to the terminal, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageWarningNoLine`.

```

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

```

```

\msg_info:nnxxxx {\module} {\name} {\arg one}
{\arg two} {\arg three} {\arg four}

```

Prints $\langle module \rangle$ message $\langle name \rangle$ to the log, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageInfoNoLine`.

<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnxxx</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	<code>\msg_log:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions. No continuation text is added.

<code>\msg_trace:nnxxxx</code>	
<code>\msg_trace:nnxxx</code>	
<code>\msg_trace:nnxx</code>	
<code>\msg_trace:nnx</code>	
<code>\msg_trace:nn</code>	<code>\msg_trace:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions. No continuation text is added.

<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	<code>\msg_none:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>

Does nothing: used for redirecting other message classes. Gobbles arguments given.

74 Redirecting messages

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*. Multiple redirections are possible. Redirection to a missing class or infinite loops will raise errors when the messages are used, rather than at the point of redirection.

<code>\msg_redirect_module:nnn</code>	<code>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</code>
---------------------------------------	--

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.

T_EXhackers note: This function can be used to make some messages ‘silent’ by default. For example, all of the `trace` messages of *<module>* could be turned off with:

```
\msg_redirect_module:nnn { module } { trace } { none }
```

```
\msg_redirect_name:nnn \msg_redirect_name:nnn {<module>} {<message>} {<class>}
```

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages.

TeXhackers note: This function can be used to make a selected message ‘silent’ without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

75 Support functions for output

```
\msg_line_context: \msg_line_context:
```

Prints the text specified in `\c_msg_on_line_tl` followed by the current line in the current input file.

TeXhackers note: This is similar to the text added to messages by L^AT_EX 2_ε’s `\PackageWarning` and `\PackageInfo`.

```
\msg_line_number: \msg_line_number:
```

Prints the current line number in the current input file.

```
\msg_newline:
\msg_two_newlines: \msg_newline:
```

Print one or two newlines with no continuation information.

76 Low-level functions

The low-level functions do not make assumptions about module names. The output functions here produce messages directly, and do not respond to redirection.

<code>\msg_generic_new:nnn</code> <code>\msg_generic_new:nn</code> <code>\msg_generic_set:nnn</code> <code>\msg_generic_set:nn</code>	<code>\msg_generic_new:nnn {<name>} {<text>} {<more text>}</code>
--	---

Creates new message $\langle name \rangle$ to produce $\langle text \rangle$ initially and $\langle more text \rangle$ if requested by the user. $\langle text \rangle$ and $\langle more text \rangle$ can use up to four macro parameters (#1 to #4), which are supplied by the message system. Inside $\langle text \rangle$ and $\langle more text \rangle$ spaces are not ignored.

<code>\msg_direct_interrupt:xxxxx</code>	<code>\msg_direct_interrupt:xxxxx {<first line>} {<text>}</code> <code>{<continuation>} {<last line>} {<more text>}</code>
--	---

Executes a T_EX error, interrupting compilation. The $\langle first line \rangle$ is displayed followed by $\langle text \rangle$ and the input prompt. $\langle more text \rangle$ is displays if requested by the user. If $\langle more text \rangle$ is blank a default is supplied. Each line of $\langle text \rangle$ (broken with `\\`) begins with $\langle continuation \rangle$ and finishes off with $\langle last line \rangle$. $\langle last line \rangle$ has a period appended to it; do not add one yourself.

<code>\msg_direct_log:xx</code> <code>\msg_direct_term:xx</code>	<code>\msg_direct_log:xx {<text>} {<continuation>}</code>
---	---

Prints $\langle text \rangle$ to either the log or terminal. New lines (broken with `\\`) start with $\langle continuation \rangle$.

77 Kernel-specific functions

<code>\msg_kernel_new:nnnn</code> <code>\msg_kernel_new:nnn</code> <code>\msg_kernel_set:nnnn</code> <code>\msg_kernel_set:nnn</code>	<code>\msg_kernel_new:nnnn {<division>} {<name>} {<text>}</code> <code>{<more text>}</code>
--	--

Creates new kernel message $\langle name \rangle$ to produce $\langle text \rangle$ initially and $\langle more text \rangle$ if requested by the user. $\langle text \rangle$ and $\langle more text \rangle$ can use up to four macro parameters (#1 to #4), which are supplied by the message system. Kernel messages are divided into $\langle divisions \rangle$, roughly equivalent to the L^AT_EX 2_ε package names used.

<code>\msg_kernel_fatal:nnxxxx</code> <code>\msg_kernel_fatal:nnxxx</code> <code>\msg_kernel_fatal:nnxx</code> <code>\msg_kernel_fatal:nnx</code> <code>\msg_kernel_fatal:nn</code>	<code>\msg_kernel_fatal:nnxx {<division>} {<name>} {<arg one>}</code> <code>{<arg two>} {<arg three>} {<arg four>}</code>
---	--

Issues kernel error message $\langle name \rangle$ for $\langle division \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The TeX run then halts. Cannot be redirected.

<code>\msg_kernel_error:nnxxxx</code> <code>\msg_kernel_error:nnxxx</code> <code>\msg_kernel_error:nnxx</code> <code>\msg_kernel_error:nnx</code> <code>\msg_kernel_error:nn</code>	<code>\msg_kernel_error:nnxx {$\langle division \rangle$} {$\langle name \rangle$} {$\langle arg one \rangle$} <code>{$\langle arg two \rangle$} {$\langle arg three \rangle$} {$\langle arg four \rangle$}</code> </code>
---	---

Issues kernel error message $\langle name \rangle$ for $\langle division \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. Cannot be redirected.

<code>\msg_kernel_warning:nnxxxx</code> <code>\msg_kernel_warning:nnxxx</code> <code>\msg_kernel_warning:nnxx</code> <code>\msg_kernel_warning:nnx</code> <code>\msg_kernel_warning:nn</code>	<code>\msg_kernel_warning:nnxx {$\langle division \rangle$} {$\langle name \rangle$} {$\langle arg one \rangle$} <code>{$\langle arg two \rangle$} {$\langle arg three \rangle$} {$\langle arg four \rangle$}</code> </code>
---	---

Prints kernel message $\langle name \rangle$ for $\langle division \rangle$ to the terminal, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

<code>\msg_kernel_info:nnxxxx</code> <code>\msg_kernel_info:nnxxx</code> <code>\msg_kernel_info:nnxx</code> <code>\msg_kernel_info:nnx</code> <code>\msg_kernel_info:nn</code>	<code>\msg_kernel_info:nnx {$\langle division \rangle$} {$\langle name \rangle$} {$\langle arg one \rangle$} <code>{$\langle arg two \rangle$} {$\langle arg three \rangle$} {$\langle arg four \rangle$}</code> </code>
--	---

Prints kernel message $\langle name \rangle$ for $\langle division \rangle$ to the log, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

<code>\msg_kernel_bug:x</code>	<code>\msg_kernel_bug:x {$\langle text \rangle$}</code>
--------------------------------	--

Short-cut for ‘This is a LaTeX bug: check coding’ errors.

<code>\msg_fatal_text:n</code>	<code>\msg_fatal_text:n {$\langle package \rangle$}</code>
--------------------------------	---

Prints ‘Fatal $\langle package \rangle$ error’ for use in error messages.

78 Variables and constants

<code>\c_msg_error_tl</code> <code>\c_msg_warning_tl</code> <code>\c_msg_info_tl</code>

Simple headers for errors. Although these are marked as constants, they could be changed for printing errors in a different language.

```

\c_msg_coding_error_text_tl
\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
\c_msg_no_info_text_tl
\c_msg_return_text_tl

```

Various pieces of text for use in messages, which are not changed by the code here although they could be to alter the language. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\c_msg_on_line_tl
```

The ‘on line’ phrase for line numbers. Although marked as a constant, they could be changed for printing errors in a different language.

```

\c_msg_text_prefix_tl
\c_msg_more_text_prefix_tl

```

Header information for storing the ‘paths’ to parts of a message. Although these are marked as constants, they could be changed for printing errors in a different language.

```

\l_msg_class_tl
\l_msg_current_class_tl
\l_msg_current_module_tl

```

Information about message method, used for filtering.

```
\l_msg_names_clist
```

List of all of the message names defined.

```

\l_msg_redirect_classes_prop
\l_msg_redirect_names_prop

```

Re-direction lists containing the class of message to convert an different one.

```
\l_msg_redirect_classes_clist
```

List so that filtering does not loop.

Part XVII

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_`.

79 Generic functions

<code>\box_new:N</code>
<code>\box_new:c</code>

`\box_new:N <box>`

Defines `<box>` to be a new variable of type `box`.

T_EXhackers note: `\box_new:N` is the equivalent of plain T_EX's `\newbox`.

<code>\if_hbox:N</code>
<code>\if_vbox:N</code>
<code>\if_box_empty:N</code>

`\if_hbox:N <box> <true code>\else: <false code>\fi:`
`\if_box_empty:N <box> <true code>\else: <false code>\fi:`

`\if_hbox:N` and `\if_vbox:N` check if `<box>` is an horizontal or vertical box resp. `\if_box_empty:N` tests if `<box>` is empty (void) and executes `code` according to the test outcome.

T_EXhackers note: These are the T_EX primitives `\ifhbox`, `\ifvbox` and `\ifvoid`.

<code>\box_if_horizontal_p:N</code>
<code>\box_if_horizontal_p:c</code>
<code>\box_if_horizontal:NTF</code>
<code>\box_if_horizontal:cTF</code>

`\box_if_horizontal:NTF <box> {\true code} {\false code}`

Tests if `<box>` is an horizontal box and executes `<code>` accordingly.

<code>\box_if_vertical_p:N</code>
<code>\box_if_vertical_p:c</code>
<code>\box_if_vertical:NTF</code>
<code>\box_if_vertical:cTF</code>

`\box_if_vertical:NTF <box> {\true code} {\false code}`

Tests if `<box>` is a vertical box and executes `<code>` accordingly.

<code>\box_if_empty_p:N</code>
<code>\box_if_empty_p:c</code>
<code>\box_if_empty:NTF</code>
<code>\box_if_empty:cTF</code>

`\box_if_empty:NTF <box> {\true code} {\false code}`

Tests if `<box>` is empty (void) and executes `code` according to the test outcome.

T_EXhackers note: `\box_if_empty:NTF` is the L^AT_EX3 function name for `\ifvoid`.

<code>\box_set_eq:NN</code>
<code>\box_set_eq:cN</code>
<code>\box_set_eq:Nc</code>
<code>\box_set_eq:cc</code>
<code>\box_set_eq_clear:NN</code>
<code>\box_set_eq_clear:cN</code>
<code>\box_set_eq_clear:Nc</code>
<code>\box_set_eq_clear:cc</code>

`\box_set_eq:NN <box1> <box2>`

Sets $\langle box_1 \rangle$ equal to $\langle box_2 \rangle$. The `_clear` versions eradicate the contents of $\langle box_2 \rangle$ afterwards.

<code>\box_gset_eq:NN</code>
<code>\box_gset_eq:cN</code>
<code>\box_gset_eq:Nc</code>
<code>\box_gset_eq:cc</code>
<code>\box_gset_eq_clear:NN</code>
<code>\box_gset_eq_clear:cN</code>
<code>\box_gset_eq_clear:Nc</code>
<code>\box_gset_eq_clear:cc</code>

`\box_gset_eq:NN <box1> <box2>`

Globally sets $\langle box_1 \rangle$ equal to $\langle box_2 \rangle$. The `_clear` versions eradicate the contents of $\langle box_2 \rangle$ afterwards.

<code>\box_set_to_last:N</code>
<code>\box_set_to_last:c</code>
<code>\box_gset_to_last:N</code>
<code>\box_gset_to_last:c</code>

`\box_set_to_last:N <box>`

Sets $\langle box \rangle$ equal to the previous box `\l_last_box` and removes `\l_last_box` from the current list (unless in outer vertical or math mode).

<code>\box_move_right:nn</code>
<code>\box_move_left:nn</code>
<code>\box_move_up:nn</code>
<code>\box_move_down:nn</code>

`\box_move_left:nn {<dimen>} {<box function>}`

Moves $\langle box \text{ function} \rangle$ $\langle dimen \rangle$ in the direction specified. $\langle box \text{ function} \rangle$ is either an operation on a box such as `\box_use:N` or a “raw” box specification like `\vbox:n{xyz}`.

<code>\box_clear:N</code>
<code>\box_clear:c</code>
<code>\box_gclear:N</code>
<code>\box_gclear:c</code>

`\box_clear:N <box>`

Clears $\langle box \rangle$ by setting it to the constant `\c_void_box`. `\box_gclear:N` does it globally.

<code>\box_use:N</code>	
<code>\box_use:c</code>	
<code>\box_use_clear:N</code>	<code>\box_use:N $\langle box \rangle$</code>
<code>\box_use_clear:c</code>	<code>\box_use_clear:N $\langle box \rangle$</code>

`\box_use:N` puts a copy of $\langle box \rangle$ on the current list while `\box_use_clear:N` puts the box on the current list and then eradicates the contents of it.

T_EXhackers note: `\box_use:N` and `\box_use_clear:N` are the T_EX primitives `\copy` and `\box` with new (descriptive) names.

<code>\box_ht:N</code>	
<code>\box_ht:c</code>	
<code>\box_dp:N</code>	
<code>\box_dp:c</code>	
<code>\box_wd:N</code>	
<code>\box_wd:c</code>	<code>\box_ht:N $\langle box \rangle$</code>

Returns the height, depth, and width of $\langle box \rangle$ for use in dimension settings.

T_EXhackers note: These are the T_EX primitives `\ht`, `\dp` and `\wd`.

<code>\box_set_dp:Nn</code>	
<code>\box_set_dp:cn</code>	<code>\box_set_dp:Nn $\langle box \rangle$ {$\langle dimension expression \rangle$}</code>

Set the depth(below the baseline) of the $\langle box \rangle$ to the value of the { $\langle dimension expression \rangle$ }. This is a local assignment.

<code>\box_set_ht:Nn</code>	
<code>\box_set_ht:cn</code>	<code>\box_set_ht:Nn $\langle box \rangle$ {$\langle dimension expression \rangle$}</code>

Set the height(above the baseline) of the $\langle box \rangle$ to the value of the { $\langle dimension expression \rangle$ }. This is a local assignment.

<code>\box_set_wd:Nn</code>	
<code>\box_set_wd:cn</code>	<code>\box_set_wd:Nn $\langle box \rangle$ {$\langle dimension expression \rangle$}</code>

Set the width of the $\langle box \rangle$ to the value of the { $\langle dimension expression \rangle$ }. This is a local assignment.

<code>\box_show:N</code>	
<code>\box_show:c</code>	<code>\box_show:N $\langle box \rangle$</code>

Writes the contents of $\langle box \rangle$ to the log file.

T_EXhackers note: This is the T_EX primitive `\showbox`.

<code>\c_empty_box</code>
<code>\l_tmpa_box</code>
<code>\l_tmpb_box</code>

`\c_empty_box` is the constantly empty box. The others are scratch boxes.

<code>\l_last_box</code>

`\l_last_box` is more or less a read-only box register managed by the engine. It denotes the last box on the current list if there is one, otherwise it is void. You can set other boxes to this box, with the result that the last box on the current list is removed at the same time (so it is with variable with side-effects).

80 Horizontal mode

<code>\hbox:n</code>

`\hbox:n {<contents>}`

Places a `hbox` of natural size.

<code>\hbox_set:Nn</code>
<code>\hbox_set:cn</code>
<code>\hbox_gset:Nn</code>
<code>\hbox_gset:cn</code>

`\hbox_set:Nn <box> {<contents>}`

Sets `<box>` to be a horizontal mode box containing `<contents>`. It has its natural size. `\hbox_gset:Nn` does it globally.

<code>\hbox_set_to_wd:Nnn</code>
<code>\hbox_set_to_wd:cnn</code>
<code>\hbox_gset_to_wd:Nnn</code>
<code>\hbox_gset_to_wd:cnn</code>

`\hbox_set_to_wd:Nnn <box> {<dimen>} {<contents>}`

Sets `<box>` to contain `<contents>` and have width `<dimen>`. `\hbox_gset_to_wd:Nn` does it globally.

<code>\hbox_to_wd:nn</code>
<code>\hbox_to_zero:n</code>

`\hbox_to_wd:nn {<dimen>} <contents>`
`\hbox_to_zero:n <contents>`

Places a `<box>` of width `<dimen>` containing `<contents>`. `\hbox_to_zero:n` is a shorthand for a width of zero.

<code>\hbox_overlap_left:n</code>
<code>\hbox_overlap_right:n</code>

`\hbox_overlap_left:n <contents>`

Places a $\langle box \rangle$ of width zero containing $\langle contents \rangle$ in a way the it overlaps with surrounding material (sticking out to the left or right).

<code>\hbox_set_inline_begin:N</code> <code>\hbox_set_inline_begin:c</code> <code>\hbox_set_inline_end:</code> <code>\hbox_gset_inline_begin:N</code> <code>\hbox_gset_inline_begin:c</code> <code>\hbox_gset_inline_end:</code>	<code>\hbox_set_inline_begin:N</code> $\langle box \rangle$ $\langle contents \rangle$ <code>\hbox_set_inline_end:</code>
---	--

Sets $\langle box \rangle$ to contain $\langle contents \rangle$. This type is useful for use in environment definitions.

<code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code>	<code>\hbox_unpack:N</code> $\langle box \rangle$
--	---

`\hbox_unpack:N` unpacks the contents of the $\langle box \rangle$ register and `\hbox_unpack_clear:N` also clears the $\langle box \rangle$ after unpacking it.

T_EXhackers note: These are the T_EX primitives `\unhcopy` and `\unhbox`.

81 Vertical mode

<code>\vbox:n</code>	<code>\vbox:n</code> $\{\langle contents \rangle\}$
----------------------	---

Places a `vbox` of natural size with baseline equal to the baseline of the last object in the box, i.e., if the last object is a line of text the box has the same depth as that line; otherwise the depth will be zero.

<code>\vbox_top:n</code>	<code>\vbox_top:n</code> $\{\langle contents \rangle\}$
--------------------------	---

Same as `\vbox:n` except that the reference point will be at the baseline of the first object in the box not the last.

<code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code>	<code>\vbox_set:Nn</code> $\langle box \rangle$ $\{\langle contents \rangle\}$
--	--

Sets $\langle box \rangle$ to be a vertical mode box containing $\langle contents \rangle$. It has its natural size and the reference point will be at the baseline of the last object in the box. `\vbox_gset:Nn` does it globally.

<code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code>	<code>\vbox_set_top:Nn</code> $\langle box \rangle$ $\{ \langle contents \rangle \}$
--	--

Sets $\langle box \rangle$ to be a vertical mode box containing $\langle contents \rangle$. It has its natural size (usually a small height and a larger depth) and the reference point will be at the baseline of the first object in the box. `\vbox_gset_top:Nn` does it globally.

<code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <code>\vbox_gset_to_ht:ccn</code>	<code>\vbox_set_to_ht:Nnn</code> $\langle box \rangle$ $\{ \langle dimen \rangle \}$ $\{ \langle contents \rangle \}$
---	---

Sets $\langle box \rangle$ to contain $\langle contents \rangle$ and have total height $\langle dimen \rangle$. `\vbox_gset_to_ht:Nn` does it globally.

<code>\vbox_set_inline_begin:N</code> <code>\vbox_set_inline_end:</code> <code>\vbox_gset_inline_begin:N</code> <code>\vbox_gset_inline_end:</code>	<code>\vbox_set_inline_begin:N</code> $\langle box \rangle$ $\langle contents \rangle$ <code>\vbox_set_inline_end:</code>
--	--

Sets $\langle box \rangle$ to contain $\langle contents \rangle$. This type is useful for use in environment definitions.

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$ $\{ \langle dimen \rangle \}$
--	--

Sets $\langle box_1 \rangle$ to contain the top $\langle dimen \rangle$ part of $\langle box_2 \rangle$.

T_EXhackers note: This is the T_EX primitive `\vsplit`.

<code>\vbox_to_ht:nn</code> <code>\vbox_to_zero:n</code>	<code>\vbox_to_ht:nn</code> $\{ \langle dimen \rangle \}$ $\langle contents \rangle$ <code>\vbox_to_zero:n</code> $\langle contents \rangle$
---	---

Places a $\langle box \rangle$ of size $\langle dimen \rangle$ containing $\langle contents \rangle$.

<code>\vbox_unpack:N</code> <code>\vbox_unpack:c</code> <code>\vbox_unpack_clear:N</code> <code>\vbox_unpack_clear:c</code>	<code>\vbox_unpack:N</code> $\langle box \rangle$
--	---

`\vbox_unpack:N` unpacks the contents of the $\langle box \rangle$ register and `\vbox_unpack_clear:N` also clears the $\langle box \rangle$ after unpacking it.

T_EXhackers note: These are the T_EX primitives `\unvcopy` and `\unvbox`.

Part XVIII

The l3xref package

Cross references

`\xref_set_label:n` `\xref_set_label:n {<name>}`

Sets a label in the text. Note that this function does not do anything else than setting the correct labels. In particular, it does not try to fix any spacing around the write node; this is a task for the `galley2` module.

`\xref_new:nn` `\xref_new:nn {<type>} {<value>}`

Defines a new cross reference type `<type>`. This defines the token list variable `\l_xref_curr_<type>_tl` with default value `<value>` which gets written fully expanded when `\xref_set_label:n` is called.

`\xref_deferred_new:nn` `\xref_deferred_new:nn {<type>} {<value>}`

Same as `\xref_new:n` except for this one, the value written happens when T_EX ships out the page. Page numbers use this one obviously.

`\xref_get_value:nn *` `\xref_get_value:nn {<type>} {<name>}`

Extracts the cross reference information of type `<type>` for the label `<name>`. This operation is expandable.

Part XIX

The l3keyval package

Key-value parsing

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.

This module provides the low-level machinery for processing arbitrary key–value lists. The `l3keys` module provides a higher-level interface for managing run-time settings using key–value input, while other parts of L^AT_EX3 also use key–value input based on `l3keyval` (for example the `xtemplate` module).

82 Features of `l3keyval`

As `l3keyval` is a low-level module, its functions are restricted to converting a *keyval list* into keys and values for further processing. Each key and value (or key alone) has to be processed further by a function provided when `l3keyval` is called. Typically, this will be *via* one of the `\KV_process...` functions:

```
\KV_process_space_removal_sanitize:NNn
\my_processor_function_one:n
\my_processor_function_two:nn
{ <keyval list> }
```

The two processor functions here handle the cases where there is only a key, and where there is both a key and value, respectively.

`l3keyval` parses key–value lists in a manner that does not double `#` tokens or expand any input. The module has processor functions which will sanitize the category codes of `=` and `,` tokens (for use in the document body) as well as faster versions which do not do this (for use inside code blocks). Spaces can be removed from each end of the key and value (again for the document body), again with faster code to be used where this is not necessary. Values which are wrapped in braces will have exactly one set removed, meaning that

```
key = {value here},
```

and

```
key = value here,
```

are treated as identical (assuming that space removal is in force). `l3keyval`

83 Functions for keyval processing

The `l3keyval` module should be accessed *via* a small set of external functions. These correctly set up the module internals for use by other parts of L^AT_EX3.

In all cases, two functions have to be supplied by the programmer to apply to the items from the `<keyval list>` after `!3keyval` has separated out the entries. The first function should take one argument, and will receive the names of keys for which no value was supplied. The second function should take two arguments: a key name and the associated value.

<code>\KV_process_space_removal_sanitiz:NNn</code>	<code>\KV_process_space_removal_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
--	---

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, and the category codes of non-braced `=` and `,` tokens are normalised so that parsing is ‘category code safe’. After parsing is completed, $\langle function_1 \rangle$ is used to process keys without values and $\langle function_2 \rangle$ deals with keys which have associated values.

<code>\KV_process_space_removal_no_sanitiz:NNn</code>	<code>\KV_process_space_removal_no_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
---	--

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, but category codes are not normalised. After parsing is completed, $\langle function_1 \rangle$ is used to process keys without values and $\langle function_2 \rangle$ deals with keys which have associated values.

<code>\KV_process_no_space_removal_no_sanitiz:NNn</code>	<code>\KV_process_no_space_removal_no_sanitiz:NNn</code> $\langle function_1 \rangle \langle function_2 \rangle \{ \langle keyval list \rangle \}$
--	---

Parses the `<keyval list>` splitting it into keys and associated values. Spaces are *not* removed from the ends of the key and value, and category codes are *not* normalised. After parsing is completed, $\langle function_1 \rangle$ is used to process keys without values and $\langle function_2 \rangle$ deals with keys which have associated values.

<code>\l_KV_remove_one_level_of_braces_bool</code>	This boolean controls whether or not one level of braces is stripped from the key and value. The default value for this boolean is <code>true</code> so that exactly one level of braces is stripped. For certain applications it is desirable to keep the braces in which case the programmer just has to set the boolean false temporarily. Only applicable when spaces are being removed.
--	--

84 Internal functions

The remaining functions provided by `!3keyval` do not have any protection for nesting of one call to the module inside another. They should therefore not be called directly by other modules.

<code>\KV_parse_no_space_removal_no_sanitize:n</code>	<code>\KV_parse_no_space_removal_no_sanitize:n</code> $\{\langle keyval list \rangle\}$
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are not removed in the parsing process and the category codes of = and , are not normalised.

<code>\KV_parse_space_removal_no_sanitize:n</code>	<code>\KV_parse_space_removal_no_sanitize:n</code> $\{\langle keyval list \rangle\}$
--	--

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value, but the category codes of = and , are not normalised.

<code>\KV_parse_space_removal_sanitize:n</code>	<code>\KV_parse_space_removal_sanitize:n</code> $\{\langle keyval list \rangle\}$
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value and the category codes of = and , are normalised at the outer level (*i.e.* only unbraced tokens are affected).

<code>\KV_key_no_value_elt:n</code>	<code>\KV_key_no_value_elt:n</code> $\{\langle key \rangle\}$
<code>\KV_key_value_elt:nn</code>	<code>\KV_key_value_elt:n</code> $\{\langle key \rangle\} \{\langle value \rangle\}$

Used by `\KV_parse...` functions to further process keys with no values and keys with values, respectively. The standard definitions are error functions: the programmer should provide appropriate definitions for both at point of use.

85 Variables and constants

<code>\c_KV_single_equal_sign_tl</code>	Constant token list to make finding = faster.
---	---

<code>\l_KV_tmpa_tl</code> <code>\l_KV_tmpb_tl</code>	Scratch token lists.
--	----------------------

<code>\l_KV_parse_tl</code> <code>\l_KV_currkey_tl</code> <code>\l_KV_currval_tl</code>	Token list variables for various parts of the parsed input.
---	---

Part XX

The l3keys package

Key–value support

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. For the user, the system normally results in input of the form

```
\PackageControlMacro{
  key-one = value one,
  key-two = value two
}
```

or

```
\PackageMacro[
  key-one = value one,
  key-two = value two
]{argument}.
```

For the programmer, the original keyval package gives only the most basic interface for this work. All key macros have to be created one at a time, and as a result the kvoptions and xkeyval packages have been written to extend the ease of creating keys. A very different approach has been provided by the pgfkeys package, which uses a key–value list to generate keys.

The l3keys package is aimed at creating a programming interface for key–value controls in L^AT_EX3. Keys are created using a key–value interface, in a similar manner to pgfkeys. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { module }
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_module_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { module }
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` is used within a document function. For L^AT_EX2_ε, a generic set up function could be created with

```

\newcommand*\SomePackageSetup[1]{%
  \@nameuse{keys_set:nn}{module}{#1}%
}

```

or to use key–value input as the optional argument for a macro:

```

\newcommand*\SomePackageMacro[2][]{%
  \begingroup
    \@nameuse{keys_set:nn}{module}{#1}%
    % Main code for \SomePackageMacro
  \endgroup
}

```

The same concepts using `xparse` for L^AT_EX3 use:

```

\DeclareDocumentCommand \SomePackageSetup { m } {
  \keys_set:nn { module } { #1 }
}
\DeclareDocumentCommand \SomePackageMacro { o m } {
  \group_begin:
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}

```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 87, it is suggested that the character ‘/’ is reserved for sub-division of keys into logical groups. Macros are *not* expanded when creating key names, and so

```

\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module } {
  \l_module_tmp_tl .code:n = code
}

```

will create a key called `\l_module_tmp_tl`, and not one called `key`.

86 Creating keys

```

\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 $\langle keyval list \rangle$ 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:
}
```

where the properties of the key begin from the `.` after the key name.

The `\keys_define:nn` function does not skip spaces in the input, and does not check the category codes for `,` and `=` tokens. This means that it is intended for use with code blocks and other environments where spaces are ignored.

<code>.bool_set:N</code>	$\langle key \rangle$ <code>.bool_set:N = $\langle bool \rangle$</code>
<code>.bool_gset:N</code>	

Defines $\langle key \rangle$ to set $\langle bool \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`). Here, $\langle bool \rangle$ is a L^AT_EX3 boolean variable (*i.e.* created using `\bool_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.choice:</code>	$\langle key \rangle$ <code>.choice:</code>
-----------------------	---

Sets $\langle key \rangle$ to act as a multiple choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section [87.1](#).

<code>.choice_code:n</code>	$\langle key \rangle$ <code>.choice_code:n = $\langle code \rangle$</code>
<code>.choice_code:x</code>	

Stores $\langle code \rangle$ for use when `.generate_choices:n` creates one or more choice sub-keys of the current key. Inside $\langle code \rangle$, `\l_keys_choice_tl` contains the name of the choice made, and `\l_keys_choice_int` is the position of the choice in the list given to `.generate_choices:n`. Choices are discussed in detail in section [87.1](#).

<code>.code:n</code>	$\langle key \rangle$ <code>.code:n = $\langle code \rangle$</code>
<code>.code:x</code>	

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is called. The $\langle code \rangle$ can include one parameter (`#1`), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The `.code:x` variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

<code>.default:n</code>	$\langle key \rangle$ <code>.default:n = $\langle default \rangle$</code>
<code>.default:V</code>	

Creates a $\langle default \rangle$ value for $\langle key \rangle$, which is used if no value is given. This will be used if only the key name is given, but not if a blank $\langle value \rangle$ is given:


```

\keys_define:nn { module } {
  key .code:n    = Hello #1,
  key .default:n = World
}
\keys_set:nn { module} {
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,    % Prints 'Hello '
}

```

T_EXhackers note: The $\langle default \rangle$ is stored as a token list variable, and therefore should not contain unescaped # tokens.

<code>.dim_set:N</code>
<code>.dim_set:c</code>
<code>.dim_gset:N</code>
<code>.dim_gset:c</code>

 $\langle key \rangle .dim_set:N = \langle dimension \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle dimension \rangle$. Here, $\langle dimension \rangle$ is a L^AT_EX3 dim variable (*i.e.* created using `\dim_new:N`) or a L^AT_EX2_ε `dimen` (*i.e.* created using `\newdimen`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.fp_set:N</code>
<code>.fp_set:c</code>
<code>.fp_gset:N</code>
<code>.fp_gset:c</code>

 $\langle key \rangle .fp_set:N = \langle floating\ point \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle floating\ point \rangle$. Here, $\langle floating\ point \rangle$ is a L^AT_EX3 fp variable (*i.e.* created using `\fp_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.generate_choices:n</code>

 $\langle key \rangle .generate_choices:n = \langle comma\ list \rangle$

Makes $\langle key \rangle$ a multiple choice key, accepting the choices specified in $\langle comma\ list \rangle$. Each choice will execute code which should previously have been defined using `.choice_code:n` or `.choice_code:x`. Choices are discussed in detail in section 87.1.

<code>.int_set:N</code>
<code>.int_set:c</code>
<code>.int_gset:N</code>
<code>.int_gset:c</code>

 $\langle key \rangle .int_set:N = \langle integer \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle integer \rangle$. Here, $\langle integer \rangle$ is a L^AT_EX3 int variable (*i.e.* created using `\int_new:N`) or a L^AT_EX2_ε `count` (*i.e.* created using `\newcount`). If the variable does not exist, it will be created at the point that the key is set up.

```
.meta:n
.meta:x <key> .meta:n = <multiple keys>
```

Makes $\langle key \rangle$ a meta-key, which will set $\langle multiple\ keys \rangle$ in one go. If $\langle key \rangle$ is given with a value at the time the key is used, then the value will be passed through to the subsidiary $\langle keys \rangle$ for processing (as #1).

```
.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c <key> .skip_set:N = <skip>
```

Sets $\langle key \rangle$ to store the value it is given in $\langle skip \rangle$, which is created if it does not already exist. Here, $\langle skip \rangle$ is a L^AT_EX3 `skip` variable (*i.e.* created using `\skip_new:N`) or a L^AT_EX 2_ε `skip` (*i.e.* created using `\newskip`). If the variable does not exist, it will be created at the point that the key is set up.

```
.tl_set:N
.tl_set:c
.tl_set_x:N
.tl_set_x:c
.tl_gset:N
.tl_gset:c
.tl_gset_x:N
.tl_gset_x:c <key> .tl_set:N = <token list variable>
```

Sets $\langle key \rangle$ to store the value it is given in $\langle token\ list\ variable \rangle$, which is created if it does not already exist. Here, $\langle token\ list\ variable \rangle$ is a L^AT_EX3 `tl` variable (*i.e.* created using `\tl_new:N`) or a L^AT_EX 2_ε macro with no arguments (*i.e.* created using `\newcommand` or `\def`). If the variable does not exist, it will be created at the point that the key is set up. The `x` variants perform an `x` expansion at the time the $\langle value \rangle$ passed to the $\langle key \rangle$ is saved to the $\langle token\ list\ variable \rangle$.

```
.value_forbidden:
.value_required: <key> .value_forbidden:
```

Flags for forbidding and requiring a $\langle value \rangle$ for $\langle key \rangle$. Giving a $\langle value \rangle$ for a $\langle key \rangle$ which has the `.value_forbidden:` property set will result in an error. In the same way, if a $\langle key \rangle$ has the `.value_required:` property set then a $\langle value \rangle$ must be given when the $\langle key \rangle$ is used.

87 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 { module } {
  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 will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

87.1 Multiple choices

Multiple choices are created by setting the `.choice:` property:

```
\keys_define:nn { module } {
  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 choices. Here, the keys can share the same code, and can be rapidly created using the `.choice_code:n` and `.generate_choices:n` properties:

```
\keys_define:nn { module } {
  key .choice_code:n = {
    You~gave~choice~‘‘\int_use:N \l_keys_choice_tl’’ ,~
    which~is~in~position~
    \int_use:N\l_keys_choice_int\space
    in~the~list.
  },
  key .generate_choices:n = {
    choice-a, choice-b, choice-c
  }
}
```

Following common computing practice, `\l_keys_choice_int` is indexed from 0 (as an offset), so that the value of `\l_keys_choice_int` for the first choice in a list will be zero. This means that `\l_keys_choice_int` can be used directly with `\if_case:w` and so on.

```
\l_keys_choice_int
\l_keys_choice_tl
```

Inside the code block for a choice generated using `.generate_choice:`, 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 0.

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 { module } {
  key .choice:n,
  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 `.generate_choices:n` (*i.e.* anything might happen!).

88 Setting keys

```
\keys_set:nn
\keys_set:nV
\keys_set:nv
```

`\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 will be illustrated later. In contrast to `\keys_define:nn`, this function does check category codes and ignore spaces, and is therefore suitable for user input.

If a key is not known, `\keys_set:nn` will look for a special `unknown` key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module } {
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_path_tl'~to~'#1'
}
```

```
\l_keys_key_tl
```

When processing an unknown key, the name of the key is available as `\l_keys_key_tl`. Note that this will have been processed using `\tl_to_str:N`. The value passed to the key (if any) is available as the macro parameter `#1`.

88.1 Examining keys: internal representation

`\keys_if_exist:nnTF` `\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}`

Tests if *<key>* exists for *<module>*, i.e. if any code has been defined for *<key>*.

TeXhackers note: The function works by testing for the existence of the internal function `\keys > <module>/<key>.cmd:n`.

`\keys_show:nn` `\keys_show:nn {<module>} {<key>}`

Shows the internal representation of a *<key>*.

TeXhackers note: Keys are stored as functions with names of the format `\keys > <module>/<key>.cmd:n`.

89 Internal functions

`\keys_bool_set:Nn` `\keys_bool_set:Nn <bool> {<scope>}`

Creates code to set *<bool>* when *<key>* is given, with setting using *<scope>* (empty or `g` for local or global, respectively). *<bool>* should be a L^AT_EX3 boolean variable.

`\keys_choice_code_store:x` `\keys_choice_code_store:x <code>`

Stores *<code>* for later use by `.generate_code:n`.

`\keys_choice_make:` `\keys_choice_make:`

Makes *<key>* a choice key.

`\keys_choices_generate:n` `\keys_choices_generate:n {<comma list>}`

Makes *<comma list>* choices for *<key>*.

`\keys_choice_find:n` `\keys_choice_find:n {<choice>}`

Searches for *<choice>* as a sub-key of *<key>*.

<code>\keys_cmd_set:nn</code>
<code>\keys_cmd_set:nx</code>

`\keys_cmd_set:nn {⟨path⟩} {⟨code⟩}`
 Creates a function for `⟨path⟩` using `⟨code⟩`.

<code>\keys_default_set:n</code>
<code>\keys_default_set:V</code>

`\keys_default_set:n {⟨default⟩}`
 Sets `⟨default⟩` for `⟨key⟩`.

<code>\keys_define_elt:n</code>
<code>\keys_define_elt:nn</code>

`\keys_define_elt:nn {⟨key⟩} {⟨value⟩}`
 Processing functions for key–value pairs when defining keys.

<code>\keys_define_key:n</code>

`\keys_define_key:n {⟨key⟩}`
 Defines `⟨key⟩`.

<code>\keys_execute:</code>

`\keys_execute:`
 Executes `⟨key⟩` (where the name of the `⟨key⟩` will be stored internally).

<code>\keys_execute_unknown:</code>

`\keys_execute_unknown:`

Handles unknown `⟨key⟩` names.

<code>\keys_if_value_requirement:nTF</code>
<code>\keys_if_value_requirement:nTF *</code>

`\keys_if_value_requirement:nTF {⟨requirement⟩} {⟨true code⟩} {⟨false code⟩}`

Check if `⟨requirement⟩` applies to `⟨key⟩`.

<code>\keys_meta_make:n</code>
<code>\keys_meta_make:x</code>

`\keys_meta_make:n {⟨keys⟩}`
 Makes `⟨key⟩` a meta-key to set `⟨keys⟩`.

<code>\keys_property_find:n</code>

`\keys_property_find:n {⟨key⟩}`

Separates `⟨key⟩` from `⟨property⟩`.

<code>\keys_property_new:nn</code>
<code>\keys_property_new_arg:nn</code>

`\keys_property_new:nn {⟨property⟩} {⟨code⟩}`

Makes a new $\langle property \rangle$ expanding to $\langle code \rangle$. The `arg` version makes properties with one argument.

`\keys_property_undefine:n` `\keys_property_undefine:n { $\langle property \rangle$ }`

Deletes $\langle property \rangle$ of $\langle key \rangle$.

`\keys_set_elt:n`
`\keys_set_elt:nn` `\keys_set_elt:nn { $\langle key \rangle$ } { $\langle value \rangle$ }`

Processing functions for key–value pairs when setting keys.

`\keys_tmp:w` `\keys_tmp:w $\langle args \rangle$`
 Used to store $\langle code \rangle$ to execute a $\langle key \rangle$.

`\keys_value_or_default:n` `\keys_value_or_default:n { $\langle value \rangle$ }`

Sets `\l_keys_value_tl` to $\langle value \rangle$, or $\langle default \rangle$ if $\langle value \rangle$ was not given and if $\langle default \rangle$ is available.

`\keys_value_requirement:n` `\keys_value_requirement:n { $\langle requirement \rangle$ }`

Sets $\langle key \rangle$ to have $\langle requirement \rangle$ concerning $\langle value \rangle$.

`\keys_variable_set:NnNN`
`\keys_variable_set:cnNN` `\keys_variable_set:NnNN $\langle var \rangle$ $\langle type \rangle$ $\langle scope \rangle$ $\langle expansion \rangle$`

Sets $\langle key \rangle$ to assign $\langle value \rangle$ to $\langle variable \rangle$. The $\langle scope \rangle$ (blank for local, `g` for global) and $\langle type \rangle$ (`tl`, `int`, etc.) are given explicitly.

90 Variables and constants

`\c_keys_properties_root_tl`
`\c_keys_root_tl` The root paths for keys and properties, used to generate the names of the functions which store these items.

`\c_keys_value_forbidden_tl`
`\c_keys_value_required_tl` Marker text containers: by storing the values the code can make comparisons slightly faster.

`\l_keys_choice_code_tl` Used to transfer code from storage when making multiple choices.

`\l_keys_module_tl`
`\l_keys_path_tl`
`\l_keys_property_tl` Various key paths need to be stored. These are flexible items that are set during the key reading process.

`\l_keys_no_value_bool` A marker for ‘no value’ as key input.

`\l_keys_value_tl` Holds the currently supplied value, in a token register as there may be # tokens.

Part XXI

The l3file package

File Loading

91 Loading files

In contrast to the l3io module, which deals with the lowest level of file management, the l3file module provides a higher level interface for handling file contents. This involves providing convenient wrappers around many of the functions in l3io to make them more generally accessible.

It is important to remember that T_EX will attempt to locate files using both the operating system path and entries in the T_EX file database (most T_EX systems use such a database). Thus the ‘current path’ for T_EX is somewhat broader than that for other programs.

`\g_file_current_name_tl` Contains the name of the current LaTeX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_job_name_tl` at the start of a L^AT_EX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`
`\file_if_exist:VTF` `\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`
Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. The branching versions then leave either `<true`

code) or *(false code)* in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

<code>\file_input:n</code>	<code>\file_input:n {<file name>}</code>
<code>\file_input:V</code>	

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function.

<code>\file_path_include:n</code>	<code>\file_path_include:n {<path>}</code>
-----------------------------------	--

Adds *<path>* to the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

<code>\file_path_remove:n</code>	<code>\file_path_remove:n {<path>}</code>
----------------------------------	---

Removes *<path>* from the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

<code>\file_list:</code>	<code>\file_list:</code>
--------------------------	--------------------------

This function will list all files loaded using `\file_input:n` in the log file.

Part XXII

The l3fp package

Floating point arithmetic

92 Floating point numbers

A floating point number is one which is stored as a mantissa and a separate exponent. This module implements arithmetic using radix 10 floating point numbers. This means that the mantissa should be a real number in the range $1 \leq |x| < 10$, with the exponent given as an integer between -99 and 99 . In the input, the exponent part is represented starting with an `e`. As this is a low-level module, error-checking is minimal. Numbers which are too large for the floating point unit to handle will result in errors, either from T_EX or from L^AT_EX. The L^AT_EX code does not check that the input will not overflow, hence the possibility of a T_EX error. On the other hand, numbers which are too small will be dropped, which will mean that extra decimal digits will simply be lost.

When parsing numbers, any missing parts will be interpreted as zero. So for example

```

\fp_set:Nn \l_my_fp { }
\fp_set:Nn \l_my_fp { . }
\fp_set:Nn \l_my_fp { - }

```

will all be interpreted as zero values without raising an error.

Operations which give an undefined result (such as division by 0) will not lead to errors. Instead special marker values are returned, which can be tested for using for example `\fp_if_undefined:N(TF)`. In this way it is possible to work with asymptotic functions without first checking the input. If these special values are carried forward in calculations they will be treated as 0.

Floating point numbers are stored in the `fp` floating point variable type. This has a standard range of functions for variable management.

92.1 Constants

`\c_e_fp` The value of the base of natural numbers, e .

`\c_one_fp` A floating point variable with permanent value 1: used for speeding up some comparisons.

`\c_pi_fp` The value of π .

`\c_undefined_fp` A special marker floating point variable representing the result of an operation which does not give a defined result (such as division by 0).

`\c_zero_fp` A permanently zero floating point variable.

92.2 Floating-point variables

`\fp_new:N`
`\fp_new:c` `\fp_new:N` *<floating point variable>*

Creates a new *<floating point variable>* or raises an error if the name is already taken. The declaration global. The *<floating point>* will initially be set to `+0.000000000e0` (the zero floating point).

`\fp_const:Nn`
`\fp_const:cn` `\fp_const:Nn` *<floating point variable>* *<value>*

Creates a new constant *<floating point variable>* or raises an error if the name is already taken. The value of the *<floating point variable>* will be set globally to the *<value>*.

<code>\fp_set_eq:NN</code>
<code>\fp_set_eq:cN</code>
<code>\fp_set_eq:Nc</code>
<code>\fp_set_eq:cc</code>

`\fp_set_eq:NN` $\langle fp\ var1 \rangle$ $\langle fp\ var2 \rangle$

Sets the value of $\langle floating\ point\ variable1 \rangle$ equal to that of $\langle floating\ point\ variable2 \rangle$. This assignment is restricted to the current T_EX group level.

<code>\fp_gset_eq:NN</code>
<code>\fp_gset_eq:cN</code>
<code>\fp_gset_eq:Nc</code>
<code>\fp_gset_eq:cc</code>

`\fp_gset_eq:NN` $\langle fp\ var1 \rangle$ $\langle fp\ var2 \rangle$

Sets the value of $\langle floating\ point\ variable1 \rangle$ equal to that of $\langle floating\ point\ variable2 \rangle$. This assignment is global and so is not limited by the current T_EX group level.

<code>\fp_zero:N</code>
<code>\fp_zero:c</code>

`\fp_zero:N` $\langle floating\ point\ variable \rangle$

Sets the $\langle floating\ point\ variable \rangle$ to +0.000000000e0 within the current scope.

<code>\fp_gzero:N</code>
<code>\fp_gzero:c</code>

`\fp_gzero:N` $\langle floating\ point\ variable \rangle$

Sets the $\langle floating\ point\ variable \rangle$ to +0.000000000e0 globally.

<code>\fp_set:Nn</code>
<code>\fp_set:cn</code>

`\fp_set:Nn` $\langle floating\ point\ variable \rangle$ $\{ \langle value \rangle \}$

Sets the $\langle floating\ point\ variable \rangle$ variable to $\langle value \rangle$ within the scope of the current T_EX group.

<code>\fp_gset:Nn</code>
<code>\fp_gset:cn</code>

`\fp_gset:Nn` $\langle floating\ point\ variable \rangle$ $\{ \langle value \rangle \}$

Sets the $\langle floating\ point\ variable \rangle$ variable to $\langle value \rangle$ globally.

<code>\fp_set_from_dim:Nn</code>
<code>\fp_set_from_dim:cn</code>

`\fp_set_from_dim:Nn` $\langle floating\ point\ variable \rangle$ $\{ \langle dimexpr \rangle \}$

Sets the $\langle floating\ point\ variable \rangle$ to the distance represented by the $\langle dimension\ expression \rangle$ in the units points. This means that distances given in other units are first converted to points before being assigned to the $\langle floating\ point\ variable \rangle$. The assignment is local.

<code>\fp_gset_from_dim:Nn</code>
<code>\fp_gset_from_dim:cn</code>

`\fp_gset_from_dim:Nn` $\langle floating\ point\ variable \rangle$ $\{ \langle dimexpr \rangle \}$

Sets the *floating point variable* to the distance represented by the *dimension expression* in the units points. This means that distances given in other units are first converted to points before being assigned to the *floating point variable*. The assignment is global.

\fp_use:N ★	\fp_use:N <i>floating point variable</i>
\fp_use:c ★	

Inserts the value of the *floating point variable* into the input stream. The value will be given as a real number without any exponent part, and will always include a decimal point. For example,

```
\fp_new:Nn \test
\fp_set:Nn \test { 1.234 e 5 }
\fp_use:N \test
```

will insert ‘12345.00000’ into the input stream. As illustrated, a floating point will always be inserted with ten significant digits given. Very large and very small values will include additional zeros for place value.

\fp_show:N	\fp_show:N <i>floating point variable</i>
\fp_show:c	

Displays the content of the *floating point variable* on the terminal.

92.3 Conversion to other formats

It is useful to be able to convert floating point variables to other forms. These functions are expandable, so that the material can be used in a variety of contexts. The `\fp_use:N` function should also be consulted in this context, as it will insert the value of the floating point variable as a real number.

\fp_to_dim:N ★	\fp_to_dim:N <i>floating point variable</i>
\fp_to_dim:c ★	

Inserts the value of the *floating point variable* into the input stream converted into a dimension in points.

\fp_to_int:N ★	\fp_to_int:N <i>floating point variable</i>
\fp_to_int:c ★	

Inserts the integer value of the *floating point variable* into the input stream. The decimal part of the number will not be included, but will be used to round the integer.

\fp_to_tl:N ★	\fp_to_tl:N <i>floating point variable</i>
\fp_to_tl:c ★	

Inserts a representation of the *floating point variable* into the input stream as a token list. The representation follows the conventions of a pocket calculator:

Floating point value	Representation
1.234000000000e0	1.234
-1.234000000000e0	-1.234
1.234000000000e3	1234
1.234000000000e13	1234e13
1.234000000000e-1	0.1234
1.234000000000e-2	0.01234
1.234000000000e-3	1.234e-3

Notice that trailing zeros are removed in this process, and that numbers which do not require a decimal part do *not* include a decimal marker.

92.4 Rounding floating point values

The module can round floating point values to either decimal places or significant figures using the usual method in which exact halves are rounded up.

<code>\fp_round_figures:Nn</code> <code>\fp_round_figures:cn</code>
--

`\fp_round_figures:Nn <floating point variable> {<target>}`

Rounds the *<floating point variable>* to the *<target>* number of significant figures (an integer expression). The rounding is carried out locally.

<code>\fp_ground_figures:Nn</code> <code>\fp_ground_figures:cn</code>
--

`\fp_ground_figures:Nn <floating point variable> {<target>}`

Rounds the *<floating point variable>* to the *<target>* number of significant figures (an integer expression). The rounding is carried out globally.

<code>\fp_round_places:Nn</code> <code>\fp_round_places:cn</code>
--

`\fp_round_places:Nn <floating point variable> {<target>}`

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out locally.

<code>\fp_ground_places:Nn</code> <code>\fp_ground_places:cn</code>
--

`\fp_ground_places:Nn <floating point variable> {<target>}`

Rounds the *<floating point variable>* to the *<target>* number of decimal places (an integer expression). The rounding is carried out globally.

92.5 Tests on floating-point values

$\backslash\text{fp_if_undefined_p:N} \star$	$\backslash\text{fp_if_undefined_p:N} \langle\text{fixed-point}\rangle$
$\backslash\text{fp_if_undefined:N}\underline{\text{TF}} \star$	$\backslash\text{fp_if_undefined:N}\underline{\text{TF}} \langle\text{fixed-point}\rangle$
	$\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

Tests if $\langle\text{floating point}\rangle$ is undefined (*i.e.* equal to the special $\backslash\text{c_undefined_fp}$ variable). The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

$\backslash\text{fp_if_zero_p:N} \star$	$\backslash\text{fp_if_zero_p:N} \langle\text{fixed-point}\rangle$
$\backslash\text{fp_if_zero:N}\underline{\text{TF}} \star$	$\backslash\text{fp_if_zero:N}\underline{\text{TF}} \langle\text{fixed-point}\rangle \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

Tests if $\langle\text{floating point}\rangle$ is equal to zero (*i.e.* equal to the special $\backslash\text{c_zero_fp}$ variable). The branching versions then leave either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

	$\backslash\text{fp_compare:nNnTF}$
$\backslash\text{fp_compare:nNn}\underline{\text{TF}}$	$\{\langle\text{floating point}_1\rangle\} \langle\text{relation}\rangle \{\langle\text{floating point}_2\rangle\}$
	$\{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$

This function compared the two $\langle\text{floating point}\rangle$ values, which may be stored as **fp** variables, using the $\langle\text{relation}\rangle$:

Equal	=
Greater than	>
Less than	<

Either $\langle\text{true code}\rangle$ or $\langle\text{false code}\rangle$ is then left in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

92.6 Unary operations

The unary operations alter the value stored within an **fp** variable.

$\backslash\text{fp_abs:N}$	
$\backslash\text{fp_abs:c}$	$\backslash\text{fp_abs:N} \langle\text{floating point variable}\rangle$

Converts the $\langle\text{floating point variable}\rangle$ to its absolute value, assigning the result within the current $\text{T}_{\text{E}}\text{X}$ group.

$\backslash\text{fp_gabs:N}$	
$\backslash\text{fp_gabs:c}$	$\backslash\text{fp_gabs:N} \langle\text{floating point variable}\rangle$

Converts the $\langle\text{floating point variable}\rangle$ to its absolute value, assigning the result globally.

<code>\fp_neg:N</code> <code>\fp_neg:c</code>
--

`\fp_neg:N` *⟨floating point variable⟩*

Reverse the sign of the *⟨floating point variable⟩*, assigning the result within the current T_EX group.

<code>\fp_gneg:N</code> <code>\fp_gneg:c</code>
--

`\fp_gneg:N` *⟨floating point variable⟩*

Reverse the sign of the *⟨floating point variable⟩*, assigning the result globally.

92.7 Arithmetic operations

Binary arithmetic operations act on the value stored in an `fp`, so for example

```
\fp_set:Nn \l_my_fp { 1.234 }
\fp_sub:Nn \l_my_fp { 5.678 }
```

sets `\l_my_fp` to the result of $1.234 - 5.678$ (*i.e.* -4.444).

<code>\fp_add:Nn</code> <code>\fp_add:cn</code>
--

`\fp_add:Nn` *⟨floating point⟩* {*⟨value⟩*}

Adds the *⟨value⟩* to the *⟨floating point⟩*, making the assignment within the current T_EX group level.

<code>\fp_gadd:Nn</code> <code>\fp_gadd:cn</code>
--

`\fp_gadd:Nn` *⟨floating point⟩* {*⟨value⟩*}

Adds the *⟨value⟩* to the *⟨floating point⟩*, making the assignment globally.

<code>\fp_sub:Nn</code> <code>\fp_sub:cn</code>
--

`\fp_sub:Nn` *⟨floating point⟩* {*⟨value⟩*}

Subtracts the *⟨value⟩* from the *⟨floating point⟩*, making the assignment within the current T_EX group level.

<code>\fp_gsub:Nn</code> <code>\fp_gsub:cn</code>
--

`\fp_gsub:Nn` *⟨floating point⟩* {*⟨value⟩*}

Subtracts the *⟨value⟩* from the *⟨floating point⟩*, making the assignment globally.

<code>\fp_mul:Nn</code> <code>\fp_mul:cn</code>
--

`\fp_mul:Nn` *⟨floating point⟩* {*⟨value⟩*}

Multiplies the *⟨floating point⟩* by the *⟨value⟩*, making the assignment within the current T_EX group level.

<code>\fp_gmul:Nn</code>
<code>\fp_gmul:cn</code>

`\fp_gmul:Nn` *<floating point>* {*<value>*}

Multiples the *<floating point>* by the *<value>*, making the assignment globally.

<code>\fp_div:Nn</code>
<code>\fp_div:cn</code>

`\fp_div:Nn` *<floating point>* {*<value>*}

Divides the *<floating point>* by the *<value>*, making the assignment within the current TeX group level. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is local.

<code>\fp_gdiv:Nn</code>
<code>\fp_gdiv:cn</code>

`\fp_gdiv:Nn` *<floating point>* {*<value>*}

Divides the *<floating point>* by the *<value>*, making the assignment globally. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is global.

92.8 Power operations

<code>\fp_pow:Nn</code>
<code>\fp_pow:cn</code>

`\fp_pow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*, which should be a positive real number or a negative integer. Mathematically invalid operations such as 0^0 will give set the *<floating point>* to `\c_undefined_fp`. The assignment is local.

<code>\fp_gpow:Nn</code>
<code>\fp_gpow:cn</code>

`\fp_gpow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*, which should be a positive real number or a negative integer. Mathematically invalid operations such as 0^0 will give set the *<floating point>* to `\c_undefined_fp`. The assignment is global.

92.9 Exponential and logarithm functions

<code>\fp_exp:Nn</code>
<code>\fp_exp:cn</code>

`\fp_exp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is local.

<code>\fp_gexp:Nn</code>
<code>\fp_gexp:cn</code>

`\fp_gexp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is global.

<code>\fp_ln:Nn</code>
<code>\fp_ln:cn</code>

`\fp_ln:Nn <floating point> {<value>}`

Calculates the natural logarithm of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is local.

<code>\fp_gln:Nn</code>
<code>\fp_gln:cn</code>

`\fp_gln:Nn <floating point> {<value>}`

Calculates the natural logarithm of the $\langle value \rangle$ and assigns this to the $\langle floating point \rangle$. The assignment is global.

92.10 Trigonometric functions

The trigonometric functions all work in radians. They accept a maximum input value of 100 000 000, as there are issues with range reduction and very large input values.

<code>\fp_sin:Nn</code>
<code>\fp_sin:cn</code>

`\fp_sin:Nn <floating point> {<value>}`

Assigns the sine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is local.

<code>\fp_gsin:Nn</code>
<code>\fp_gsin:cn</code>

`\fp_gsin:Nn <floating point> {<value>}`

Assigns the sine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is global.

<code>\fp_cos:Nn</code>
<code>\fp_cos:cn</code>

`\fp_cos:Nn <floating point> {<value>}`

Assigns the cosine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is local.

<code>\fp_gcos:Nn</code>
<code>\fp_gcos:cn</code>

`\fp_gcos:Nn <floating point> {<value>}`

Assigns the cosine of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is global.

<code>\fp_tan:Nn</code>
<code>\fp_tan:cn</code>

`\fp_tan:Nn <floating point> {<value>}`

Assigns the tangent of the $\langle value \rangle$ to the $\langle floating point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is local.

<code>\fp_gtan:Nn</code>
<code>\fp_gtan:cn</code>

`\fp_gtan:Nn` $\langle floating\ point \rangle$ $\{ \langle value \rangle \}$

Assigns the tangent of the $\langle value \rangle$ to the $\langle floating\ point \rangle$. The $\langle value \rangle$ should be given in radians. The assignment is global.

92.11 Notes on the floating point unit

As calculation of the elemental transcendental functions is computationally expensive compared to storage of results, after calculating a trigonometric function, exponent, *etc.* the module stored the result for reuse. Thus the performance of the module for repeated operations, most probably trigonometric functions, should be much higher than if the values were re-calculated every time they were needed.

Anyone with experience of programming floating point calculations will know that this is a complex area. The aim of the unit is to be accurate enough for the likely applications in a typesetting context. The arithmetic operations are therefore intended to provide ten digit accuracy with the last digit accurate to ± 1 . The elemental transcendental functions may not provide such high accuracy in every case, although the design aim has been to provide 10 digit accuracy for cases likely to be relevant in typesetting situations. A good overview of the challenges in this area can be found in J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd edition, Birkhäuser Boston, New York, USA, 2006.

The internal representation of numbers is tuned to the needs of the underlying \TeX system. This means that the format is somewhat different from that used in, for example, computer floating point units. Programming in \TeX makes it most convenient to use a radix 10 system, using \TeX count registers for storage and taking advantage where possible of delimited arguments.

Part XXIII

The \l3luatex package

Lua \TeX -specific functions

93 Breaking out to Lua

The Lua \TeX 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 pdf \TeX or X \TeX these will raise an error: use `\engine_if_luatex:T` to avoid this. Details of coding the Lua \TeX engine are detailed in the Lua \TeX manual.

<code>\lua_now:n *</code>
<code>\lua_now:x *</code>

`\lua_now:n {⟨token list⟩}`

The $\langle token\ list \rangle$ is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting $\langle Lua\ input \rangle$ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ immediately, and in an expandable manner.

T_EXhackers note: `\lua_now:x` is the LuaT_EX primitive `\directlua` renamed.

<code>\lua_shipout:n</code>
<code>\lua_shipout:x</code>

`\lua_shipout:x {⟨token list⟩}`

The $\langle token\ list \rangle$ is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting $\langle Lua\ input \rangle$ 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 $\langle Lua\ input \rangle$ during the page-building routine: no T_EX expansion of the $\langle Lua\ input \rangle$ will occur at this stage.

T_EXhackers note: At a T_EX level, the $\langle Lua\ input \rangle$ is stored as a ‘whatsit’.

<code>\lua_shipout_x:n</code>
<code>\lua_shipout_x:x</code>

`\lua_shipout:n {⟨token list⟩}`

The $\langle token\ list \rangle$ is first tokenized by T_EX, which will include converting line ends to spaces in the usual T_EX manner and which respects currently-applicable T_EX category codes. The resulting $\langle Lua\ input \rangle$ 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 $\langle Lua\ input \rangle$ during the page-building routine: the $\langle Lua\ input \rangle$ is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).

T_EXhackers note: `\lua_sjhipout_x:n` is the LuaT_EX primitive `\latelua` named using the L^AT_EX3 scheme.

At a T_EX level, the $\langle Lua\ input \rangle$ is stored as a ‘whatsit’.

94 Category code tables

As well as providing methods to break out into Lua, there are places where additional L^AT_EX3 functions are provided by the LuaT_EX engine. In particular, LuaT_EX provides

category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by `\ExplSyntaxOn` and `ExplSyntaxOff` when using the `LuaTeX` engine.

`\cctab_new:N` `\cctab_new:N` *<category code table>*

Creates a new category code table, initially with the codes as used by `IniTeX`.

`\cctab_gset:Nn` `\cctab_gset:Nn` *<category code table>*
{<category code set up>}

Sets the *<category code table>* to apply the category codes which apply when the prevailing regime is modified by the *<category code set up>*. Thus within a standard code block the starting point will be the code applied by `\c_code_cctab`. The assignment of the table is global: the underlying primitive does not respect grouping.

`\cctab_begin:N` `\cctab_begin:N` *<category code table>*

Switches the category codes in force to those stored in the *<category code table>*. The prevailing codes before the function is called are added to a stack, for use with `\cctab_end:`.

`\cctab_end:` `\cctab_end:`

Ends the scope of a *<category code table>* started using `\cctab_begin:N`, retuning the codes to those in force before the matching `\cctab_begin:N` was used.

`\c_code_cctab` Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOn`.

`\c_document_cctab` Category code table for a standard `LaTeX` document. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOff`.

`\c_initex_cctab` Category code table as set up by `IniTeX`.

`\c_other_cctab` Category code table where all characters have category code 12 (other).

`\c_string_cctab` Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIV

Implementation

95 l3names implementation

This is the base part of $\text{\LaTeX}3$ defining things like `catcodes` and redefining the \TeX primitives, as well as setting up the code to load `expl3` modules in $\text{\LaTeX}2_{\epsilon}$.

95.1 Internal functions

<code>\ExplSyntaxStatus</code>
<code>\ExplSyntaxPopStack</code>
<code>\ExplSyntaxStack</code>

Functions used to track the state of the catcode regime.

<code>\@pushfilename</code>
<code>\@popfilename</code>

Re-definitions of \LaTeX 's file-loading functions to support `\ExplSyntax`.

95.2 Bootstrap code

The very first thing to do is to bootstrap the $\text{Init}\text{\TeX}$ system so that everything else will actually work. \TeX does not start with some pretty basic character codes set up.

```
1 <!*package>
2 \catcode '\{ = 1 \relax
3 \catcode '\} = 2 \relax
4 \catcode '\# = 6 \relax
5 \catcode '\^ = 7 \relax
6 <!/package>
```

Tab characters should not show up in the code, but to be on the safe side.

```
7 <!*package>
8 \catcode '\^^I = 10 \relax
9 <!/package>
```

For $\text{Lua}\text{\TeX}$ the extra primitives need to be enabled before they can be use. No `\ifdefined` yet, so do it the old-fashioned way. The primitive `\strcmp` is simulated using some Lua code, which currently has to be applied to every job as the Lua code is not part of the format. Thanks to Taco Hoekwater for this code. The odd `\csname` business is needed so that the later deletion code will work.

```

10 <!*package>
11 \begingroup\expandafter\expandafter\expandafter\endgroup
12 \expandafter\ifx\csname directlua\endcsname\relax
13 \else
14 \directlua
15 {
16     tex.enableprimitives('',tex.extraprimitives ())
17     lua.bytecode[1] = function ()
18         function strcmp (A, B)
19             if A == B then
20                 tex.write("0")
21             elseif A < B then
22                 tex.write("-1")
23             else
24                 tex.write("1")
25             end
26         end
27     end
28     lua.bytecode[1]()
29 }
30 \everyjob\expandafter
31 {\csname tex_directlua:D\endcsname{lua.bytecode[1]()}}
32 \long\edef\pdfstrcmp#1#2%
33 {%
34     \expandafter\noexpand\csname tex_directlua:D\endcsname
35     {%
36         strcmp(%
37             "\noexpand\luaescapestring{#1}",%
38             "\noexpand\luaescapestring{#2}"%
39         )%
40     }%
41 }
42 \fi
43 </!package>

```

When loaded as a package this can all be handed off to other L^AT_EX 2_ε code.

```

44 <!*package>
45 \def\@tempa{%
46     \def\@tempa{%
47         \RequirePackage{luatex}%
48         \RequirePackage{pdftexcmds}%
49         \let\pdfstrcmp\pdf@strcmp
50     }
51 \begingroup\expandafter\expandafter\expandafter\endgroup
52 \expandafter\ifx\csname directlua\endcsname\relax
53 \else
54     \expandafter\@tempa
55 \fi
56 </package>

```

X_qTeX calls the primitive `\strcmp`, so there needs to be a check for that too.

```

57 \begingroup\expandafter\expandafter\expandafter\endgroup
58 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
59 \let\pdfstrcmp\strcmp
60 \fi

```

95.3 Requirements

Currently, the code requires the ε -TeX primitives and functionality equivalent to `\pdfstrcmp`. Any package which provides the later will provide the former, so the test can be done only for `\pdfstrcmp`.

```

61 \begingroup\expandafter\expandafter\expandafter\endgroup
62 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
63 <*package>
64 \PackageError{l3names}{Required primitive not found: \protect\pdfstrcmp}
65 {%
66 LaTeX3 requires the e-TeX primitives and
67 \string\pdfstrcmp.\MessageBreak
68 These are available in engine versions: \MessageBreak
69 - pdfTeX 1.30 \MessageBreak
70 - XeTeX 0.9994 \MessageBreak
71 - LuaTeX 0.60 \MessageBreak
72 or later. \MessageBreak
73 \MessageBreak
74 Loading of l3names will abort!
75 }
76 </package>
77 <!*package>
78 \newlinechar'\^^J\relax
79 \errhelp{%
80 LaTeX3 requires the e-TeX primitives and
81 \string\pdfstrcmp. ^^J
82 These are available in engine versions: ^^J
83 - pdfTeX 1.30 ^^J
84 - XeTeX 0.9994 ^^J
85 - LuaTeX 0.60 ^^J
86 or later. ^^J
87 For pdfTeX and XeTeX the '-etex' command-line switch is also
88 needed.^^J
89 ^^J
90 Format building will abort!
91 }
92 </!package>
93 \expandafter\endinput
94 \fi

```

95.4 Catcode assignments

Catcodes for `begingroup`, `endgroup`, macro parameter, superscript, and `tab`, are all assigned before the start of the documented code. (See the beginning of `l3names.dtx`.)

Reason for `\endlinechar=32` is that a line ending with a backslash will be interpreted as the token `_` which seems most natural and since spaces are ignored it works as we intend elsewhere.

Before we do this we must however record the settings for the catcode regime as it was when we start changing it.

```
95 <*initex | package>
96 \protected\edef\ExplSyntaxOff{
97   \unexpanded{\ifodd \ExplSyntaxStatus\relax
98     \def\ExplSyntaxStatus{0}
99   }
100   \catcode 126=\the \catcode 126 \relax
101   \catcode 32=\the \catcode 32 \relax
102   \catcode 9=\the \catcode 9 \relax
103   \endlinechar =\the \endlinechar \relax
104   \catcode 95=\the \catcode 95 \relax
105   \catcode 58=\the \catcode 58 \relax
106   \catcode 124=\the \catcode 124 \relax
107   \catcode 38=\the \catcode 38 \relax
108   \catcode 94=\the \catcode 94 \relax
109   \catcode 34=\the \catcode 34 \relax
110   \noexpand\fi
111 }
112 \catcode126=10\relax % tilde is a space char.
113 \catcode32=9\relax   % space is ignored
114 \catcode9=9\relax    % tab also ignored
115 \endlinechar=32\relax % endline is space
116 \catcode95=11\relax  % underscore letter
117 \catcode58=11\relax  % colon letter
118 \catcode124=12\relax % vert bar, other
119 \catcode38=4\relax   % ampersand, alignment token
120 \catcode34=12\relax  % doublequote, other
121 \catcode94=7\relax   % caret, math superscript
```

95.5 Setting up primitive names

Here is the function that renames \TeX 's primitives.

Normally the old name is left untouched, but the possibility of undefining the original names is made available by `docstrip` and package options. If nothing else, this gives a way of checking what ‘old code’ a package depends on...

If the package option ‘`removeoldnames`’ is used then some trick code is run after the end of this file, to skip past the code which has been inserted by $\text{\LaTeX 2}_{\epsilon}$ to manage the file

name stack, this code would break if run once the T_EX primitives have been undefined. (What a surprise!) **The option has been temporarily disabled.**

To get things started, give a new name for `\let`.

```
122 \let \tex_let:D \let
123 </initex | package>
```

and now an internal function to possibly remove the old name: for the moment.

```
124 <*initex>
125 \long \def \name_undefine:N #1 {
126   \tex_let:D #1 \c_undefined
127 }
128 </initex>

129 <*package>
130 \DeclareOption{removeoldnames}{
131   \long\def\name_undefine:N#1{
132     \tex_let:D#1\c_undefined}}

133 \DeclareOption{keepoldnames}{
134   \long\def\name_undefine:N#1{}}

135 \ExecuteOptions{keepoldnames}

136 \ProcessOptions
137 </package>
```

The internal function to give the new name and possibly undefine the old name.

```
138 <*initex | package>
139 \long \def \name_primitive:NN #1#2 {
140   \tex_let:D #2 #1
141   \name_undefine:N #1
142 }
```

95.6 Reassignment of primitives

In the current incarnation of this package, all T_EX 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.

```
143 \name_primitive:NN \tex_space:D
144 \name_primitive:NN \tex_italiccor:D
145 \name_primitive:NN \tex_hyphen:D
```

Now all the other primitives.

146	\name_primitive:NN \let	\tex_let:D
147	\name_primitive:NN \def	\tex_def:D
148	\name_primitive:NN \edef	\tex_edef:D
149	\name_primitive:NN \gdef	\tex_gdef:D
150	\name_primitive:NN \xdef	\tex_xdef:D
151	\name_primitive:NN \chardef	\tex_chardef:D
152	\name_primitive:NN \countdef	\tex_countdef:D
153	\name_primitive:NN \dimendef	\tex_dimendef:D
154	\name_primitive:NN \skipdef	\tex_skipdef:D
155	\name_primitive:NN \muskipdef	\tex_muskipdef:D
156	\name_primitive:NN \mathchardef	\tex_mathchardef:D
157	\name_primitive:NN \toksdef	\tex_toksdef:D
158	\name_primitive:NN \futurelet	\tex_futurelet:D
159	\name_primitive:NN \advance	\tex_advance:D
160	\name_primitive:NN \divide	\tex_divide:D
161	\name_primitive:NN \multiply	\tex_multiply:D
162	\name_primitive:NN \font	\tex_font:D
163	\name_primitive:NN \fam	\tex_fam:D
164	\name_primitive:NN \global	\tex_global:D
165	\name_primitive:NN \long	\tex_long:D
166	\name_primitive:NN \outer	\tex_outer:D
167	\name_primitive:NN \setlanguage	\tex_setlanguage:D
168	\name_primitive:NN \globaldefs	\tex_globaldefs:D
169	\name_primitive:NN \afterassignment	\tex_afterassignment:D
170	\name_primitive:NN \aftergroup	\tex_aftergroup:D
171	\name_primitive:NN \expandafter	\tex_expandafter:D
172	\name_primitive:NN \noexpand	\tex_noexpand:D
173	\name_primitive:NN \begingroup	\tex_begingroup:D
174	\name_primitive:NN \endgroup	\tex_endgroup:D
175	\name_primitive:NN \halign	\tex_halign:D
176	\name_primitive:NN \valign	\tex_valign:D
177	\name_primitive:NN \cr	\tex_cr:D
178	\name_primitive:NN \crr	\tex_crr:D
179	\name_primitive:NN \noalign	\tex_noalign:D
180	\name_primitive:NN \omit	\tex_omit:D
181	\name_primitive:NN \span	\tex_span:D
182	\name_primitive:NN \tabskip	\tex_tabskip:D
183	\name_primitive:NN \everycr	\tex_everycr:D
184	\name_primitive:NN \if	\tex_if:D
185	\name_primitive:NN \ifcase	\tex_ifcase:D
186	\name_primitive:NN \ifcat	\tex_ifcat:D
187	\name_primitive:NN \ifnum	\tex_ifnum:D
188	\name_primitive:NN \ifodd	\tex_ifodd:D
189	\name_primitive:NN \ifdim	\tex_ifdim:D
190	\name_primitive:NN \ifeof	\tex_ifeof:D
191	\name_primitive:NN \ifhbox	\tex_ifhbox:D
192	\name_primitive:NN \ifvbox	\tex_ifvbox:D
193	\name_primitive:NN \ifvoid	\tex_ifvoid:D

194	\name_primitive:NN \ifx	\tex_ifx:D
195	\name_primitive:NN \iffalse	\tex_iffalse:D
196	\name_primitive:NN \iftrue	\tex_iftrue:D
197	\name_primitive:NN \ifhmode	\tex_ifhmode:D
198	\name_primitive:NN \ifmmode	\tex_ifmmode:D
199	\name_primitive:NN \ifvmode	\tex_ifvmode:D
200	\name_primitive:NN \ifinner	\tex_ifinner:D
201	\name_primitive:NN \else	\tex_else:D
202	\name_primitive:NN \fi	\tex_fi:D
203	\name_primitive:NN \or	\tex_or:D
204	\name_primitive:NN \immediate	\tex_immediate:D
205	\name_primitive:NN \closeout	\tex_closeout:D
206	\name_primitive:NN \openin	\tex_openin:D
207	\name_primitive:NN \openout	\tex_openout:D
208	\name_primitive:NN \read	\tex_read:D
209	\name_primitive:NN \write	\tex_write:D
210	\name_primitive:NN \closein	\tex_closein:D
211	\name_primitive:NN \newlinechar	\tex_newlinechar:D
212	\name_primitive:NN \input	\tex_input:D
213	\name_primitive:NN \endinput	\tex_endinput:D
214	\name_primitive:NN \inputlineno	\tex_inputlineno:D
215	\name_primitive:NN \errmessage	\tex_errmessage:D
216	\name_primitive:NN \message	\tex_message:D
217	\name_primitive:NN \show	\tex_show:D
218	\name_primitive:NN \showthe	\tex_showthe:D
219	\name_primitive:NN \showbox	\tex_showbox:D
220	\name_primitive:NN \showlists	\tex_showlists:D
221	\name_primitive:NN \errhelp	\tex_errhelp:D
222	\name_primitive:NN \errorcontextlines	\tex_errorcontextlines:D
223	\name_primitive:NN \tracingcommands	\tex_tracingcommands:D
224	\name_primitive:NN \tracinglostchars	\tex_tracinglostchars:D
225	\name_primitive:NN \tracingmacros	\tex_tracingmacros:D
226	\name_primitive:NN \tracingonline	\tex_tracingonline:D
227	\name_primitive:NN \tracingoutput	\tex_tracingoutput:D
228	\name_primitive:NN \tracingpages	\tex_tracingpages:D
229	\name_primitive:NN \tracingparagraphs	\tex_tracingparagraphs:D
230	\name_primitive:NN \tracingrestores	\tex_tracingrestores:D
231	\name_primitive:NN \tracingstats	\tex_tracingstats:D
232	\name_primitive:NN \pausing	\tex_pausing:D
233	\name_primitive:NN \showboxbreadth	\tex_showboxbreadth:D
234	\name_primitive:NN \showboxdepth	\tex_showboxdepth:D
235	\name_primitive:NN \batchmode	\tex_batchmode:D
236	\name_primitive:NN \errorstopmode	\tex_errorstopmode:D
237	\name_primitive:NN \nonstopmode	\tex_nonstopmode:D
238	\name_primitive:NN \scrollmode	\tex_scrollmode:D
239	\name_primitive:NN \end	\tex_end:D
240	\name_primitive:NN \csname	\tex_csname:D
241	\name_primitive:NN \endcsname	\tex_endcsname:D
242	\name_primitive:NN \ignorespaces	\tex_ignorespaces:D
243	\name_primitive:NN \relax	\tex_relax:D

244	\name_primitive:NN	\the	\tex_the:D
245	\name_primitive:NN	\mag	\tex_mag:D
246	\name_primitive:NN	\language	\tex_language:D
247	\name_primitive:NN	\mark	\tex_mark:D
248	\name_primitive:NN	\topmark	\tex_topmark:D
249	\name_primitive:NN	\firstmark	\tex_firstmark:D
250	\name_primitive:NN	\botmark	\tex_botmark:D
251	\name_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
252	\name_primitive:NN	\splitbotmark	\tex_splitbotmark:D
253	\name_primitive:NN	\fontname	\tex_fontname:D
254	\name_primitive:NN	\escapechar	\tex_escapechar:D
255	\name_primitive:NN	\endlinechar	\tex_endlinechar:D
256	\name_primitive:NN	\mathchoice	\tex_mathchoice:D
257	\name_primitive:NN	\delimiter	\tex_delimiter:D
258	\name_primitive:NN	\mathaccent	\tex_mathaccent:D
259	\name_primitive:NN	\mathchar	\tex_mathchar:D
260	\name_primitive:NN	\mskip	\tex_mskip:D
261	\name_primitive:NN	\radical	\tex_radical:D
262	\name_primitive:NN	\vcenter	\tex_vcenter:D
263	\name_primitive:NN	\mkern	\tex_mkern:D
264	\name_primitive:NN	\above	\tex_above:D
265	\name_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
266	\name_primitive:NN	\atop	\tex_atop:D
267	\name_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
268	\name_primitive:NN	\over	\tex_over:D
269	\name_primitive:NN	\overwithdelims	\tex_overwithdelims:D
270	\name_primitive:NN	\displaystyle	\tex_displaystyle:D
271	\name_primitive:NN	\textstyle	\tex_textstyle:D
272	\name_primitive:NN	\scriptstyle	\tex_scriptstyle:D
273	\name_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
274	\name_primitive:NN	\nonscript	\tex_nonscript:D
275	\name_primitive:NN	\eqno	\tex_eqno:D
276	\name_primitive:NN	\leqno	\tex_leqno:D
277	\name_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
278	\name_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
279	\name_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
280	\name_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
281	\name_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
282	\name_primitive:NN	\displayindent	\tex_displayindent:D
283	\name_primitive:NN	\displaywidth	\tex_displaywidth:D
284	\name_primitive:NN	\everydisplay	\tex_everydisplay:D
285	\name_primitive:NN	\predisplaysize	\tex_predisplaysize:D
286	\name_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
287	\name_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
288	\name_primitive:NN	\mathbin	\tex_mathbin:D
289	\name_primitive:NN	\mathclose	\tex_mathclose:D
290	\name_primitive:NN	\mathinner	\tex_mathinner:D
291	\name_primitive:NN	\mathop	\tex_mathop:D
292	\name_primitive:NN	\displaylimits	\tex_displaylimits:D
293	\name_primitive:NN	\limits	\tex_limits:D

294	<code>\name_primitive:NN \nolimits</code>	<code>\tex_nolimits:D</code>
295	<code>\name_primitive:NN \mathopen</code>	<code>\tex_mathopen:D</code>
296	<code>\name_primitive:NN \mathord</code>	<code>\tex_mathord:D</code>
297	<code>\name_primitive:NN \mathpunct</code>	<code>\tex_mathpunct:D</code>
298	<code>\name_primitive:NN \mathrel</code>	<code>\tex_mathrel:D</code>
299	<code>\name_primitive:NN \overline</code>	<code>\tex_overline:D</code>
300	<code>\name_primitive:NN \underline</code>	<code>\tex_underline:D</code>
301	<code>\name_primitive:NN \left</code>	<code>\tex_left:D</code>
302	<code>\name_primitive:NN \right</code>	<code>\tex_right:D</code>
303	<code>\name_primitive:NN \binoppenalty</code>	<code>\tex_binoppenalty:D</code>
304	<code>\name_primitive:NN \relpenalty</code>	<code>\tex_relpenalty:D</code>
305	<code>\name_primitive:NN \delimitershortfall</code>	<code>\tex_delimitershortfall:D</code>
306	<code>\name_primitive:NN \delimiterfactor</code>	<code>\tex_delimiterfactor:D</code>
307	<code>\name_primitive:NN \nulldelimiterspace</code>	<code>\tex_nulldelimiterspace:D</code>
308	<code>\name_primitive:NN \everymath</code>	<code>\tex_everymath:D</code>
309	<code>\name_primitive:NN \mathsurround</code>	<code>\tex_mathsurround:D</code>
310	<code>\name_primitive:NN \medmuskip</code>	<code>\tex_medmuskip:D</code>
311	<code>\name_primitive:NN \thinmuskip</code>	<code>\tex_thinmuskip:D</code>
312	<code>\name_primitive:NN \thickmuskip</code>	<code>\tex_thickmuskip:D</code>
313	<code>\name_primitive:NN \scriptspace</code>	<code>\tex_scriptspace:D</code>
314	<code>\name_primitive:NN \noboundary</code>	<code>\tex_noboundary:D</code>
315	<code>\name_primitive:NN \accent</code>	<code>\tex_accent:D</code>
316	<code>\name_primitive:NN \char</code>	<code>\tex_char:D</code>
317	<code>\name_primitive:NN \discretionary</code>	<code>\tex_discretionary:D</code>
318	<code>\name_primitive:NN \hfil</code>	<code>\tex_hfil:D</code>
319	<code>\name_primitive:NN \hfilneg</code>	<code>\tex_hfilneg:D</code>
320	<code>\name_primitive:NN \hfill</code>	<code>\tex_hfill:D</code>
321	<code>\name_primitive:NN \hskip</code>	<code>\tex_hskip:D</code>
322	<code>\name_primitive:NN \hss</code>	<code>\tex_hss:D</code>
323	<code>\name_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
324	<code>\name_primitive:NN \vfilneg</code>	<code>\tex_vfilneg:D</code>
325	<code>\name_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
326	<code>\name_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
327	<code>\name_primitive:NN \vss</code>	<code>\tex_vss:D</code>
328	<code>\name_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
329	<code>\name_primitive:NN \kern</code>	<code>\tex_kern:D</code>
330	<code>\name_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
331	<code>\name_primitive:NN \hrule</code>	<code>\tex_hrule:D</code>
332	<code>\name_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
333	<code>\name_primitive:NN \leaders</code>	<code>\tex_leaders:D</code>
334	<code>\name_primitive:NN \cleaders</code>	<code>\tex_cleaders:D</code>
335	<code>\name_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
336	<code>\name_primitive:NN \lastkern</code>	<code>\tex_lastkern:D</code>
337	<code>\name_primitive:NN \lastskip</code>	<code>\tex_lastskip:D</code>
338	<code>\name_primitive:NN \indent</code>	<code>\tex_indent:D</code>
339	<code>\name_primitive:NN \par</code>	<code>\tex_par:D</code>
340	<code>\name_primitive:NN \noindent</code>	<code>\tex_noindent:D</code>
341	<code>\name_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
342	<code>\name_primitive:NN \baselineskip</code>	<code>\tex_baselineskip:D</code>
343	<code>\name_primitive:NN \lineskip</code>	<code>\tex_lineskip:D</code>

344	\name_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
345	\name_primitive:NN	\clubpenalty	\tex_clubpenalty:D
346	\name_primitive:NN	\widowpenalty	\tex_widowpenalty:D
347	\name_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
348	\name_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
349	\name_primitive:NN	\linepenalty	\tex_linepenalty:D
350	\name_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
351	\name_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
352	\name_primitive:NN	\adjdemerits	\tex_adjdemerits:D
353	\name_primitive:NN	\hangafter	\tex_hangafter:D
354	\name_primitive:NN	\hangindent	\tex_hangindent:D
355	\name_primitive:NN	\parshape	\tex_parshape:D
356	\name_primitive:NN	\hsize	\tex_hsize:D
357	\name_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
358	\name_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
359	\name_primitive:NN	\leftskip	\tex_leftskip:D
360	\name_primitive:NN	\rightskip	\tex_rightskip:D
361	\name_primitive:NN	\looseness	\tex_looseness:D
362	\name_primitive:NN	\parskip	\tex_parskip:D
363	\name_primitive:NN	\parindent	\tex_parindent:D
364	\name_primitive:NN	\uchyph	\tex_uchyph:D
365	\name_primitive:NN	\emergencystretch	\tex_emergencystretch:D
366	\name_primitive:NN	\pretolerance	\tex_pretolerance:D
367	\name_primitive:NN	\tolerance	\tex_tolerance:D
368	\name_primitive:NN	\spaceskip	\tex_spaceskip:D
369	\name_primitive:NN	\xspaceskip	\tex_xspaceskip:D
370	\name_primitive:NN	\parfillskip	\tex_parfillskip:D
371	\name_primitive:NN	\everypar	\tex_everypar:D
372	\name_primitive:NN	\prevgraf	\tex_prevgraf:D
373	\name_primitive:NN	\spacefactor	\tex_spacefactor:D
374	\name_primitive:NN	\shipout	\tex_shipout:D
375	\name_primitive:NN	\vsize	\tex_vsize:D
376	\name_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
377	\name_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
378	\name_primitive:NN	\topskip	\tex_topskip:D
379	\name_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
380	\name_primitive:NN	\maxdepth	\tex_maxdepth:D
381	\name_primitive:NN	\output	\tex_output:D
382	\name_primitive:NN	\deadcycles	\tex_deadcycles:D
383	\name_primitive:NN	\pagedepth	\tex_pagedepth:D
384	\name_primitive:NN	\pagestretch	\tex_pagestretch:D
385	\name_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
386	\name_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
387	\name_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
388	\name_primitive:NN	\pageshrink	\tex_pageshrink:D
389	\name_primitive:NN	\pagegoal	\tex_pagegoal:D
390	\name_primitive:NN	\pagetotal	\tex_pagetotal:D
391	\name_primitive:NN	\outputpenalty	\tex_outputpenalty:D
392	\name_primitive:NN	\hoffset	\tex_hoffset:D
393	\name_primitive:NN	\voffset	\tex_voffset:D

394	\name_primitive:NN	\insert	\tex_insert:D
395	\name_primitive:NN	\holdinginserts	\tex_holdinginserts:D
396	\name_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
397	\name_primitive:NN	\insertpenalties	\tex_insertpenalties:D
398	\name_primitive:NN	\lower	\tex_lower:D
399	\name_primitive:NN	\moveleft	\tex_moveleft:D
400	\name_primitive:NN	\moveright	\tex_moveright:D
401	\name_primitive:NN	\raise	\tex_raise:D
402	\name_primitive:NN	\copy	\tex_copy:D
403	\name_primitive:NN	\lastbox	\tex_lastbox:D
404	\name_primitive:NN	\vsplit	\tex_vsplit:D
405	\name_primitive:NN	\unhbox	\tex_unhbox:D
406	\name_primitive:NN	\unhcopy	\tex_unhcopy:D
407	\name_primitive:NN	\unvbox	\tex_unvbox:D
408	\name_primitive:NN	\unvcopy	\tex_unvcopy:D
409	\name_primitive:NN	\setbox	\tex_setbox:D
410	\name_primitive:NN	\hbox	\tex_hbox:D
411	\name_primitive:NN	\vbox	\tex_vbox:D
412	\name_primitive:NN	\vtop	\tex_vtop:D
413	\name_primitive:NN	\prevdepth	\tex_prevdepth:D
414	\name_primitive:NN	\badness	\tex_badness:D
415	\name_primitive:NN	\hbadness	\tex_hbadness:D
416	\name_primitive:NN	\vbadness	\tex_vbadness:D
417	\name_primitive:NN	\hfuzz	\tex_hfuzz:D
418	\name_primitive:NN	\vfuzz	\tex_vfuzz:D
419	\name_primitive:NN	\overfullrule	\tex_overfullrule:D
420	\name_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
421	\name_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
422	\name_primitive:NN	\splittopskip	\tex_splittopskip:D
423	\name_primitive:NN	\everyhbox	\tex_everyhbox:D
424	\name_primitive:NN	\everyvbox	\tex_everyvbox:D
425	\name_primitive:NN	\nullfont	\tex_nullfont:D
426	\name_primitive:NN	\textfont	\tex_textfont:D
427	\name_primitive:NN	\scriptfont	\tex_scriptfont:D
428	\name_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
429	\name_primitive:NN	\fontdimen	\tex_fontdimen:D
430	\name_primitive:NN	\hyphenchar	\tex_hyphenchar:D
431	\name_primitive:NN	\skewchar	\tex_skewchar:D
432	\name_primitive:NN	\defaultshyphenchar	\tex_defaultshyphenchar:D
433	\name_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
434	\name_primitive:NN	\number	\tex_number:D
435	\name_primitive:NN	\romannumeral	\tex_romannumeral:D
436	\name_primitive:NN	\string	\tex_string:D
437	\name_primitive:NN	\lowercase	\tex_lowercase:D
438	\name_primitive:NN	\uppercase	\tex_uppercase:D
439	\name_primitive:NN	\meaning	\tex_meaning:D
440	\name_primitive:NN	\penalty	\tex_penalty:D
441	\name_primitive:NN	\unpenalty	\tex_unpenalty:D
442	\name_primitive:NN	\lastpenalty	\tex_lastpenalty:D
443	\name_primitive:NN	\special	\tex_special:D

444	<code>\name_primitive:NN \dump</code>	<code>\tex_dump:D</code>
445	<code>\name_primitive:NN \patterns</code>	<code>\tex_patterns:D</code>
446	<code>\name_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
447	<code>\name_primitive:NN \time</code>	<code>\tex_time:D</code>
448	<code>\name_primitive:NN \day</code>	<code>\tex_day:D</code>
449	<code>\name_primitive:NN \month</code>	<code>\tex_month:D</code>
450	<code>\name_primitive:NN \year</code>	<code>\tex_year:D</code>
451	<code>\name_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
452	<code>\name_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
453	<code>\name_primitive:NN \count</code>	<code>\tex_count:D</code>
454	<code>\name_primitive:NN \dimen</code>	<code>\tex_dimen:D</code>
455	<code>\name_primitive:NN \skip</code>	<code>\tex_skip:D</code>
456	<code>\name_primitive:NN \toks</code>	<code>\tex_toks:D</code>
457	<code>\name_primitive:NN \muskip</code>	<code>\tex_muskip:D</code>
458	<code>\name_primitive:NN \box</code>	<code>\tex_box:D</code>
459	<code>\name_primitive:NN \wd</code>	<code>\tex_wd:D</code>
460	<code>\name_primitive:NN \ht</code>	<code>\tex_ht:D</code>
461	<code>\name_primitive:NN \dp</code>	<code>\tex_dp:D</code>
462	<code>\name_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
463	<code>\name_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
464	<code>\name_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
465	<code>\name_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
466	<code>\name_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
467	<code>\name_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

Since L^AT_EX3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

468	<code>\name_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
469	<code>\name_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
470	<code>\name_primitive:NN \unless</code>	<code>\etex_unless:D</code>
471	<code>\name_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
472	<code>\name_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
473	<code>\name_primitive:NN \marks</code>	<code>\etex_marks:D</code>
474	<code>\name_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
475	<code>\name_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
476	<code>\name_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
477	<code>\name_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
478	<code>\name_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
479	<code>\name_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
480	<code>\name_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
481	<code>\name_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
482	<code>\name_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
483	<code>\name_primitive:NN \readline</code>	<code>\etex_readline:D</code>
484	<code>\name_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
485	<code>\name_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
486	<code>\name_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
487	<code>\name_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
488	<code>\name_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
489	<code>\name_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>

490	<code>\name_primitive:NN \currentifttype</code>	<code>\etex_currentifttype:D</code>
491	<code>\name_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
492	<code>\name_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
493	<code>\name_primitive:NN \currentgrouptype</code>	<code>\etex_currentgrouptype:D</code>
494	<code>\name_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
495	<code>\name_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
496	<code>\name_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
497	<code>\name_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
498	<code>\name_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
499	<code>\name_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
500	<code>\name_primitive:NN \fontchardp</code>	<code>\etex_fontchardp:D</code>
501	<code>\name_primitive:NN \fontcharwd</code>	<code>\etex_fontcharwd:D</code>
502	<code>\name_primitive:NN \fontcharic</code>	<code>\etex_fontcharic:D</code>
503	<code>\name_primitive:NN \parshapeindent</code>	<code>\etex_parshapeindent:D</code>
504	<code>\name_primitive:NN \parshapelength</code>	<code>\etex_parshapelength:D</code>
505	<code>\name_primitive:NN \parshapedimen</code>	<code>\etex_parshapedimen:D</code>
506	<code>\name_primitive:NN \numexpr</code>	<code>\etex_numexpr:D</code>
507	<code>\name_primitive:NN \dimexpr</code>	<code>\etex_dimexpr:D</code>
508	<code>\name_primitive:NN \glueexpr</code>	<code>\etex_glueexpr:D</code>
509	<code>\name_primitive:NN \muexpr</code>	<code>\etex_muexpr:D</code>
510	<code>\name_primitive:NN \gluestretch</code>	<code>\etex_gluestretch:D</code>
511	<code>\name_primitive:NN \glueshrink</code>	<code>\etex_glueshrink:D</code>
512	<code>\name_primitive:NN \gluestretchorder</code>	<code>\etex_gluestretchorder:D</code>
513	<code>\name_primitive:NN \glueshrinkorder</code>	<code>\etex_glueshrinkorder:D</code>
514	<code>\name_primitive:NN \gluetomu</code>	<code>\etex_gluetomu:D</code>
515	<code>\name_primitive:NN \mutoglu</code>	<code>\etex_mutoglu:D</code>
516	<code>\name_primitive:NN \lastlinefit</code>	<code>\etex_lastlinefit:D</code>
517	<code>\name_primitive:NN \interlinepenalties</code>	<code>\etex_interlinepenalties:D</code>
518	<code>\name_primitive:NN \clubpenalties</code>	<code>\etex_clubpenalties:D</code>
519	<code>\name_primitive:NN \widowpenalties</code>	<code>\etex_widowpenalties:D</code>
520	<code>\name_primitive:NN \displaywidowpenalties</code>	<code>\etex_displaywidowpenalties:D</code>
521	<code>\name_primitive:NN \middle</code>	<code>\etex_middle:D</code>
522	<code>\name_primitive:NN \savinghyphcodes</code>	<code>\etex_savinghyphcodes:D</code>
523	<code>\name_primitive:NN \savingvdiscards</code>	<code>\etex_savingvdiscards:D</code>
524	<code>\name_primitive:NN \pagediscards</code>	<code>\etex_pagediscards:D</code>
525	<code>\name_primitive:NN \splitdiscards</code>	<code>\etex_splitdiscards:D</code>
526	<code>\name_primitive:NN \TeXETstate</code>	<code>\etex_TeXXETstate:D</code>
527	<code>\name_primitive:NN \beginL</code>	<code>\etex_beginL:D</code>
528	<code>\name_primitive:NN \endL</code>	<code>\etex_endL:D</code>
529	<code>\name_primitive:NN \beginR</code>	<code>\etex_beginR:D</code>
530	<code>\name_primitive:NN \endR</code>	<code>\etex_endR:D</code>
531	<code>\name_primitive:NN \predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
532	<code>\name_primitive:NN \everyeof</code>	<code>\etex_everyeof:D</code>
533	<code>\name_primitive:NN \protected</code>	<code>\etex_protected:D</code>

All major distributions use pdf ϵ -T ϵ X as engine so we add these names as well. Since the pdfT ϵ X team has been very good at prefixing most primitives with pdf (so far only five do not start with pdf) we do not give them a double pdf prefix. The list below covers pdfT ϵ Xv 1.30.4.

```

534 %% integer registers:
535 \name_primitive:NN \pdfoutput \pdf_output:D
536 \name_primitive:NN \pdfminorversion \pdf_minorversion:D
537 \name_primitive:NN \pdfcompresslevel \pdf_compresslevel:D
538 \name_primitive:NN \pdfdecimaldigits \pdf_decimaldigits:D
539 \name_primitive:NN \pdfimageresolution \pdf_imageresolution:D
540 \name_primitive:NN \pdfpkresolution \pdf_pkresolution:D
541 \name_primitive:NN \pdftracingfonts \pdf_tracingfonts:D
542 \name_primitive:NN \pdfuniqueresname \pdf_uniqueresname:D
543 \name_primitive:NN \pdfadjustspacing \pdf_adjustspacing:D
544 \name_primitive:NN \pdfprotrudechars \pdf_protrudechars:D
545 \name_primitive:NN \efcode \pdf_efcode:D
546 \name_primitive:NN \lpcode \pdf_lpcode:D
547 \name_primitive:NN \rpcode \pdf_rpcode:D
548 \name_primitive:NN \pdfforcepagebox \pdf_forcepagebox:D
549 \name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
550 \name_primitive:NN \pdfinclusionerrorlevel \pdf_inclusionerrorlevel:D
551 \name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
552 \name_primitive:NN \pdfimagehicolor \pdf_imagehicolor:D
553 \name_primitive:NN \pdfimageapplygamma \pdf_imageapplygamma:D
554 \name_primitive:NN \pdfgamma \pdf_gamma:D
555 \name_primitive:NN \pdfimagegamma \pdf_imagegamma:D
556 %% dimen registers:
557 \name_primitive:NN \pdfhorigin \pdf_horigin:D
558 \name_primitive:NN \pdfvorigin \pdf_vorigin:D
559 \name_primitive:NN \pdfpagewidth \pdf_pagewidth:D
560 \name_primitive:NN \pdfpageheight \pdf_pageheight:D
561 \name_primitive:NN \pdflinkmargin \pdf_linkmargin:D
562 \name_primitive:NN \pdfdestmargin \pdf_destmargin:D
563 \name_primitive:NN \pdfthreadmargin \pdf_threadmargin:D
564 %% token registers:
565 \name_primitive:NN \pdfpagesattr \pdf_pagesattr:D
566 \name_primitive:NN \pdfpageattr \pdf_pageattr:D
567 \name_primitive:NN \pdfpageresources \pdf_pageresources:D
568 \name_primitive:NN \pdfpkmode \pdf_pkmode:D
569 %% expandable commands:
570 \name_primitive:NN \pdftexrevision \pdf_texrevision:D
571 \name_primitive:NN \pdftexbanner \pdf_texbanner:D
572 \name_primitive:NN \pdfcreationdate \pdf_creationdate:D
573 \name_primitive:NN \pdfpageref \pdf_pageref:D
574 \name_primitive:NN \pdfxformname \pdf_xformname:D
575 \name_primitive:NN \pdffontname \pdf_fontname:D
576 \name_primitive:NN \pdffontobjnum \pdf_fontobjnum:D
577 \name_primitive:NN \pdffontsize \pdf_fontsize:D
578 \name_primitive:NN \pdfincludechars \pdf_includechars:D
579 \name_primitive:NN \leftmarginkern \pdf_leftmarginkern:D
580 \name_primitive:NN \rightmarginkern \pdf_rightmarginkern:D
581 \name_primitive:NN \pdfescapestring \pdf_escapestring:D
582 \name_primitive:NN \pdfescapename \pdf_escapename:D
583 \name_primitive:NN \pdfescapehex \pdf_escapehex:D

```

```

584 \name_primitive:NN \pdfunescapehex \pdf_unescapehex:D
585 \name_primitive:NN \pdfstrcmp \pdf_strcmp:D
586 \name_primitive:NN \pdfuniformdeviate \pdf_uniformdeviate:D
587 \name_primitive:NN \pdfnormaldeviate \pdf_normaldeviate:D
588 \name_primitive:NN \pdfmdfivesum \pdf_mdfivesum:D
589 \name_primitive:NN \pdffilemoddate \pdf_filemoddate:D
590 \name_primitive:NN \pdffilesize \pdf_filesize:D
591 \name_primitive:NN \pdffiledump \pdf_filedump:D
592 %% read-only integers:
593 \name_primitive:NN \pdftexversion \pdf_texversion:D
594 \name_primitive:NN \pdflastobj \pdf_lastobj:D
595 \name_primitive:NN \pdflastxform \pdf_lastxform:D
596 \name_primitive:NN \pdflastximage \pdf_lastximage:D
597 \name_primitive:NN \pdflastximagepages \pdf_lastximagepages:D
598 \name_primitive:NN \pdflastannot \pdf_lastannot:D
599 \name_primitive:NN \pdflastxpos \pdf_lastxpos:D
600 \name_primitive:NN \pdflastypos \pdf_lastypos:D
601 \name_primitive:NN \pdflastdemerits \pdf_lastdemerits:D
602 \name_primitive:NN \pdfelapsedtime \pdf_elapsedtime:D
603 \name_primitive:NN \pdfrandomseed \pdf_randomseed:D
604 \name_primitive:NN \pdfshellescape \pdf_shellescape:D
605 %% general commands:
606 \name_primitive:NN \pdfobj \pdf_obj:D
607 \name_primitive:NN \pdfrefobj \pdf_refobj:D
608 \name_primitive:NN \pdfxform \pdf_xform:D
609 \name_primitive:NN \pdfrefxform \pdf_refxform:D
610 \name_primitive:NN \pdfximage \pdf_ximage:D
611 \name_primitive:NN \pdfrefximage \pdf_refximage:D
612 \name_primitive:NN \pdfannot \pdf_annot:D
613 \name_primitive:NN \pdfstartlink \pdf_startlink:D
614 \name_primitive:NN \pdfendlink \pdf_endlink:D
615 \name_primitive:NN \pdfoutline \pdf_outline:D
616 \name_primitive:NN \pdfdest \pdf_dest:D
617 \name_primitive:NN \pdfthread \pdf_thread:D
618 \name_primitive:NN \pdfstartthread \pdf_startthread:D
619 \name_primitive:NN \pdfendthread \pdf_endthread:D
620 \name_primitive:NN \pdfsavepos \pdf_savepos:D
621 \name_primitive:NN \pdfinfo \pdf_info:D
622 \name_primitive:NN \pdfcatalog \pdf_catalog:D
623 \name_primitive:NN \pdfnames \pdf_names:D
624 \name_primitive:NN \pdfmapfile \pdf_mapfile:D
625 \name_primitive:NN \pdfmapline \pdf_mapline:D
626 \name_primitive:NN \pdffontattr \pdf_fontattr:D
627 \name_primitive:NN \pdftrailer \pdf_trailer:D
628 \name_primitive:NN \pdffontexpand \pdf_fontexpand:D
629 %%\name_primitive:NN \vadjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
630 \name_primitive:NN \pdfliteral \pdf_literal:D
631 %%\name_primitive:NN \special <pdfspecial spec>
632 \name_primitive:NN \pdfresettimer \pdf_resettimer:D
633 \name_primitive:NN \pdfsetrandomseed \pdf_setrandomseed:D

```

```
634 \name_primitive:NN \pdfnoligatures \pdf_noligatures:D
```

Only a little bit of XeTeX and LuaTeX at the moment.

```
635 \name_primitive:NN \XeTeXversion \xetex_version:D
636 \name_primitive:NN \catcodetable \luatex_catcodetable:D
637 \name_primitive:NN \directlua \luatex_directlua:D
638 \name_primitive:NN \initcatcodetable \luatex_initcatcodetable:D
639 \name_primitive:NN \latelua \luatex_latelua:D
640 \name_primitive:NN \savecatcodetable \luatex_savecatcodetable:D
```

XeTeX adds `\stricmp` to the set of primitives, with the same implementation as `\pdfstricmp` but a different name. To avoid having to worry about this later, the same internal name is used. Note that we want to use the original primitive name here so that `\name_undefine:N` (if used) will remove it.

```
641 \tex_begingroup:D
642 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
643 \tex_endgroup:D
644 \tex_expandafter:D \tex_ifx:D \tex_csname:D xetex_version:D \tex_endcsname:D
645 \tex_relax:D \tex_else:D
646 \name_primitive:NN \stricmp \pdf_stricmp:D
647 \tex_fi:D
```

95.7 expl3 code switches

`\ExplSyntaxOn`
`\ExplSyntaxOff`
`\ExplSyntaxStatus`

Here we define functions that are used to turn on and off the special conventions used in the kernel of L^AT_EX3.

First of all, the space, tab and the return characters will all be ignored inside L^AT_EX3 code, the latter because `endline` is set to a space instead. When space characters are needed in L^AT_EX3 code the `~` character will be used for that purpose.

Specification of the desired behavior:

- `ExplSyntax` can be either On or Off.
- The On switch is `<null>` if `ExplSyntax` is on.
- The Off switch is `<null>` if `ExplSyntax` is off.
- If the On switch is issued and not `<null>`, it records the current catcode scheme just prior to it being issued.
- An Off switch restores the catcode scheme to what it was just prior to the previous On switch.

```
648 \etex_protected:D \tex_def:D \ExplSyntaxOn {
649 \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
650 \tex_else:D
```

```

651 \etex_protected:D \tex_edef:D \ExplSyntaxOff {
652   \etex_unexpanded:D{
653     \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
654     \tex_def:D \ExplSyntaxStatus{0}
655   }
656   \tex_catcode:D 126=\tex_the:D \tex_catcode:D 126 \tex_relax:D
657   \tex_catcode:D 32=\tex_the:D \tex_catcode:D 32 \tex_relax:D
658   \tex_catcode:D 9=\tex_the:D \tex_catcode:D 9 \tex_relax:D
659   \tex_endlinechar:D =\tex_the:D \tex_endlinechar:D \tex_relax:D
660   \tex_catcode:D 95=\tex_the:D \tex_catcode:D 95 \tex_relax:D
661   \tex_catcode:D 58=\tex_the:D \tex_catcode:D 58 \tex_relax:D
662   \tex_catcode:D 124=\tex_the:D \tex_catcode:D 124 \tex_relax:D
663   \tex_catcode:D 38=\tex_the:D \tex_catcode:D 38 \tex_relax:D
664   \tex_catcode:D 94=\tex_the:D \tex_catcode:D 94 \tex_relax:D
665   \tex_catcode:D 34=\tex_the:D \tex_catcode:D 34 \tex_relax:D
666   \tex_noexpand:D \tex_fi:D
667 }
668 \tex_def:D \ExplSyntaxStatus { 1 }
669 \tex_catcode:D 126=10 \tex_relax:D % tilde is a space char.
670 \tex_catcode:D 32=9 \tex_relax:D % space is ignored
671 \tex_catcode:D 9=9 \tex_relax:D % tab also ignored
672 \tex_endlinechar:D =32 \tex_relax:D % endline is space
673 \tex_catcode:D 95=11 \tex_relax:D % underscore letter
674 \tex_catcode:D 58=11 \tex_relax:D % colon letter
675 \tex_catcode:D 124=12 \tex_relax:D % vertical bar, other
676 \tex_catcode:D 38=4 \tex_relax:D % ampersand, alignment token
677 \tex_catcode:D 94=7 \tex_relax:D % caret, math superscript
678 \tex_catcode:D 34=12 \tex_relax:D % doublequote, other
679 \tex_fi:D
680 }

```

At this point we better set the status.

```

681 \tex_def:D \ExplSyntaxStatus { 1 }

```

`\ExplSyntaxNamesOn` Sometimes we need to be able to use names from the kernel of L^AT_EX3 without adhering it's conventions according to space characters. These macros provide the necessary settings.

```

682 \etex_protected:D \tex_def:D \ExplSyntaxNamesOn {
683   \tex_catcode:D '\_ =11 \tex_relax:D
684   \tex_catcode:D '\: =11 \tex_relax:D
685 }
686 \etex_protected:D \tex_def:D \ExplSyntaxNamesOff {
687   \tex_catcode:D '\_ =8 \tex_relax:D
688   \tex_catcode:D '\: =12 \tex_relax:D
689 }

```

95.8 Package loading

`\GetIdInfo` Extract all information from a cvs or svn field. The formats are slightly different but at least the information is in the same positions so we check in the date format so see if it contains a / after the four-digit year. If it does it is cvs else svn and we extract information. To be on the safe side we ensure that spaces in the argument are seen.

```

\filedescription
\filename
\fileversion
\fileauthor
\filedate
\filenameext
\filetimestamp
\GetIdInfoAuxI:w
\GetIdInfoAuxII:w
\GetIdInfoAuxCVS:w
\GetIdInfoAuxSVN:w

690 \etex_protected:D \tex_def:D \GetIdInfo {
691   \tex_begingroup:D
692   \tex_catcode:D 32=10 \tex_relax:D % needed? Probably for now.
693   \GetIdInfoMaybeMissing:w
694 }

695 \etex_protected:D \tex_def:D\GetIdInfoMaybeMissing:w$#1$#2{
696   \tex_def:D \l_kernel_tmpa_tl {#1}
697   \tex_def:D \l_kernel_tmpb_tl {Id}
698   \tex_ifx:D \l_kernel_tmpa_tl \l_kernel_tmpb_tl
699   \tex_def:D \l_kernel_tmpa_tl {
700     \tex_endgroup:D
701     \tex_def:D\filedescription{#2}
702     \tex_def:D\filename      {[unknown~name]}
703     \tex_def:D\fileversion   {000}
704     \tex_def:D\fileauthor    {[unknown~author]}
705     \tex_def:D\filedate      {0000/00/00}
706     \tex_def:D\filenameext   {[unknown~ext]}
707     \tex_def:D\filetimestamp {[unknown~timestamp]}
708   }
709   \tex_else:D
710     \tex_def:D \l_kernel_tmpa_tl {\GetIdInfoAuxI:w$#1$#2}
711   \tex_fi:D
712   \l_kernel_tmpa_tl
713 }

714 \etex_protected:D \tex_def:D\GetIdInfoAuxI:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
715   \tex_endgroup:D
716   \tex_def:D\filename{#2}
717   \tex_def:D\fileversion{#4}
718   \tex_def:D\filedescription{#9}
719   \tex_def:D\fileauthor{#7}
720   \GetIdInfoAuxII:w #5\tex_relax:D
721   #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
722 }

723 \etex_protected:D \tex_def:D\GetIdInfoAuxII:w #1#2#3#4#5#6\tex_relax:D{
724   \tex_ifx:D#5/
725     \tex_expandafter:D\GetIdInfoAuxCVS:w
726   \tex_else:D
727     \tex_expandafter:D\GetIdInfoAuxSVN:w
728   \tex_fi:D
729 }
```

```

730 \etex_protected:D \tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
731                               #2\tex_relax:D#3\tex_relax:D{
732   \tex_def:D\filedate{#2}
733   \tex_def:D\filenameext{#1}
734   \tex_def:D\filetimestamp{#3}

```

When creating the format we want the information in the log straight away.

```

735 <initex>\tex_immediate:D\tex_write:D-1
736 <initex> {\filename;~ v\fileversion,~\filedate;~\filedescription}
737 }
738 \etex_protected:D \tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2-#3-#4
739                               \tex_relax:D#5Z\tex_relax:D{
740   \tex_def:D\filenameext{#1}
741   \tex_def:D\filedate{#2/#3/#4}
742   \tex_def:D\filetimestamp{#5}
743 <-package>\tex_immediate:D\tex_write:D-1
744 <-package> {\filename;~ v\fileversion,~\filedate;~\filedescription}
745 }
746 </initex | package>

```

Finally some corrections in the case we are running over L^AT_EX 2_ε.

We want to set things up so that experimental packages and regular packages can coexist with the former using the L^AT_EX3 programming catcode settings. Since it cannot be the task of the end user to know how a package is constructed under the hood we make it so that the experimental packages have to identify themselves. As an example it can be done as

```

\RequirePackage{l3names}
\ProvidesExplPackage{agent}{2007/08/28}{007}{bonding module}

```

or by using the `\file{field}` informations from `\GetIdInfo` as the packages in this distribution do like this:

```

\RequirePackage{l3names}
\GetIdInfo$Id: l3names.dtx 2122 2011-01-08 09:14:28Z joseph $
    {L3 Experimental Box module}
\ProvidesExplPackage
    {\filename}{\filedate}{\fileversion}{\filedescription}

```

```

\ProvidesExplPackage
\ProvidesExplClass
\ProvidesExplFile

```

First up is the identification. Rather trivial as we don't allow for options just yet.

```

747 <*package>
748 \etex_protected:D \tex_def:D \ProvidesExplPackage#1#2#3#4{
749   \ProvidesPackage{#1}[#2~v#3~#4]
750   \ExplSyntaxOn

```

```

751 }
752 \etex_protected:D \tex_def:D \ProvidesExplClass#1#2#3#4{
753   \ProvidesClass{#1}[#2~v#3~#4]
754   \ExplSyntaxOn
755 }
756 \etex_protected:D \tex_def:D \ProvidesExplFile#1#2#3#4{
757   \ProvidesFile{#1}[#2~v#3~#4]
758   \ExplSyntaxOn
759 }

```

`\@pushfilename` The idea behind the code is to record whether or not the L^AT_EX3 syntax is on or off when
`\@popfilename` about to load a file with class or package extension. This status stored in the parameter `\ExplSyntaxStatus` and set by `\ExplSyntaxOn` and `\ExplSyntaxOff` to 1 and 0 respectively is pushed onto the stack `\ExplSyntaxStack`. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again. The whole thing is a bit problematical. So let's take a look at what the desired behavior is: A package or class which declares itself of Expl type by using `\ProvidesExplClass` or `\ProvidesExplPackage` should automatically ensure the correct catcode scheme as soon as the identification part is over. Similarly, a package or class which uses the traditional `\ProvidesClass` or `\ProvidesPackage` commands should go back to the traditional catcode scheme. An example:

```

\RequirePackage{l3names}
\ProvidesExplPackage{foobar}{2009/05/07}{0.1}{Foobar package}
\cs_new:Npn \foo_bar:nn #1#2 {#1,#2}
...
\RequirePackage{array}
...
\cs_new:Npn \foo_bar:nnn #1#2#3 {#3,#2,#1}

```

Inside the `array` package, everything should behave as normal under traditional L^AT_EX but as soon as we are back at the top level, we should use the new catcode regime.

Whenever L^AT_EX inputs a package file or similar, it calls upon `\@pushfilename` to push the name, the extension and the catcode of `@` of the file it was currently processing onto a file name stack. Similarly, after inputting such a file, this file name stack is popped again and the catcode of `@` is set to what it was before. If it is a package within package, `@` maintains catcode 11 whereas if it is package within document preamble `@` is reset to what it was in the preamble (which is usually catcode 12). We wish to adopt a similar technique. Every time an Expl package or class is declared, they will issue an `\ExplSyntaxOn`. Then whenever we are about to load another file, we will first push this status onto a stack and then turn it off again. Then when done loading a file, we pop the stack and if `\ExplSyntax` was On right before, so should it be now. The only problem with this is that we cannot guarantee that we get to the file name stack very early on. Therefore, if the `\ExplSyntaxStack` is empty when trying to pop it, we ensure to turn `\ExplSyntax` off again.

`\@pushfilename` is prepended with a small function pushing the current `ExplSyntaxStatus` (true/false) onto a stack. Then the current catcode regime is recorded and `ExplSyntax` is switched off.

`\@popfilename` is appended with a function for popping the `ExplSyntax` stack. However, chances are we didn't get to hook into the file stack early enough so `LATEX` might try to pop the file name stack while the `ExplSyntaxStack` is empty. If the latter is empty, we just switch off `ExplSyntax`.

```

760 \tex_edef:D \@pushfilename{
761   \etex_unexpanded:D{
762     \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus \ExplSyntaxStack }
763     \ExplSyntaxOff
764   }
765   \etex_unexpanded:D\tex_expandafter:D{\@pushfilename }
766 }
767 \tex_edef:D \@popfilename{
768   \etex_unexpanded:D\tex_expandafter:D{\@popfilename
769     \tex_if:D 2\ExplSyntaxStack 2
770     \ExplSyntaxOff
771     \tex_else:D
772     \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\q_stop
773     \tex_fi:D
774   }
775 }

```

`\ExplSyntaxPopStack` Popping the stack is simple: Take the first token which is either 0 (false) or 1 (true) and
`\ExplSyntaxStack` test if it is odd. Save the rest. The stack is initially empty set to 0 signalling that before `l3names` was loaded, the `ExplSyntax` was off.

```

776 \etex_protected:D\tex_def:D\ExplSyntaxPopStack#1#2\q_stop{
777   \tex_def:D\ExplSyntaxStack{#2}
778   \tex_ifodd:D#1\tex_relax:D
779     \ExplSyntaxOn
780   \tex_else:D
781     \ExplSyntaxOff
782   \tex_fi:D
783 }
784 \tex_def:D \ExplSyntaxStack{0}

```

95.9 Finishing up

A few of the ‘primitives’ assigned above have already been stolen by `LATEX`, so assign them by hand to the saved real primitive.

```

785 \tex_let:D\tex_input:D      \@@input
786 \tex_let:D\tex_underline:D  \@@underline
787 \tex_let:D\tex_end:D        \@@end

```

```

788 \tex_let:D\tex_everymath:D      \frozen@everymath
789 \tex_let:D\tex_everydisplay:D   \frozen@everydisplay
790 \tex_let:D\tex_italiccorr:D     @@italiccorr
791 \tex_let:D\tex_hyphen:D         @@hyph
792 \tex_let:D\luatex_catcodetable:D \luatexcatcodetable
793 \tex_let:D\luatex_initcatcodetable:D \luatexinitcatcodetable
794 \tex_let:D\luatex_latelua:D      \luatexlatelua
795 \tex_let:D\luatex_savecatcodetable:D \luatexsavecatcodetable

```

\TeX has a nasty habit of inserting a command with the name `\par` so we had better make sure that that command at least has a definition.

```

796 \tex_let:D\par          \tex_par:D

```

This is the end for `l3names` when used on top of $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X} 2_{\varepsilon}$:

```

797 \tex_ifx:D\name_undefine:N\@gobble
798 \tex_def:D\name_pop_stack:w{}
799 \tex_else:D

```

But if traditional \TeX code is disabled, do this...

As mentioned above, The $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X} 2_{\varepsilon}$ package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the \TeX primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```

800 \tex_def:D\ProvidesPackage{
801   \tex_begingroup:D
802   \ExplSyntaxOff
803   \package_provides:w}

804 \tex_def:D\package_provides:w#1#2[#3]{
805   \tex_endgroup:D
806   \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
807   \tex_expandafter:D\tex_xdef:D
808   \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}

```

In this case the catcode preserving stack is not maintained and `\ExplSyntaxOn` conventions stay in force once on. You'll need to turn then off explicitly with `\ExplSyntaxOff` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```

809 \tex_def:D\name_pop_stack:w#1\relax{%
810   \ExplSyntaxOff
811   \tex_expandafter:D\p@pfilename\@currnamestack\@nil
812   \tex_let:D\default@ds\@unknownoptionerror
813   \tex_global:D\tex_let:D\ds@\@empty
814   \tex_global:D\tex_let:D\@declaredoptions\@empty}

```

```

815 \tex_def:D\@p@pfilename#1#2#3#4\@nil{%
816   \tex_gdef:D\@currname{#1}%
817   \tex_gdef:D\@current{#2}%
818   \tex_catcode:D'\@#3%
819   \tex_gdef:D\@currnamestack{#4}}

820 \tex_def:D\NeedsTeXFormat#1{}
821 \tex_def:D\RequirePackage#1{
822   \tex_expandafter:D\tex_ifx:D
823     \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
824     \ExplSyntaxOn
825     \tex_input:D#1.sty\tex_relax:D
826   \tex_fi:D}
827 \tex_fi:D

```

The `\futurelet` just forces the special end of file marker to vanish, so the argument of `\name_pop_stack:w` does not cause an end-of-file error. (Normally I use `\expandafter` for this trick, but here the next token is in fact `\let` and that may be undefined.)

```

828 \tex_futurelet:D\name_tmp:\name_pop_stack:w

```

expl3 dependency checks We want the `expl3` bundle to be loaded ‘as one’; this command is used to ensure that one of the 13 packages isn’t loaded on its own.

```

829 <*\initex>
830 \etex_protected:D\tex_def:D \package_check_loaded_expl: {
831   \@ifpackageloaded{expl3}{\{
832     \PackageError{expl3}{Cannot~load~the~expl3~modules~separately}{
833       The~expl3~modules~cannot~be~loaded~separately;\MessageBreak
834       please~\protect\usepackage{expl3}~instead.
835     }
836   }
837 }
838 <\/!\initex>

839 <\/package>

```

95.10 Showing memory usage

This section is from some old code from 1993; it’d be good to work out how it should be used in our code today.

During the development of the L^AT_EX3 kernel we need to be able to keep track of the memory usage. Therefore we generate empty pages while loading the kernel code, just to be able to check the memory usage.

```

840 <*\showmemory>
841 \g_trace_statistics_status=2\scan_stop:

```

```

842 \cs_set_nopar:Npn\showMemUsage{
843   \if_horizontal_mode:
844     \tex_errmessage:D{Wrong~ mode~ H:~ something~ triggered~
845       hmode~ above}
846   \else:
847     \tex_message:D{Mode ~ okay}
848   \fi:
849   \tex_shipout:D\hbox:w{ }
850 }
851 \showMemUsage
852 \showmemory

```

96 l3basics implementation

We need l3names to get things going but we actually need it very early on, so it is loaded at the very top of the file l3basics.dtx. Also, most of the code below won't run until l3expan has been loaded.

96.1 Renaming some T_EX primitives (again)

`\cs_set_eq:NwN` 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 do a few now, just to get started.⁷

```

853 \*package
854 \ProvidesExplPackage
855   {\filename}{\filedate}{\fileversion}{\filedescription}
856 \package_check_loaded_expl:
857 \*package
858 \*initex | package
859 \tex_let:D \cs_set_eq:NwN          \tex_let:D

```

<code>\if_true:</code>	Then some conditionals.		
<code>\if_false:</code>			
<code>\or:</code>	860 <code>\cs_set_eq:NwN</code>	<code>\if_true:</code>	<code>\tex_iftrue:D</code>
<code>\else:</code>	861 <code>\cs_set_eq:NwN</code>	<code>\if_false:</code>	<code>\tex_iffalse:D</code>
<code>\fi:</code>	862 <code>\cs_set_eq:NwN</code>	<code>\or:</code>	<code>\tex_or:D</code>
<code>\reverse_if:N</code>	863 <code>\cs_set_eq:NwN</code>	<code>\else:</code>	<code>\tex_else:D</code>
<code>\if:w</code>	864 <code>\cs_set_eq:NwN</code>	<code>\fi:</code>	<code>\tex_fi:D</code>
<code>\if_bool:N</code>	865 <code>\cs_set_eq:NwN</code>	<code>\reverse_if:N</code>	<code>\etex_unless:D</code>
<code>\if_predicate:w</code>	866 <code>\cs_set_eq:NwN</code>	<code>\if:w</code>	<code>\tex_if:D</code>
<code>\if_charcode:w</code>	867 <code>\cs_set_eq:NwN</code>	<code>\if_bool:N</code>	<code>\tex_ifodd:D</code>
<code>\if_catcode:w</code>	868 <code>\cs_set_eq:NwN</code>	<code>\if_predicate:w</code>	<code>\tex_ifodd:D</code>
	869 <code>\cs_set_eq:NwN</code>	<code>\if_charcode:w</code>	<code>\tex_if:D</code>
	870 <code>\cs_set_eq:NwN</code>	<code>\if_catcode:w</code>	<code>\tex_ifcat:D</code>

⁷This 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.

<code>\if_meaning:w</code>			
	871	<code>\cs_set_eq:NwN</code>	<code>\if_meaning:w</code> <code>\tex_ifx:D</code>
<code>\if_mode_math:</code>		T _E X lets us detect some if its modes.	
<code>\if_mode_horizontal:</code>			
<code>\if_mode_vertical:</code>	872	<code>\cs_set_eq:NwN</code>	<code>\if_mode_math:</code> <code>\tex_ifmmode:D</code>
<code>\if_mode_inner:</code>	873	<code>\cs_set_eq:NwN</code>	<code>\if_mode_horizontal:</code> <code>\tex_ifhmode:D</code>
	874	<code>\cs_set_eq:NwN</code>	<code>\if_mode_vertical:</code> <code>\tex_ifvmode:D</code>
	875	<code>\cs_set_eq:NwN</code>	<code>\if_mode_inner:</code> <code>\tex_ifinner:D</code>
<code>\if_cs_exist:N</code>			
<code>\if_cs_exist:w</code>	876	<code>\cs_set_eq:NwN</code>	<code>\if_cs_exist:N</code> <code>\etex_ifdefined:D</code>
	877	<code>\cs_set_eq:NwN</code>	<code>\if_cs_exist:w</code> <code>\etex_ifcsname:D</code>
<code>\exp_after:wN</code>		The three <code>\exp_</code> functions are used in the <code>l3expan</code> module where they are described.	
<code>\exp_not:N</code>			
<code>\exp_not:n</code>	878	<code>\cs_set_eq:NwN</code>	<code>\exp_after:wN</code> <code>\tex_expandafter:D</code>
	879	<code>\cs_set_eq:NwN</code>	<code>\exp_not:N</code> <code>\tex_noexpand:D</code>
	880	<code>\cs_set_eq:NwN</code>	<code>\exp_not:n</code> <code>\etex_unexpanded:D</code>
<code>\iow_shipout_x:Nn</code>			
<code>\token_to_meaning:N</code>	881	<code>\cs_set_eq:NwN</code>	<code>\iow_shipout_x:Nn</code> <code>\tex_write:D</code>
<code>\token_to_str:N</code>	882	<code>\cs_set_eq:NwN</code>	<code>\token_to_meaning:N</code> <code>\tex_meaning:D</code>
<code>\token_to_str:c</code>	883	<code>\cs_set_eq:NwN</code>	<code>\token_to_str:N</code> <code>\tex_string:D</code>
<code>\cs:w</code>	884	<code>\cs_set_eq:NwN</code>	<code>\cs:w</code> <code>\tex_csname:D</code>
<code>\cs_end:</code>	885	<code>\cs_set_eq:NwN</code>	<code>\cs_end:</code> <code>\tex_endcsname:D</code>
<code>\cs_meaning:N</code>	886	<code>\cs_set_eq:NwN</code>	<code>\cs_meaning:N</code> <code>\tex_meaning:D</code>
<code>\cs_meaning:c</code>	887	<code>\tex_def:D</code>	<code>\cs_meaning:c</code> <code>{\exp_args:Nc\cs_meaning:N}</code>
<code>\cs_show:N</code>	888	<code>\cs_set_eq:NwN</code>	<code>\cs_show:N</code> <code>\tex_show:D</code>
<code>\cs_show:c</code>	889	<code>\tex_def:D</code>	<code>\cs_show:c</code> <code>{\exp_args:Nc\cs_show:N}</code>
	890	<code>\tex_def:D</code>	<code>\token_to_str:c</code> <code>{\exp_args:Nc\token_to_str:N}</code>
<code>\scan_stop:</code>		The next three are basic functions for which there also exist versions that are safe inside	
<code>\group_begin:</code>		alignments. These safe versions are defined in the <code>l3prg</code> module.	
<code>\group_end:</code>	891	<code>\cs_set_eq:NwN</code>	<code>\scan_stop:</code> <code>\tex_relax:D</code>
	892	<code>\cs_set_eq:NwN</code>	<code>\group_begin:</code> <code>\tex_begingroup:D</code>
	893	<code>\cs_set_eq:NwN</code>	<code>\group_end:</code> <code>\tex_endgroup:D</code>
<code>\group_execute_after:N</code>			
	894	<code>\cs_set_eq:NwN</code>	<code>\group_execute_after:N</code> <code>\tex_aftergroup:D</code>
<code>\pref_global:D</code>			
<code>\pref_long:D</code>			
<code>\pref_protected:D</code>	895	<code>\cs_set_eq:NwN</code>	<code>\pref_global:D</code> <code>\tex_global:D</code>
	896	<code>\cs_set_eq:NwN</code>	<code>\pref_long:D</code> <code>\tex_long:D</code>
	897	<code>\cs_set_eq:NwN</code>	<code>\pref_protected:D</code> <code>\etex_protected:D</code>

96.2 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

All assignment functions in L^AT_EX3 should be naturally robust; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren't.

```

\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
898 \cs_set_eq:NwN \cs_set_nopar:Npn \tex_def:D
899 \cs_set_eq:NwN \cs_set_nopar:Npx \tex_edef:D
900 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npn {
901   \pref_long:D \cs_set_nopar:Npn
902 }
903 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npx {
904   \pref_long:D \cs_set_nopar:Npx
905 }
906 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn {
907   \pref_protected:D \cs_set_nopar:Npn
908 }
909 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx {
910   \pref_protected:D \cs_set_nopar:Npx
911 }
912 \cs_set_protected_nopar:Npn \cs_set_protected:Npn {
913   \pref_protected:D \pref_long:D \cs_set_nopar:Npn
914 }
915 \cs_set_protected_nopar:Npn \cs_set_protected:Npx {
916   \pref_protected:D \pref_long:D \cs_set_nopar:Npx
917 }

```

Global versions of the above functions.

```

\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
918 \cs_set_eq:NwN \cs_gset_nopar:Npn \tex_gdef:D
919 \cs_set_eq:NwN \cs_gset_nopar:Npx \tex_xdef:D
920 \cs_set_protected_nopar:Npn \cs_gset:Npn {
921   \pref_long:D \cs_gset_nopar:Npn
922 }
923 \cs_set_protected_nopar:Npn \cs_gset:Npx {
924   \pref_long:D \cs_gset_nopar:Npx
925 }
926 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn {
927   \pref_protected:D \cs_gset_nopar:Npn
928 }
929 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx {
930   \pref_protected:D \cs_gset_nopar:Npx
931 }
932 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn {
933   \pref_protected:D \pref_long:D \cs_gset_nopar:Npn
934 }
935 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx {
936   \pref_protected:D \pref_long:D \cs_gset_nopar:Npx
937 }

```

96.3 Selecting tokens

`\use:c` This macro grabs its argument and returns a csname from it.

```
938 \cs_set:Npn \use:c #1 { \cs:w#1\cs_end: }
```

`\use:x` Fully expands its argument and passes it to the input stream. Uses `\cs_tmp:` as a scratch register but does not affect it.

```
939 \cs_set_protected:Npn \use:x #1 {
940   \group_begin:
941     \cs_set:Npx \cs_tmp: {#1}
942     \exp_after:wN
943   \group_end:
944   \cs_tmp:
945 }
```

`\use:n` These macro grabs its arguments and returns it back to the input (with outer braces removed). `\use:n` is defined earlier for bootstrapping.

```
\use:nnn
\use:nnnn
946 \cs_set:Npn \use:n #1 {#1}
947 \cs_set:Npn \use:nn #1#2 {#1#2}
948 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
949 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

`\use_i:nn` These macros are needed to provide functions with true and false cases, as introduced by Michael some time ago. By using `\exp_after:wN \use_i:nn \else:` constructions it is possible to write code where the true or false case is able to access the following tokens from the input stream, which is not possible if the `\c_true_bool` syntax is used.

```
950 \cs_set:Npn \use_i:nn #1#2 {#1}
951 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

`\use_i:nnnn` We also need something for picking up arguments from a longer list.

```
\use_ii:nnn
\use_iii:nnn
\use_i:nnnn
\use_ii:nnnn
\use_iii:nnnn
\use_iv:nnnn
\use_i_ii:nnn
952 \cs_set:Npn \use_i:nnn #1#2#3{#1}
953 \cs_set:Npn \use_ii:nnn #1#2#3{#2}
954 \cs_set:Npn \use_iii:nnn #1#2#3{#3}
955 \cs_set:Npn \use_i:nnnn #1#2#3#4{#1}
956 \cs_set:Npn \use_ii:nnnn #1#2#3#4{#2}
957 \cs_set:Npn \use_iii:nnnn #1#2#3#4{#3}
958 \cs_set:Npn \use_iv:nnnn #1#2#3#4{#4}
959 \cs_set:Npn \use_i_ii:nnn #1#2#3{#1#2}
```

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil` or `\q_stop` resp.

```
\use_none_delimit_by_q_stop:w
\use_none_delimit_by_q_recursion_stop:w
960 \cs_set:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
961 \cs_set:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
962 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop {}
```

`\use_i_delimit_by_q_nil: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.

`\use_i_delimit_by_q_stop:nw`

`\delimit_by_q_recursion_stop:nw`

```

963 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
964 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
965 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}

```

`\use_i_after_fi:nw` Returns the first argument after ending the conditional.

`\use_i_after_else:nw`

`\use_i_after_or:nw`

`\use_i_after_orelse:nw`

```

966 \cs_set:Npn \use_i_after_fi:nw #1\fi:{\fi: #1}
967 \cs_set:Npn \use_i_after_else:nw #1\else:#2\fi:{\fi: #1}
968 \cs_set:Npn \use_i_after_or:nw #1\or: #2\fi: {\fi:#1}
969 \cs_set:Npn \use_i_after_orelse:nw #1 #2#3\fi: {\fi:#1}

```

96.4 Gobbling tokens from input

`\use_none:n` 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 defining `\use_none:nnn` and above as separate calls of `\use_none:n` and `\use_none:nn` is slightly faster, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

`\use_none:nn`

`\use_none:nnn`

`\use_none:nnnn`

`\use_none:nnnnn`

`\use_none:nnnnnn`

`\use_none:nnnnnnn`

`\use_none:nnnnnnnn`

`\use_none:nnnnnnnnn`

```

970 \cs_set:Npn \use_none:n #1{}
971 \cs_set:Npn \use_none:nn #1#2{}
972 \cs_set:Npn \use_none:nnn #1#2#3{}
973 \cs_set:Npn \use_none:nnnn #1#2#3#4{}
974 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5{}
975 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6{}
976 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7{}
977 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8{}
978 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9{}

```

96.5 Expansion control from l3expan

`\exp_args:Nc` Moved here for now as it is going to be used right away.

```

979 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}

```

96.6 Conditional processing and definitions

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 T_EX 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 T_EX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` These break statements put T_EX in a *⟨true⟩* or *⟨false⟩* state. The idea is that the expansion of `\tex_romannumeral:D \c_zero` is *⟨null⟩* so we set off a `\tex_romannumeral:D`. It will on its way expand any `\else:` or `\fi:` that are waiting to be discarded anyway before finally arriving at the `\c_zero` we will place right after the conditional. After this expansion has terminated, we issue either `\if_true:` or `\if_false:` to put T_EX in the correct state.

```

980 \cs_set:Npn \prg_return_true: { \exp_after:wN\if_true:\tex_romannumeral:D }
981 \cs_set:Npn \prg_return_false: { \exp_after:wN\if_false:\tex_romannumeral:D }

```

An extended state space could instead utilize `\tex_ifcase:D`:

```

\cs_set:Npn \prg_return_true: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_zero \tex_romannumeral:D
}
\cs_set:Npn \prg_return_false: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_one \tex_romannumeral:D
}
\cs_set:Npn \prg_return_error: {
  \exp_after:wN\tex_ifcase:D \exp_after:wN \c_two \tex_romannumeral:D
}

```

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. Call aux function to grab parameters, split the base function into name and signature and then use, e.g., `\cs_set:Npn` to define it with.

```

\prg_new_conditional:Npnn
\set_protected_conditional:Npnn
\new_protected_conditional:Npnn
982 \cs_set_protected:Npn \prg_set_conditional:Npnn #1{
983   \prg_get_parm_aux:nw{
984     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
985     \cs_set:Npn {parm}
986   }
987 }
988 \cs_set_protected:Npn \prg_new_conditional:Npnn #1{
989   \prg_get_parm_aux:nw{
990     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
991     \cs_new:Npn {parm}
992   }

```

```

993 }
994 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn #1{
995   \prg_get_parm_aux:nw{
996     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
997     \cs_set_protected:Npn {parm}
998   }
999 }
1000 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn #1{
1001   \prg_get_parm_aux:nw{
1002     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1003     \cs_new_protected:Npn {parm}
1004   }
1005 }

```

```

\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn

```

The user functions for the types automatically inserting the correct parameter text based on the signature. Call aux function after calculating number of arguments, split the base function into name and signature and then use, e.g., `\cs_set:Npn` to define it with.

```

1006 \cs_set_protected:Npn \prg_set_conditional:Nnn #1{
1007   \exp_args:Nnf \prg_get_count_aux:nn{
1008     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1009     \cs_set:Npn {count}
1010   }{\cs_get_arg_count_from_signature:N #1}
1011 }
1012 \cs_set_protected:Npn \prg_new_conditional:Nnn #1{
1013   \exp_args:Nnf \prg_get_count_aux:nn{
1014     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1015     \cs_new:Npn {count}
1016   }{\cs_get_arg_count_from_signature:N #1}
1017 }
1018
1019 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn #1{
1020   \exp_args:Nnf \prg_get_count_aux:nn{
1021     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1022     \cs_set_protected:Npn {count}
1023   }{\cs_get_arg_count_from_signature:N #1}
1024 }
1025
1026 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn #1{
1027   \exp_args:Nnf \prg_get_count_aux:nn{
1028     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1029     \cs_new_protected:Npn {count}
1030   }{\cs_get_arg_count_from_signature:N #1}
1031 }

```

```

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

```

The obvious setting-equal functions.

```

1032 \cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3 {
1033   \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3}

```

```

1034 }
1035 \cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3 {
1036   \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3}
1037 }

```

\prg_get_parm_aux:nw For the Npnn type we must grab the parameter text before continuing. We make this a very generic function that takes one argument before reading everything up to a left brace. Something similar for the Nnn type.

```

1038 \cs_set:Npn \prg_get_count_aux:nn #1#2 {#1{#2}}
1039 \cs_set:Npn \prg_get_parm_aux:nw #1#2#{#1{#2}}

```

e_conditional_parm_aux:nnNNnnnn generate_conditional_parm_aux:nw 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. For the time being, we do not use this piece of information but could well throw an error. The fourth argument is how to define this function, the fifth is the text **parm** or **count** for which version to use to define the functions, the sixth is the parameters to use (possibly empty) or number of arguments, the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms.

```

1040 \cs_set:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8{
1041   \prg_generate_conditional_aux:nnw{#5}{
1042     #4{#1}{#2}{#6}{#8}
1043   }#7,?, \q_recursion_stop
1044 }

```

Looping through the list of desired forms. First is the text **parm** or **count**, second is five arguments packed together and third is the form. Use text and form to call the correct type.

```

1045 \cs_set:Npn \prg_generate_conditional_aux:nnw #1#2#3,{
1046   \if:w ?#3
1047     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1048   \fi:
1049   \use:c{prg_generate_#3_form_#1:Nnnnn} #2
1050   \prg_generate_conditional_aux:nnw{#1}{#2}
1051 }

```

\prg_generate_p_form_parm:Nnnnn \prg_generate_TF_form_parm:Nnnnn \prg_generate_T_form_parm:Nnnnn \prg_generate_F_form_parm:Nnnnn How to generate the various forms. The **parm** types here takes the following arguments: 1: how to define (an N-type), 2: name, 3: signature, 4: parameter text (or empty), 5: replacement.

```

1052 \cs_set:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5{
1053   \exp_args:Nc #1 {#2_p:#3}#4{#5 \c_zero
1054     \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
1055   }
1056 }

```

```

1057 \cs_set:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5{
1058   \exp_args:Nc#1 {#2:#3TF}#4{#5 \c_zero
1059     \exp_after:wN \use_i:nn \else: \exp_after:wN \use_ii:nn \fi:
1060   }
1061 }
1062 \cs_set:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5{
1063   \exp_args:Nc#1 {#2:#3T}#4{#5 \c_zero
1064     \else:\exp_after:wN\use_none:nn\fi:\use:n
1065   }
1066 }
1067 \cs_set:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5{
1068   \exp_args:Nc#1 {#2:#3F}#4{#5 \c_zero
1069     \exp_after:wN\use_none:nn\fi:\use:n
1070   }
1071 }

```

```

prg_generate_p_form_count:Nnnnn
prg_generate_TF_form_count:Nnnnn
prg_generate_T_form_count:Nnnnn
prg_generate_F_form_count:Nnnnn

```

How to generate the various forms. The count types here use a number to insert the correct parameter text, otherwise like the parm functions above.

```

1072 \cs_set:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5{
1073   \cs_generate_from_arg_count:cNnn {#2_p:#3} #1 {#4}{#5 \c_zero
1074     \exp_after:wN\c_true_bool\else:\exp_after:wN\c_false_bool\fi:
1075   }
1076 }
1077 \cs_set:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5{
1078   \cs_generate_from_arg_count:cNnn {#2:#3TF} #1 {#4}{#5 \c_zero
1079     \exp_after:wN\use_i:nn\else:\exp_after:wN\use_ii:nn\fi:
1080   }
1081 }
1082 \cs_set:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5{
1083   \cs_generate_from_arg_count:cNnn {#2:#3T} #1 {#4}{#5 \c_zero
1084     \else:\exp_after:wN\use_none:nn\fi:\use:n
1085   }
1086 }
1087 \cs_set:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5{
1088   \cs_generate_from_arg_count:cNnn {#2:#3F} #1 {#4}{#5 \c_zero
1089     \exp_after:wN\use_none:nn\fi:\use:n
1090   }
1091 }

```

```

prg_set_eq_conditional_aux:NNNn
prg_set_eq_conditional_aux:NNNw

```

```

1092 \cs_set:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4 {
1093   \prg_set_eq_conditional_aux:NNNw #1#2#3#4,?,\q_recursion_stop
1094 }

```

Manual clist loop over argument #4.

```

1095 \cs_set:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4, {
1096   \if:w ? #4 \scan_stop:

```

```

1097 \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1098 \fi:
1099 #1 {
1100 \exp_args:Nnc \cs_split_function:NN #2 {prg_conditional_form_#4:nnn}
1101 }{
1102 \exp_args:Nnc \cs_split_function:NN #3 {prg_conditional_form_#4:nnn}
1103 }
1104 \prg_set_eq_conditional_aux:NNW #1{#2}{#3}
1105 }

1106 \cs_set:Npn \prg_conditional_form_p:nnn #1#2#3 {#1_p:#2}
1107 \cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 {#1:#2TF}
1108 \cs_set:Npn \prg_conditional_form_T:nnn #1#2#3 {#1:#2T}
1109 \cs_set:Npn \prg_conditional_form_F:nnn #1#2#3 {#1:#2F}

```

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 Here are the canonical boolean values.
\c_false_bool
1110 \tex_chardef:D \c_true_bool = 1~
1111 \tex_chardef:D \c_false_bool = 0~

```

96.7 Dissecting a control sequence

```

\cs_to_str:N This converts a control sequence into the character string of its name, removing the
\cs_to_str_aux:w leading escape character. This turns out to be a non-trivial matter as there are different
cases:

```

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of `\tex_escapechar:D` is negative;
- when the escape character is a space.

The route chosen is this: If `\token_to_str:N \a` produces a non-space escape char, then this will produce two tokens. If the escape char is non-printable, only one token is produced. If the escape char is a space, then a space token plus one token character token is produced. If we augment the result of this expansion with the letters `ax` we get the following three scenarios (with $\langle X \rangle$ being a printable non-space escape character):

- $\langle X \rangle \text{aax}$

- aax
- aax

In the second and third case, putting an auxiliary function in front reading unlimited arguments will treat them the same, removing the space token for us automatically. Therefore, if we test the second and third argument of what such a function reads, in case 1 we will get true and in cases 2 and 3 we will get false. If we choose to optimize for the usual case of a printable escape char, we can do it like this (again getting TeX to remove the leading space for us):

```

1112 \cs_set_nopar:Npn \cs_to_str:N {
1113   \if:w \exp_after:wN \cs_str_aux:w\token_to_str:N \a ax\q_stop
1114   \else:
1115     \exp_after:wN \exp_after:wN\exp_after:wN \use_ii:nn
1116   \fi:
1117   \exp_after:wN \use_none:n \token_to_str:N
1118 }
1119 \cs_set:Npn \cs_str_aux:w #1#2#3#4\q_stop{#2#3}

```

`\cs_split_function:NN` 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, `\cs_split_function:NN\foo_bar:cnx\use_i:nnn` as input becomes `\use_i:nnn {foo_bar}{cnx}\c_true_bool`.

Can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

```

1120 \group_begin:
1121   \tex_lccode:D '\@ = '\: \scan_stop:
1122   \tex_catcode:D '\@ = 12~
1123   \tex_lowercase:D {
1124     \group_end:

```

First ensure that we actually get a properly evaluated str as we don't know how many expansions `\cs_to_str:N` requires. Insert extra colon to catch the error cases.

```

1125 \cs_set:Npn \cs_split_function:NN #1#2{
1126   \exp_after:wN \cs_split_function_aux:w
1127   \tex_romannumeral:D -'\q \cs_to_str:N #1 @a \q_stop #2
1128 }

```

If no colon in the name, #2 is a with catcode 11 and #3 is empty. If colon in the name, then either #2 is a colon or the first letter of the signature. The letters here have catcode 12. If a colon was given we need to a) split off the colon and quark at the end and b) ensure

we return the name, signature and boolean true We can't use `\quark_if_no_value:NTF` yet but this is very safe anyway as all tokens have catcode 12.

```

1129 \cs_set:Npn \cs_split_function_aux:w #1@#2#3\q_stop#4{
1130   \if_meaning:w a#2
1131     \exp_after:wN \use_i:nn
1132   \else:
1133     \exp_after:wN\use_ii:nn
1134   \fi:
1135   {#4{#1}{}\c_false_bool}
1136   {\cs_split_function_auxii:w#2#3\q_stop #4{#1}}
1137 }
1138 \cs_set:Npn \cs_split_function_auxii:w #1@a\q_stop#2#3{
1139   #2{#3}{#1}\c_true_bool
1140 }

```

End of lowercase

```

1141 }

```

`\cs_get_function_name:N` Now returning the name is trivial: just discard the last two arguments. Similar for
`\cs_get_function_signature:N` signature.

```

1142 \cs_set:Npn \cs_get_function_name:N #1 {
1143   \cs_split_function:NN #1\use_i:nnn
1144 }
1145 \cs_set:Npn \cs_get_function_signature:N #1 {
1146   \cs_split_function:NN #1\use_ii:nnn
1147 }

```

96.8 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 `\tex_relax:D` token. A control sequence is said to be *free* (to be defined) if it does not already exist and also meets the requirement that it does not contain a D signature. The reasoning behind this is that most of the time, a check for a free control sequence is when we wish to make a new control sequence and we do not want to let the user define a new “do not use” control sequence.

`\cs_if_exist_p:N` Two versions for checking existence. For the N form we firstly check for `\tex_relax:D`
`\cs_if_exist_p:c` and then if it is in the hash table. There is no problem when inputting something like
`\cs_if_exist:NTF` `\else:` or `\fi:` as TeX will only ever skip input in case the token tested against is
`\cs_if_exist:cTF` `\tex_relax:D`.

```

1148 \prg_set_conditional:Npnn \cs_if_exist:N #1 {p,TF,T,F}{
1149   \if_meaning:w #1\tex_relax:D
1150   \prg_return_false:
1151   \else:

```

```

1152     \if_cs_exist:N #1
1153     \prg_return_true:
1154   \else:
1155     \prg_return_false:
1156   \fi:
1157 \fi:
1158 }

```

For the `c` form we firstly check if it is in the hash table and then for `\tex_relax:D` 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.

```

1159 \prg_set_conditional:Npnn \cs_if_exist:c #1 {p,TF,T,F}{
1160   \if_cs_exist:w #1 \cs_end:
1161     \exp_after:wN \use_i:nn
1162   \else:
1163     \exp_after:wN \use_ii:nn
1164   \fi:
1165   {
1166     \exp_after:wN \if_meaning:w \cs:w #1\cs_end: \tex_relax:D
1167     \prg_return_false:
1168   \else:
1169     \prg_return_true:
1170   \fi:
1171   }
1172 \prg_return_false:
1173 }

```

```

\cs_if_do_not_use_p:N
\cs_if_do_not_use_aux:nnN

```

```

1174 \cs_set:Npn \cs_if_do_not_use_p:N #1{
1175   \cs_split_function:NN #1 \cs_if_do_not_use_aux:nnN
1176 }
1177 \cs_set:Npn \cs_if_do_not_use_aux:nnN #1#2#3{
1178   \str_if_eq_p:nn { D } {#2}
1179 }

```

`\cs_if_free_p:N` The simple implementation is one using the boolean expression parser: If it is exists or is do not use, then return false.

```

\cs_if_free_p:c
\cs_if_free:N $\overline{TF}$ 
\cs_if_free:c $\overline{TF}$ 
\prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
  \bool_if:nTF {\cs_if_exist_p:N #1 || \cs_if_do_not_use_p:N #1}
  {\prg_return_false:}{\prg_return_true:}
}

```

However, this functionality may not be available this early on. We do something similar: The numerical values of true and false is one and zero respectively, which we can use.

The problem again here is that the token we are checking may in fact be something that can disturb the scanner, so we have to be careful. We would like to do minimal evaluation so we ensure this.

```

1180 \prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
1181   \tex_ifnum:D \cs_if_exist_p:N #1 =\c_zero
1182   \exp_after:wN \use_i:nn
1183   \else:
1184   \exp_after:wN \use_ii:nn
1185   \fi:
1186   {
1187     \tex_ifnum:D \cs_if_do_not_use_p:N #1 =\c_zero
1188     \prg_return_true:
1189   \else:
1190     \prg_return_false:
1191   \fi:
1192   }
1193   \prg_return_false:
1194 }
1195 \cs_set_nopar:Npn \cs_if_free_p:c{\exp_args:Nc\cs_if_free_p:N}
1196 \cs_set_nopar:Npn \cs_if_free:cTF{\exp_args:Nc\cs_if_free:NTF}
1197 \cs_set_nopar:Npn \cs_if_free:cT{\exp_args:Nc\cs_if_free:NT}
1198 \cs_set_nopar:Npn \cs_if_free:cF{\exp_args:Nc\cs_if_free:NF}

```

96.9 Defining and checking (new) functions

`\c_minus_one` We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\c_zero` which is used by some functions in the `l3alloc` module.
`\c_zero` log
`\c_sixteen` 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! The actual allocation mechanism is in `l3alloc` and as \TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```

1199 <!*initex>
1200 \cs_set_eq:NwN \c_minus_one\m@ne
1201 </!initex>
1202 <!*package>
1203 \tex_countdef:D \c_minus_one = 10 ~
1204 \c_minus_one = -1 ~
1205 </!package>
1206 \tex_chardef:D \c_sixteen = 16~
1207 \tex_chardef:D \c_zero = 0~

```

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 definitions here are only temporary, they will be redefined later on.

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal.

```
1208 \cs_set_protected_nopar:Npn \iow_log:x {
1209   \tex_immediate:D \iow_shipout_x:Nn \c_minus_one
1210 }
1211 \cs_set_protected_nopar:Npn \iow_term:x {
1212   \tex_immediate:D \iow_shipout_x:Nn \c_sixteen
1213 }
```

`\msg_kernel_bug:x` This will show internal errors.

```
1214 \cs_set_protected_nopar:Npn \msg_kernel_bug:x #1 {
1215   \iow_term:x { This~is~a~LaTeX~bug:~check~coding! }
1216   \tex_errmessage:D {#1}
1217 }
```

`\cs_record_meaning:N` This macro will be used later on for tracing purposes. But we need some more modules to define it, so we just give some dummy definition here.

```
1218 <*trace>
1219 \cs_set:Npn \cs_record_meaning:N #1{}
1220 </trace>
```

`\chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:MN` etc. to make sure
`\chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks if `<cname>` 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!

```
1221 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1{
1222   \cs_if_free:NF #1
1223   {
1224     \msg_kernel_bug:x {Command~name~'\token_to_str:N #1'~already~defined!~
1225                       Current~meaning: \ \ \c_space_tl \c_space_tl \token_to_meaning:N #1
1226                       }
1227   }
1228   <*trace>
1229   \cs_record_meaning:N#1
1230   % \iow_term:x{Defining~\token_to_str:N #1~on~%}
1231   \iow_log:x{Defining~\token_to_str:N #1~on~
1232             line~\tex_the:D \tex_inputlineno:D}
1233   </trace>
1234 }
1235 \cs_set_protected_nopar:Npn \chk_if_free_cs:c {
```

```

1236 \exp_args:Nc \chk_if_free_cs:N
1237 }
1238 <*package>
1239 \tex_ifodd:D \@l@expl@log@functions@bool \else
1240 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1 {
1241   \cs_if_free:NF #1
1242   {
1243     \msg_kernel_bug:x
1244     {
1245       Command~name~'\token_to_str:N #1'~already~defined!~
1246       Current~meaning: \ \c_space_tl \c_space_tl \token_to_meaning:N #1
1247     }
1248   }
1249 }
1250 \fi
1251 </package>

```

`\chk_if_exist_cs:N` This function issues a warning message when the control sequence in its argument does not exist.

```

1252 \cs_set_protected_nopar:Npn \chk_if_exist_cs:N #1 {
1253   \cs_if_exist:NF #1
1254   {
1255     \msg_kernel_bug:x {Command~ '\token_to_str:N #1'~
1256                       not~ yet~ defined!}
1257   }
1258 }
1259 \cs_set_protected_nopar:Npn \chk_if_exist_cs:c {
1260   \exp_args:Nc \chk_if_exist_cs:N
1261 }

```

`\str_if_eq_p:nn` 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.

`\str_if_eq_p:xx`

`\str_if_eq:xxTF`

```

1262 \prg_set_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF } {
1263   \tex_ifnum:D \pdf_strcmp:D
1264   { \etex_unexpanded:D {#1} } { \etex_unexpanded:D {#2} }
1265   = \c_zero
1266   \prg_return_true: \else: \prg_return_false: \fi:
1267 }
1268 \prg_set_conditional:Npnn \str_if_eq:xx #1#2 { p , T , F , TF } {
1269   \tex_ifnum:D \pdf_strcmp:D {#1} {#2} = \c_zero
1270   \prg_return_true: \else: \prg_return_false: \fi:
1271 }

```

`\cs_if_eq_name_p:NN` An application of the above function, already streamlined for speed, so I put it in here.

```

1272 \prg_set_conditional:Npnn \cs_if_eq_name:NN #1#2{p}{
1273   \str_if_eq_p:nn {#1} {#2}
1274 }

```

96.10 More new definitions

```

\cs_new_nopar:Npn Global versions of the above functions.
\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
1275 \cs_set:Npn \cs_tmp:w #1#2 {
1276   \cs_set_protected_nopar:Npn #1 ##1
1277   {
1278     \chk_if_free_cs:N ##1
1279     #2 ##1
1280   }
1281 }
1282 \cs_tmp:w \cs_new_nopar:Npn \cs_gset_nopar:Npn
1283 \cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
1284 \cs_tmp:w \cs_new:Npn \cs_gset:Npn
1285 \cs_tmp:w \cs_new:Npx \cs_gset:Npx
1286 \cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1287 \cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1288 \cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
1289 \cs_tmp:w \cs_new_protected:Npx \cs_gset_protected:Npx

```

`\cs_set_nopar:cpn` 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` $\langle string \rangle \langle rep-text \rangle$ will turn $\langle string \rangle$ into a `csname` and then assign $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1290 \cs_set:Npn \cs_tmp:w #1#2{
1291   \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 }
1292 }
1293 \cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1294 \cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1295 \cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1296 \cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1297 \cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1298 \cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments.

`\cs_set:cpx` We may also do this globally.

```

\cs_gset:cpn
\cs_gset:cpx
\cs_new:cpn
\cs_new:cpx
1299 \cs_tmp:w \cs_set:cpn \cs_set:Npn
1300 \cs_tmp:w \cs_set:cpx \cs_set:Npx
1301 \cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1302 \cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1303 \cs_tmp:w \cs_new:cpn \cs_new:Npn
1304 \cs_tmp:w \cs_new:cpx \cs_new:Npx

```

`\cs_set_protected_nopar:cpn` 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_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
1305 \cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1306 \cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1307 \cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1308 \cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1309 \cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1310 \cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

\cs_set_protected:cpn
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpx
\cs_new_protected:cpn
\cs_new_protected:cpx
1311 \cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1312 \cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1313 \cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1314 \cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1315 \cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1316 \cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

BACKWARDS COMPATIBILITY:

```

1317 \cs_set_eq:NwN \cs_gnew_nopar:Npn \cs_new_nopar:Npn
1318 \cs_set_eq:NwN \cs_gnew:Npn \cs_new:Npn
1319 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npn \cs_new_protected_nopar:Npn
1320 \cs_set_eq:NwN \cs_gnew_protected:Npn \cs_new_protected:Npn
1321 \cs_set_eq:NwN \cs_gnew_nopar:Npx \cs_new_nopar:Npx
1322 \cs_set_eq:NwN \cs_gnew:Npx \cs_new:Npx
1323 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npx \cs_new_protected_nopar:Npx
1324 \cs_set_eq:NwN \cs_gnew_protected:Npx \cs_new_protected:Npx
1325 \cs_set_eq:NwN \cs_gnew_nopar:cpn \cs_new_nopar:cpn
1326 \cs_set_eq:NwN \cs_gnew:cpn \cs_new:cpn
1327 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpn \cs_new_protected_nopar:cpn
1328 \cs_set_eq:NwN \cs_gnew_protected:cpn \cs_new_protected:cpn
1329 \cs_set_eq:NwN \cs_gnew_nopar:cpx \cs_new_nopar:cpx
1330 \cs_set_eq:NwN \cs_gnew:cpx \cs_new:cpx
1331 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpx \cs_new_protected_nopar:cpx
1332 \cs_set_eq:NwN \cs_gnew_protected:cpx \cs_new_protected:cpx

```

`\use_0_parameter:` For using parameters, i.e., when you need to define a function to process three parameters. See `xparse` for an application.

```

\use_0_parameter:
\use_1_parameter:
\use_2_parameter:
\use_3_parameter:
\use_4_parameter:
\use_5_parameter:
\use_6_parameter:
\use_7_parameter:
\use_8_parameter:
\use_9_parameter:
1333 \cs_set_nopar:cpn{use_0_parameter:}{}
1334 \cs_set_nopar:cpn{use_1_parameter:}{{##1}}
1335 \cs_set_nopar:cpn{use_2_parameter:}{{##1}{##2}}
1336 \cs_set_nopar:cpn{use_3_parameter:}{{##1}{##2}{##3}}
1337 \cs_set_nopar:cpn{use_4_parameter:}{{##1}{##2}{##3}{##4}}
1338 \cs_set_nopar:cpn{use_5_parameter:}{{##1}{##2}{##3}{##4}{##5}}
1339 \cs_set_nopar:cpn{use_6_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}}

```

```

1340 \cs_set_nopar:cpn{use_7_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}}
1341 \cs_set_nopar:cpn{use_8_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}{\##8}}
1342 \cs_set_nopar:cpn{use_9_parameter:}{\##1}{\##2}{\##3}{\##4}{\##5}{\##6}{\##7}{\##8}{\##9}}
1343
1344

```

96.11 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.

`\cs_set_eq:cN` The = sign allows us to define funny char tokens like = itself or `_` with this function. For the definition of `\c_space_chartok{~}` to work we need the `~` after the =.

`\cs_set_eq:Nc`
`\cs_set_eq:cc`

`\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’.

The `c` variants are not protected in order for their arguments to be constructed in the correct context.

```

1345 \cs_set_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1=~ }
1346 \cs_set_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1347 \cs_set_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1348 \cs_set_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }

```

`\cs_new_eq:NN`
`\cs_new_eq:cN`
`\cs_new_eq:Nc`
`\cs_new_eq:cc`

```

1349 \cs_new_protected:Npn \cs_new_eq:NN #1 {
1350   \chk_if_free_cs:N #1
1351   \pref_global:D \cs_set_eq:NN #1
1352 }
1353 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1354 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1355 \cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

`\cs_gset_eq:NN`
`\cs_gset_eq:cN`
`\cs_gset_eq:Nc`
`\cs_gset_eq:cc`

```

1356 \cs_new_protected:Npn \cs_gset_eq:NN { \pref_global:D \cs_set_eq:NN }
1357 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1358 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1359 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }

```

BACKWARDS COMPATIBILITY

```

1360 \cs_set_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
1361 \cs_set_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
1362 \cs_set_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
1363 \cs_set_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc

```

96.12 Undefined functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some
`\cs_undefine:c` function that isn't in use any longer.

```
\cs_gundefine:N  
\cs_gundefine:c  
1364 \cs_new_protected_nopar:Npn \cs_undefine:N #1 {  
1365   \cs_set_eq:NN #1 \c_undefined:D  
1366 }  
1367 \cs_new_protected_nopar:Npn \cs_undefine:c #1 {  
1368   \cs_set_eq:cN {#1} \c_undefined:D  
1369 }  
1370 \cs_new_protected_nopar:Npn \cs_gundefine:N #1 {  
1371   \cs_gset_eq:NN #1 \c_undefined:D  
1372 }  
1373 \cs_new_protected_nopar:Npn \cs_gundefine:c #1 {  
1374   \cs_gset_eq:cN {#1} \c_undefined:D  
1375 }
```

96.13 Diagnostic wrapper functions

```
\kernel_register_show:N  
\kernel_register_show:c  
1376 \cs_new_nopar:Npn \kernel_register_show:N #1 {  
1377   \cs_if_exist:NTF #1  
1378   {  
1379     \tex_showthe:D #1  
1380   }  
1381   {  
1382     \msg_kernel_bug:x {Register~ '\token_to_str:N #1'~ is~ not~ defined.}  
1383   }  
1384 }  
1385 \cs_new_nopar:Npn \kernel_register_show:c { \exp_args:Nc \int_show:N }
```

96.14 Engine specific definitions

`\c_xetex_is_engine_bool` In some cases it will be useful to know which engine we're running. Don't provide a `_p`
`\c_luatex_is_engine_bool` predicate because the `_bool` is used for the same thing.

```
\xetex_if_engine:TF  
\luatex_if_engine:TF  
1386 \cs_if_exist:NTF \xetex_version:D  
1387 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_true_bool }  
1388 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool }  
1389 \prg_new_conditional:Npnn \xetex_if_engine: {TF,T,F} {  
1390   \if_bool:N \c_xetex_is_engine_bool  
1391     \prg_return_true: \else: \prg_return_false: \fi:  
1392 }  
  
1393 \cs_if_exist:NTF \luatex_directlua:D  
1394 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_true_bool }
```

```

1395 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool }
1396 \prg_set_conditional:Npnn \xetex_if_engine: {TF,T,F}{
1397   \if_bool:N \c_xetex_is_engine_bool \prg_return_true:
1398   \else: \prg_return_false: \fi:
1399 }
1400 \prg_set_conditional:Npnn \luatex_if_engine: {TF,T,F}{
1401   \if_bool:N \c_luatex_is_engine_bool \prg_return_true:
1402   \else: \prg_return_false: \fi:
1403 }

```

96.15 Scratch functions

`\prg_do_nothing:` I don't think this function belongs here, but one place is as good as any other. I want to use this function when I want to express 'no operation'. It is for example used in templates where depending on the users settings we have to either select an function that does something, or one that does nothing.

```

1404 \cs_new_nopar:Npn \prg_do_nothing: {}

```

96.16 Defining functions from a given number of arguments

`\cs_get_arg_count_from_signature:N` Counting the number of tokens in the signature, i.e., the number of arguments the function should take. If there is no signature, we return that there is -1 arguments to signal an error. Otherwise we insert the string 9876543210 after the signature. If the signature is empty, the number we want is 0 so we remove the first nine tokens and return the tenth. Similarly, if the signature is `nnn` we want to remove the nine tokens `nnn987654` and return 3. Therefore, we simply remove the first nine tokens and then return the tenth.

```

1405 \cs_set:Npn \cs_get_arg_count_from_signature:N #1{
1406   \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN
1407 }
1408 \cs_set:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3{
1409   \if_predicate:w #3 % \bool_if:NTF here
1410     \exp_after:wN \use_i:nn
1411   \else:
1412     \exp_after:wN\use_ii:nn
1413   \fi:
1414   {
1415     \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
1416     \use_none:nnnnnnnn #2 9876543210\q_stop
1417   }
1418   {-1}
1419 }
1420 \cs_set:Npn \cs_get_arg_count_from_signature_auxii:w #1#2\q_stop{#1}

```

A variant form we need right away.


```

1421 \cs_set_nopar:Npn \cs_get_arg_count_from_signature:c {
1422   \exp_args:Nc \cs_get_arg_count_from_signature:N
1423 }

```

```

\cs_generate_from_arg_count:NNnn
ate_from_arg_count_error_msg:Nn

```

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

```

1424 \cs_set:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4{
1425   \tex_ifcase:D \etex_numexpr:D #3\tex_relax:D
1426   \use_i_after_orelse:nw{#2#1}
1427   \or:
1428   \use_i_after_orelse:nw{#2#1 ##1}
1429   \or:
1430   \use_i_after_orelse:nw{#2#1 ##1##2}
1431   \or:
1432   \use_i_after_orelse:nw{#2#1 ##1##2##3}
1433   \or:
1434   \use_i_after_orelse:nw{#2#1 ##1##2##3##4}
1435   \or:
1436   \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5}
1437   \or:
1438   \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6}
1439   \or:
1440   \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7}
1441   \or:
1442   \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8}
1443   \or:
1444   \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8##9}
1445   \else:
1446   \use_i_after_fi:nw{
1447     \cs_generate_from_arg_count_error_msg:Nn#1{#3}
1448     \use_none:n % to remove replacement text
1449   }
1450   \fi:
1451   {#4}
1452 }

```

A variant form we need right away.

```

1453 \cs_set_nopar:Npn \cs_generate_from_arg_count:cNnn {
1454   \exp_args:Nc \cs_generate_from_arg_count:NNnn
1455 }

```

The error message. Elsewhere we use the value of -1 to signal a missing colon in a function, so provide a hint for help on this.

```

1456 \cs_set:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2 {
1457   \msg_kernel_bug:x {
1458     You're~ trying~ to~ define~ the~ command~ '\token_to_str:N #1'~
1459     with~ \use:n{\tex_the:D\etex_numexpr:D #2\tex_relax:D} ~
1460     arguments~ but~ I~ only~ allow~ 0-9~arguments.~Perhaps~you~
1461     forgot~to~use~a~colon~in~the~function~name?~
1462     I~ can~ probably~ not~ help~ you~ here
1463   }
1464 }

```

96.17 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, i.e., the number of arguments is read from the signature.

```

\cs_set:Nn We want to define \cs_set:Nn as
\cs_set:Nx
\cs_set_nopar:Nn \cs_set_protected:Npn \cs_set:Nn #1#2{
\cs_set_nopar:Nx \cs_generate_from_arg_count:NNnn #1\cs_set:Npn
\cs_set_protected:Nn {\cs_get_arg_count_from_signature:N #1}{#2}
\cs_set_protected:Nx }
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn In short, to define \cs_set:Nn we need just use \cs_set:Npn, everything else is the same
\cs_gset:Nx for each variant. Therefore, we can make it simpler by temporarily defining a function
\cs_gset_nopar:Nn to do this for us.
\cs_gset_nopar:Nx
\cs_gset_protected:Nn 1465 \cs_set:Npn \cs_tmp:w #1#2#3{
\cs_gset_protected:Nx 1466 \cs_set_protected:cpx {cs_#1:#2}##1##2{
\cs_gset_protected:Nx 1467 \exp_not:N \cs_generate_from_arg_count:NNnn ##1
\cs_gset_protected_nopar:Nn 1468 \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
\cs_gset_protected_nopar:Nx 1469 {\exp_not:N\cs_get_arg_count_from_signature:N ##1}{##2}
1470 }
1471 }

```

Then we define the 32 variants beginning with N.

```

1472 \cs_tmp:w {set}{Nn}{Npn}
1473 \cs_tmp:w {set}{Nx}{Npx}
1474 \cs_tmp:w {set_nopar}{Nn}{Npn}
1475 \cs_tmp:w {set_nopar}{Nx}{Npx}
1476 \cs_tmp:w {set_protected}{Nn}{Npn}
1477 \cs_tmp:w {set_protected}{Nx}{Npx}
1478 \cs_tmp:w {set_protected_nopar}{Nn}{Npn}
1479 \cs_tmp:w {set_protected_nopar}{Nx}{Npx}
1480 \cs_tmp:w {gset}{Nn}{Npn}
1481 \cs_tmp:w {gset}{Nx}{Npx}

```

```

1482 \cs_tmp:w {gset_nopar}{Nn}{Npn}
1483 \cs_tmp:w {gset_nopar}{Nx}{Npx}
1484 \cs_tmp:w {gset_protected}{Nn}{Npn}
1485 \cs_tmp:w {gset_protected}{Nx}{Npx}
1486 \cs_tmp:w {gset_protected_nopar}{Nn}{Npn}
1487 \cs_tmp:w {gset_protected_nopar}{Nx}{Npx}

```

```

\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx
1488 \cs_tmp:w {new}{Nn}{Npn}
1489 \cs_tmp:w {new}{Nx}{Npx}
1490 \cs_tmp:w {new_nopar}{Nn}{Npn}
1491 \cs_tmp:w {new_nopar}{Nx}{Npx}
1492 \cs_tmp:w {new_protected}{Nn}{Npn}
1493 \cs_tmp:w {new_protected}{Nx}{Npx}
1494 \cs_tmp:w {new_protected_nopar}{Nn}{Npn}
1495 \cs_tmp:w {new_protected_nopar}{Nx}{Npx}

```

Then something similar for the c variants.

```

\cs_set_protected:Npn \cs_set:cn #1#2{
  \cs_generate_from_arg_count:cNnn {#1}\cs_set:Npn
    {\cs_get_arg_count_from_signature:c {#1}}{#2}
}

```

```

1496 \cs_set:Npn \cs_tmp:w #1#2#3{
1497   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1498     \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
1499     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1500     {\exp_not:N\cs_get_arg_count_from_signature:c {##1}}{##2}
1501   }
1502 }

```

The 32 c variants.

```

\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
1503 \cs_tmp:w {set}{cn}{Npn}
1504 \cs_tmp:w {set}{cx}{Npx}
1505 \cs_tmp:w {set_nopar}{cn}{Npn}
1506 \cs_tmp:w {set_nopar}{cx}{Npx}
1507 \cs_tmp:w {set_protected}{cn}{Npn}
1508 \cs_tmp:w {set_protected}{cx}{Npx}
1509 \cs_tmp:w {set_protected_nopar}{cn}{Npn}
1510 \cs_tmp:w {set_protected_nopar}{cx}{Npx}
1511 \cs_tmp:w {gset}{cn}{Npn}
1512 \cs_tmp:w {gset}{cx}{Npx}
1513 \cs_tmp:w {gset_nopar}{cn}{Npn}
1514 \cs_tmp:w {gset_nopar}{cx}{Npx}
1515 \cs_tmp:w {gset_protected}{cn}{Npn}

```

```

1516 \cs_tmp:w {gset_protected}{cx}{Npx}
1517 \cs_tmp:w {gset_protected_nopar}{cn}{Npn}
1518 \cs_tmp:w {gset_protected_nopar}{cx}{Npx}

\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
1519 \cs_tmp:w {new}{cn}{Npn}
1520 \cs_tmp:w {new}{cx}{Npx}
1521 \cs_tmp:w {new_nopar}{cn}{Npn}
1522 \cs_tmp:w {new_nopar}{cx}{Npx}
1523 \cs_tmp:w {new_protected}{cn}{Npn}
1524 \cs_tmp:w {new_protected}{cx}{Npx}
1525 \cs_tmp:w {new_protected_nopar}{cn}{Npn}
1526 \cs_tmp:w {new_protected_nopar}{cx}{Npx}

\cs_if_eq_p:NN Check if two control sequences are identical.
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NN $\overline{TF}$ 
\cs_if_eq:cN $\overline{TF}$ 
\cs_if_eq:Nc $\overline{TF}$ 
\cs_if_eq:cc $\overline{TF}$ 
1527 \prg_set_conditional:Npnn \cs_if_eq:NN #1#2{p,TF,T,F}{
1528   \if_meaning:w #1#2
1529   \prg_return_true: \else: \prg_return_false: \fi:
1530 }
1531 \cs_new_nopar:Npn \cs_if_eq_p:cN {\exp_args:Nc \cs_if_eq_p:NN}
1532 \cs_new_nopar:Npn \cs_if_eq:cN $\overline{TF}$  {\exp_args:Nc \cs_if_eq:NN $\overline{TF}$ }
1533 \cs_new_nopar:Npn \cs_if_eq:cNT {\exp_args:Nc \cs_if_eq:NNT}
1534 \cs_new_nopar:Npn \cs_if_eq:cNF {\exp_args:Nc \cs_if_eq:NNF}
1535 \cs_new_nopar:Npn \cs_if_eq_p:Nc {\exp_args:NNc \cs_if_eq_p:NN}
1536 \cs_new_nopar:Npn \cs_if_eq:Nc $\overline{TF}$  {\exp_args:NNc \cs_if_eq:NN $\overline{TF}$ }
1537 \cs_new_nopar:Npn \cs_if_eq:NcT {\exp_args:NNc \cs_if_eq:NNT}
1538 \cs_new_nopar:Npn \cs_if_eq:NcF {\exp_args:NNc \cs_if_eq:NNF}
1539 \cs_new_nopar:Npn \cs_if_eq_p:cc {\exp_args:Ncc \cs_if_eq_p:NN}
1540 \cs_new_nopar:Npn \cs_if_eq:cc $\overline{TF}$  {\exp_args:Ncc \cs_if_eq:NN $\overline{TF}$ }
1541 \cs_new_nopar:Npn \cs_if_eq:ccT {\exp_args:Ncc \cs_if_eq:NNT}
1542 \cs_new_nopar:Npn \cs_if_eq:ccF {\exp_args:Ncc \cs_if_eq:NNF}

1543 \</initex | package>

1544 \< *showmemory>
1545 \showMemUsage
1546 \< /showmemory>

```

97 l3expan implementation

97.1 Internal functions and variables

$\boxed{\backslash\exp_after:wN}$ $\backslash\exp_after:wN \langle token_1 \rangle \langle token_2 \rangle$

This will expand $\langle token_2 \rangle$ once before processing $\langle token_1 \rangle$. This is similar to $\backslash\exp_args:No$ except that no braces are put around the result of expanding $\langle token_2 \rangle$.

T_EXhackers note: This is the primitive `\expandafter` which was renamed to fit into the naming conventions of L^AT_EX3.

`\l_exp_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.

`\exp_eval_register:N *`
`\exp_eval_register:c *` `\exp_eval_register:N <register>`

These functions evaluates a register as part of a `V` or `v` expansion (respectively). A register might exist as one of two things: A parameter-less non-long, non-protected macro or a built-in T_EX register such as `\count`.

`\exp_eval_error_msg:w` `\exp_eval_error_msg:w <register>`

Used to generate an error message if a variable called as part of a `v` or `V` expansion is defined as `\scan_stop:`. This typically indicates that an incorrect cs name has been used.

`\n::`
`\N::`
`\c::`
`\o::`
`\f::`
`\x::`
`\v::`
`\V::`
`\:::` `\cs_set_nopar:Npn \exp_args:Ncof {\::c\::o\::f\:::}`

Internal forms for the base expansion types.

97.2 Module code

We start by ensuring that the required packages are loaded.

```
1547 <*package>
1548 \ProvidesExplPackage
1549   {\filename}{\filedate}{\fileversion}{\filedescription}
1550 \package_check_loaded_expl:
1551 </package>
1552 <*initex | package>
```

```

\exp_after:wN These are defined in l3basics.
\exp_not:N
\exp_not:n
1553 <*bootstrap>
1554 \cs_set_eq:NwN \exp_after:wN \tex_expandafter:D
1555 \cs_set_eq:NwN \exp_not:N \tex_noexpand:D
1556 \cs_set_eq:NwN \exp_not:n \etex_unexpanded:D
1557 </bootstrap>

```

97.3 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 L^AT_EX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.⁸)

The definition of expansion functions with this technique happens in section 97.5. In section 97.4 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l_exp_tl` We need a scratch token list variable. We don't use `tl` methods so that `l3expan` can be loaded earlier.

```

1558 \cs_new_nopar:Npn \l_exp_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.

`\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 auxilliary 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 will require 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.

`\exp_arg_next_nobrace:nnn`

```

1559 \cs_new:Npn\exp_arg_next:nnn#1#2#3{
1560   #2\:::{#3{#1}}
1561 }
1562 \cs_new:Npn\exp_arg_next_nobrace:nnn#1#2#3{
1563   #2\:::{#3#1}
1564 }

```

⁸However, some primitives have certain characteristics that means that their arguments undergo an `x` type expansion but the primitive is in fact still expandable. We shall make it very clear when such a function is expandable.

`\:::` The end marker is just another name for the identity function.

```
1565 \cs_new:Npn\:::#1{#1}
```

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
1566 \cs_new:Npn\::n#1\:::#2#3{
1567   #1\:::{#2{#3}}
1568 }
```

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
1569 \cs_new:Npn\::N#1\:::#2#3{
1570   #1\:::{#2#3}
1571 }
```

`\::c` This function is used to skip an argument that is turned into as control sequence without expansion.

```
1572 \cs_new:Npn\::c#1\:::#2#3{
1573   \exp_after:wN\exp_arg_next:nbr:nnn\cs:w #3\cs_end:{#1}{#2}
1574 }
```

`\::o` This function is used to expand an argument once.

```
1575 \cs_new:Npn\::o#1\:::#2#3{
1576   \exp_after:wN\exp_arg_next:nnn\exp_after:wN{#3}{#1}{#2}
1577 }
```

`\::f` This function is used to expand a token list until the first unexpandable token is found.
`\exp_stop_f:` The underlying `\tex_romannumeral:D -'0` expands everything in its way to find something terminating the number and thereby expands the function in front of 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 case the scanner hits a number, this number also terminates the scanning and is left untouched. 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:NwN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NwN`. Since the expansion of `\tex_romannumeral:D -'0` 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.

```
1578 \cs_new:Npn\::f#1\:::#2#3{
1579   \exp_after:wN\exp_arg_next:nnn
1580   \exp_after:wN{\tex_romannumeral:D -'0 #3}
1581   {#1}{#2}
1582 }
1583 \cs_new_nopar:Npn \exp_stop_f: {~}
```

`\::x` This function is used to expand an argument fully. We could use the new expandable primitive `\expanded` here, but we don't want to create incompatibilities between engines.

```

1584 \cs_new_protected:Npn \::x #1 \::: #2#3 {
1585   \cs_set_nopar:Npx \l_exp_tl {{#3}}
1586   \exp_after:wN \exp_arg_next:nnn \l_exp_tl {#1}{#2}
1587 }

```

`\::v` These functions return the value of a register, i.e., one of `tl`, `num`, `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 sequence `\tex_romannumeral:D -'0` sets off an `f` type expansion. The argument is returned in braces.

```

1588 \cs_new:Npn \::V#1\::: #2#3{
1589   \exp_after:wN\exp_arg_next:nnn
1590   \exp_after:wN{
1591     \tex_romannumeral:D -'0
1592     \exp_eval_register:N #3
1593   }
1594   {#1}{#2}
1595 }
1596 \cs_new:Npn \::v#1\::: #2#3{
1597   \exp_after:wN\exp_arg_next:nnn
1598   \exp_after:wN{
1599     \tex_romannumeral:D -'0
1600     \exp_eval_register:c {#3}
1601   }
1602   {#1}{#2}
1603 }

```

`\exp_eval_register:N` 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 `\tex_the:D` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\tex_the:D` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\tex_relax:D`.

```

1604 \cs_set_nopar:Npn \exp_eval_register:N #1{
1605   \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 `\tex_relax:D`. 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.

```

1606     \if_meaning:w \tex_relax:D #1
1607     \exp_eval_error_msg:w
1608     \fi:

```

The next bit requires some explanation. The function must be initiated by the sequence `\tex_romannumeral:D -'0` and we want to terminate this expansion chain by inserting an `\exp_stop_f:` token. However, we have to expand the register `#1` before we do that. If it is a T_EX register, we need to execute the sequence `\exp_after:wN\exp_stop_f:\tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN\exp_stop_f: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

1609     \else:
1610     \exp_after:wN \use_i_ii:nnn
1611     \fi:
1612     \exp_after:wN \exp_stop_f: \tex_the:D #1
1613   }
1614   \cs_set_nopar:Npn \exp_eval_register:c #1{
1615     \exp_after:wN\exp_eval_register:N\cs:w #1\cs_end:
1616   }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
\exp_eval_error_msg:w ...erroneous variable used!

```

```

1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

```

```

1617 \group_begin:%
1618 \tex_catcode:D'\!=11\tex_relax:D%
1619 \tex_catcode:D'\ =11\tex_relax:D%
1620 \cs_gset:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
1621   \fi:\fi:\erroneous variable used!}%
1622 \group_end:%

```

97.4 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 `\pref_global:D` for example. This together with the fact that the ‘general’ concept above is slower means that we should convert whenever possible and perhaps remove all remaining occurrences by hand-encoding in the end.

\exp_args:No
\exp_args:NNo
\exp_args:NNNo

```
1623 \cs_new:Npn \exp_args:No #1#2{\exp_after:wN#1\exp_after:wN{#2}}
1624 \cs_new:Npn \exp_args:NNo #1#2#3{\exp_after:wN#1\exp_after:wN#2
1625   \exp_after:wN{#3}}
1626 \cs_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:wN#1\exp_after:wN#2
1627   \exp_after:wN#3\exp_after:wN{#4}}
```

\exp_args:Nc
\exp_args:cc
\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc

Here are the functions that turn their argument into csnames but are expandable.

```
1628 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
1629 \cs_new:Npn \exp_args:cc #1#2{\cs:w #1\exp_after:wN\cs_end:\cs:w #2\cs_end:}
1630 \cs_new:Npn \exp_args:NNc #1#2#3{\exp_after:wN#1\exp_after:wN#2
1631   \cs:w#3\cs_end:}
1632 \cs_new:Npn \exp_args:Ncc #1#2#3{\exp_after:wN#1
1633   \cs:w#2\exp_after:wN\cs_end:\cs:w#3\cs_end:}
1634 \cs_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:wN#1
1635   \cs:w#2\exp_after:wN\cs_end:\cs:w#3\exp_after:wN
1636   \cs_end:\cs:w #4\cs_end:}
```

\exp_args:Nco

If we force that the third argument always has braces, we could implement this function with less tokens and only two arguments.

```
1637 \cs_new:Npn \exp_args:Nco #1#2#3{\exp_after:wN#1\cs:w#2\exp_after:wN
1638   \cs_end:\exp_after:wN{#3}}
```

97.5 Definitions with the ‘general’ technique

\exp_args:Nf
\exp_args:Nv
\exp_args:Nv
\exp_args:Nx

```
1639 \cs_set_nopar:Npn \exp_args:Nf {\::f\:::}
1640 \cs_set_nopar:Npn \exp_args:Nv {\::v\:::}
1641 \cs_set_nopar:Npn \exp_args:NV {\::V\:::}
1642 \cs_set_protected_nopar:Npn \exp_args:Nx {\::x\:::}
```

\exp_args:NNV

Here are the actual function definitions, using the helper functions above.

\exp_args:NNv
\exp_args:NNf
\exp_args:NNx
\exp_args:NNV
\exp_args:Ncx
\exp_args:Nfo
\exp_args:Nff
\exp_args:Ncf
\exp_args:Nco
\exp_args:Nnf
\exp_args:Nno
\exp_args:NnV
\exp_args:Nnx
\exp_args:Noo
\exp_args:Noc
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```
1643 \cs_set_nopar:Npn \exp_args:NNf {\::N\::f\:::}
1644 \cs_set_nopar:Npn \exp_args:NNv {\::N\::v\:::}
1645 \cs_set_nopar:Npn \exp_args:NNV {\::N\::V\:::}
1646 \cs_set_protected_nopar:Npn \exp_args:NNx {\::N\::x\:::}
1647
1648 \cs_set_protected_nopar:Npn \exp_args:Ncx {\::c\:::x\:::}
1649 \cs_set_nopar:Npn \exp_args:Nfo {\::f\:::o\:::}
1650 \cs_set_nopar:Npn \exp_args:Nff {\::f\:::f\:::}
1651 \cs_set_nopar:Npn \exp_args:Ncf {\::c\:::f\:::}
1652 \cs_set_nopar:Npn \exp_args:Nnf {\::n\:::f\:::}
1653 \cs_set_nopar:Npn \exp_args:Nno {\::n\:::o\:::}
```

```

1654 \cs_set_nopar:Npn \exp_args:NnV {\::n\::V\:::}
1655 \cs_set_protected_nopar:Npn \exp_args:Nnx {\::n\::x\:::}
1656
1657 \cs_set_nopar:Npn \exp_args:Noc {\::o\::c\:::}
1658 \cs_set_nopar:Npn \exp_args:Noo {\::o\::o\:::}
1659 \cs_set_protected_nopar:Npn \exp_args:Nox {\::o\::x\:::}
1660
1661 \cs_set_nopar:Npn \exp_args:NVV {\::V\::V\:::}
1662
1663 \cs_set_protected_nopar:Npn \exp_args:Nxo {\::x\::o\:::}
1664 \cs_set_protected_nopar:Npn \exp_args:Nxx {\::x\::x\:::}

\exp_args:Ncco
\exp_args:Nccx
\exp_args:Ncnx
1665 \cs_set_nopar:Npn \exp_args:NNNV {\::N\::N\::V\:::}
1666
\exp_args:NcNc
1667 \cs_set_nopar:Npn \exp_args:NNno {\::N\::n\::o\:::}
\exp_args:NcNo
1668 \cs_set_protected_nopar:Npn \exp_args:NNnx {\::N\::n\::x\:::}
\exp_args:NNno
1669 \cs_set_nopar:Npn \exp_args:NNoo {\::N\::o\::o\:::}
\exp_args:NNNV
1670 \cs_set_protected_nopar:Npn \exp_args:NNox {\::N\::o\::x\:::}
\exp_args:Nnno
1671
\exp_args:Nnnx
1672 \cs_set_nopar:Npn \exp_args:Nnnc {\::n\::n\::c\:::}
\exp_args:Nnox
1673 \cs_set_nopar:Npn \exp_args:Nnno {\::n\::n\::o\:::}
\exp_args:Nooo
1674 \cs_set_protected_nopar:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
\exp_args:Noox
1675 \cs_set_protected_nopar:Npn \exp_args:Nnox {\::n\::o\::x\:::}
\exp_args:Nnnc
1676
\exp_args:NNnx
1677 \cs_set_nopar:Npn \exp_args:NcNc {\::c\::N\::c\:::}
\exp_args:NNno
1678 \cs_set_nopar:Npn \exp_args:NcNo {\::c\::N\::o\:::}
\exp_args:NNoo
1679 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
\exp_args:NNox
1680 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\:::}
1681 \cs_set_protected_nopar:Npn \exp_args:Nccx {\::c\::c\::x\:::}
1682 \cs_set_protected_nopar:Npn \exp_args:Ncnx {\::c\::n\::x\:::}
1683
1684 \cs_set_protected_nopar:Npn \exp_args:Noox {\::o\::o\::x\:::}
1685 \cs_set_nopar:Npn \exp_args:Nooo {\::o\::o\::o\:::}

```

97.6 Preventing expansion

```

\exp_not:o
\exp_not:f
\exp_not:v
\exp_not:V
1686 \cs_new:Npn\exp_not:o#1{\exp_not:n\exp_after:wN{#1}}
1687 \cs_new:Npn\exp_not:f#1{
1688   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 #1}
1689 }
1690 \cs_new:Npn\exp_not:v#1{
1691   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:c {#1}}
1692 }
1693 \cs_new:Npn\exp_not:V#1{
1694   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:N #1}

```

```
1695 }
```

`\exp_not:c` A helper function.

```
1696 \cs_new:Npn\exp_not:c#1{\exp_after:wN\exp_not:N\cs:w#1\cs_end:}
```

97.7 Defining function variants

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`

`\cs_generate_variant_aux:nnNn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

`\cs_generate_variant_aux:nnw`

`\cs_generate_variant_aux:N`

Split up the original base function to grab its name and signature consisting of k letters. Then we wish to iterate through the list of variant argument specifiers, and for each one 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. For example, for a base function `\tl_set:Nn` which needs a `c` variant form, we want the new signature to be `cn`.

```
1697 \cs_new_protected:Npn \cs_generate_variant:Nn #1 {
1698   \chk_if_exist_cs:N #1
1699   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNn
1700 }
```

We discard the boolean and then set off a loop through the desired variant forms.

```
1701 \cs_set:Npn \cs_generate_variant_aux:nnNn #1#2#3#4{
1702   \cs_generate_variant_aux:nnw {#1}{#2} #4,?,\q_recursion_stop
1703 }
```

Next is the real work to be done. We now have 1: base name, 2: base signature, 3: beginning of variant signature. To construct the new `csname` and the `\exp_args:Ncc` form, we need the variant signature. In our example, we wanted to discard the first two letters of the base signature because the variant form started with `cc`. This is the same as putting first `cc` in the signature and then `\use_none:nn` followed by the base signature `NNn`. We therefore call a small loop that outputs an `n` for each letter in the variant signature and use this to call the correct `\use_none:` variant. Firstly though, we check whether to terminate the loop.

```
1704 \cs_set:Npn \cs_generate_variant_aux:nnw #1 #2 #3, {
1705   \if:w ? #3
1706   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1707   \fi:
```

Then check if the variant form has already been defined.

```
1708 \cs_if_free:cTF {
1709   #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1710 }
1711 {
```

If not, then define it and then additionally check if the `\exp_args:N` form needed is defined.

```

1712   \cs_generate_variant_aux:ccpx { #1 : #2 }
1713   {
1714     #1:#3 \use:c{use_none:\cs_generate_variant_aux:N #3 ?}#2
1715   }
1716   {
1717     \exp_not:c { exp_args:N #3} \exp_not:c {#1:#2}
1718   }
1719   \cs_generate_internal_variant:n {#3}
1720 }

```

Otherwise tell that it was already defined.

```

1721 {
1722   \iow_log:x{
1723     Variant~\token_to_str:c {
1724       #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1725     }~already~defined;~ not~ changing~ it~on~line~
1726     \tex_the:D \tex_inputlineno:D
1727   }
1728 }

```

Recurse.

```

1729   \cs_generate_variant_aux:nnw{#1}{#2}
1730 }

```

The small loop for defining the required number of ns. Break when seeing a ?.

```

1731 \cs_set:Npn \cs_generate_variant_aux:N #1{
1732   \if:w ?#1 \exp_after:wN\use_none:nn \fi: n \cs_generate_variant_aux:N
1733 }

```

`\cs_generate_variant_aux:Ncp` The idea here is to pick up protected parent functions, using the nature of the meaning string that they generate. The test here is almost the same as `\tl_if_empty:nTF`, but
`\cs_generate_variant_aux:ccpx` has to be hard-coded as that function is not yet available and because it has to match
`\cs_generate_variant_aux:w` both long and short macros.

```

1734 \group_begin:
1735   \tex_lccode:D '\Z = '\d \scan_stop:
1736   \tex_lccode:D '\? ='\ \ \scan_stop:
1737   \tex_catcode:D '\P = 12 \scan_stop:
1738   \tex_catcode:D '\R = 12 \scan_stop:
1739   \tex_catcode:D '\O = 12 \scan_stop:
1740   \tex_catcode:D '\T = 12 \scan_stop:
1741   \tex_catcode:D '\E = 12 \scan_stop:
1742   \tex_catcode:D '\C = 12 \scan_stop:
1743   \tex_catcode:D '\Z = 12 \scan_stop:

```

```

1744 \tex_lowercase:D {
1745   \group_end:
1746   \cs_new_nopar:Npn \_cs_generate_variant_aux:Ncpx #1
1747   {
1748     \exp_after:wN \_cs_generate_variant_aux:w
1749     \tex_meaning:D #1 ? PROTECTEZ \q_stop
1750   }
1751   \cs_new_nopar:Npn \_cs_generate_variant_aux:ccpx
1752   { \exp_args:Nc \_cs_generate_variant_aux:Ncpx}
1753   \cs_new:Npn \_cs_generate_variant_aux:w
1754   #1 ? PROTECTEZ #2 \q_stop
1755   {
1756     \exp_after:wN \tex_ifx:D \exp_after:wN
1757     \q_no_value \etex_detokenize:D {#1} \q_no_value
1758     \exp_after:wN \cs_new_protected_nopar:cpx
1759     \tex_else:D
1760     \exp_after:wN \cs_new_nopar:cpx
1761     \tex_fi:D
1762   }
1763 }

```

`\cs_generate_internal_variant:n` Test if `\exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in `#1`

```

1764 \cs_new_protected:Npn \cs_generate_internal_variant:n #1 {
1765   \cs_if_free:cT { \exp_args:N #1 }{

```

We use `new` to log the definition if we have to make one.

```

1766     \cs_new:cpx { \exp_args:N #1 }
1767     { \cs_generate_internal_variant_aux:n #1 : }
1768   }
1769 }

```

`\generate_internal_variant_aux:n` This command grabs char by char outputting `\::#1` (not expanded further) until we see a `\.` That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```

1770 \cs_new:Npn \cs_generate_internal_variant_aux:n #1 {
1771   \exp_not:c{::#1}
1772   \if_meaning:w #1 :
1773     \exp_after:wN \use_none:n
1774   \fi:
1775   \cs_generate_internal_variant_aux:n
1776 }

```

97.8 Last-unbraced versions

`\exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
1777 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1778 \cs_new:Npn \::f_unbraced \:::#1#2 {
1779   \exp_after:wN \exp_arg_last_unbraced:nn
1780   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1781 }
1782 \cs_new:Npn \::o_unbraced \:::#1#2 {
1783   \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2 }{#1}
1784 }
1785 \cs_new:Npn \::V_unbraced \:::#1#2 {
1786   \exp_after:wN \exp_arg_last_unbraced:nn
1787   \exp_after:wN { \tex_romannumeral:D -'0 \exp_eval_register:N #2 } {#1}
1788 }
1789 \cs_new:Npn \::v_unbraced \:::#1#2 {
1790   \exp_after:wN \exp_arg_last_unbraced:nn
1791   \exp_after:wN {
1792     \tex_romannumeral:D -'0 \exp_eval_register:c {#2}
1793   } {#1}
1794 }

```

`\exp_last_unbraced:NV` Now the business end.

```

\exp_last_unbraced:No
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo
1795 \cs_new_nopar:Npn \exp_last_unbraced:Nf { \::f_unbraced \::: }
1796 \cs_new_nopar:Npn \exp_last_unbraced:NV { \::V_unbraced \::: }
1797 \cs_new_nopar:Npn \exp_last_unbraced:No { \::o_unbraced \::: }
1798 \cs_new_nopar:Npn \exp_last_unbraced:Nv { \::v_unbraced \::: }
1799 \cs_new_nopar:Npn \exp_last_unbraced:NcV {
1800   \::c \::V_unbraced \:::
1801 }
1802 \cs_new_nopar:Npn \exp_last_unbraced:NNV {
1803   \::N \::V_unbraced \:::
1804 }
1805 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3 {
1806   \exp_after:wN #1 \exp_after:wN #2 #3
1807 }
1808 \cs_new_nopar:Npn \exp_last_unbraced:NNNV {
1809   \::N \::N \::V_unbraced \:::
1810 }
1811 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4 {
1812   \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4
1813 }

```

97.9 Items held from earlier

`\str_if_eq_p:Vn` These cannot come earlier as they need `\cs_generate_variant:Nn`.

```

\str_if_eq:VnTF
\str_if_eq_p:on
\str_if_eq:onTF
\str_if_eq_p:nV
\str_if_eq:nVTF
\str_if_eq_p:no
\str_if_eq:noTF
\str_if_eq_p:VV
\str_if_eq:VVTF

```

```

1814 \cs_generate_variant:Nn \str_if_eq_p:nn { V }
1815 \cs_generate_variant:Nn \str_if_eq_p:nn { o }
1816 \cs_generate_variant:Nn \str_if_eq_p:nn { nV }
1817 \cs_generate_variant:Nn \str_if_eq_p:nn { no }
1818 \cs_generate_variant:Nn \str_if_eq_p:nn { VV }
1819 \cs_generate_variant:Nn \str_if_eq:nnT { V }
1820 \cs_generate_variant:Nn \str_if_eq:nnT { o }
1821 \cs_generate_variant:Nn \str_if_eq:nnT { nV }
1822 \cs_generate_variant:Nn \str_if_eq:nnT { no }
1823 \cs_generate_variant:Nn \str_if_eq:nnT { VV }
1824 \cs_generate_variant:Nn \str_if_eq:nnF { V }
1825 \cs_generate_variant:Nn \str_if_eq:nnF { o }
1826 \cs_generate_variant:Nn \str_if_eq:nnF { nV }
1827 \cs_generate_variant:Nn \str_if_eq:nnF { no }
1828 \cs_generate_variant:Nn \str_if_eq:nnF { VV }
1829 \cs_generate_variant:Nn \str_if_eq:nnTF { V }
1830 \cs_generate_variant:Nn \str_if_eq:nnTF { o }
1831 \cs_generate_variant:Nn \str_if_eq:nnTF { nV }
1832 \cs_generate_variant:Nn \str_if_eq:nnTF { no }
1833 \cs_generate_variant:Nn \str_if_eq:nnTF { VV }

1834 </initex | package>

```

Show token usage:

```

1835 <*showmemory>
1836 \showMemUsage
1837 </showmemory>

```

98 l3prg implementation

98.1 Variables

\l_tmpa_bool \g_tmpa_bool

Reserved booleans.

\g_prg_inline_level_int

Global variable to track the nesting of the stepwise inline loop.

98.2 Module code

We start by ensuring that the required packages are loaded.

```

1838 <*package>

```



```

1839 \ProvidesExplPackage
1840   {\filename}{\filedate}{\fileversion}{\filedescription}
1841 \package_check_loaded_expl:
1842 </package>
1843 <*initex | package>

```

`\prg_return_true:` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder that
`\prg_return_false:` that is the case!

```

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

```

98.3 Choosing modes

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 `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```

1844 \prg_set_conditional:Npnn \mode_if_vertical: {p,TF,T,F}{
1845   \if_mode_vertical:
1846   \prg_return_true: \else: \prg_return_false: \fi:
1847 }

```

`\mode_if_horizontal_p:` For testing horizontal mode.

`\mode_if_horizontal:` TF

```

1848 \prg_set_conditional:Npnn \mode_if_horizontal: {p,TF,T,F}{
1849   \if_mode_horizontal:
1850   \prg_return_true: \else: \prg_return_false: \fi:
1851 }

```

`\mode_if_inner_p:` For testing inner mode.

`\mode_if_inner:` TF

```

1852 \prg_set_conditional:Npnn \mode_if_inner: {p,TF,T,F}{
1853   \if_mode_inner:
1854   \prg_return_true: \else: \prg_return_false: \fi:
1855 }

```

`\mode_if_math_p:` For testing math mode. Uses the kern-save `\scan_align_safe_stop:`.

`\mode_if_math:` TF

```

1856 \prg_set_conditional:Npnn \mode_if_math: {p,TF,T,F}{
1857   \scan_align_safe_stop: \if_mode_math:
1858   \prg_return_true: \else: \prg_return_false: \fi:
1859 }

```

Alignment safe grouping and scanning

`\group_align_safe_begin:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: 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 will get some sort of weird error message because the underlying `\tex_futurelet:D` will store 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 T_EX 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 T_EXbook*...

```
1860 \cs_new_nopar:Npn \group_align_safe_begin: {
1861   \if_false:{\fi:\if_num:w'=\c_zero\fi:}
1862 \cs_new_nopar:Npn \group_align_safe_end:   {\if_num:w'=\c_zero}\fi:}
```

`\scan_align_safe_stop:` When T_EX is in the beginning of an align cell (right after the `\cr`) it is in a somewhat strange mode as it is looking ahead to find an `\tex_omit:D` or `\tex_noalign:D` and hasn’t looked at the preamble yet. Thus an `\tex_ifmmode:D` test will always fail unless we insert `\scan_stop:` to stop T_EX’s scanning ahead. On the other hand we don’t want to insert a `\scan_stop:` every time as that will destroy kerning between letters⁹ Unfortunately there is no way to detect if we’re in the beginning of an alignment cell as they have different characteristics depending on column number etc. However we *can* detect if we’re in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that `\scan_stop:` is only inserted iff a) we’re in the outer part of an alignment cell and b) the last node *wasn’t* a char node or a ligature node.

```
1863 \cs_new_nopar:Npn \scan_align_safe_stop: {
1864   \int_compare:nNnT \etex_currentgrouptype:D = \c_six
1865   {
1866     \int_compare:nNnF \etex_lastnodetype:D = \c_zero
1867     {
1868       \int_compare:nNnF \etex_lastnodetype:D = \c_seven
1869       \scan_stop:
1870     }
1871   }
1872 }
```

98.4 Producing n copies

`\prg_replicate:nn` This function uses a cascading csname technique by David Kastrup (who else :-)
`\prg_replicate_aux:N`
`\prg_replicate_first_aux:N`

⁹Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

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. Finally we must ensure that the cascade comes to a peaceful end so we make it so that the original csnme `\TeX` is creating is simply `\prg_do_nothing:` expanding to nothing.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it will severely affect 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. An alternative approach is to create a string of m's with `\int_to_roman: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.

```

1873 \cs_new_nopar:Npn \prg_replicate:nn #1{
1874   \cs:w prg_do_nothing:
1875   \exp_after:wN\prg_replicate_first_aux:N
1876   \tex_romannumeral:D -'\q \int_eval:n{#1} \cs_end:
1877   \cs_end:
1878 }
1879 \cs_new_nopar:Npn \prg_replicate_aux:N#1{
1880   \cs:w prg_replicate_#1:n\prg_replicate_aux:N
1881 }
1882 \cs_new_nopar:Npn \prg_replicate_first_aux:N#1{
1883   \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
1884 }

```

Then comes all the functions that do the hard work of inserting all the copies.

```

1885 \cs_new_nopar:Npn \prg_replicate_ :n #1{ }% no, this is not a typo!
1886 \cs_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1887 \cs_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1888 \cs_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1889 \cs_new:cpn {prg_replicate_3:n}#1{
1890   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1891 \cs_new:cpn {prg_replicate_4:n}#1{
1892   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1893 \cs_new:cpn {prg_replicate_5:n}#1{
1894   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1895 \cs_new:cpn {prg_replicate_6:n}#1{
1896   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1897 \cs_new:cpn {prg_replicate_7:n}#1{

```

```

1898 \cs_end:{#1#1#1#1#1#1#1#1#1#1}
1899 \cs_new:cpn {prg_replicate_8:n}{#1{
1900   \cs_end:{#1#1#1#1#1#1#1#1#1#1}
1901   \cs_new:cpn {prg_replicate_9:n}{#1{
1902     \cs_end:{#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...

```

1903 \cs_new:cpn {prg_replicate_first_-\n}#1{\cs_end:  \ERROR }
1904 \cs_new:cpn {prg_replicate_first_0:n}#1{\cs_end:  }
1905 \cs_new:cpn {prg_replicate_first_1:n}#1{\cs_end:  #1}
1906 \cs_new:cpn {prg_replicate_first_2:n}#1{\cs_end:  #1#1}
1907 \cs_new:cpn {prg_replicate_first_3:n}#1{\cs_end:  #1#1#1}
1908 \cs_new:cpn {prg_replicate_first_4:n}#1{\cs_end:  #1#1#1#1}
1909 \cs_new:cpn {prg_replicate_first_5:n}#1{\cs_end:  #1#1#1#1#1}
1910 \cs_new:cpn {prg_replicate_first_6:n}#1{\cs_end:  #1#1#1#1#1#1}
1911 \cs_new:cpn {prg_replicate_first_7:n}#1{\cs_end:  #1#1#1#1#1#1#1}
1912 \cs_new:cpn {prg_replicate_first_8:n}#1{\cs_end:  #1#1#1#1#1#1#1#1}
1913 \cs_new:cpn {prg_replicate_first_9:n}#1{\cs_end:  #1#1#1#1#1#1#1#1#1}

```

```

\prg_stepwise_function:nnnN
prg_stepwise_function_incr:nnnN
prg_stepwise_function_decr:nnnN

```

A stepwise function. Firstly we check the direction of the steps **#2** since that will depend on which test we should use. If the step is positive we use a greater than test, otherwise a less than test. If the test comes out true exit, otherwise perform **#4**, add the step to **#1** and try again with this new value of **#1**.

```

1914 \cs_new:Npn \prg_stepwise_function:nnnN #1#2{
1915   \int_compare:nNnTF{#2}<\c_zero
1916     {\exp_args:Nf\prg_stepwise_function_decr:nnnN }
1917     {\exp_args:Nf\prg_stepwise_function_incr:nnnN }
1918     {\int_eval:n{#1}}{#2}
1919 }
1920 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4{
1921   \int_compare:nNnF {#1}>{#3}
1922   {
1923     #4{#1}
1924     \exp_args:Nf \prg_stepwise_function_incr:nnnN
1925     {\int_eval:n{#1 + #2}}
1926     {#2}{#3}{#4}
1927   }
1928 }
1929 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4{
1930   \int_compare:nNnF {#1}<{#3}
1931   {
1932     #4{#1}
1933     \exp_args:Nf \prg_stepwise_function_decr:nnnN
1934     {\int_eval:n{#1 + #2}}
1935     {#2}{#3}{#4}
1936   }
1937 }

```

<pre> \g_prg_inline_level_int \prg_stepwise_inline:nnnn \prg_stepwise_inline_decr:nnnn \prg_stepwise_inline_incr:nnnn </pre>	<p>This function uses the same approach as for instance <code>\clist_map_inline:Nn</code> to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed csname. Must make assignments global when we maintain our own stack.</p>
--	---

```

1938 \int_new:N\g_prg_inline_level_int
1939 \cs_new_protected:Npn\prg_stepwise_inline:nnnn #1#2#3#4{
1940   \int_gincr:N \g_prg_inline_level_int
1941   \cs_gset_nopar:cpn{prg_stepwise_inline\_int_use:N\g_prg_inline_level_int :n}##1{#4}
1942   \int_compare:nNnTF {#2}<\c_zero
1943   {\exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
1944   {\exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
1945   {prg_stepwise_inline\_int_use:N\g_prg_inline_level_int :n}
1946   {\int_eval:n{#1}} {#2} {#3}
1947   \int_gdecr:N \g_prg_inline_level_int
1948 }
1949 \cs_new:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4{
1950   \int_compare:nNnF {#2}>{#4}
1951   {
1952     #1{#2}
1953     \exp_args:NNf \prg_stepwise_inline_incr:Nnnn #1
1954     {\int_eval:n{#2 + #3}} {#3}{#4}
1955   }
1956 }
1957 \cs_new:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4{
1958   \int_compare:nNnF {#2}<{#4}
1959   {
1960     #1{#2}
1961     \exp_args:NNf \prg_stepwise_inline_decr:Nnnn #1
1962     {\int_eval:n{#2 + #3}} {#3}{#4}
1963   }
1964 }

```

<pre> \prg_stepwise_variable:nnnNn \prg_stepwise_variable_decr:nnnNn \prg_stepwise_variable_incr:nnnNn </pre>	<p>Almost the same as above. Just store the value in #4 and execute #5.</p>
---	---

```

1965 \cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2 {
1966   \int_compare:nNnTF {#2}<\c_zero
1967   {\exp_args:Nf\prg_stepwise_variable_decr:nnnNn}
1968   {\exp_args:Nf\prg_stepwise_variable_incr:nnnNn}
1969   {\int_eval:n{#1}}{#2}
1970 }
1971 \cs_new_protected:Npn \prg_stepwise_variable_incr:nnnNn #1#2#3#4#5 {
1972   \int_compare:nNnF {#1}>{#3}
1973   {
1974     \cs_set_nopar:Npn #4{#1} #5
1975     \exp_args:Nf \prg_stepwise_variable_incr:nnnNn
1976     {\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1977   }
1978 }
1979 \cs_new_protected:Npn \prg_stepwise_variable_decr:nnnNn #1#2#3#4#5 {

```

```

1980 \int_compare:nNnF {#1}<{#3}
1981 {
1982   \cs_set_nopar:Npn #4{#1} #5
1983   \exp_args:Nf \prg_stepwise_variable_decr:nnnNn
1984   {\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
1985 }
1986 }

```

98.5 Booleans

For normal booleans we set them to either `\c_true_bool` or `\c_false_bool` and then use `\if_bool:N` to choose the right branch. The functions return either the TF, T, or F case *after* ending the `\if_bool:N`. We only define the N versions here as the c versions can easily be constructed with the expansion module.

```

\bool_new:N Defining and setting a boolean is easy.
\bool_new:c
\bool_set_true:N 1987 \cs_new_protected_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\bool_set_true:c 1988 \cs_new_protected_nopar:Npn \bool_new:c #1 { \cs_new_eq:cN {#1} \c_false_bool }
\bool_set_false:N 1989 \cs_new_protected_nopar:Npn \bool_set_true:N #1 { \cs_set_eq:NN #1 \c_true_bool }
\bool_set_false:c 1990 \cs_new_protected_nopar:Npn \bool_set_true:c #1 { \cs_set_eq:cN {#1} \c_true_bool }
\bool_gset_true:N 1991 \cs_new_protected_nopar:Npn \bool_set_false:N #1 { \cs_set_eq:NN #1 \c_false_bool }
\bool_gset_true:c 1992 \cs_new_protected_nopar:Npn \bool_set_false:c #1 { \cs_set_eq:cN {#1} \c_false_bool }
\bool_gset_false:N 1993 \cs_new_protected_nopar:Npn \bool_gset_true:N #1 { \cs_gset_eq:NN #1 \c_true_bool }
\bool_gset_false:c 1994 \cs_new_protected_nopar:Npn \bool_gset_true:c #1 { \cs_gset_eq:cN {#1} \c_true_bool }
\bool_gset_false:c 1995 \cs_new_protected_nopar:Npn \bool_gset_false:N #1 { \cs_gset_eq:NN #1 \c_false_bool }
1996 \cs_new_protected_nopar:Npn \bool_gset_false:c #1 { \cs_gset_eq:cN {#1} \c_false_bool }

\bool_set_eq:NN Setting a boolean to another is also pretty easy.
\bool_set_eq:Nc
\bool_set_eq:cN 1997 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:cc 1998 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
1999 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
2000 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
2001 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
2002 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
2003 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
2004 \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc

```

`\l_tmpa_bool` A few booleans just if you need them.
`\g_tmpa_bool`

```

2005 \bool_new:N \l_tmpa_bool
2006 \bool_new:N \g_tmpa_bool

```

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just
`\bool_if_p:c` be input directly.
`\bool_if:N \underline{TF}`
`\bool_if:c \underline{TF}`

```

2007 \prg_set_conditional:Npnn \bool_if:N #1 {p,TF,T,F}{
2008   \if_bool:N #1 \prg_return_true: \else: \prg_return_false: \fi:
2009 }
2010 \cs_generate_variant:Nn \bool_if_p:N {c}
2011 \cs_generate_variant:Nn \bool_if:N {NTF {c}}
2012 \cs_generate_variant:Nn \bool_if:NT {c}
2013 \cs_generate_variant:Nn \bool_if:NF {c}

```

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The ‘while’
 \bool_while_do:cn version executes the code as long as the boolean is true; the ‘until’ version executes the
 \bool_until_do:Nn code as long as the boolean is false.
 \bool_until_do:cn

```

2014 \cs_new:Npn \bool_while_do:Nn #1 #2 {
2015   \bool_if:NT #1 {#2 \bool_while_do:Nn #1 {#2}}
2016 }
2017 \cs_generate_variant:Nn \bool_while_do:Nn {c}

2018 \cs_new:Npn \bool_until_do:Nn #1 #2 {
2019   \bool_if:NF #1 {#2 \bool_until_do:Nn #1 {#2}}
2020 }
2021 \cs_generate_variant:Nn \bool_until_do:Nn {c}

```

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested
 \bool_do_while:cn after executing the body. Otherwise identical to the above functions.
 \bool_do_until:Nn
 \bool_do_until:cn

```

2022 \cs_new:Npn \bool_do_while:Nn #1 #2 {
2023   #2 \bool_if:NT #1 {\bool_do_while:Nn #1 {#2}}
2024 }
2025 \cs_generate_variant:Nn \bool_do_while:Nn {c}

2026 \cs_new:Npn \bool_do_until:Nn #1 #2 {
2027   #2 \bool_if:NF #1 {\bool_do_until:Nn #1 {#2}}
2028 }
2029 \cs_generate_variant:Nn \bool_do_until:Nn {c}

```

98.6 Parsing boolean expressions

\bool_if_p:n Evaluating the truth value of a list of predicates is done using an input syntax somewhat
 \bool_if:nTF similar to the one found in other programming languages with (and) for grouping, !
 \bool_get_next:N for logical ‘Not’, && for logical ‘And’ and || for logical Or. We shall use the terms Not,
 \bool_cleanup:N And, Or, Open and Close for these operations.

\bool_choose:NN Any expression is terminated by a Close operation. Evaluation happens from left to right
 \bool_!:w in the following manner using a GetNext function:

```

\bool_Not:w
\bool_Not:w
\bool_(w
\bool_p:w
\bool_8_1:w
\bool_I_1:w
\bool_8_0:w
\bool_I_0:w
\bool_)_0:w
\bool_)_1:w
\bool_S_0:w
\bool_S_1:w

```

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.

- If a Not is seen, insert a negating function (if-even in this case) and call GetNext.
- If none of the above, start evaluating a new expression by reinserting the token found (this is supposed to be a predicate function) in front of Eval.

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:

$\langle true \rangle$ And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ And Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle false \rangle$.

$\langle true \rangle$ Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ Close Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ Close Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the following semantics:

$\langle true \rangle$ Stop Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ Stop Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\tex_number:D` operation. First we issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for T_EX. We also need to finish this special group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a `S` following the last Close operation.

```

2030 \cs_new:Npn \bool_if_p:n #1{
2031   \group_align_safe_begin:
2032   \bool_get_next:N ( #1 )S
2033 }
```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```

2034 \cs_new:Npn \bool_get_next:N #1{
2035   \use:c {
```



```

2036     bool_
2037     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2038     :w
2039   } #1
2040 }

```

This variant gets called when a NOT has just been entered. It (eventually) results in a reversal of the logic of the directly following material.

```

2041 \cs_new:Npn \bool_get_not_next:N #1{
2042   \use:c {
2043     bool_not_
2044     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2045     :w
2046   } #1
2047 }

```

We need these later on to nullify the unity operation !!.

```

2048 \cs_new:Npn \bool_get_next:NN #1#2{
2049   \bool_get_next:N #2
2050 }
2051 \cs_new:Npn \bool_get_not_next:NN #1#2{
2052   \bool_get_not_next:N #2
2053 }

```

The Not operation. Discard the token read and reverse the truth value of the next expression if there are brackets; otherwise if we're coming up to a ! then we don't need to reverse anything (but we then want to continue scanning ahead in case some fool has written !!(...)); otherwise we have a boolean that we can reverse here and now.

```

2054 \cs_new:cpn { bool_!:w } #1#2 {
2055   \if_meaning:w ( #2
2056     \exp_after:wN \bool_Not:w
2057   \else:
2058     \if_meaning:w ! #2
2059     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN
2060   \else:
2061     \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
2062   \fi:
2063   \fi:
2064   #2
2065 }

```

Variant called when already inside a NOT. Essentially the opposite of the above.

```

2066 \cs_new:cpn { bool_not_!:w } #1#2 {
2067   \if_meaning:w ( #2
2068     \exp_after:wN \bool_not_Not:w
2069   \else:
2070     \if_meaning:w ! #2

```

```

2071     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
2072   \else:
2073     \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
2074   \fi:
2075 \fi:
2076 #2
2077 }

```

These occur when processing `!(...)`. The idea is to use a variant of `\bool_get_next:N` that finishes its parsing with a logic reversal. Of course, the double logic reversal gets us back to where we started.

```

2078 \cs_new:Npn \bool_Not:w {
2079   \exp_after:wN \tex_number:D \bool_get_not_next:N
2080 }
2081 \cs_new:Npn \bool_not_Not:w {
2082   \exp_after:wN \tex_number:D \bool_get_next:N
2083 }

```

These occur when processing `!<bool>` and can be evaluated directly.

```

2084 \cs_new:Npn \bool_Not:N #1 {
2085   \exp_after:wN \bool_p:w
2086   \if_meaning:w #1 \c_true_bool
2087     \c_false_bool
2088   \else:
2089     \c_true_bool
2090   \fi:
2091 }
2092 \cs_new:Npn \bool_not_Not:N #1 {
2093   \exp_after:wN \bool_p:w
2094   \if_meaning:w #1 \c_true_bool
2095     \c_true_bool
2096   \else:
2097     \c_false_bool
2098   \fi:
2099 }

```

The Open operation. Discard the token read and start a sub-expression. `\bool_get_next:N` continues building up the logical expressions as usual; `\bool_not_cleanup:N` is what reverses the logic if we're inside `!(...)`.

```

2100 \cs_new:cpn {bool_( :w)}#1{
2101   \exp_after:wN \bool_cleanup:N \tex_number:D \bool_get_next:N
2102 }
2103 \cs_new:cpn {bool_not_( :w)}#1{
2104   \exp_after:wN \bool_not_cleanup:N \tex_number:D \bool_get_next:N
2105 }

```

Otherwise just evaluate the predicate and look for And, Or or Close afterward.

```

2106 \cs_new:cpn {bool_p:w}{\exp_after:wN \bool_cleanup:N \tex_number:D }
2107 \cs_new:cpn {bool_not_p:w}{\exp_after:wN \bool_not_cleanup:N \tex_number:D }

```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```

2108 \cs_new_nopar:Npn \bool_cleanup:N #1{
2109   \exp_after:wN \bool_choose:NN \exp_after:wN #1
2110   \int_to_roman:w-'\q
2111 }
2112 \cs_new_nopar:Npn \bool_not_cleanup:N #1{
2113   \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
2114   \int_to_roman:w-'\q
2115 }

```

Branching the six way switch. Reversals should be reasonably straightforward. When programming this, however, I got things around the wrong way a few times. (Will's hacks onto Morten's code, that is.)

```

2116 \cs_new_nopar:Npn \bool_choose:NN #1#2{ \use:c{bool_#2_#1:w} }
2117 \cs_new_nopar:Npn \bool_not_choose:NN #1#2{ \use:c{bool_not_#2_#1:w} }

```

Continues scanning. Must remove the second & or |.

```

2118 \cs_new_nopar:cpn{bool_&_1:w}&{\bool_get_next:N}
2119 \cs_new_nopar:cpn{bool_|_0:w}|{\bool_get_next:N}
2120 \cs_new_nopar:cpn{bool_not_&_0:w}&{\bool_get_next:N}
2121 \cs_new_nopar:cpn{bool_not_|_1:w}|{\bool_get_next:N}

```

Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```

2122 \cs_new_nopar:cpn{bool_)_0:w}{ \c_false_bool }
2123 \cs_new_nopar:cpn{bool_)_1:w}{ \c_true_bool }
2124 \cs_new_nopar:cpn{bool_not_)_0:w}{ \c_true_bool }
2125 \cs_new_nopar:cpn{bool_not_)_1:w}{ \c_false_bool }
2126 \cs_new_nopar:cpn{bool_S_0:w}{\group_align_safe_end: \c_false_bool }
2127 \cs_new_nopar:cpn{bool_S_1:w}{\group_align_safe_end: \c_true_bool }

```

When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the () manually.

```

2128 \cs_new:cpn{bool_&_0:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
2129 \cs_new:cpn{bool_|_1:w}|{\bool_eval_skip_to_end:Nw \c_true_bool}
2130 \cs_new:cpn{bool_not_&_1:w}&{\bool_eval_skip_to_end:Nw \c_false_bool}
2131 \cs_new:cpn{bool_not_|_0:w}|{\bool_eval_skip_to_end:Nw \c_true_bool}

```

```

\bool_eval_skip_to_end:Nw
\bool_eval_skip_to_end_aux:Nw
\bool_eval_skip_to_end_auxii:Nw

```

There is always at least one) waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to

detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool  && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a () pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an Open so we remove another () pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a Close and again find Open tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

This whole operation could be made a lot simpler if we were allowed to do simple pattern matching. With a new enough pdfTeX one can do that sort of thing to test for existence of particular tokens.

```
2132 \cs_new:Npn \bool_eval_skip_to_end:Nw #1#2){
2133   \bool_eval_skip_to_end_aux:Nw #1 #2(\q_no_value\q_stop{#2}
2134 }
```

If no right parenthesis, then #3 is no_value and we are done, return the boolean #1. If there is, we need to grab a () pair and then recurse

```
2135 \cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2(#3#4\q_stop#5{
2136   \quark_if_no_value:NTF #3
2137   { #1 }
2138   { \bool_eval_skip_to_end_auxii:Nw #1 #5 }
2139 }
```

keep the boolean, throw away anything up to the (as it is irrelevant, remove a () pair but remember to reinsert #3 as it may contain (tokens!

```
2140 \cs_new:Npn \bool_eval_skip_to_end_auxii:Nw #1#2(#3){
2141   \bool_eval_skip_to_end:Nw #1#3 )
2142 }
```

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning
`\bool_set:cn` `\c_true_bool` or `\c_false_bool`.

```
\bool_gset:Nn 2143 \cs_new:Npn \bool_set:Nn #1#2 {\tex_chardef:D #1 = \bool_if_p:n {#2}}
\bool_gset:cn 2144 \cs_new:Npn \bool_gset:Nn #1#2 {
2145   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2146 }
2147 \cs_generate_variant:Nn \bool_set:Nn {c}
2148 \cs_generate_variant:Nn \bool_gset:Nn {c}
```

`\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.

```
2149 \cs_new:Npn \bool_not_p:n #1{ \bool_if_p:n{!(#1)} }
```

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2150 \cs_new:Npn \bool_xor_p:nn #1#2 {
2151   \int_compare:nNnTF {\bool_if_p:n { #1 }} = {\bool_if_p:n { #2 }}
2152   {\c_false_bool}{\c_true_bool}
2153 }
```

```

2154 \prg_set_conditional:Npn \bool_if:n #1 {TF,T,F}{
2155   \if_predicate:w \bool_if_p:n{#1}
2156   \prg_return_true: \else: \prg_return_false: \fi:
2157 }

```

```

\bool_while_do:nn #1 : Predicate test
\bool_until_do:nn #2 : Code to execute
\bool_do_while:nn
\bool_do_until:nn
2158 \cs_new:Npn \bool_while_do:nn #1#2 {
2159   \bool_if:nT {#1} { #2 \bool_while_do:nn {#1}{#2} }
2160 }
2161 \cs_new:Npn \bool_until_do:nn #1#2 {
2162   \bool_if:nF {#1} { #2 \bool_until_do:nn {#1}{#2} }
2163 }
2164 \cs_new:Npn \bool_do_while:nn #1#2 {
2165   #2 \bool_if:nT {#1} { \bool_do_while:nn {#1}{#2} }
2166 }
2167 \cs_new:Npn \bool_do_until:nn #1#2 {
2168   #2 \bool_if:nF {#1} { \bool_do_until:nn {#1}{#2} }
2169 }

```

98.7 Case switch

\prg_case_int:nnn This case switch is in reality quite simple. It takes three arguments:
\prg_case_int_aux:nnn

1. An integer expression you wish to find.
2. A list of pairs of $\{\langle integer\ expr \rangle\} \{\langle code \rangle\}$. The list can be as long as is desired and $\langle integer\ expr \rangle$ can be negative.
3. The code to be executed if the value wasn't found.

We don't need the else case here yet, so leave it dangling in the input stream.

```

2170 \cs_new:Npn \prg_case_int:nnn #1 #2 {

```

We will be parsing on #1 for each step so we might as well evaluate it first in case it is complicated.

```

2171   \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n{#1}} #2

```

The ? below is just so there are enough arguments when we reach the end. And it made you look. ;-)

```

2172   \q_recursion_tail ? \q_recursion_stop
2173 }
2174 \cs_new:Npn \prg_case_int_aux:nnn #1#2#3{

```

If we reach the end, return the else case. We just remove braces.

```
2175 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
```

Otherwise we compare (which evaluates #2 for us)

```
2176 \int_compare:nNnTF{#1}={#2}
```

If true, we want to remove the remainder of the list, the else case and then execute the code specified. `\prg_end_case:nw {#3}` does just that in one go. This means `f` style expansion works the way one wants it to work.

```
2177 { \prg_end_case:nw {#3} }
2178 { \prg_case_int_aux:nnn {#1}}
2179 }
```

`\prg_case_dim:nnn` Same as `\prg_case_dim:nnn` except it is for *<dim>* registers.
`\prg_case_dim_aux:nnn`

```
2180 \cs_new:Npn \prg_case_dim:nnn #1 #2 {
2181   \exp_args:No \prg_case_dim_aux:nnn {\dim_eval:n{#1}} #2
2182   \q_recursion_tail ? \q_recursion_stop
2183 }
2184 \cs_new:Npn \prg_case_dim_aux:nnn #1#2#3{
2185   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2186   \dim_compare:nNnTF{#1}={#2}
2187   { \prg_end_case:nw {#3} }
2188   { \prg_case_dim_aux:nnn {#1}}
2189 }
```

`\prg_case_str:nnn` Same as `\prg_case_dim:nnn` except it is for strings.
`\prg_case_str_aux:nnn`

```
2190 \cs_new:Npn \prg_case_str:nnn #1 #2 {
2191   \prg_case_str_aux:nnn {#1} #2
2192   \q_recursion_tail ? \q_recursion_stop
2193 }
2194 \cs_new:Npn \prg_case_str_aux:nnn #1#2#3{
2195   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2196   \str_if_eq:xxTF{#1}{#2}
2197   { \prg_end_case:nw {#3} }
2198   { \prg_case_str_aux:nnn {#1}}
2199 }
```

`\prg_case_tl:Nnn` Same as `\prg_case_dim:nnn` except it is for token list variables.
`\prg_case_tl_aux:NNn`

```
2200 \cs_new:Npn \prg_case_tl:Nnn #1 #2 {
2201   \prg_case_tl_aux:NNn #1 #2
2202   \q_recursion_tail ? \q_recursion_stop
2203 }
2204 \cs_new:Npn \prg_case_tl_aux:NNn #1#2#3{
2205   \quark_if_recursion_tail_stop_do:Nn #2{\use:n}
```

```

2206 \tl_if_eq:NNTF #1 #2
2207 { \prg_end_case:nw {#3} }
2208 { \prg_case_tl_aux:NNn #1}
2209 }

```

`\prg_end_case:nw` Ending a case switch is always performed the same way so we optimize for this. `#1` is the code to execute, `#2` the remainder, and `#3` the dangling else case.

```

2210 \cs_new:Npn \prg_end_case:nw #1#2\q_recursion_stop#3{#1}

```

98.8 Sorting

`\prg_define_quicksort:nnn` `#1` is the name, `#2` and `#3` are the tokens enclosing the argument. For the somewhat strange `<clist>` type which doesn't enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```

\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}

```

defines the user function `\seq_quicksort:n` and furthermore expects to use the two functions `\seq_quicksort_compare:nnTF` which compares the items and `\seq_quicksort_function:n` which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the `seq` type a sequence is a token list variable, so one additionally has to define

```

\cs_set_nopar:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}

```

For details on the implementation see “Sorting in T_EX’s Mouth” by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```

2211 \cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
2212   \cs_set:cpx{#1_quicksort:n}##1{
2213     \exp_not:c{#1_quicksort_start_partition:w} ##1
2214     \exp_not:n{#2\q_nil#3\q_stop}
2215   }
2216   \cs_set:cpx{#1_quicksort_braced:n}##1{
2217     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
2218     \exp_not:N\q_nil\exp_not:N\q_stop
2219   }
2220   \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2221     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2222     \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
2223   }
2224   \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2225     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2226     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
2227   }

```


Now for doing the partitions.

```

2228 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2229   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2230   {
2231     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2232     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2233     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2234   }
2235   {##1}{##2}{##3}{##4}
2236 }
2237 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2238   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2239   {
2240     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2241     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2242     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2243   }
2244   {##1}{##2}{##3}{##4}
2245 }
2246 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2247   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2248   {
2249     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2250     \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2251     \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2252   }
2253   {##1}{##2}{##3}{##4}
2254 }
2255 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2256   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnw}
2257   {
2258     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2259     \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2260     \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2261   }
2262   {##1}{##2}{##3}{##4}
2263 }

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2264 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2265   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2266 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2267   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2268 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2269   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2270 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2271   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}

```

```

2272 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2273   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##3}
2274 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2275   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3{##4}}}
2276 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2277   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}##2}{##3}
2278 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2279   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2{##4}}{##3}}

```

Finally, the big kahuna! This is where the sub-lists are sorted.

```

2280 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2281   \exp_not:c{#1_quicksort_braced:n}{##2}
2282   \exp_not:c{#1_quicksort_function:n}{##1}
2283   \exp_not:c{#1_quicksort_braced:n}{##3}
2284 }
2285 }

```

`\prg_quicksort:n` A simple version. Sorts a list of tokens, uses the function `\prg_quicksort_compare:nnTF` to compare items, and places the function `\prg_quicksort_function:n` in front of each of them.

```

2286 \prg_define_quicksort:nnn {prg}{f}{f}

```

`\prg_quicksort_function:n`
`\prg_quicksort_compare:nnTF`

```

2287 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2288 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}

```

98.9 Variable type and scope

`\prg_variable_get_scope:N` Expandable functions to find the type of a variable, and to return g if the variable is global. The trick for `\prg_variable_get_scope:N` is the same as that in `\cs_split_function:NN`, but it can be simplified as the requirements here are less complex.

`\prg_variable_get_scope_aux:w`
`\prg_variable_get_type:N`
`\prg_variable_get_type:w`

```

2289 \group_begin:
2290   \tex_lccode:D '\& = '\g \tex_relax:D
2291   \tex_catcode:D '\& = \c_twelve \tex_relax:D
2292   \tl_to_lowercase:n {
2293     \group_end:
2294     \cs_new_nopar:Npn \prg_variable_get_scope:N #1 {
2295       \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
2296       { \cs_to_str:N #1 \exp_stop_f: \q_stop }
2297     }
2298     \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop {
2299       \token_if_eq_meaning:NNT & #1 {g}
2300     }
2301   }

```

```

2302 \group_begin:
2303   \tex_lccode:D '\& = '\_ \tex_relax:D
2304   \tex_catcode:D '\& = \c_twelve \tex_relax:D
2305   \tl_to_lowercase:n {
2306     \group_end:
2307     \cs_new_nopar:Npn \prg_variable_get_type:N #1 {
2308       \exp_after:wN \p;rg_variable_get_type_aux:w
2309       \token_to_str:N #1 & a \q_stop
2310     }
2311     \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop {
2312       \token_if_eq_meaning:NNTF a #2 {
2313         #1
2314       }{
2315         \prg_variable_get_type_aux:w #2#3 \q_stop
2316       }
2317     }
2318   }

```

98.10 Mapping to variables

`\prg_new_map_functions:Nn` The idea here is to generate all of the various mapping functions in one go. Everything is done with expansion so that the performance hit is taken at definition time and not at point of use. The inline version uses a counter as this keeps things nestable, and global to avoid problems with, for example, table cells.

`\prg_set_map_functions:Nn`

```

2319 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 {
2320   \cs_if_free:cTF { #2 _map_function:NN }
2321   { \prg_set_map_functions:Nn #1 {#2} }
2322   {
2323     \msg_kernel_error:nxx { code } { csname-already-defined }
2324     { \token_to_str:c { #2 _map_function:NN } }
2325   }
2326 }
2327 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 {
2328   \cs_gset_nopar:cpx { #2 _map_function:NN } ##1##2
2329   {
2330     \exp_not:N \tl_if_empty:NF ##1
2331     {
2332       \exp_not:N \exp_after:wN
2333       \exp_not:c { #2 _map_function_aux:Nw }
2334       \exp_not:N \exp_after:wN ##2 ##1
2335       \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2336     }
2337   }
2338   \cs_gset:cpx { #2 _map_function:nN } ##1##2
2339   {
2340     \exp_not:N \tl_if_blank:nF {##1}
2341     {
2342       \exp_not:c { #2 _map_function_aux:Nw } ##2 ##1

```

```

2343         \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2344     }
2345 }
2346 \cs_gset:cpx { #2 _map_function_aux:Nw } ##1##2 #1
2347 {
2348     \exp_not:N \quark_if_recursion_tail_stop:n {##2}
2349     ##1 {##2}
2350     \exp_not:c { #2 _map_function_aux:Nw } ##1
2351 }
2352 \cs_if_free:cT { g_ #2 _map_inline_int }
2353 { \int_new:c { g_ #2 _map_inline_int } }
2354 \cs_gset_protected_nopar:cpx { #2 _map_inline:Nn } ##1##2
2355 {
2356     \exp_not:N \tl_if_empty:NF ##1
2357     {
2358         \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2359         \cs_gset:cpn
2360         {
2361             #2 _map_inline_
2362             \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2363             :n
2364         }
2365         #####1 {##2}
2366         \exp_not:N \exp_last_unbraced:NcV
2367         \exp_not:c { #2 _map_function_aux:Nw }
2368         {
2369             #2 _map_inline_
2370             \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2371             :n
2372         }
2373         ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2374         \exp_not:N \int_gdecr:N \exp_not:c { g_ #2 _map_inline_int }
2375     }
2376 }
2377 \cs_gset_protected:cpx { #2 _map_inline:nn } ##1##2
2378 {
2379     \exp_not:N \tl_if_empty:nF {##1}
2380     {
2381         \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2382         \cs_gset:cpn
2383         {
2384             #2 _map_inline_
2385             \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2386             :n
2387         }
2388         #####1 {##2}
2389         \exp_not:N \exp_args:Nc
2390         \exp_not:c { #2 _map_function_aux:Nw }
2391         {
2392             #2 _map_inline_

```

```

2393         \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2394         :n
2395     }
2396     ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2397     \exp_not:N \int_gdecr:N \exp_not:c { g_ #2 _map_inline_int }
2398 }
2399 }
2400 \cs_gset_eq:cN { #2 _map_break: }
2401 \use_none_delimit_by_q_recursion_stop:w
2402 }

```

That's it (for now).

```

2403 </initex | package>

2404 <*showmemory>
2405 \showMemUsage
2406 </showmemory>

```

99 l3quark implementation

We start by ensuring that the required packages are loaded. We check for `l3expan` since this a basic package that is essential for use of any higher-level package.

```

2407 <*package>
2408 \ProvidesExplPackage
2409   {\filename}{\filedate}{\fileversion}{\filedescription}
2410 \package_check_loaded_expl:
2411 </package>
2412 <*initex | package>

```

`\quark_new:N` Allocate a new quark.

```

2413 \cs_new_protected_nopar:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }

```

`\q_stop` `\q_stop` is often used as a marker in parameter text, `\q_no_value` is the canonical missing value, and `\q_nil` represents a nil pointer in some data structures.

`\q_nil`

```

2414 \quark_new:N \q_stop
2415 \quark_new:N \q_no_value
2416 \quark_new:N \q_nil

```

`\q_error` We need two additional quarks. `\q_error` delimits the end of the computation for purposes of error recovery. `\q_mark` is used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

`\q_mark`

```

2417 \quark_new:N\q_error
2418 \quark_new:N\q_mark

```

`\q_recursion_tail` 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.

```
2419 \quark_new:N\q_recursion_tail
2420 \quark_new:N\q_recursion_stop
```

`\quark_if_recursion_tail_stop:n` When doing recursions it is easy to spend a lot of time testing if we found the end marker.
`\quark_if_recursion_tail_stop:N` To avoid this, we use a recursion end marker every time we do this kind of task. Also, if
`\quark_if_recursion_tail_stop:o` the recursion end marker is found, we wrap things up and finish.

```
2421 \cs_new:Npn \quark_if_recursion_tail_stop:n #1 {
2422   \exp_after:wN\if_meaning:w
2423     \quark_if_recursion_tail_aux:w #1?\q_stop\q_recursion_tail\q_recursion_tail
2424     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2425   \fi:
2426 }
2427 \cs_new:Npn \quark_if_recursion_tail_stop:N #1 {
2428   \if_meaning:w#1\q_recursion_tail
2429     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2430   \fi:
2431 }
2432 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n {o}
```

`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:Nn`
`\quark_if_recursion_tail_stop_do:on`

```
2433 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2 {
2434   \exp_after:wN\if_meaning:w
2435     \quark_if_recursion_tail_aux:w #1?\q_stop\q_recursion_tail\q_recursion_tail
2436     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2437   \else:
2438     \exp_after:wN\use_none:n
2439   \fi:
2440   {#2}
2441 }
2442 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2 {
2443   \if_meaning:w #1\q_recursion_tail
2444     \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2445   \else:
2446     \exp_after:wN\use_none:n
2447   \fi:
2448   {#2}
2449 }
2450 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn {on}
```

`\quark_if_recursion_tail_aux:w`

```
2451 \cs_new:Npn \quark_if_recursion_tail_aux:w #1#2 \q_stop \q_recursion_tail {#1}
```

`\quark_if_no_value_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_no_value_p:n` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value:NTF` wrongly given a string like aabc instead of a single token.¹⁰
`\quark_if_no_value:nTF`

```

2452 \prg_new_conditional:Nnn \quark_if_no_value:N {p,TF,T,F} {
2453   \if_meaning:w \q_no_value #1
2454   \prg_return_true: \else: \prg_return_false: \fi:
2455 }

```

These tests are easy with `\pdf_strcmp:D` available.

```

2456 \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
2457   \if_num:w \pdf_strcmp:D
2458   {\exp_not:N \q_no_value}
2459   {\exp_not:n{#1}} = \c_zero
2460   \prg_return_true: \else: \prg_return_false:
2461   \fi:
2462 }

```

`\quark_if_nil_p:N` A function to check for the presence of `\q_nil`.

`\quark_if_nil:NTF`

```

2463 \prg_new_conditional:Nnn \quark_if_nil:N {p,TF,T,F} {
2464   \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi:
2465 }

```

`\quark_if_nil_p:n` A function to check for the presence of `\q_nil`.

`\quark_if_nil_p:V`
`\quark_if_nil_p:o`
`\quark_if_nil:nTF`
`\quark_if_nil:VTF`
`\quark_if_nil:oTF`

```

2466 \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
2467   \if_num:w \pdf_strcmp:D
2468   {\exp_not:N \q_nil}
2469   {\exp_not:n{#1}} = \c_zero
2470   \prg_return_true: \else: \prg_return_false:
2471   \fi:
2472 }
2473 \cs_generate_variant:Nn \quark_if_nil_p:n {V}
2474 \cs_generate_variant:Nn \quark_if_nil:nTF {V}
2475 \cs_generate_variant:Nn \quark_if_nil:nT {V}
2476 \cs_generate_variant:Nn \quark_if_nil:nF {V}
2477 \cs_generate_variant:Nn \quark_if_nil_p:n {o}
2478 \cs_generate_variant:Nn \quark_if_nil:nTF {o}
2479 \cs_generate_variant:Nn \quark_if_nil:nT {o}
2480 \cs_generate_variant:Nn \quark_if_nil:nF {o}

```

Show token usage:

```

2481
2482 < *showmemory >
2483 \showMemUsage
2484 < /showmemory >

```

¹⁰It may still loop in special circumstances however!

100 l3token implementation

100.1 Documentation of internal functions

`\l_peek_true_tl`
`\l_peek_false_tl`

These token list variables are used internally when choosing either the true or false branches of a test.

`\l_peek_search_tl`

Used to store `\l_peek_search_token`.

`\peek_tmp:w`

Scratch function used to gobble tokens from the input stream.

`\l_peek_true_aux_tl`
`\c_peek_true_remove_next_tl`

These token list variables are used internally when choosing either the true or false branches of a test.

`\peek_ignore_spaces_execute_branches:`
`\peek_ignore_spaces_aux:`

Functions used to ignore space tokens in the input stream.

100.2 Module code

First a few required packages to get this going.

```
2485 <*package>
2486 \ProvidesExplPackage
2487   {\filename}{\filedate}{\fileversion}{\filedescription}
2488 \package_check_loaded_expl:
2489 </package>
2490 <*initex | package>
```

100.3 Character tokens

```
\char_set_catcode:w
\char_set_catcode:nn
\char_value_catcode:w
\char_value_catcode:n
\char_show_value_catcode:w
\char_show_value_catcode:n
2491 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
2492 \cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2 {
```



```

2493 \char_set_catcode:w #1 = \int_eval:w #2\int_eval_end:
2494 }
2495 \cs_new_nopar:Npn \char_value_catcode:w { \int_use:N \tex_catcode:D }
2496 \cs_new_nopar:Npn \char_value_catcode:n #1 {
2497   \char_value_catcode:w \int_eval:w #1\int_eval_end:
2498 }
2499 \cs_new_nopar:Npn \char_show_value_catcode:w {
2500   \tex_showthe:D \tex_catcode:D
2501 }
2502 \cs_new_nopar:Npn \char_show_value_catcode:n #1 {
2503   \char_show_value_catcode:w \int_eval:w #1\int_eval_end:
2504 }

```

```

\char_make_escape:N
\char_make_begin_group:N
\char_make_end_group:N
\char_make_math_shift:N
\char_make_alignment:N
\char_make_end_line:N
\char_make_parameter:N
\char_make_math_superscript:N
\char_make_math_subscript:N
\char_make_ignore:N
\char_make_space:N
\char_make_letter:N
\char_make_other:N
\char_make_active:N
\char_make_comment:N
\char_make_invalid:N

```

```

2505 \cs_new_protected_nopar:Npn \char_make_escape:N #1 { \char_set_catcode:nn {'#1} {\c
2506 \cs_new_protected_nopar:Npn \char_make_begin_group:N #1 { \char_set_catcode:nn {'#1} {\c
2507 \cs_new_protected_nopar:Npn \char_make_end_group:N #1 { \char_set_catcode:nn {'#1} {\c
2508 \cs_new_protected_nopar:Npn \char_make_math_shift:N #1 { \char_set_catcode:nn {'#1} {\c
2509 \cs_new_protected_nopar:Npn \char_make_alignment:N #1 { \char_set_catcode:nn {'#1} {\c
2510 \cs_new_protected_nopar:Npn \char_make_end_line:N #1 { \char_set_catcode:nn {'#1} {\c
2511 \cs_new_protected_nopar:Npn \char_make_parameter:N #1 { \char_set_catcode:nn {'#1} {\c
2512 \cs_new_protected_nopar:Npn \char_make_math_superscript:N #1 { \char_set_catcode:nn {'#1} {\c
2513 \cs_new_protected_nopar:Npn \char_make_math_subscript:N #1 { \char_set_catcode:nn {'#1} {\c
2514 \cs_new_protected_nopar:Npn \char_make_ignore:N #1 { \char_set_catcode:nn {'#1} {\c
2515 \cs_new_protected_nopar:Npn \char_make_space:N #1 { \char_set_catcode:nn {'#1} {\c
2516 \cs_new_protected_nopar:Npn \char_make_letter:N #1 { \char_set_catcode:nn {'#1} {\c
2517 \cs_new_protected_nopar:Npn \char_make_other:N #1 { \char_set_catcode:nn {'#1} {\c
2518 \cs_new_protected_nopar:Npn \char_make_active:N #1 { \char_set_catcode:nn {'#1} {\c
2519 \cs_new_protected_nopar:Npn \char_make_comment:N #1 { \char_set_catcode:nn {'#1} {\c
2520 \cs_new_protected_nopar:Npn \char_make_invalid:N #1 { \char_set_catcode:nn {'#1} {\c

```

```

\char_make_escape:n
\char_make_begin_group:n
\char_make_end_group:n
\char_make_math_shift:n
\char_make_alignment:n
\char_make_end_line:n
\char_make_parameter:n
\char_make_math_superscript:n
\char_make_math_subscript:n
\char_make_ignore:n
\char_make_space:n
\char_make_letter:n
\char_make_other:n
\char_make_active:n
\char_make_comment:n
\char_make_invalid:n

```

```

2521 \cs_new_protected_nopar:Npn \char_make_escape:n #1 { \char_set_catcode:nn {#1} {\c
2522 \cs_new_protected_nopar:Npn \char_make_begin_group:n #1 { \char_set_catcode:nn {#1} {\c
2523 \cs_new_protected_nopar:Npn \char_make_end_group:n #1 { \char_set_catcode:nn {#1} {\c
2524 \cs_new_protected_nopar:Npn \char_make_math_shift:n #1 { \char_set_catcode:nn {#1} {\c
2525 \cs_new_protected_nopar:Npn \char_make_alignment:n #1 { \char_set_catcode:nn {#1} {\c
2526 \cs_new_protected_nopar:Npn \char_make_end_line:n #1 { \char_set_catcode:nn {#1} {\c
2527 \cs_new_protected_nopar:Npn \char_make_parameter:n #1 { \char_set_catcode:nn {#1} {\c
2528 \cs_new_protected_nopar:Npn \char_make_math_superscript:n #1 { \char_set_catcode:nn {#1} {\c
2529 \cs_new_protected_nopar:Npn \char_make_math_subscript:n #1 { \char_set_catcode:nn {#1} {\c
2530 \cs_new_protected_nopar:Npn \char_make_ignore:n #1 { \char_set_catcode:nn {#1} {\c
2531 \cs_new_protected_nopar:Npn \char_make_space:n #1 { \char_set_catcode:nn {#1} {\c
2532 \cs_new_protected_nopar:Npn \char_make_letter:n #1 { \char_set_catcode:nn {#1} {\c
2533 \cs_new_protected_nopar:Npn \char_make_other:n #1 { \char_set_catcode:nn {#1} {\c
2534 \cs_new_protected_nopar:Npn \char_make_active:n #1 { \char_set_catcode:nn {#1} {\c
2535 \cs_new_protected_nopar:Npn \char_make_comment:n #1 { \char_set_catcode:nn {#1} {\c
2536 \cs_new_protected_nopar:Npn \char_make_invalid:n #1 { \char_set_catcode:nn {#1} {\c

```

```

\char_set_mathcode:w Math codes.
\char_set_mathcode:nn
\char_gset_mathcode:w
\char_gset_mathcode:nn
\char_value_mathcode:w
\char_value_mathcode:n
\char_show_value_mathcode:w
\char_show_value_mathcode:n

2537 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
2538 \cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2 {
2539   \char_set_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2540 }
2541 \cs_new_protected_nopar:Npn \char_gset_mathcode:w { \pref_global:D \tex_mathcode:D }
2542 \cs_new_protected_nopar:Npn \char_gset_mathcode:nn #1#2 {
2543   \char_gset_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2544 }
2545 \cs_new_nopar:Npn \char_value_mathcode:w { \int_use:N \tex_mathcode:D }
2546 \cs_new_nopar:Npn \char_value_mathcode:n #1 {
2547   \char_value_mathcode:w \int_eval:w #1\int_eval_end:
2548 }
2549 \cs_new_nopar:Npn \char_show_value_mathcode:w { \tex_showthe:D \tex_mathcode:D }
2550 \cs_new_nopar:Npn \char_show_value_mathcode:n #1 {
2551   \char_show_value_mathcode:w \int_eval:w #1\int_eval_end:
2552 }

\char_set_lccode:w
\char_set_lccode:nn
\char_value_lccode:w
\char_value_lccode:n
\char_show_value_lccode:w
\char_show_value_lccode:n

2553 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
2554 \cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2{
2555   \char_set_lccode:w #1 = \int_eval:w #2\int_eval_end:
2556 }
2557 \cs_new_nopar:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
2558 \cs_new_nopar:Npn \char_value_lccode:n #1{\char_value_lccode:w
2559   \int_eval:w #1\int_eval_end:}
2560 \cs_new_nopar:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}
2561 \cs_new_nopar:Npn \char_show_value_lccode:n #1{
2562   \char_show_value_lccode:w \int_eval:w #1\int_eval_end:}

\char_set_uccode:w
\char_set_uccode:nn
\char_value_uccode:w
\char_value_uccode:n
\char_show_value_uccode:w
\char_show_value_uccode:n

2563 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
2564 \cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2{
2565   \char_set_uccode:w #1 = \int_eval:w #2\int_eval_end:
2566 }
2567 \cs_new_nopar:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}
2568 \cs_new_nopar:Npn \char_value_uccode:n #1{\char_value_uccode:w
2569   \int_eval:w #1\int_eval_end:}
2570 \cs_new_nopar:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
2571 \cs_new_nopar:Npn \char_show_value_uccode:n #1{
2572   \char_show_value_uccode:w \int_eval:w #1\int_eval_end:}

\char_set_sfcode:w
\char_set_sfcode:nn
\char_value_sfcode:w
\char_value_sfcode:n
\char_show_value_sfcode:w
\char_show_value_sfcode:n

2573 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
2574 \cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2 {
2575   \char_set_sfcode:w #1 = \int_eval:w #2\int_eval_end:

```

```

2576 }
2577 \cs_new_nopar:Npn \char_value_sfcode:w { \int_use:N \tex_sfcode:D }
2578 \cs_new_nopar:Npn \char_value_sfcode:n #1 {
2579   \char_value_sfcode:w \int_eval:w #1\int_eval_end:
2580 }
2581 \cs_new_nopar:Npn \char_show_value_sfcode:w { \tex_showthe:D \tex_sfcode:D }
2582 \cs_new_nopar:Npn \char_show_value_sfcode:n #1 {
2583   \char_show_value_sfcode:w \int_eval:w #1\int_eval_end:
2584 }

```

100.4 Generic tokens

`\token_new:Nn` Creates a new token.

```

2585 \cs_new_protected_nopar:Npn \token_new:Nn #1#2 {\cs_new_eq:NN #1#2}

```

`\c_group_begin_token` `\c_group_end_token` `\c_math_shift_token` `\c_alignment_tab_token` `\c_parameter_token` `\c_math_superscript_token` `\c_math_subscript_token` `\c_space_token` `\c_letter_token` `\c_other_char_token` `\c_active_char_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

2586 \cs_new_eq:NN \c_group_begin_token {
2587 \cs_new_eq:NN \c_group_end_token }
2588 \group_begin:
2589 \char_set_catcode:nn{'\*}{3}
2590 \token_new:Nn \c_math_shift_token {*}
2591 \char_set_catcode:nn{'\*}{4}
2592 \token_new:Nn \c_alignment_tab_token {*}
2593 \token_new:Nn \c_parameter_token {#}
2594 \token_new:Nn \c_math_superscript_token {^}
2595 \char_set_catcode:nn{'\*}{8}
2596 \token_new:Nn \c_math_subscript_token {*}
2597 \token_new:Nn \c_space_token {~}
2598 \token_new:Nn \c_letter_token {a}
2599 \token_new:Nn \c_other_char_token {1}
2600 \char_set_catcode:nn{'\*}{13}
2601 \cs_gset_nopar:Npn \c_active_char_token {\exp_not:N*}
2602 \group_end:

```

`\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.

```

2603 \prg_new_conditional:Nnn \token_if_group_begin:N {p,TF,T,F} {
2604   \if_catcode:w \exp_not:N #1\c_group_begin_token
2605   \prg_return_true: \else: \prg_return_false: \fi:
2606 }

```

`\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.

```

2607 \prg_new_conditional:Nnn \token_if_group_end:N {p,TF,T,F} {

```

```

2608 \if_catcode:w \exp_not:N #1\c_group_end_token
2609 \prg_return_true: \else: \prg_return_false: \fi:
2610 }

```

`\token_if_math_shift_p:N` Check if token is a math shift token. We use the constant `\c_math_shift_token` for
`\token_if_math_shift:NTF` this.

```

2611 \prg_new_conditional:Nnn \token_if_math_shift:N {p,TF,T,F} {
2612 \if_catcode:w \exp_not:N #1\c_math_shift_token
2613 \prg_return_true: \else: \prg_return_false: \fi:
2614 }

```

`\token_if_alignment_tab_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token`
`\token_if_alignment_tab:NTF` for this.

```

2615 \prg_new_conditional:Nnn \token_if_alignment_tab:N {p,TF,T,F} {
2616 \if_catcode:w \exp_not:N #1\c_alignment_tab_token
2617 \prg_return_true: \else: \prg_return_false: \fi:
2618 }

```

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:NTF` We have to trick T_EX a bit to avoid an error message.

```

2619 \prg_new_conditional:Nnn \token_if_parameter:N {p,TF,T,F} {
2620 \exp_after:wN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
2621 \prg_return_true: \else: \prg_return_false: \fi:
2622 }

```

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token`
`\token_if_math_superscript:NTF` for this.

```

2623 \prg_new_conditional:Nnn \token_if_math_superscript:N {p,TF,T,F} {
2624 \if_catcode:w \exp_not:N #1\c_math_superscript_token
2625 \prg_return_true: \else: \prg_return_false: \fi:
2626 }

```

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token`
`\token_if_math_subscript:NTF` for this.

```

2627 \prg_new_conditional:Nnn \token_if_math_subscript:N {p,TF,T,F} {
2628 \if_catcode:w \exp_not:N #1\c_math_subscript_token
2629 \prg_return_true: \else: \prg_return_false: \fi:
2630 }

```

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.
`\token_if_space:NTF`

```

2631 \prg_new_conditional:Nnn \token_if_space:N {p,TF,T,F} {
2632 \if_catcode:w \exp_not:N #1\c_space_token
2633 \prg_return_true: \else: \prg_return_false: \fi:
2634 }

```

<code>\token_if_letter_p:N</code> <code>\token_if_letter:N\underline{TF}</code>	<p>Check if token is a letter token. We use the constant <code>\c_letter_token</code> for this.</p> <pre> 2635 \prg_new_conditional:Nnn \token_if_letter:N {p,TF,T,F} { 2636 \if_catcode:w \exp_not:N #1\c_letter_token 2637 \prg_return_true: \else: \prg_return_false: \fi: 2638 } </pre>
<code>\token_if_other_char_p:N</code> <code>\token_if_other_char:N\underline{TF}</code>	<p>Check if token is an other char token. We use the constant <code>\c_other_char_token</code> for this.</p> <pre> 2639 \prg_new_conditional:Nnn \token_if_other_char:N {p,TF,T,F} { 2640 \if_catcode:w \exp_not:N #1\c_other_char_token 2641 \prg_return_true: \else: \prg_return_false: \fi: 2642 } </pre>
<code>\token_if_active_char_p:N</code> <code>\token_if_active_char:N\underline{TF}</code>	<p>Check if token is an active char token. We use the constant <code>\c_active_char_token</code> for this.</p> <pre> 2643 \prg_new_conditional:Nnn \token_if_active_char:N {p,TF,T,F} { 2644 \if_catcode:w \exp_not:N #1\c_active_char_token 2645 \prg_return_true: \else: \prg_return_false: \fi: 2646 } </pre>
<code>\token_if_eq_meaning_p:NN</code> <code>\token_if_eq_meaning:NN\underline{TF}</code>	<p>Check if the tokens #1 and #2 have same meaning.</p> <pre> 2647 \prg_new_conditional:Nnn \token_if_eq_meaning:NN {p,TF,T,F} { 2648 \if_meaning:w #1 #2 2649 \prg_return_true: \else: \prg_return_false: \fi: 2650 } </pre>
<code>\token_if_eq_catcode_p:NN</code> <code>\token_if_eq_catcode:NN\underline{TF}</code>	<p>Check if the tokens #1 and #2 have same category code.</p> <pre> 2651 \prg_new_conditional:Nnn \token_if_eq_catcode:NN {p,TF,T,F} { 2652 \if_catcode:w \exp_not:N #1 \exp_not:N #2 2653 \prg_return_true: \else: \prg_return_false: \fi: 2654 } </pre>
<code>\token_if_eq_charcode_p:NN</code> <code>\token_if_eq_charcode:NN\underline{TF}</code>	<p>Check if the tokens #1 and #2 have same character code.</p> <pre> 2655 \prg_new_conditional:Nnn \token_if_eq_charcode:NN {p,TF,T,F} { 2656 \if_charcode:w \exp_not:N #1 \exp_not:N #2 2657 \prg_return_true: \else: \prg_return_false: \fi: 2658 } </pre>
<code>\token_if_macro_p:N</code> <code>\token_if_macro:N\underline{TF}</code> <code>\token_if_macro_p_aux:w</code>	<p>When a token is a macro, <code>\token_to_meaning:N</code> will always output something like <code>\long macro:#1->#1</code> so we simply check to see if the meaning contains <code>-></code>. Argument #2 in the code below will be empty if the string <code>-></code> isn't present, proof that the token was not a macro (which is why we reverse the emptiness test). However this function will fail</p>

on its own auxiliary function (and a few other private functions as well) but that should certainly never be a problem!

```

2659 \prg_new_conditional:Nnn \token_if_macro:N {p,TF,T,F} {
2660   \exp_after:wN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_stop
2661 }
2662 \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 -> #2 \q_stop{
2663   \if_predicate:w \tl_if_empty_p:n{#2}
2664   \prg_return_false: \else: \prg_return_true: \fi:
2665 }

```

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. We use `\scan_stop:` for this.
`\token_if_cs:NTF`

```

2666 \prg_new_conditional:Nnn \token_if_cs:N {p,TF,T,F} {
2667   \if_predicate:w \token_if_eq_catcode_p:NN \scan_stop: #1
2668   \prg_return_true: \else: \prg_return_false: \fi:

```

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert
`\token_if_expandable:N` `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable.

```

2669 \prg_new_conditional:Nnn \token_if_expandable:N {p,TF,T,F} {
2670   \cs_if_exist:NTF #1 {
2671     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2672     \prg_return_false: \else: \prg_return_true: \fi:
2673   } {
2674     \prg_return_false:
2675   }
2676 }

```

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to
`\token_if_mathchardef_p:N` do some checkups on which characters are output by `\token_to_meaning:N`. As usual,
`\token_if_int_register_p:N` these characters have catcode 12 so we must do some serious substitutions in the code
`\token_if_skip_register_p:N` below...
`\token_if_dim_register_p:N`

```

2677 \group_begin:
2678 \char_set_lccode:nn {'\T}{'\T}
2679 \char_set_lccode:nn {'\F}{'\F}
2680 \char_set_lccode:nn {'\X}{'\n}
2681 \char_set_lccode:nn {'\Y}{'\t}
2682 \char_set_lccode:nn {'\Z}{'\d}
2683 \char_set_lccode:nn {'\?}{'\}
2684 \tl_map_inline:nn{\X\Y\Z\M\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
2685   {\char_set_catcode:nn {'#1}{12}}

```

`\token_if_protected_macro_p:N` We convert the token list to lowercase and restore the catcode and lowercase code changes.
`\token_if_protected_long_macro_p:N`

```

2686 \tl_to_lowercase:n{
2687   \group_end:

```

`\token_if_dim_register:N`
`\token_if_skip_register:N`
`\token_if_int_register:N`
`\token_if_toks_register:N`
`\token_if_chardef_p_aux:w`
`\token_if_mathchardef_p_aux:w`
`\token_if_int_register_p_aux:w`
`\token_if_skip_register_p_aux:w`
`\token_if_dim_register_p_aux:w`
`\token_if_toks_register_p_aux:w`
`\token_if_protected_macro_p_aux:w`
`\token_if_protected_long_macro_p_aux:w`

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since \TeX thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`.

```

2688 \prg_new_conditional:Nnn \token_if_chardef:N {p,TF,T,F} {
2689   \exp_after:wN \token_if_chardef_aux:w
2690   \token_to_meaning:N #1?CHAR"\q_stop
2691 }
2692 \cs_new_nopar:Npn \token_if_chardef_aux:w #1?CHAR"#2\q_stop{
2693   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2694 }

2695 \prg_new_conditional:Nnn \token_if_mathchardef:N {p,TF,T,F} {
2696   \exp_after:wN \token_if_mathchardef_aux:w
2697   \token_to_meaning:N #1?MAYHCHAR"\q_stop
2698 }
2699 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1?MAYHCHAR"#2\q_stop{
2700   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2701 }

```

Integer registers are a little more difficult since they expand to `\count<number>` and there is also a primitive `\countdef`. So we have to check for that primitive as well.

```

2702 \prg_new_conditional:Nnn \token_if_int_register:N {p,TF,T,F} {
2703   \if_meaning:w \tex_countdef:D #1
2704   \prg_return_false:
2705   \else:
2706     \exp_after:wN \token_if_int_register_aux:w
2707     \token_to_meaning:N #1?COUXY\q_stop
2708   \fi:
2709 }
2710 \cs_new_nopar:Npn \token_if_int_register_aux:w #1?COUXY#2\q_stop{
2711   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2712 }

```

Skip registers are done the same way as the integer registers.

```

2713 \prg_new_conditional:Nnn \token_if_skip_register:N {p,TF,T,F} {
2714   \if_meaning:w \tex_skipdef:D #1
2715   \prg_return_false:
2716   \else:
2717     \exp_after:wN \token_if_skip_register_aux:w
2718     \token_to_meaning:N #1?SKIP\q_stop
2719   \fi:
2720 }
2721 \cs_new_nopar:Npn \token_if_skip_register_aux:w #1?SKIP#2\q_stop{
2722   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2723 }

```

Dim registers. No news here

```

2724 \prg_new_conditional:Nnn \token_if_dim_register:N {p,TF,T,F} {
2725   \if_meaning:w \tex_dimendef:D #1
2726   \c_false_bool
2727   \else:
2728     \exp_after:wN \token_if_dim_register_aux:w
2729     \token_to_meaning:N #1?ZIMEX\q_stop
2730   \fi:
2731 }
2732 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1?ZIMEX#2\q_stop{
2733   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2734 }

```

Toks registers.

```

2735 \prg_new_conditional:Nnn \token_if_toks_register:N {p,TF,T,F} {
2736   \if_meaning:w \tex_toksdef:D #1
2737   \prg_return_false:
2738   \else:
2739     \exp_after:wN \token_if_toks_register_aux:w
2740     \token_to_meaning:N #1?YOKS\q_stop
2741   \fi:
2742 }
2743 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1?YOKS#2\q_stop{
2744   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2745 }

```

Protected macros.

```

2746 \prg_new_conditional:Nnn \token_if_protected_macro:N {p,TF,T,F} {
2747   \exp_after:wN \token_if_protected_macro_aux:w
2748   \token_to_meaning:N #1?PROYECYEZ~MACRO\q_stop
2749 }
2750 \cs_new_nopar:Npn \token_if_protected_macro_aux:w #1?PROYECYEZ~MACRO#2\q_stop{
2751   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2752 }

```

Long macros.

```

2753 \prg_new_conditional:Nnn \token_if_long_macro:N {p,TF,T,F} {
2754   \exp_after:wN \token_if_long_macro_aux:w
2755   \token_to_meaning:N #1?LOXG~MACRO\q_stop
2756 }
2757 \cs_new_nopar:Npn \token_if_long_macro_aux:w #1?LOXG~MACRO#2\q_stop{
2758   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2759 }

```

Finally protected long macros where we for once don't have to add an extra test since there is no primitive for the combined prefixes.

```

2760 \prg_new_conditional:Nnn \token_if_protected_long_macro:N {p,TF,T,F} {
2761   \exp_after:wN \token_if_protected_long_macro_aux:w

```



```

2762 \token_to_meaning:N #1?PROYECY EZ?LOXG~MACRO\q_stop
2763 }
2764 \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w #1
2765 ?PROYECY EZ?LOXG~MACRO#2\q_stop{
2766 \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2767 }

```

Finally the `\tl_to_lowercase:n` ends!

```

2768 }

```

We do not provide a function for testing if a control sequence is “outer” since we don’t use that in L^AT_EX3.

```

et_prefix_arg_replacement_aux:w
\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N

```

In the `xparse` package 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.

```

2769 \group_begin:
2770 \char_set_lccode:nn {'?}{\:}
2771 \char_set_catcode:nn{'M}{12}
2772 \char_set_catcode:nn{'A}{12}
2773 \char_set_catcode:nn{'C}{12}
2774 \char_set_catcode:nn{'R}{12}
2775 \char_set_catcode:nn{'0}{12}
2776 \tl_to_lowercase:n{
2777 \group_end:
2778 \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_stop#4{
2779 #4{#1}{#2}{#3}
2780 }
2781 \cs_new_nopar:Npn \token_get_prefix_spec:N #1{
2782 \token_if_macro:NTF #1{
2783 \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2784 \token_to_meaning:N #1\q_stop\use_i:nnn
2785 }{\scan_stop:}
2786 }
2787 \cs_new_nopar:Npn \token_get_arg_spec:N #1{
2788 \token_if_macro:NTF #1{
2789 \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2790 \token_to_meaning:N #1\q_stop\use_ii:nnn
2791 }{\scan_stop:}
2792 }
2793 \cs_new_nopar:Npn \token_get_replacement_spec:N #1{
2794 \token_if_macro:NTF #1{
2795 \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2796 \token_to_meaning:N #1\q_stop\use_iii:nnn
2797 }{\scan_stop:}

```

```

2798 }
2799 }

```

Useless code: because we can!

`\token_if_primitive_p:N` It is rather hard to determine if a token is a primitive. First we can check if it is a
`\token_if_primitive_p_aux:N` control sequence or active character. If either, we check if it is a macro. Then we can go
`\token_if_primitive:N \underline{T} \underline{F}` through a tedious process of testing for different register types... I don't actually think
this function is useful but you never know.

```

2800 \prg_new_conditional:Nnn \token_if_primitive:N {p,TF,T,F} {
2801   \if_predicate:w \token_if_cs_p:N #1
2802     \if_predicate:w \token_if_macro_p:N #1
2803       \prg_return_false:
2804     \else:
2805       \token_if_primitive_p_aux:N #1
2806     \fi:
2807   \else:
2808     \if_predicate:w \token_if_active_char_p:N #1
2809       \if_predicate:w \token_if_macro_p:N #1
2810         \prg_return_false:
2811       \else:
2812         \token_if_primitive_p_aux:N #1
2813       \fi:
2814     \else:
2815       \prg_return_false:
2816     \fi:
2817   \fi:
2818 }
2819 \cs_new_nopar:Npn \token_if_primitive_p_aux:N #1{
2820   \if_predicate:w \token_if_chardef_p:N #1 \c_false_bool
2821   \else:
2822     \if_predicate:w \token_if_mathchardef_p:N #1 \prg_return_false:
2823   \else:
2824     \if_predicate:w \token_if_int_register_p:N #1 \prg_return_false:
2825   \else:
2826     \if_predicate:w \token_if_skip_register_p:N #1 \prg_return_false:
2827   \else:
2828     \if_predicate:w \token_if_dim_register_p:N #1 \prg_return_false:
2829   \else:
2830     \if_predicate:w \token_if_toks_register_p:N #1 \prg_return_false:
2831   \else:

```

We made it!

```

2832       \prg_return_true:
2833     \fi:
2834   \fi:
2835 \fi:

```

```

2836     \fi:
2837     \fi:
2838 \fi:
2839 }

```

100.5 Peeking ahead at the next token

```

\l_peek_token
\g_peek_token
\l_peek_search_token

```

We define some other tokens which will initially be the character ?.

```

2840 \token_new:Nn \l_peek_token {?}
2841 \token_new:Nn \g_peek_token {?}
2842 \token_new:Nn \l_peek_search_token {?}

```

`\peek_after:NN` `\peek_after:NN` takes two argument where the first is a function acting on `\l_peek_token` and the second is the next token in the input stream which `\l_peek_token` is set equal to. `\peek_gafter:NN` does the same globally to `\g_peek_token`.

```

2843 \cs_new_protected_nopar:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
2844 \cs_new_protected_nopar:Npn \peek_gafter:NN {
2845   \pref_global:D \tex_futurelet:D \g_peek_token
2846 }

```

For normal purposes there are four main 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.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the `<true>` and `<false>` cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a `<toks>` register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tl` Two dedicated token list variables that store the true and false cases.
`\l_peek_false_tl`

```
2847 \tl_new:N \l_peek_true_tl
2848 \tl_new:N \l_peek_false_tl
```

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```
2849 \cs_new_nopar:Npn \peek_tmp:w {}
```

`\l_peek_search_tl` We also use this token list variable for storing the token we want to compare. This turns out to be useful.

```
2850 \tl_new:N \l_peek_search_tl
```

`\peek_token_generic:NNTF` #1 : the function to execute (obey or ignore spaces, etc.),
#2 : the special token we're looking for.

```
2851 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4 {
2852   \cs_set_eq:NN \l_peek_search_token #2
2853   \tl_set:Nn \l_peek_search_tl {#2}
2854   \tl_set:Nn \l_peek_true_tl { \group_align_safe_end: #3 }
2855   \tl_set:Nn \l_peek_false_tl { \group_align_safe_end: #4 }
2856   \group_align_safe_begin:
2857     \peek_after:NN #1
2858 }
2859 \cs_new_protected:Npn \peek_token_generic:NNT #1#2#3 {
2860   \peek_token_generic:NNTF #1#2 {#3} {}
2861 }
2862 \cs_new_protected:Npn \peek_token_generic:NNF #1#2#3 {
2863   \peek_token_generic:NNTF #1#2 {} {#3}
2864 }
```

`\peek_token_remove_generic:NNTF` If we want to be able to remove any character from the input stream we might as well do it the same way for all characters so we define this as little differently from above.

```
2865 \cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4 {
2866   \cs_set_eq:NN \l_peek_search_token #2
2867   \tl_set:Nn \l_peek_search_tl {#2}
2868   \tl_set:Nn \l_peek_true_aux_tl {#3}
2869   \tl_set_eq:NN \l_peek_true_tl \c_peek_true_remove_next_tl
2870   \tl_set:Nn \l_peek_false_tl { \group_align_safe_end: #4 }
2871   \group_align_safe_begin:
2872     \peek_after:NN #1
2873 }
2874 \cs_new:Npn \peek_token_remove_generic:NNT #1#2#3 {
2875   \peek_token_remove_generic:NNTF #1#2 {#3} {}
2876 }
2877 \cs_new:Npn \peek_token_remove_generic:NNF #1#2#3 {
2878   \peek_token_remove_generic:NNTF #1#2 {} {#3}
2879 }
```

`\l_peek_true_aux_tl` Two token list variables to help with removing the character from the input stream.
`\c_peek_true_remove_next_tl`

```

2880 \tl_new:N \l_peek_true_aux_tl
2881 \tl_const:Nn \c_peek_true_remove_next_tl {\group_align_safe_end:
2882 \tex_afterassignment:D \l_peek_true_aux_tl \cs_set_eq:NN \peek_tmp:w
2883 }

```

`\peek_execute_branches_meaning:` There are three major tests between tokens in T_EX: meaning, catcode and charcode.
`\peek_execute_branches_catcode:` Hence we define three basic test functions that set in after the ignoring phase is over and
`\peek_execute_branches_charcode:` done with.
`\execute_branches_charcode_aux:NN`

```

2884 \cs_new_nopar:Npn \peek_execute_branches_meaning: {
2885   \if_meaning:w \l_peek_token \l_peek_search_token
2886   \exp_after:wN \l_peek_true_tl
2887   \else:
2888     \exp_after:wN \l_peek_false_tl
2889   \fi:
2890 }
2891 \cs_new_nopar:Npn \peek_execute_branches_catcode: {
2892   \if_catcode:w \exp_not:N\l_peek_token \exp_not:N\l_peek_search_token
2893   \exp_after:wN \l_peek_true_tl
2894   \else:
2895     \exp_after:wN \l_peek_false_tl
2896   \fi:
2897 }

```

For the charcode version we do things a little differently. We want to check the token directly but if we do this we face problems if the next thing in the input stream is a braced group or a space token. The braced group would be read as a complete argument and the space would be gobbled by T_EX's argument reading routines. Hence we test for both of these and if one of them is found we just execute the false result directly since no one should ever try to use the `charcode` function for searching for `\c_group_begin_token` or `\c_space_token`. The same is true for `\c_group_end_token`, as this can only occur if the function is at the end of a group.

```

2898 \cs_new_nopar:Npn \peek_execute_branches_charcode: {
2899   \bool_if:nTF {
2900     \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
2901     \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token ||
2902     \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2903   }
2904   { \l_peek_false_tl }

```

Otherwise we call a small auxiliary function that just grabs the next token. We can do that because it really is a single token; we just have insert it again afterwards. Also we stored the token we were looking for in the token list variable `\l_peek_search_tl` so we unpack it again for this function.

```

2905   { \exp_after:wN \peek_execute_branches_charcode_aux:NN \l_peek_search_tl }
2906 }

```

Then we just do the usual `\if_charcode:w` comparison. We also remember to insert `#2` again after executing the true or false branches.

```

2907 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
2908   \if_charcode:w \exp_not:N #1\exp_not:N#2
2909     \exp_after:wN \l_peek_true_tl
2910   \else:
2911     \exp_after:wN \l_peek_false_tl
2912   \fi:
2913   #2
2914 }

```

`\peek_def_aux:nnnn` This function aids defining conditional variants without too much repeated code. I hope
`\peek_def_aux_ii:nnnnn` that it doesn't detract too much from the readability.

```

2915 \cs_new_nopar:Npn \peek_def_aux:nnnn #1#2#3#4 {
2916   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { TF }
2917   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { T  }
2918   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { F  }
2919 }
2920 \cs_new_protected_nopar:Npn \peek_def_aux_ii:nnnnn #1#2#3#4#5 {
2921   \cs_new_nopar:cpx { #1 #5 } {
2922     \tl_if_empty:nF {#2} {
2923       \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 }
2924     }
2925     \exp_not:c { #3 #5 }
2926     \exp_not:n { #4 }
2927   }
2928 }

```

`\peek_meaning:N \underline{T} \underline{F}` Here we use meaning comparison with `\if_meaning:w`.

```

2929 \peek_def_aux:nnnn
2930 { peek_meaning:N }
2931 {}
2932 { peek_token_generic:NN }
2933 { \peek_execute_branches_meaning: }

```

`\peek_meaning_ignore_spaces:N \underline{T} \underline{F}`

```

2934 \peek_def_aux:nnnn
2935 { peek_meaning_ignore_spaces:N }
2936 { \peek_execute_branches_meaning: }
2937 { peek_token_generic:NN }
2938 { \peek_ignore_spaces_execute_branches: }

```

`\peek_meaning_remove:N \underline{T} \underline{F}`

```

2939 \peek_def_aux:nnnn

```

```

2940 { peek_meaning_remove:N }
2941 {}
2942 { peek_token_remove_generic:NN }
2943 { \peek_execute_branches_meaning: }

```

eaning_remove_ignore_spaces:NTF

```

2944 \peek_def_aux:nnnn
2945 { peek_meaning_remove_ignore_spaces:N }
2946 { \peek_execute_branches_meaning: }
2947 { peek_token_remove_generic:NN }
2948 { \peek_ignore_spaces_execute_branches: }

```

\peek_catcode:NTF Here we use catcode comparison with \if_catcode:w.

```

2949 \peek_def_aux:nnnn
2950 { peek_catcode:N }
2951 {}
2952 { peek_token_generic:NN }
2953 { \peek_execute_branches_catcode: }

```

\peek_catcode_ignore_spaces:NTF

```

2954 \peek_def_aux:nnnn
2955 { peek_catcode_ignore_spaces:N }
2956 { \peek_execute_branches_catcode: }
2957 { peek_token_generic:NN }
2958 { \peek_ignore_spaces_execute_branches: }

```

\peek_catcode_remove:NTF

```

2959 \peek_def_aux:nnnn
2960 { peek_catcode_remove:N }
2961 {}
2962 { peek_token_remove_generic:NN }
2963 { \peek_execute_branches_catcode: }

```

atcode_remove_ignore_spaces:NTF

```

2964 \peek_def_aux:nnnn
2965 { peek_catcode_remove_ignore_spaces:N }
2966 { \peek_execute_branches_catcode: }
2967 { peek_token_remove_generic:NN }
2968 { \peek_ignore_spaces_execute_branches: }

```

\peek_charcode:NTF Here we use charcode comparison with \if_charcode:w.

```

2969 \peek_def_aux:nnnn
2970 { peek_charcode:N }

```

```

2971 {}
2972 { peek_token_generic:NN }
2973 { \peek_execute_branches_charcode: }

peek_charcode_ignore_spaces:N $\mathcal{T}$ F

2974 \peek_def_aux:nnnn
2975 { peek_charcode_ignore_spaces:N }
2976 { \peek_execute_branches_charcode: }
2977 { peek_token_generic:NN }
2978 { \peek_ignore_spaces_execute_branches: }

\peek_charcode_remove:N $\mathcal{T}$ F

2979 \peek_def_aux:nnnn
2980 { peek_charcode_remove:N }
2981 {}
2982 { peek_token_remove_generic:NN }
2983 { \peek_execute_branches_charcode: }

arcode_remove_ignore_spaces:N $\mathcal{T}$ F

2984 \peek_def_aux:nnnn
2985 { peek_charcode_remove_ignore_spaces:N }
2986 { \peek_execute_branches_charcode: }
2987 { peek_token_remove_generic:NN }
2988 { \peek_ignore_spaces_execute_branches:}

\peek_ignore_spaces_aux: Throw away a space token and search again. We could define this in a more devious
ignore_spaces_execute_branches: way where the auxiliary function gobbles the space token but then what do we do if we
decide that a certain function should ignore more than one specific token? For example
someone might find it interesting to define a \peek_ function that ignores a's and b's! Or
maybe different kinds of "funny spaces"... Therefore I have decided to use this version
which uses \tex_afterassignment:D to call the auxiliary function after the next token
has been removed by \cs_set_eq:NN. That way it is easily extensible.

2989 \cs_new_nopar:Npn \peek_ignore_spaces_aux: {
2990 \peek_after:NN \peek_ignore_spaces_execute_branches:
2991 }
2992 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches: {
2993 \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
2994 { \tex_afterassignment:D \peek_ignore_spaces_aux:
2995 \cs_set_eq:NN \peek_tmp:w
2996 }
2997 \peek_execute_branches:
2998 }

2999 </initex | package>

3000 <*showmemory>
3001 \showMemUsage
3002 </showmemory>

```


101 l3int implementation

101.1 Internal functions and variables

<code>\int_advance:w</code>

`\int_advance:w <int register> <optional 'by'> <number> <space>`
 Increments the count register by the specified amount.

T_EXhackers note: This is T_EX's `\advance`.

<code>\int_convert_number_to_letter:n *</code>
--

`\int_convert_number_to_letter:n <{integer expression}>`

Internal function for turning a number for a different base into a letter or digit.

<code>\int_pre_eval_one_arg:Nn</code>	<code>\int_pre_eval_one_arg:Nn <function> <{integer expression}></code>
<code>\int_pre_eval_two_args:Nnn</code>	<code>\int_pre_eval_one_arg:Nnn <function> <{int expr₁>}</code>

`<{int expr2>}`

These are expansion helpers; they evaluate their integer expressions before handing them off to the specified *<function>*.

<code>\int_get_sign_and_digits:n *</code>	
<code>\int_get_sign:n</code>	<code>*</code>
<code>\int_get_digits:n</code>	<code>*</code>

`\int_get_sign_and_digits:n <{number}>`

From an argument that may or may not include a + or - sign, these functions expand to the respective components of the number.

101.2 Module loading and primitives definitions

We start by ensuring that the required packages are loaded.

```

3003 <*package>
3004 \ProvidesExplPackage
3005   {\filename}{\filedate}{\fileversion}{\filedescription}
3006 \package_check_loaded_expl:
3007 </package>
3008 <*initex | package>

```

<pre> \int_value:w \int_eval:n \int_eval:w \int_eval_end: \if_int_compare:w \if_int_odd:w \if_num:w \if_case:w \int_to_roman:w \int_advance:w </pre>	<p>Here are the remaining primitives for number comparisons and expressions.</p> <pre> 3009 \cs_set_eq:NN \int_value:w \tex_number:D 3010 \cs_set_eq:NN \int_eval:w \etex_numexpr:D </pre>
--	--

```

3011 \cs_set_protected:Npn \int_eval_end: {\tex_relax:D}
3012 \cs_set_eq:NN \if_int_compare:w \tex_ifnum:D
3013 \cs_new_eq:NN \if_num:w \tex_ifnum:D
3014 \cs_set_eq:NN \if_int_odd:w \tex_ifodd:D
3015 \cs_new_eq:NN \if_case:w \tex_ifcase:D
3016 \cs_new_eq:NN \int_to_roman:w \tex_romannumeral:D
3017 \cs_new_eq:NN \int_advance:w \tex_advance:D

```

`\int_eval:n` Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream.

```

3018 \cs_set:Npn \int_eval:n #1{
3019   \int_value:w \int_eval:w #1\int_eval_end:
3020 }

```

101.3 Allocation and setting

`\int_new:N` For the L^AT_EX3 format:

`\int_new:c`

```

3021 <*initex>
3022 \alloc_new:nnnN {int} {11} {\c_max_register_int} \tex_countdef:D
3023 </initex>

```

For ‘l3in2e’:

```

3024 <*package>
3025 \cs_new_protected_nopar:Npn \int_new:N #1 {
3026   \chk_if_free_cs:N #1
3027   \newcount #1
3028 }
3029 </package>

3030 \cs_generate_variant:Nn \int_new:N {c}

```

`\int_set:Nn` Setting counters is again something that I would like to make uniform at the moment to get a better overview.

`\int_set:cn`

`\int_gset:Nn`

`\int_gset:cn`

```

3031 \cs_new_protected_nopar:Npn \int_set:Nn #1#2{#1 \int_eval:w #2\int_eval_end:
3032 <*check>
3033 \chk_local_or_pref_global:N #1
3034 </check>
3035 }
3036 \cs_new_protected_nopar:Npn \int_gset:Nn {
3037 <*check>
3038   \pref_global_chk:
3039 </check>
3040 <-check> \pref_global:D
3041   \int_set:Nn }
3042 \cs_generate_variant:Nn\int_set:Nn {cn}
3043 \cs_generate_variant:Nn\int_gset:Nn {cn}

```

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another.

```

\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
3044 \cs_new_protected_nopar:Npn \int_set_eq:NN #1#2 {
3045   \int_set:Nn #1 {#2}
3046 }
3047 \cs_generate_variant:Nn \int_set_eq:NN { c }
3048 \cs_generate_variant:Nn \int_set_eq:NN { Nc }
3049 \cs_generate_variant:Nn \int_set_eq:NN { cc }
3050 \cs_new_protected_nopar:Npn \int_gset_eq:NN #1#2 {
3051   \int_gset:Nn #1 {#2}
3052 }
3053 \cs_generate_variant:Nn \int_gset_eq:NN { c }
3054 \cs_generate_variant:Nn \int_gset_eq:NN { Nc }
3055 \cs_generate_variant:Nn \int_gset_eq:NN { cc }

```

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_decr:N
\int_gincr:N
\int_gdecr:N
\int_incr:c
\int_decr:c
\int_gincr:c
\int_gdecr:c
3056 \cs_new_protected_nopar:Npn \int_incr:N #1{\int_advance:w#1\c_one
3057 <*check>
3058   \chk_local_or_pref_global:N #1
3059 </check>
3060 }
3061 \cs_new_protected_nopar:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
3062 <*check>
3063   \chk_local_or_pref_global:N #1
3064 </check>
3065 }
3066 \cs_new_protected_nopar:Npn \int_gincr:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```

3067 <*check>
3068   \pref_global_chk:
3069 </check>
3070 <-check> \pref_global:D
3071   \int_incr:N}
3072 \cs_new_protected_nopar:Npn \int_gdecr:N {
3073 <*check>
3074   \pref_global_chk:
3075 </check>
3076 <-check> \pref_global:D
3077   \int_decr:N}

```

With the `\int_add:Nn` functions we can shorten the above code. If this makes it too slow ...

```

3078 \cs_set_protected_nopar:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
3079 \cs_set_protected_nopar:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
3080 \cs_set_protected_nopar:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
3081 \cs_set_protected_nopar:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}

3082 \cs_generate_variant:Nn \int_incr:N {c}
3083 \cs_generate_variant:Nn \int_decr:N {c}
3084 \cs_generate_variant:Nn \int_gincr:N {c}
3085 \cs_generate_variant:Nn \int_gdecr:N {c}

```

\int_zero:N Functions that reset an $\langle int \rangle$ register to zero.

```

\int_zero:c
\int_gzero:N 3086 \cs_new_protected_nopar:Npn \int_zero:N #1 {#1=\c_zero}
\int_gzero:c 3087 \cs_generate_variant:Nn \int_zero:N {c}

3088 \cs_new_protected_nopar:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
3089 \cs_generate_variant:Nn \int_gzero:N {c}

```

\int_add:Nn Adding and subtracting to and from a counter ... We should think of using these functions

```

\int_add:cn
\int_gadd:Nn 3090 \cs_new_protected_nopar:Npn \int_add:Nn #1#2{
\int_gadd:cn

```

\int_sub:Nn We need to say by in case the first argument is a register accessed by its number, e.g.,
\int_sub:cn \count23. Not that it should ever happen but...
\int_gsub:Nn

```

\int_gsub:cn 3091 \int_advance:w #1 by \int_eval:w #2\int_eval_end:
3092 \*check
3093 \chk_local_or_pref_global:N #1
3094 \</check>
3095 }
3096 \cs_new_nopar:Npn \int_sub:Nn #1#2{
3097 \int_advance:w #1-\int_eval:w #2\int_eval_end:
3098 \*check
3099 \chk_local_or_pref_global:N #1
3100 \</check>
3101 }
3102 \cs_new_protected_nopar:Npn \int_gadd:Nn {
3103 \*check
3104 \pref_global_chk:
3105 \</check>
3106 \<-check> \pref_global:D
3107 \int_add:Nn }
3108 \cs_new_protected_nopar:Npn \int_gsub:Nn {
3109 \*check
3110 \pref_global_chk:
3111 \</check>
3112 \<-check> \pref_global:D
3113 \int_sub:Nn }
3114 \cs_generate_variant:Nn \int_add:Nn {cn}

```

```

3115 \cs_generate_variant:Nn \int_gadd:Nn {cn}
3116 \cs_generate_variant:Nn \int_sub:Nn {cn}
3117 \cs_generate_variant:Nn \int_gsub:Nn {cn}

```

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
3118 \cs_new_eq:NN \int_use:N \tex_the:D
3119 \cs_new_nopar:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}

```

`\int_show:N`
`\int_show:c`

```

3120 \cs_new_eq:NN \int_show:N \kernel_register_show:N
3121 \cs_new_eq:NN \int_show:c \kernel_register_show:c

```

`\int_to_arabic:n` Nothing exciting here.

```

3122 \cs_new_nopar:Npn \int_to_arabic:n #1{ \int_eval:n{#1}}

```

`\int_roman_lcuc_mapping:Nnn` Using TeX's built-in feature for producing roman numerals has some surprising features. One is the the characters resulting from `\int_to_roman:w` have category code 12 so they may fail in certain comparison tests. Therefore we use a mapping from the character TeX produces to the character we actually want which will give us letters with category code 11.

```

3123 \cs_new_protected_nopar:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
3124   \cs_set_nopar:cpn {int_to_lc_roman_#1:}{#2}
3125   \cs_set_nopar:cpn {int_to_uc_roman_#1:}{#3}
3126 }

```

Here are the default mappings. I haven't found any examples of say Turkish doing the mapping `i \i` `I` but at least there is a possibility for it if needed. Note: I have now asked a Turkish person and he tells me they do the `i I` mapping.

```

3127 \int_roman_lcuc_mapping:Nnn i i I
3128 \int_roman_lcuc_mapping:Nnn v v V
3129 \int_roman_lcuc_mapping:Nnn x x X
3130 \int_roman_lcuc_mapping:Nnn l l L
3131 \int_roman_lcuc_mapping:Nnn c c C
3132 \int_roman_lcuc_mapping:Nnn d d D
3133 \int_roman_lcuc_mapping:Nnn m m M

```

For the delimiter we cheat and let it gobble its arguments instead.

```

3134 \int_roman_lcuc_mapping:Nnn Q \use_none:nn \use_none:nn

```

`\int_to_roman:n` The commands for producing the lower and upper case roman numerals run a loop on one character at a time and also carries some information for upper or lower case with
`\int_to_Roman:n` it. We put it through `\int_eval:n` first which is safer and more flexible.
`\int_to_roman_lcuc:NN`

```

3135 \cs_new_nopar:Npn \int_to_roman:n #1 {
3136   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN l
3137   \int_to_roman:w \int_eval:n {#1} Q
3138 }
3139 \cs_new_nopar:Npn \int_to_Roman:n #1 {
3140   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN u
3141   \int_to_roman:w \int_eval:n {#1} Q
3142 }
3143 \cs_new_nopar:Npn \int_to_roman_lcuc:NN #1#2{
3144   \use:c {int_to_#lc_roman_#2:}
3145   \int_to_roman_lcuc:NN #1
3146 }

```

`\int_convert_to_symbols:nnn` 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. This is more or less the same as `\int_convert_number_with_rule:nnN` but 'pre-packaged'.

```

3147 \cs_new_nopar:Npn \int_convert_to_symbols:nnn #1#2#3 {
3148   \int_compare:nNnTF {#1} > {#2}
3149   {
3150     \exp_args:Nf \int_convert_to_symbols:nnn
3151     { \int_div_truncate:nn { #1 - 1 } {#2} } {#2} {#3}
3152     \exp_args:Nf \prg_case_int:nnn
3153     { \int_eval:n { 1 + \int_mod:nn { #1 - 1 } {#2} } }
3154     {#3} { }
3155   }
3156   { \exp_args:Nf \prg_case_int:nnn { \int_eval:n {#1} } {#3} { } }
3157 }

```

`\int_convert_number_with_rule:nnN` This is our major workhorse for conversions. #1 is the number we want converted, #2 is the base number, and #3 is the function converting the number. This function expects to receive a non-negative integer and as such is ideal for something using `\if_case:w` internally.

The basic example is this: We want to convert the number 50 (#1) into an alphabetic equivalent **ax**. For the English language our list contains 26 elements so this is our argument #2 while the function #3 just turns 1 into a, 2 into b, etc. Hence our goal is to turn 50 into the sequence #3{1}#1{24} so what we do is to first divide 50 by 26 and truncating the result returning 1. Then before we execute this we call the function again but this time on the result of the remainder of the division. This goes on until the remainder is less than or equal to the base number where we just call the function #3 directly on the number.

We do a little pre-expansion of the arguments below as they otherwise have a tendency to grow quite large.

```

3158 \cs_set_nopar:Npn \int_convert_number_with_rule:nnN #1#2#3{
3159   \int_compare:nNnTF {#1}>{#2}
3160   {
3161     \exp_args:Nf \int_convert_number_with_rule:nnN
3162     { \int_div_truncate:nn {#1-1}{#2} }{#2}
3163     #3

```

Note that we have to nudge our modulus function so it won't return 0 as that wouldn't work with `\if_case:w` when that expects a positive number to produce a letter.

```

3164     \exp_args:Nf #3 { \int_eval:n{1+\int_mod:nn {#1-1}{#2}} }
3165   }
3166   { \exp_args:Nf #3{ \int_eval:n{#1} } }
3167 }

```

As can be seen it is even simpler to convert to number systems that contain 0, since then we don't have to add or subtract 1 here and there.

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```

3168 \cs_new_nopar:Npn \int_to_alph:n #1 {
3169   \int_convert_to_symbols:nnn {#1} { 26 }
3170   {
3171     { 1 } { a }
3172     { 2 } { b }
3173     { 3 } { c }
3174     { 4 } { d }
3175     { 5 } { e }
3176     { 6 } { f }
3177     { 7 } { g }
3178     { 8 } { h }
3179     { 9 } { i }
3180     { 10 } { j }
3181     { 11 } { k }
3182     { 12 } { l }
3183     { 13 } { m }
3184     { 14 } { n }
3185     { 15 } { o }
3186     { 16 } { p }
3187     { 17 } { q }
3188     { 18 } { r }
3189     { 19 } { s }
3190     { 20 } { t }
3191     { 21 } { u }
3192     { 22 } { v }
3193     { 23 } { w }

```

```

3194     { 24 } { x }
3195     { 25 } { y }
3196     { 26 } { z }
3197   }
3198 }
3199 \cs_new_nopar:Npn \int_to_Alph:n #1 {
3200   \int_convert_to_symbols:nnn {#1} { 26 }
3201   {
3202     { 1 } { A }
3203     { 2 } { B }
3204     { 3 } { C }
3205     { 4 } { D }
3206     { 5 } { E }
3207     { 6 } { F }
3208     { 7 } { G }
3209     { 8 } { H }
3210     { 9 } { I }
3211     { 10 } { J }
3212     { 11 } { K }
3213     { 12 } { L }
3214     { 13 } { M }
3215     { 14 } { N }
3216     { 15 } { O }
3217     { 16 } { P }
3218     { 17 } { Q }
3219     { 18 } { R }
3220     { 19 } { S }
3221     { 20 } { T }
3222     { 21 } { U }
3223     { 22 } { V }
3224     { 23 } { W }
3225     { 24 } { X }
3226     { 25 } { Y }
3227     { 26 } { Z }
3228   }
3229 }

```

`\int_to_symbol:n` Turning a number into a symbol is also easy enough.

```

3230 \cs_new_nopar:Npn \int_to_symbol:n #1{
3231   \mode_if_math:TF
3232   {
3233     \int_convert_number_with_rule:nnN {#1}{9}
3234     \int_symbol_math_conversion_rule:n
3235   }
3236   {
3237     \int_convert_number_with_rule:nnN {#1}{9}
3238     \int_symbol_text_conversion_rule:n
3239   }
3240 }

```


`\int_symbol_math_conversion_rule:n`
`\int_symbol_text_conversion_rule:n`

Nothing spectacular here.

```

3241 \cs_new_nopar:Npn \int_symbol_math_conversion_rule:n #1 {
3242   \if_case:w #1
3243     \or: *
3244     \or: \dagger
3245     \or: \ddagger
3246     \or: \mathsection
3247     \or: \mathparagraph
3248     \or: \/
3249     \or: **
3250     \or: \dagger\dagger
3251     \or: \ddagger\ddagger
3252   \fi:
3253 }
3254 \cs_new_nopar:Npn \int_symbol_text_conversion_rule:n #1 {
3255   \if_case:w #1
3256     \or: \textasteriskcentered
3257     \or: \textdagger
3258     \or: \textdaggerdbl
3259     \or: \textsection
3260     \or: \textparagraph
3261     \or: \textbardbl
3262     \or: \textasteriskcentered\textasteriskcentered
3263     \or: \textdagger\textdagger
3264     \or: \textdaggerdbl\textdaggerdbl
3265   \fi:
3266 }
```

`\l_tmpa_int`
`\l_tmpb_int`
`\l_tmpc_int`
`\g_tmpa_int`
`\g_tmpb_int`

We provide four local and two global scratch counters, maybe we need more or less.

```

3267 \int_new:N \l_tmpa_int
3268 \int_new:N \l_tmpb_int
3269 \int_new:N \l_tmpc_int
3270 \int_new:N \g_tmpa_int
3271 \int_new:N \g_tmpb_int
```

`\int_pre_eval_one_arg:Nn`
`\int_pre_eval_two_args:Nnn`

These are handy when handing down values to other functions. All they do is evaluate the number in advance.

```

3272 \cs_set_nopar:Npn \int_pre_eval_one_arg:Nn #1#2{
3273   \exp_args:Nf#1{\int_eval:n{#2}}}}
3274 \cs_set_nopar:Npn \int_pre_eval_two_args:Nnn #1#2#3{
3275   \exp_args:Nff#1{\int_eval:n{#2}}{\int_eval:n{#3}}
3276 }
```

101.4 Scanning and conversion

`\int_from_roman:n`
`\int_from_roman_aux:NN`
`\int_from_roman_end:w`
`\int_from_roman_clean_up: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 T_EX.

```

3277 \cs_new_nopar:Npn \int_from_roman:n #1 {
3278   \tl_if_blank:nF {#1}
3279   {
3280     \tex_expandafter:D \int_from_roman_end:w
3281     \tex_number:D \etex_numexpr:D
3282     \int_from_roman_aux:NN #1 Q \q_stop
3283   }
3284 }
3285 \cs_new_nopar:Npn \int_from_roman_aux:NN #1#2 {
3286   \str_if_eq:nnTF {#1} { Q }
3287   {#1#2}
3288   {
3289     \str_if_eq:nnTF {#2} { Q }
3290     {
3291       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3292       { \int_from_roman_clean_up:w }
3293       +
3294       \use:c { c_int_from_roman_ #1 _int }
3295       #2
3296     }
3297     {
3298       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3299       { \int_from_roman_clean_up:w }
3300       \cs_if_exist:cF { c_int_from_roman_ #2 _int }
3301       { \int_from_roman_clean_up:w }
3302       \int_compare:nNnTF
3303       { \use:c { c_int_from_roman_ #1 _int } }
3304       <
3305       { \use:c { c_int_from_roman_ #2 _int } }
3306       {
3307         + \use:c { c_int_from_roman_ #2 _int }
3308         - \use:c { c_int_from_roman_ #1 _int }
3309         \int_from_roman_aux:NN
3310       }
3311       {
3312         + \use:c { c_int_from_roman_ #1 _int }
3313         \int_from_roman_aux:NN #2
3314       }
3315     }
3316   }
3317 }
3318 \cs_new_nopar:Npn \int_from_roman_end:w #1 Q #2 \q_stop {
3319   \tl_if_empty:nTF {#2} {#1} {#2}
3320 }
3321 \cs_new_nopar:Npn \int_from_roman_clean_up:w #1 Q { + 0 Q -1 }

```

```

\int_convert_from_base_ten:nn
\int_convert_from_base_ten_aux:nnn
\int_convert_number_to_letter:n

```

Converting from base ten (#1) to a second base (#2) starts with a simple sign check. As the input is base 10 TeX can then do the actual work with the sign itself.

```

3322 \cs_new:Npn \int_convert_from_base_ten:nn #1#2 {
3323   \int_compare:nNnTF {#1} < { 0 }
3324   {
3325     -
3326     \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3327     { } { \int_eval:n { 0 - ( #1 ) } } {#2}
3328   }
3329   {
3330     \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3331     { } { \int_eval:n {#1} } {#2}
3332   }
3333 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is build up as argument #1, which is why it starts off empty in the above. At each pass, the value in #2 is checked to see if it is less than the new base (#3). If it is the it is converted directly and the rest of the output is added in. 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 the remainder is carried forward to the next round.S

```

3334 \cs_new:Npn \int_convert_from_base_ten_aux:nnn #1#2#3 {
3335   \int_compare:nNnTF {#2} < {#3}
3336   {
3337     \int_convert_number_to_letter:n {#2}
3338     #1
3339   }
3340   {
3341     \exp_args:Nff \int_convert_from_base_ten_aux:nnn
3342     {
3343       \int_convert_number_to_letter:n
3344       { \int_mod:nn {#2} {#3} }
3345       #1
3346     }
3347     { \int_div_truncate:nn {#2} {#3} }
3348     {#3}
3349   }
3350 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged.

```

3351 \cs_new:Npn \int_convert_number_to_letter:n #1 {
3352   \prg_case_int:nnn { #1 - 9 }
3353   {
3354     { 1 } { A }
3355     { 2 } { B }
3356     { 3 } { C }

```

```

3357     { 4 } { D }
3358     { 5 } { E }
3359     { 6 } { F }
3360     { 7 } { G }
3361     { 8 } { H }
3362     { 9 } { I }
3363     { 10 } { J }
3364     { 11 } { K }
3365     { 12 } { L }
3366     { 13 } { M }
3367     { 14 } { N }
3368     { 15 } { O }
3369     { 16 } { P }
3370     { 17 } { Q }
3371     { 18 } { R }
3372     { 19 } { S }
3373     { 20 } { T }
3374     { 21 } { U }
3375     { 22 } { V }
3376     { 23 } { W }
3377     { 24 } { X }
3378     { 25 } { Y }
3379     { 26 } { Z }
3380   }
3381   {#1}
3382 }

```

```

\int_convert_to_base_ten:nn
\int_convert_to_base_ten_aux:nn
\int_convert_to_base_ten_aux:nnN
\int_convert_to_base_ten_aux:N
  \int_get_sign_and_digits:n
    \int_get_sign:n
    \int_get_digits:n
\int_get_sign_and_digits_aux:nnnn
\int_get_sign_and_digits_aux:nnnn

```

Conversion to base ten means stripping off the sign then iterating through the input one token at a time. The total number is then added up as the code loops.

```

3383 \cs_new:Npn \int_convert_to_base_ten:nn #1#2 {
3384   \int_eval:n
3385   {
3386     \int_get_sign:n {#1}
3387     \exp_args:Nf \int_convert_to_base_ten_aux:nn
3388       { \int_get_digits:n {#1} } {#2}
3389   }
3390 }
3391 \cs_new:Npn \int_convert_to_base_ten_aux:nn #1#2 {
3392   \int_convert_to_base_ten_aux:nnN { 0 } { #2 } #1 \q_nil
3393 }
3394 \cs_new:Npn \int_convert_to_base_ten_aux:nnN #1#2#3 {
3395   \quark_if_nil:NTF #3
3396   {#1}
3397   {
3398     \exp_args:Nf \int_convert_to_base_ten_aux:nnN
3399       { \int_eval:n { #1 * #2 + \int_convert_to_base_ten_aux:N #3 } }
3400     {#2}
3401   }
3402 }

```

The conversion here will take lower or upper case letters and turn them into the appropriate number, hence the two-part nature of the function.

```

3403 \cs_new:Npn \int_convert_to_base_ten_aux:N #1 {
3404   \int_compare:nNnTF { '#1 } < { 58 }
3405     {#1}
3406     {
3407       \int_eval:n
3408         { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3409     }
3410 }

```

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```

3411 \cs_new:Npn \int_get_sign_and_digits:n #1 {
3412   \int_get_sign_and_digits_aux:nNNN {#1}
3413   \c_true_bool \c_true_bool \c_true_bool
3414 }
3415 \cs_new:Npn \int_get_sign:n #1 {
3416   \int_get_sign_and_digits_aux:nNNN {#1}
3417   \c_true_bool \c_true_bool \c_false_bool
3418 }
3419 \cs_new:Npn \int_get_digits:n #1 {
3420   \int_get_sign_and_digits_aux:nNNN {#1}
3421   \c_true_bool \c_false_bool \c_true_bool
3422 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3423 \cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4 {
3424   \tl_if_head_eq_charcode:fNnTF {#1} -
3425   {
3426     \bool_if:NnTF #2
3427     {
3428       \int_get_sign_and_digits_aux:oNNN
3429         { \use_none:n #1 } \c_false_bool #3#4
3430     }
3431     {
3432       \int_get_sign_and_digits_aux:oNNN
3433         { \use_none:n #1 } \c_true_bool #3#4
3434     }
3435   }
3436   {
3437     \tl_if_head_eq_charcode:fNnTF {#1} +
3438     { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
3439     {

```

```

3440         \bool_if:NT #3 { \bool_if:NF #2 - }
3441         \bool_if:NT #4 {#1}
3442     }
3443 }
3444 }
3445 \cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN { o }

```

\int_from_binary:n Wrappers around the generic function.

```

\int_from_hexadecimal:n
\int_from_octal:n
\int_to_binary:n
\int_to_hexadecimal:n
\int_to_octal:n
3446 \cs_new:Npn \int_from_binary:n #1 {
3447     \int_convert_to_base_ten:nn {#1} { 2 }
3448 }
3449 \cs_new:Npn \int_from_hexadecimal:n #1 {
3450     \int_convert_to_base_ten:nn {#1} { 16 }
3451 }
3452 \cs_new:Npn \int_from_octal:n #1 {
3453     \int_convert_to_base_ten:nn {#1} { 8 }
3454 }
3455 \cs_new:Npn \int_to_binary:n #1 {
3456     \int_convert_from_base_ten:nn {#1} { 2 }
3457 }
3458 \cs_new:Npn \int_to_hexadecimal:n #1 {
3459     \int_convert_from_base_ten:nn {#1} { 16 }
3460 }
3461 \cs_new:Npn \int_to_octal:n #1 {
3462     \int_convert_from_base_ten:nn {#1} { 8 }
3463 }

```

\int_from_alph:n The aim here is to iterate through the input, converting one letter at a time to a number.
\int_from_alph_aux:n The same approach is also used for base conversion, but this needs a different final
\int_from_alph_aux:nN auxiliary.
\int_from_alph_aux:N

```

3464 \cs_new:Npn \int_from_alph:n #1 {
3465     \int_eval:n
3466     {
3467         \int_get_sign:n {#1}
3468         \exp_args:Nf \int_from_alph_aux:n
3469         { \int_get_digits:n {#1} }
3470     }
3471 }
3472 \cs_new:Npn \int_from_alph_aux:n #1 {
3473     \int_from_alph_aux:nN { 0 } #1 \q_nil
3474 }
3475 \cs_new:Npn \int_from_alph_aux:nN #1#2 {
3476     \quark_if_nil:NTF #2
3477     {#1}
3478     {
3479         \exp_args:Nf \int_from_alph_aux:nN
3480         { \int_eval:n { #1 * 26 + \int_from_alph_aux:N #2 } }

```

```

3481     }
3482   }
3483   \cs_new:Npn \int_from_alph_aux:N #1 {
3484     \int_eval:n
3485     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
3486   }

```

`\int_compare_p:n` Comparison tests using a simple syntax where only one set of braces is required and
`\int_compare:nTF` additional operators such as `!=` and `>=` are supported. First some notes on the idea behind this. We wish to support writing code like

```
\int_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }
```

In other words, we want to somehow add the missing `\int_eval:w` where required. We can start evaluating from the left using `\int_eval:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let T_EX evaluate this left hand side of the (in)equality.

```

3487 \prg_set_conditional:Npnn \int_compare:n #1{p,TF,T,F}{
3488   \exp_after:wN \int_compare_auxi:w \int_value:w
3489   \int_eval:w #1\q_stop
3490 }

```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `<null>`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3491 \cs_set:Npn \int_compare_auxi:w #1#2\q_stop{
3492   \exp_after:wN \int_compare_auxii:w \tex_romannumeral:D
3493   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3494 }

```

This leaves the first relation symbol in front and assuming the right hand side has been input, at least one other token as well. We support the following forms: `=`, `<`, `>` and the extended `!=`, `==`, `<=` and `>=`. All the extended forms have an extra `=` so we check if that is present as well. Then use specific function to perform the test.

```

3495 \cs_set:Npn \int_compare_auxii:w #1#2#3\q_mark{
3496   \use:c{
3497     int_compare_
3498     #1 \if_meaning:w =#2 = \fi:
3499     :w}
3500 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3501 \cs_set:cpn {int_compare_=:w} #1=#2\q_stop{
3502   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3503     \prg_return_true: \else: \prg_return_false: \fi:
3504 }

```

So is the one using == – we just have to use == in the parameter text.

```

3505 \cs_set:cpn {int_compare_==:w} #1==#2\q_stop{
3506   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3507     \prg_return_true: \else: \prg_return_false: \fi:
3508 }

```

Not equal is just about reversing the truth value.

```

3509 \cs_set:cpn {int_compare_!=:w} #1!=#2\q_stop{
3510   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3511     \prg_return_false: \else: \prg_return_true: \fi:
3512 }

```

Less than and greater than are also straight forward.

```

3513 \cs_set:cpn {int_compare_<:w} #1<#2\q_stop{
3514   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3515     \prg_return_true: \else: \prg_return_false: \fi:
3516 }
3517 \cs_set:cpn {int_compare_>:w} #1>#2\q_stop{
3518   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3519     \prg_return_true: \else: \prg_return_false: \fi:
3520 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

3521 \cs_set:cpn {int_compare_<=:w} #1<=#2\q_stop{
3522   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3523     \prg_return_false: \else: \prg_return_true: \fi:
3524 }
3525 \cs_set:cpn {int_compare_>=:w} #1>=#2\q_stop{
3526   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3527     \prg_return_false: \else: \prg_return_true: \fi:
3528 }

```

\int_compare_p:nNn More efficient but less natural in typing.

\int_compare:nNnTF

```

3529 \prg_set_conditional:Npnn \int_compare:nNn #1#2#3{p}{
3530   \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3
3531   \int_eval_end:
3532   \prg_return_true: \else: \prg_return_false: \fi:

```



```

3533 }
3534 \cs_set_nopar:Npn \int_compare:nNnT #1#2#3 {
3535   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3536   \tex_expandafter:D \use:n
3537   \tex_else:D
3538   \tex_expandafter:D \use_none:n
3539   \tex_fi:D
3540 }
3541 \cs_set_nopar:Npn \int_compare:nNnF #1#2#3 {
3542   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3543   \tex_expandafter:D \use_none:n
3544   \tex_else:D
3545   \tex_expandafter:D \use:n
3546   \tex_fi:D
3547 }
3548 \cs_set_nopar:Npn \int_compare:nNnTF #1#2#3 {
3549   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3550   \tex_expandafter:D \use_i:nn
3551   \tex_else:D
3552   \tex_expandafter:D \use_ii:nn
3553   \tex_fi:D
3554 }

```

`\int_max:nn` Functions for min, max, and absolute value.

`\int_min:nn`

`\int_abs:n`

```

3555 \cs_set:Npn \int_abs:n #1{
3556   \int_value:w
3557   \if_int_compare:w \int_eval:w #1<\c_zero
3558     -
3559   \fi:
3560   \int_eval:w #1\int_eval_end:
3561 }
3562 \cs_set:Npn \int_max:nn #1#2{
3563   \int_value:w \int_eval:w
3564   \if_int_compare:w
3565     \int_eval:w #1>\int_eval:w #2\int_eval_end:
3566     #1
3567   \else:
3568     #2
3569   \fi:
3570   \int_eval_end:
3571 }
3572 \cs_set:Npn \int_min:nn #1#2{
3573   \int_value:w \int_eval:w
3574   \if_int_compare:w
3575     \int_eval:w #1<\int_eval:w #2\int_eval_end:
3576     #1
3577   \else:
3578     #2
3579   \fi:

```

```

3580 \int_eval_end:
3581 }

```

`\int_div_truncate:nn` As `\int_eval:w` rounds the result of a division we also provide a version that truncates the result.
`\int_div_round:nn`
`\int_mod:nn`

Initial version didn't work correctly with eTeX's implementation.

```

3582 %\cs_set:Npn \int_div_truncate_raw:nn #1#2 {
3583 % \int_eval:n{ (2*#1 - #2) / (2* #2) }
3584 %}

```

New version by Heiko:

```

3585 \cs_set:Npn \int_div_truncate:nn #1#2 {
3586 \int_value:w \int_eval:w
3587 \if_int_compare:w \int_eval:w #1 = \c_zero
3588 0
3589 \else:
3590 (#1
3591 \if_int_compare:w \int_eval:w #1 < \c_zero
3592 \if_int_compare:w \int_eval:w #2 < \c_zero
3593 -( #2 +
3594 \else:
3595 +( #2 -
3596 \fi:
3597 \else:
3598 \if_int_compare:w \int_eval:w #2 < \c_zero
3599 +( #2 +
3600 \else:
3601 -( #2 -
3602 \fi:
3603 \fi:
3604 1)/2)
3605 \fi:
3606 /(#2)
3607 \int_eval_end:
3608 }

```

For the sake of completeness:

```

3609 \cs_set:Npn \int_div_round:nn #1#2 {\int_eval:n{(#1)/(#2)}}

```

Finally there's the modulus operation.

```

3610 \cs_set:Npn \int_mod:nn #1#2 {
3611 \int_value:w
3612 \int_eval:w
3613 #1 - \int_div_truncate:nn {#1}{#2} * (#2)
3614 \int_eval_end:
3615 }

```

`\int_if_odd_p:n` A predicate function.
`\int_if_odd:nTF`
`\int_if_even_p:n`
`\int_if_even:nTF`

```

3616 \prg_set_conditional:Npnn \int_if_odd:n #1 {p,TF,T,F} {
3617   \if_int_odd:w \int_eval:w #1\int_eval_end:
3618   \prg_return_true: \else: \prg_return_false: \fi:
3619 }
3620 \prg_set_conditional:Npnn \int_if_even:n #1 {p,TF,T,F} {
3621   \if_int_odd:w \int_eval:w #1\int_eval_end:
3622   \prg_return_false: \else: \prg_return_true: \fi:
3623 }
```

`\int_while_do:nn` These are quite easy given the above functions. The while versions test first and then
`\int_until_do:nn` execute the body. The do_while does it the other way round.
`\int_do_while:nn`
`\int_do_until:nn`

```

3624 \cs_set:Npn \int_while_do:nn #1#2{
3625   \int_compare:nT {#1}{#2} \int_while_do:nn {#1}{#2}}
3626 }
3627 \cs_set:Npn \int_until_do:nn #1#2{
3628   \int_compare:nF {#1}{#2} \int_until_do:nn {#1}{#2}}
3629 }
3630 \cs_set:Npn \int_do_while:nn #1#2{
3631   #2 \int_compare:nT {#1}{\int_do_while:nNnn {#1}{#2}}
3632 }
3633 \cs_set:Npn \int_do_until:nn #1#2{
3634   #2 \int_compare:nF {#1}{\int_do_until:nn {#1}{#2}}
3635 }
```

`\int_while_do:nNnn` As above but not using the more natural syntax.
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

```

3636 \cs_set:Npn \int_while_do:nNnn #1#2#3#4{
3637   \int_compare:nNnT {#1}#2{#3}{#4} \int_while_do:nNnn {#1}#2{#3}{#4}}
3638 }
3639 \cs_set:Npn \int_until_do:nNnn #1#2#3#4{
3640   \int_compare:nNnF {#1}#2{#3}{#4} \int_until_do:nNnn {#1}#2{#3}{#4}}
3641 }
3642 \cs_set:Npn \int_do_while:nNnn #1#2#3#4{
3643   #4 \int_compare:nNnT {#1}#2{#3}{\int_do_while:nNnn {#1}#2{#3}{#4}}
3644 }
3645 \cs_set:Npn \int_do_until:nNnn #1#2#3#4{
3646   #4 \int_compare:nNnF {#1}#2{#3}{\int_do_until:nNnn {#1}#2{#3}{#4}}
3647 }
```

101.5 Defining constants

`\int_const:Nn` As stated, most constants can be defined as `\tex_chardef:D` or `\tex_mathchardef:D`
`\int_const:cn` but that's engine dependent.

```

3648 \cs_new_protected_nopar:Npn \int_const:Nn #1#2 {
```

```

3649 \int_compare:nTF { #2 > \c_minus_one }
3650 {
3651   \int_compare:nTF { #2 > \c_max_register_int }
3652   {
3653     \int_new:N #1
3654     \int_gset:Nn #1 {#2}
3655   }
3656   {
3657     \chk_if_free_cs:N #1
3658     \tex_global:D \tex_mathchardef:D #1 = \int_eval:n {#2}
3659   }
3660 }
3661 {
3662   \int_new:N #1
3663   \int_gset:Nn #1 {#2}
3664 }
3665 }
3666 \cs_generate_variant:Nn \int_const:Nn { c }

```

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`

```

3667 \tex_mathchardef:D \c_max_register_int = 32767 \scan_stop:

```

`\c_minus_one` And the usual constants, others are still missing. Please, make every constant a real constant at least for the moment. We can easily convert things in the end when we have found what constants are used in critical places and what not.

```

\c_zero
\c_one
\c_two
\c_three 3668 %% \tex_countdef:D \c_minus_one = 10 \scan_stop:
\c_four 3669 %% \c_minus_one = -1 \scan_stop: %% in l3basics
\c_five 3670 %\int_const:Nn \c_zero {0}
\c_six 3671 \int_const:Nn \c_one {1}
\c_seven 3672 \int_const:Nn \c_two {2}
\c_eight 3673 \int_const:Nn \c_three {3}
\c_nine 3674 \int_const:Nn \c_four {4}
\c_ten 3675 \int_const:Nn \c_five {5}
\c_eleven 3676 \int_const:Nn \c_six {6}
\c_twelve 3677 \int_const:Nn \c_seven {7}
\c_thirteen 3678 \int_const:Nn \c_eight {8}
\c_fourteen 3679 \int_const:Nn \c_nine {9}
\c_fifteen 3680 \int_const:Nn \c_ten {10}
\c_sixteen 3681 \int_const:Nn \c_eleven {11}
\c_thirty_two 3682 \int_const:Nn \c_twelve {12}
\c_hundred_one 3683 \int_const:Nn \c_thirteen {13}
\c_twohundred_fifty_five 3684 \int_const:Nn \c_fourteen {14}
\c_twohundred_fifty_six 3685 \int_const:Nn \c_fifteen {15}
\c_thousand 3686 %% \tex_chardef:D \c_sixteen = 16 \scan_stop: %% in l3basics
\c_ten_thousand 3687 \int_const:Nn \c_thirty_two {32}
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three
\c_ten_thousand_four
\c_twenty_thousand

```

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```

3688 \int_const:Nn \c_hundred_one {101}
3689 \int_const:Nn \c_twohundred_fifty_five{255}
3690 \int_const:Nn \c_twohundred_fifty_six {256}
3691 \int_const:Nn \c_thousand {1000}
3692 \int_const:Nn \c_ten_thousand {10000}
3693 \int_const:Nn \c_ten_thousand_one {10001}
3694 \int_const:Nn \c_ten_thousand_two {10002}
3695 \int_const:Nn \c_ten_thousand_three {10003}
3696 \int_const:Nn \c_ten_thousand_four {10004}
3697 \int_const:Nn \c_twenty_thousand {20000}

```

`\c_max_int` The largest number allowed is $2^{31} - 1$

```

3698 \int_const:Nn \c_max_int {2147483647}

```

`\c_int_from_roman_i_int` Delayed from earlier.

```

\c_int_from_roman_v_int
\c_int_from_roman_x_int
\l_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
3699 \int_const:cn { c_int_from_roman_i_int } { 1 }
3700 \int_const:cn { c_int_from_roman_v_int } { 5 }
3701 \int_const:cn { c_int_from_roman_x_int } { 10 }
3702 \int_const:cn { c_int_from_roman_l_int } { 50 }
3703 \int_const:cn { c_int_from_roman_c_int } { 100 }
3704 \int_const:cn { c_int_from_roman_d_int } { 500 }
3705 \int_const:cn { c_int_from_roman_m_int } { 1000 }
3706 \int_const:cn { c_int_from_roman_I_int } { 1 }
3707 \int_const:cn { c_int_from_roman_V_int } { 5 }
3708 \int_const:cn { c_int_from_roman_X_int } { 10 }
3709 \int_const:cn { c_int_from_roman_L_int } { 50 }
3710 \int_const:cn { c_int_from_roman_C_int } { 100 }
3711 \int_const:cn { c_int_from_roman_D_int } { 500 }
3712 \int_const:cn { c_int_from_roman_M_int } { 1000 }

```

Needed from the `tl` module:

```

3713 \int_new:N \g_tl_inline_level_int

```

101.6 Backwards compatibility

```

3714 \cs_set_eq:NN \intexpr_value:w \int_value:w
3715 \cs_set_eq:NN \intexpr_eval:w \int_eval:w
3716 \cs_set_eq:NN \intexpr_eval_end: \int_eval_end:
3717 \cs_set_eq:NN \if_intexpr_compare:w \if_int_compare:w
3718 \cs_set_eq:NN \if_intexpr_odd:w \if_int_odd:w
3719 \cs_set_eq:NN \if_intexpr_case:w \if_case:w
3720 \cs_set_eq:NN \intexpr_eval:n \int_eval:n
3721

```

```

3722 \cs_set_eq:NN \intexpr_compare_p:n \int_compare_p:n
3723 \cs_set_eq:NN \intexpr_compare:nTF \int_compare:nTF
3724 \cs_set_eq:NN \intexpr_compare:nT \int_compare:nT
3725 \cs_set_eq:NN \intexpr_compare:nF \int_compare:nF
3726
3727 \cs_set_eq:NN \intexpr_compare_p:nNn \int_compare_p:nNn
3728 \cs_set_eq:NN \intexpr_compare:nNnTF \int_compare:nNnTF
3729 \cs_set_eq:NN \intexpr_compare:nNnT \int_compare:nNnT
3730 \cs_set_eq:NN \intexpr_compare:nNnF \int_compare:nNnF
3731
3732 \cs_set_eq:NN \intexpr_abs:n \int_abs:n
3733 \cs_set_eq:NN \intexpr_max:nn \int_max:nn
3734 \cs_set_eq:NN \intexpr_min:nn \int_min:nn
3735
3736 \cs_set_eq:NN \intexpr_div_truncate:nn \int_div_truncate:nn
3737 \cs_set_eq:NN \intexpr_div_round:nn \int_div_round:nn
3738 \cs_set_eq:NN \intexpr_mod:nn \int_mod:nn
3739
3740 \cs_set_eq:NN \intexpr_if_odd_p:n \int_if_odd_p:n
3741 \cs_set_eq:NN \intexpr_if_odd:nTF \int_if_odd:nTF
3742 \cs_set_eq:NN \intexpr_if_odd:nT \int_if_odd:nT
3743 \cs_set_eq:NN \intexpr_if_odd:nF \int_if_odd:nF
3744
3745 \cs_set_eq:NN \intexpr_if_even_p:n \int_if_even_p:n
3746 \cs_set_eq:NN \intexpr_if_even:nTF \int_if_even:nTF
3747 \cs_set_eq:NN \intexpr_if_even:nT \int_if_even:nT
3748 \cs_set_eq:NN \intexpr_if_even:nF \int_if_even:nF
3749
3750 \cs_set_eq:NN \intexpr_while_do:nn \int_while_do:nn
3751 \cs_set_eq:NN \intexpr_until_do:nn \int_until_do:nn
3752 \cs_set_eq:NN \intexpr_do_while:nn \int_do_while:nn
3753 \cs_set_eq:NN \intexpr_do_until:nn \int_do_until:nn
3754
3755 \cs_set_eq:NN \intexpr_while_do:nNnn \int_while_do:nNnn
3756 \cs_set_eq:NN \intexpr_until_do:nNnn \int_until_do:nNnn
3757 \cs_set_eq:NN \intexpr_do_while:nNnn \int_do_while:nNnn
3758 \cs_set_eq:NN \intexpr_do_until:nNnn \int_do_until:nNnn
3759 </initex | package>

```

Show token usage:

```

3760 <*showmemory>
3761 \showMemUsage
3762 </showmemory>

```

102 l3skip implementation

We start by ensuring that the required packages are loaded.

```

3763 <*package>
3764 \ProvidesExplPackage
3765   {\filename}{\filedate}{\fileversion}{\filedescription}
3766 \package_check_loaded_expl:
3767 </package>
3768 <*initex | package>

```

102.1 Skip registers

`\skip_new:N` Allocation of a new internal registers.

```

\skip_new:c
3769 <*initex>
3770 \alloc_new:nnnN {skip} \c_zero \c_max_register_int \tex_skipdef:D
3771 </initex>
3772 <*package>
3773 \cs_new_protected_nopar:Npn \skip_new:N #1 {
3774   \chk_if_free_cs:N #1
3775   \newskip #1
3776 }
3777 </package>
3778 \cs_generate_variant:Nn \skip_new:N {c}

```

`\skip_set:Nn` Setting skips is again something that I would like to make uniform at the moment to get
`\skip_set:cn` a better overview.

```

\skip_gset:Nn
\skip_gset:cn
3779 \cs_new_protected_nopar:Npn \skip_set:Nn #1#2 {
3780   #1 \etex_glueexpr:D #2 \scan_stop:
3781 <*check>
3782 \chk_local_or_pref_global:N #1
3783 </check>
3784 }
3785 \cs_new_protected_nopar:Npn \skip_gset:Nn {
3786 <*check>
3787   \pref_global_chk:
3788 </check>
3789 <-check> \pref_global:D
3790   \skip_set:Nn
3791 }
3792 \cs_generate_variant:Nn \skip_set:Nn {cn}
3793 \cs_generate_variant:Nn \skip_gset:Nn {cn}

```

`\skip_zero:N` Reset the register to zero.

```

\skip_gzero:N
\skip_zero:c
\skip_gzero:c
3794 \cs_new_protected_nopar:Npn \skip_zero:N #1{
3795   #1\c_zero_skip \scan_stop:
3796 <*check>
3797   \chk_local_or_pref_global:N #1
3798 </check>
3799 }
3800 \cs_new_protected_nopar:Npn \skip_gzero:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```

3801 <*check>
3802   \pref_global_chk:
3803 </check>
3804 <-check> \pref_global:D
3805   \skip_zero:N
3806 }
3807 \cs_generate_variant:Nn \skip_zero:N {c}
3808 \cs_generate_variant:Nn \skip_gzero:N {c}

```

`\skip_add:Nn` Adding and subtracting to and from `<skip>s`

`\skip_add:cn`

`\skip_gadd:Nn`

`\skip_gadd:cn`

`\skip_sub:Nn`

`\skip_gsub:Nn`

```

3809 \cs_new_protected_nopar:Npn \skip_add:Nn #1#2 {

```

We need to say by in case the first argument is a register accessed by its number, e.g., `\skip23`.

```

3810   \tex_advance:D#1 by \etex_glueexpr:D #2 \scan_stop:
3811 <*check>
3812   \chk_local_or_pref_global:N #1
3813 </check>
3814 }
3815 \cs_generate_variant:Nn \skip_add:Nn {cn}

3816 \cs_new_protected_nopar:Npn \skip_sub:Nn #1#2{
3817   \tex_advance:D #1 -\etex_glueexpr:D #2 \scan_stop:
3818 <*check>
3819   \chk_local_or_pref_global:N #1
3820 </check>
3821 }

3822 \cs_new_protected_nopar:Npn \skip_gadd:Nn {
3823 <*check>
3824   \pref_global_chk:
3825 </check>
3826 <-check> \pref_global:D
3827   \skip_add:Nn
3828 }
3829 \cs_generate_variant:Nn \skip_gadd:Nn {cn}

3830 \cs_new_nopar:Npn \skip_gsub:Nn {
3831 <*check>
3832   \pref_global_chk:
3833 </check>
3834 <-check> \pref_global:D

```



```

3835 \skip_sub:Nn
3836 }

\skip_horizontal:N Inserting skips.
\skip_horizontal:c
\skip_horizontal:n 3837 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_vertical:N 3838 \cs_generate_variant:Nn \skip_horizontal:N {c}
\skip_vertical:c 3839 \cs_new_nopar:Npn \skip_horizontal:n #1 {
\skip_vertical:n 3840 \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop:
3841 }
3842 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
3843 \cs_generate_variant:Nn \skip_vertical:N {c}
3844 \cs_new_nopar:Npn \skip_vertical:n #1 {
3845 \skip_vertical:N \etex_glueexpr:D #1 \scan_stop:
3846 }

\skip_use:N Here is how skip registers are accessed:
\skip_use:c 3847 \cs_new_eq:NN \skip_use:N \tex_the:D
3848 \cs_generate_variant:Nn \skip_use:N {c}

\skip_show:N Diagnostics.
\skip_show:c 3849 \cs_new_eq:NN \skip_show:N \kernel_register_show:N
3850 \cs_generate_variant:Nn \skip_show:N {c}

\skip_eval:n Evaluating a calc expression.

3851 \cs_new_protected_nopar:Npn \skip_eval:n #1 {
3852 \tex_the:D \etex_glueexpr:D #1 \scan_stop:
3853 }

\l_tmpa_skip We provide three local and two global scratch registers, maybe we need more or less.
\l_tmpb_skip
\l_tmpc_skip 3854 %%\chk_if_free_cs:N \l_tmpa_skip
3855 %%\tex_skipdef:D\l_tmpa_skip 255 %currently taken up by \skip@
\g_tmpa_skip 3856 \skip_new:N \l_tmpa_skip
\g_tmpb_skip 3857 \skip_new:N \l_tmpb_skip
3858 \skip_new:N \l_tmpc_skip
3859 \skip_new:N \g_tmpa_skip
3860 \skip_new:N \g_tmpb_skip

\c_zero_skip
\c_max_skip 3861 \<!*package>
3862 \skip_new:N \c_zero_skip
3863 \skip_set:Nn \c_zero_skip {0pt}
3864 \skip_new:N \c_max_skip

```

```

3865 \skip_set:Nn \c_max_skip {16383.99999pt}
3866 \!/package>
3867 \*!initex>
3868 \cs_set_eq:NN \c_zero_skip \z@
3869 \cs_set_eq:NN \c_max_skip \maxdimen
3870 \!/initex>

```

`\skip_if_infinite_glue_p:n`
`\skip_if_infinite_glue:nTF`

With ε -TEX we all of a sudden get access to a lot of information we should otherwise consider ourselves lucky to get. One is the stretch and shrink components of a skip register and the order of those components. `\skip_if_infinite_glue:nTF` tests it directly by looking at the stretch and shrink order. If either of the predicate functions return $\langle true \rangle$ `\bool_if:nTF` will return $\langle true \rangle$ and the logic test will take the true branch.

```

3871 \prg_new_conditional:Nnn \skip_if_infinite_glue:n {p,TF,T,F} {
3872   \bool_if:nTF {
3873     \int_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
3874     \int_compare_p:nNn {\etex_glueshrinkorder:D #1 } > \c_zero
3875   } {\prg_return_true:} {\prg_return_false:}
3876 }

```

`\p_split_finite_else_action:nnNN`

This macro is useful when performing error checking in certain circumstances. If the $\langle skip \rangle$ 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 global.

```

3877 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
3878   \skip_if_infinite_glue:nTF {#1}
3879   {
3880     #3 = \c_zero_skip
3881     #4 = \c_zero_skip
3882     #2
3883   }
3884   {
3885     #3 = \etex_gluestretch:D #1 \scan_stop:
3886     #4 = \etex_glueshrink:D #1 \scan_stop:
3887   }
3888 }

```

102.2 Dimen registers

`\dim_new:N` Allocating $\langle dim \rangle$ registers...
`\dim_new:c`

```

3889 \*!initex>
3890 \alloc_new:nnnN {dim} \c_zero \c_max_register_int \tex_dimendef:D
3891 \!/initex>
3892 \*package>
3893 \cs_new_protected_nopar:Npn \dim_new:N #1 {
3894   \chk_if_free_cs:N #1

```

```

3895 \newdimen #1
3896 }
3897 </package>
3898 \cs_generate_variant:Nn \dim_new:N {c}

```

`\dim_set:Nn` We add `\dim_eval:n` in order to allow simple arithmetic and a space just for those using `\dimen1` or alike. See OR!

```

\dim_set:cn
\dim_set:Nc
\dim_gset:Nn
\dim_gset:cn
\dim_gset:Nc
\dim_gset:cc

3899 \cs_new_protected_nopar:Npn \dim_set:Nn #1#2 {
3900   #1~ \etex_dimexpr:D #2 \scan_stop:
3901 }
3902 \cs_generate_variant:Nn \dim_set:Nn {cn,Nc}

3903 \cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
3904 \cs_generate_variant:Nn \dim_gset:Nn {cn,Nc,cc}

```

`\dim_set_max:Nn` Setting maximum and minimum values is simply a case of so build-in comparison. This only applies to dimensions as skips are not ordered.

```

\dim_set_max:cn
\dim_set_min:Nn
\dim_set_min:cn
\dim_gset_max:Nn
\dim_gset_max:cn
\dim_gset_min:Nn
\dim_gset_min:cn

3905 \cs_new_protected_nopar:Npn \dim_set_max:Nn #1#2 {
3906   \dim_compare:nNnT {#1} < {#2} { \dim_set:Nn #1 {#2} }
3907 }
3908 \cs_generate_variant:Nn \dim_set_max:Nn { c }
3909 \cs_new_protected_nopar:Npn \dim_set_min:Nn #1#2 {
3910   \dim_compare:nNnT {#1} > {#2} { \dim_set:Nn #1 {#2} }
3911 }
3912 \cs_generate_variant:Nn \dim_set_min:Nn { c }
3913 \cs_new_protected_nopar:Npn \dim_gset_max:Nn #1#2 {
3914   \dim_compare:nNnT {#1} < {#2} { \dim_gset:Nn #1 {#2} }
3915 }
3916 \cs_generate_variant:Nn \dim_gset_max:Nn { c }
3917 \cs_new_protected_nopar:Npn \dim_gset_min:Nn #1#2 {
3918   \dim_compare:nNnT {#1} > {#2} { \dim_gset:Nn #1 {#2} }
3919 }
3920 \cs_generate_variant:Nn \dim_gset_min:Nn { c }

```

`\dim_zero:N` Resetting.

```

\dim_gzero:N
\dim_zero:c
\dim_gzero:c

3921 \cs_new_protected_nopar:Npn \dim_zero:N #1 { #1\c_zero_skip }
3922 \cs_generate_variant:Nn \dim_zero:N {c}

3923 \cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3924 \cs_generate_variant:Nn \dim_gzero:N {c}

```

`\dim_add:Nn` Addition.

```

\dim_add:cn
\dim_add:Nc
\dim_gadd:Nn
\dim_gadd:cn

3925 \cs_new_protected_nopar:Npn \dim_add:Nn #1#2{

```

We need to say by in case the first argument is a register accessed by its number, e.g., `\dimen23`.

```

3926      \tex_advance:D#1 by \etex_dimexpr:D #2 \scan_stop:
3927    }
3928    \cs_generate_variant:Nn \dim_add:Nn {cn,Nc}

3929    \cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3930    \cs_generate_variant:Nn \dim_gadd:Nn {cn}

```

`\dim_sub:Nn` Subtracting.

```

\dim_sub:cn
\dim_sub:Nc
\dim_gsub:Nn
\dim_gsub:cn

3931    \cs_new_protected_nopar:Npn \dim_sub:Nn #1#2 { \tex_advance:D#1-#2\scan_stop: }
3932    \cs_generate_variant:Nn \dim_sub:Nn {cn,Nc}

3933    \cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3934    \cs_generate_variant:Nn \dim_gsub:Nn {cn}

```

`\dim_use:N` Accessing a $\langle dim \rangle$.

```

\dim_use:c

3935    \cs_new_eq:NN \dim_use:N \tex_the:D
3936    \cs_generate_variant:Nn \dim_use:N {c}

```

`\dim_show:N` Diagnostics.

```

\dim_show:c

3937    \cs_new_eq:NN \dim_show:N \kernel_register_show:N
3938    \cs_generate_variant:Nn \dim_show:N {c}

```

`\l_tmpa_dim` Some scratch registers.

```

\l_tmpb_dim
\l_tmppc_dim
\l_tmppd_dim
\g_tmpa_dim
\g_tmppb_dim

3939    \dim_new:N \l_tmpa_dim
3940    \dim_new:N \l_tmppb_dim
3941    \dim_new:N \l_tmppc_dim
3942    \dim_new:N \l_tmppd_dim
3943    \dim_new:N \g_tmpa_dim
3944    \dim_new:N \g_tmppb_dim

```

`\c_zero_dim` Just aliases.

```

\c_max_dim

3945    \cs_new_eq:NN \c_zero_dim \c_zero_skip
3946    \cs_new_eq:NN \c_max_dim \c_max_skip

```

`\dim_eval:n` Evaluating a calc expression.

```

3947    \cs_new_protected_nopar:Npn \dim_eval:n #1 {
3948      \tex_the:D \etex_dimexpr:D #1 \scan_stop:
3949    }

```

```

\if_dim:w Primitives.
\dim_value:w
\dim_eval:w
\dim_eval_end:
3950 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
3951 \cs_set_eq:NN \dim_value:w \tex_number:D
3952 \cs_set_eq:NN \dim_eval:w \tex_dimexpr:D
3953 \cs_set_protected:Npn \dim_eval_end: {\tex_relax:D}

```

```

\dim_compare_p:nNn
\dim_compare:nNnTF
3954 \prg_new_conditional:Nnn \dim_compare:nNn {p,TF,T,F} {
3955   \if_dim:w \tex_dimexpr:D #1 #2 \tex_dimexpr:D #3 \scan_stop:
3956   \prg_return_true: \else: \prg_return_false: \fi:
3957 }

```

`\dim_compare_p:n` [This code plus comments lifted directly from the `\int_compare:nTF` function.] Comparison tests using a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. First some notes on the idea behind this. We wish to support writing code like

```
\dim_compare_p:n { 5 + \l_tmpa_dim != 4 - \l_tmpb_dim }
```

In other words, we want to somehow add the missing `\dim_eval:w` where required. We can start evaluating from the left using `\dim:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let \TeX evaluate this left hand side of the (in)equality.

```

3958 \prg_new_conditional:Npnn \dim_compare:n #1 {p,TF,T,F} {
3959   \exp_after:wN \dim_compare_auxi:w \dim_value:w
3960   \dim_eval:w #1 \q_stop
3961 }

```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `\langle null \rangle`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3962 \cs_new:Npn \dim_compare_auxi:w #1#2 \q_stop {
3963   \exp_after:wN \dim_compare_auxii:w \tex_romannumeral:D
3964   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3965 }

```

This leaves the first relation symbol in front and assuming the right hand side has been input, at least one other token as well. We support the following forms: `=`, `<`, `>` and the extended `!=`, `==`, `<=` and `>=`. All the extended forms have an extra `=` so we check if that is present as well. Then use specific function to perform the test.

```

3966 \cs_new:Npn \dim_compare_auxii:w #1#2#3\q_mark{
3967   \use:c{
3968     dim_compare_ #1 \if_meaning:w =#2 = \fi:
3969     :w}
3970 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3971 \cs_new:cpn {dim_compare_=:w} #1 = #2 \q_stop {
3972   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3973   \prg_return_true: \else: \prg_return_false: \fi:
3974 }

```

So is the one using == – we just have to use == in the parameter text.

```

3975 \cs_new:cpn {dim_compare_==:w} #1 == #2 \q_stop {
3976   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3977   \prg_return_true: \else: \prg_return_false: \fi:
3978 }

```

Not equal is just about reversing the truth value.

```

3979 \cs_new:cpn {dim_compare_!=:w} #1 != #2 \q_stop {
3980   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3981   \prg_return_false: \else: \prg_return_true: \fi:
3982 }

```

Less than and greater than are also straight forward.

```

3983 \cs_new:cpn {dim_compare_<:w} #1 < #2 \q_stop {
3984   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
3985   \prg_return_true: \else: \prg_return_false: \fi:
3986 }
3987 \cs_new:cpn {dim_compare_>:w} #1 > #2 \q_stop {
3988   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
3989   \prg_return_true: \else: \prg_return_false: \fi:
3990 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

3991 \cs_new:cpn {dim_compare_<=:w} #1 <= #2 \q_stop {
3992   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
3993   \prg_return_false: \else: \prg_return_true: \fi:
3994 }
3995 \cs_new:cpn {dim_compare_>=:w} #1 >= #2 \q_stop {
3996   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
3997   \prg_return_false: \else: \prg_return_true: \fi:
3998 }

```

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
3999 \cs_new_nopar:Npn \dim_while_do:nNnn #1#2#3#4{
4000   \dim_compare:nNnT {#1}#2{#3}{#4 \dim_while_do:nNnn {#1}#2{#3}{#4}}
4001 }
4002 \cs_new_nopar:Npn \dim_until_do:nNnn #1#2#3#4{
4003   \dim_compare:nNnF {#1}#2{#3}{#4 \dim_until_do:nNnn {#1}#2{#3}{#4}}
4004 }
4005 \cs_new_nopar:Npn \dim_do_while:nNnn #1#2#3#4{
4006   #4 \dim_compare:nNnT {#1}#2{#3}{\dim_do_while:nNnn {#1}#2{#3}{#4}}
4007 }
4008 \cs_new_nopar:Npn \dim_do_until:nNnn #1#2#3#4{
4009   #4 \dim_compare:nNnF {#1}#2{#3}{\dim_do_until:nNnn {#1}#2{#3}{#4}}
4010 }

```

102.3 Muskip

`\muskip_new:N` And then we add muskips.

```

4011 <*initex>
4012 \alloc_new:nnnN {muskip} \c_zero \c_max_register_int \tex_muskipdef:D
4013 </initex>
4014 <*package>
4015 \cs_new_protected_nopar:Npn \muskip_new:N #1 {
4016   \chk_if_free_cs:N #1
4017   \newmuskip #1
4018 }
4019 </package>

```

`\muskip_set:Nn` Simple functions for muskips.

```

\muskip_gset:Nn
\muskip_add:Nn
\muskip_gadd:Nn
\muskip_sub:Nn
\muskip_gsub:Nn
4020 \cs_new_protected_nopar:Npn \muskip_set:Nn#1#2{#1\etex_muexpr:D#2\scan_stop:}
4021 \cs_new_protected_nopar:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
4022 \cs_new_protected_nopar:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
4023 \cs_new_protected_nopar:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
4024 \cs_new_protected_nopar:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
4025 \cs_new_protected_nopar:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}

```

`\muskip_use:N` Accessing a `<muskip>`.

```

4026 \cs_new_eq:NN \muskip_use:N \tex_the:D

```

`\muskip_show:N`

```

4027 \cs_new_eq:NN \muskip_show:N \kernel_register_show:N
4028 </initex | package>

```

103 l3tl implementation

We start by ensuring that the required packages are loaded.

```
4029 <*package>
4030 \ProvidesExplPackage
4031   {\filename}{\filedate}{\fileversion}{\filedescription}
4032 \package_check_loaded_expl:
4033 </package>

4034 <*initex | package>
```

A token list variable is a control sequence that holds tokens. The interface is similar to that for token registers, but beware that the behavior vis á vis `\cs_set_nopar:Npx` etc. ... is different. (You see this comes from Denys' implementation.)

103.1 Functions

`\tl_new:N` We provide one allocation function (which checks that the name is not used) and two
`\tl_new:c` clear functions that locally or globally clear the token list. The allocation function has
`\tl_new:Nn` two arguments to specify an initial value. This is the only way to give values to constants.
`\tl_new:cn`
`\tl_new:Nx`

```
4035 \cs_new_protected:Npn \tl_new:Nn #1#2{
4036   \chk_if_free_cs:N #1
```

If checking we don't allow constants to be defined.

```
4037 <*check>
4038   \chk_var_or_const:N #1
4039 </check>
```

Otherwise any variable type is allowed.

```
4040   \cs_gset_nopar:Npn #1{#2}
4041 }
4042 \cs_generate_variant:Nn \tl_new:Nn {cn}
4043 \cs_new_protected:Npn \tl_new:Nx #1#2{
4044   \chk_if_free_cs:N #1
4045 <check> \chk_var_or_const:N #1
4046   \cs_gset_nopar:Npx #1{#2}
4047 }
4048 \cs_new_protected_nopar:Npn \tl_new:N #1{\tl_new:Nn #1{}}
4049 \cs_new_protected_nopar:Npn \tl_new:c #1{\tl_new:cn {#1}{}}
```

`\tl_const:Nn` For creating constant token lists: there is not actually anything here that cannot be achieved using `\tl_new:N` and `\tl_set:Nn`

```
4050 \cs_new_protected:Npn \tl_const:Nn #1#2 {
4051   \tl_new:N #1
4052   \tl_gset:Nn #1 {#2}
4053 }
```


`\tl_use:N` Perhaps this should just be enabled when checking?

`\tl_use:c`

```

4054 \cs_new_nopar:Npn \tl_use:N #1 {
4055   \if_meaning:w #1 \tex_relax:D

```

If $\langle tl\ var.\rangle$ equals `\tex_relax:D` it is probably stemming from a `\cs:w... \cs_end:` that was created by mistake somewhere.

```

4056   \msg_kernel_bug:x {Token~list~variable~ '\token_to_str:N #1'~
4057                      has~ an~ erroneous~ structure!}
4058   \else:
4059     \exp_after:wN #1
4060   \fi:
4061 }
4062 \cs_generate_variant:Nn \tl_use:N {c}

```

`\tl_show:N` Showing a $\langle tl\ var.\rangle$ is just `\show`ing it and I don't really care about checking that it's malformed at this stage.

`\tl_show:c`

`\tl_show:n`

```

4063 \cs_new_nopar:Npn \tl_show:N #1 { \cs_show:N #1 }
4064 \cs_generate_variant:Nn \tl_show:N {c}
4065 \cs_set_eq:NN \tl_show:n \etex_showtokens:D

```

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens.

```

\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: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:cx

```

```

4066 \cs_new_protected:Npn \tl_set:Nn #1#2 {
4067   \cs_set_nopar:Npx #1 { \exp_not:n {#2} }
4068 }
4069 \cs_new_protected:Npn \tl_set:Nx #1#2 {
4070   \cs_set_nopar:Npx #1 {#2}
4071 }
4072 \cs_new_protected:Npn \tl_gset:Nn #1#2 {
4073   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4074 }
4075 \cs_new_protected:Npn \tl_gset:Nx #1#2 {
4076   \cs_gset_nopar:Npx #1 {#2}
4077 }
4078 \cs_generate_variant:Nn \tl_set:Nn { NV }
4079 \cs_generate_variant:Nn \tl_set:Nn { Nv }
4080 \cs_generate_variant:Nn \tl_set:Nn { No }
4081 \cs_generate_variant:Nn \tl_set:Nn { Nf }
4082 \cs_generate_variant:Nn \tl_set:Nn { cV }
4083 \cs_generate_variant:Nn \tl_set:Nn { c }
4084 \cs_generate_variant:Nn \tl_set:Nn { cv }
4085 \cs_generate_variant:Nn \tl_set:Nn { co }
4086 \cs_generate_variant:Nn \tl_set:Nx { c }
4087 \cs_generate_variant:Nn \tl_gset:Nn { NV }
4088 \cs_generate_variant:Nn \tl_gset:Nn { Nv }
4089 \cs_generate_variant:Nn \tl_gset:Nn { No }
4090 \cs_generate_variant:Nn \tl_gset:Nn { Nf }

```

```

4091 \cs_generate_variant:Nn \tl_gset:Nn { c }
4092 \cs_generate_variant:Nn \tl_gset:Nn { cV }
4093 \cs_generate_variant:Nn \tl_gset:Nn { cv }
4094 \cs_generate_variant:Nn \tl_gset:Nx { c }

```

\tl_set_eq:NN For setting token list variables equal to each other. First checking:

```

\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cN
\tl_set_eq:cc
4095 \tl_set_eq:cc { *check }
4096 \cs_new_protected_nopar:Npn \tl_set_eq:NN #1#2{
4097   \chk_exist_cs:N #1 \cs_set_eq:NN #1#2
4098   \chk_local_or_pref_global:N #1 \chk_var_or_const:N #2
4099 }
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
4100 \tl_gset_eq:cc { *check }
4101 \cs_new_protected_nopar:Npn \tl_gset_eq:NN #1#2{
4102   \chk_exist_cs:N #1 \cs_gset_eq:NN #1#2
4103   \chk_global:N #1 \chk_var_or_const:N #2
4104 }
4105 \tl_gset_eq:cc { /check }

```

Non-checking versions are easy.

```

4105 \tl_set_eq:Nc { !check }
4106 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4107 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4108 \tl_gset_eq:Nc { !check }

```

The rest again with the expansion module.

```

4109 \cs_generate_variant:Nn \tl_set_eq:NN { Nc,c,cc }
4110 \cs_generate_variant:Nn \tl_gset_eq:NN { Nc,c,cc }

```

\tl_clear:N Clearing a token list variable.

```

\tl_clear:c
\tl_gclear:N
\tl_gclear:c
4111 \cs_new_protected_nopar:Npn \tl_clear:N #1{\tl_set_eq:NN #1\c_empty_tl}
4112 \cs_generate_variant:Nn \tl_clear:N { c }
4113 \cs_new_protected_nopar:Npn \tl_gclear:N #1{\tl_gset_eq:NN #1\c_empty_tl}
4114 \cs_generate_variant:Nn \tl_gclear:N { c }

```

\tl_clear_new:N These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.

```

\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c
4115 \tl_clear_new:c { *check }
4116 \cs_new_protected_nopar:Npn \tl_clear_new:N #1{
4117   \chk_var_or_const:N #1
4118   \if_predicate:w \cs_if_exist_p:N #1
4119     \tl_clear:N #1
4120   \else:
4121     \tl_new:N #1
4122   \fi:
4123 }
4124 \tl_clear_new:c { /check }
4125 \cs_new_eq:NN \tl_clear_new:N \tl_clear:N
4126 \cs_generate_variant:Nn \tl_clear_new:N { c }

```

\tl_gclear_new:N These are the global versions of the above.

```
\tl_gclear_new:c
4127 <*check>
4128 \cs_new_protected_nopar:Npn \tl_gclear_new:N #1{
4129   \chk_var_or_const:N #1
4130   \if_predicate:w \cs_if_exist_p:N #1
4131     \tl_gclear:N #1
4132   \else:
4133     \tl_new:N #1
4134   \fi:}
4135 </check>
4136 <-check>\cs_new_eq:NN \tl_gclear_new:N \tl_gclear:N
4137 \cs_generate_variant:Nn \tl_gclear_new:N {c}
```

\tl_put_right:Nn Adding to one end of a token list is done partially using hand tuned functions for performance reasons.

```
\tl_put_right:NV
\tl_put_right:Nv
\tl_put_right:No
\tl_put_right:Nx
\tl_put_right:cn
\tl_put_right:cV
\tl_put_right:cv
\tl_put_right:cx
\tl_gput_right:Nn
\tl_gput_right:NV
\tl_gput_right:Nv
\tl_gput_right:No
\tl_gput_right:Nx
\tl_gput_right:cn
\tl_gput_right:cV
\tl_gput_right:cv
\tl_gput_right:co
\tl_gput_right:cx
4138 \cs_new_protected:Npn \tl_put_right:Nn #1#2 {
4139   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4140 }
4141 \cs_new_protected:Npn \tl_put_right:Nv #1#2 {
4142   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4143 }
4144 \cs_new_protected:Npn \tl_put_right:Nv #1#2 {
4145   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4146 }
4147 \cs_new_protected:Npn \tl_put_right:Nx #1#2 {
4148   \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 }
4149 }
4150 \cs_new_protected:Npn \tl_put_right:No #1#2 {
4151   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4152 }
4153 \cs_new_protected:Npn \tl_gput_right:Nn #1#2 {
4154   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4155 }
4156 \cs_new_protected:Npn \tl_gput_right:NV #1#2 {
4157   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4158 }
4159 \cs_new_protected:Npn \tl_gput_right:Nv #1#2 {
4160   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4161 }
4162 \cs_new_protected:Npn \tl_gput_right:No #1#2 {
4163   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4164 }
4165 \cs_new_protected:Npn \tl_gput_right:Nx #1#2 {
4166   \cs_gset_nopar:Npx #1 { \exp_not:o #1 #2 }
4167 }
4168 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4169 \cs_generate_variant:Nn \tl_put_right:Nv { c }
4170 \cs_generate_variant:Nn \tl_put_right:Nv { c }
```

```

4171 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4172 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4173 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4174 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
4175 \cs_generate_variant:Nn \tl_gput_right:No { c }
4176 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

\tl_put_left:Nn Adding to the left is basically the same as putting on the right.

```

\tl_put_left:NV
\tl_put_left:Nv
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:cv
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:Nv
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:cv
\tl_gput_left:cx
4177 \cs_new_protected:Npn \tl_put_left:Nn #1#2 {
4178   \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4179 }
4180 \cs_new_protected:Npn \tl_put_left:NV #1#2 {
4181   \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4182 }
4183 \cs_new_protected:Npn \tl_put_left:Nv #1#2 {
4184   \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4185 }
4186 \cs_new_protected:Npn \tl_put_left:Nx #1#2 {
4187   \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 }
4188 }
4189 \cs_new_protected:Npn \tl_put_left:No #1#2 {
4190   \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4191 }
4192 \cs_new_protected:Npn \tl_gput_left:Nn #1#2 {
4193   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4194 }
4195 \cs_new_protected:Npn \tl_gput_left:NV #1#2 {
4196   \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4197 }
4198 \cs_new_protected:Npn \tl_gput_left:Nv #1#2 {
4199   \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4200 }
4201 \cs_new_protected:Npn \tl_gput_left:No #1#2 {
4202   \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4203 }
4204 \cs_new_protected:Npn \tl_gput_left:Nx #1#2 {
4205   \cs_gset_nopar:Npx #1 { #2 \exp_not:o #1 }
4206 }
4207 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4208 \cs_generate_variant:Nn \tl_put_left:NV { c }
4209 \cs_generate_variant:Nn \tl_put_left:Nv { c }
4210 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4211 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4212 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4213 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
4214 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

\tl_gset:Nc These two functions are included because they are necessary in Denys' implementations.
\tl_set:Nc

The `:Nc` convention (see the expansion module) is very unusual at first sight, but it works nicely over all modules, so we would like to keep it.

Construct a control sequence on the fly from `#2` and save it in `#1`.

```

4215 \cs_new_protected_nopar:Npn \tl_gset:Nc {
4216   \*check
4217   \pref_global_chk:
4218   \</check>
4219   \<-check> \pref_global:D
4220   \tl_set:Nc}

```

`\pref_global_chk:` will turn the variable `check` in `\tl_set:No` into a global check.

```

4221 \cs_new_protected_nopar:Npn \tl_set:Nc #1#2{\tl_set:No #1{\cs:w#2\cs_end:}}

```

103.2 Variables and constants

`\c_job_name_tl` Inherited from the `expl3` name for the primitive: this needs to actually contain the text of the jobname rather than the name of the primitive, of course.

```

4222 \tl_new:N \c_job_name_tl
4223 \tl_set:Nx \c_job_name_tl { \tex_jobname:D }

```

`\c_empty_tl` Two constants which are often used.

```

4224 \tl_const:Nn \c_empty_tl { }

```

`\c_space_tl` A space as a token list (as opposed to as a character).

```

4225 \tl_const:Nn \c_space_tl { ~ }

```

`\g_tmpa_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.

```

4226 \tl_new:N \g_tmpa_tl
4227 \tl_new:N \g_tmpb_tl

```

`\l_kernel_testa_tl` Local temporaries. These are the ones for test routines. This means that one can safely use other temporaries when calling test routines.

```

4228 \tl_new:N \l_kernel_testa_tl
4229 \tl_new:N \l_kernel_testb_tl

```

`\l_tmpa_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.

```

4230 \tl_new:N \l_tmpa_tl
4231 \tl_new:N \l_tmpb_tl

```

`\l_kernel_tmpa_tl` These are local temporary token list variables reserved for use by the kernel. They should
`\l_kernel_tmpp_tl` not be used by other modules.

```
4232 \tl_new:N \l_kernel_tmpa_tl
4233 \tl_new:N \l_kernel_tmpp_tl
```

103.3 Predicates and conditionals

We also provide a few conditionals, both in expandable form (with `\c_true_bool`) and in ‘brace-form’, the latter are denoted by TF at the end, as explained elsewhere.

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the
`\tl_if_empty_p:c` proper code from their argument(s).

```
\tl_if_empty:N $\overline{TF}$ 
\tl_if_empty:c $\overline{TF}$ 
4234 \prg_set_conditional:Npnn \tl_if_empty:N #1 {p,TF,T,F} {
4235   \if_meaning:w #1 \c_empty_tl
4236   \prg_return_true: \else: \prg_return_false: \fi:
4237 }
4238 \cs_generate_variant:Nn \tl_if_empty_p:N {c}
4239 \cs_generate_variant:Nn \tl_if_empty:N $\overline{TF}$  {c}
4240 \cs_generate_variant:Nn \tl_if_empty:NT {c}
4241 \cs_generate_variant:Nn \tl_if_empty:NF {c}
```

`\tl_if_eq_p:NN` Returns `\c_true_bool` iff the two token list variables are equal.

```
\tl_if_eq_p:Nc
\tl_if_eq_p:cN
\tl_if_eq_p:cc
\tl_if_eq:NN $\overline{TF}$ 
\tl_if_eq:Nc $\overline{TF}$ 
\tl_if_eq:cN $\overline{TF}$ 
\tl_if_eq:cc $\overline{TF}$ 
4242 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 {p,TF,T,F} {
4243   \if_meaning:w #1 #2 \prg_return_true: \else: \prg_return_false: \fi:
4244 }
4245 \cs_generate_variant:Nn \tl_if_eq_p:NN {Nc,c,cc}
4246 \cs_generate_variant:Nn \tl_if_eq:NN $\overline{TF}$  {Nc,c,cc}
4247 \cs_generate_variant:Nn \tl_if_eq:NNT {Nc,c,cc}
4248 \cs_generate_variant:Nn \tl_if_eq:NNF {Nc,c,cc}
```

`\tl_if_eq:nn \overline{TF}` A simple store and compare routine.

```
\l_tl_tmpa_tl
\l_tl_tmpp_tl
4249 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF } {
4250   \group_begin:
4251     \tl_set:Nn \l_tl_tmpa_tl {#1}
4252     \tl_set:Nn \l_tl_tmpp_tl {#2}
4253     \tex_ifx:D \l_tl_tmpa_tl \l_tl_tmpp_tl
4254     \group_end:
4255     \prg_return_true:
4256   \tex_else:D
4257     \group_end:
4258     \prg_return_false:
4259   \tex_fi:D
4260 }
4261 \tl_new:N \l_tl_tmpa_tl
4262 \tl_new:N \l_tl_tmpp_tl
```

`\tl_if_empty_p:n` It would be tempting to just use `\if_meaning:w\q_nil#1\q_nil` as a test since this
`\tl_if_empty_p:V` works really well. However it fails on a token list starting with `\q_nil` of course but
`\tl_if_empty_p:o` more troubling is the case where argument is a complete conditional such as `\if_true:`
`\tl_if_empty:nTF` `a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out
`\tl_if_empty:VTF` false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the
`\tl_if_empty:oTF` end starts executing... A safer route is to convert the entire token list into harmless
characters first and then compare that. This way the test will even accept `\q_nil` as the
first token.

```

4263 \prg_new_conditional:Npnn \tl_if_empty:n #1 {p,TF,T,F} {
4264   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4265   \prg_return_true: \else: \prg_return_false: \fi:
4266 }
4267 \cs_generate_variant:Nn \tl_if_empty_p:n {V}
4268 \cs_generate_variant:Nn \tl_if_empty:nTF {V}
4269 \cs_generate_variant:Nn \tl_if_empty:nT {V}
4270 \cs_generate_variant:Nn \tl_if_empty:nF {V}
4271 \cs_generate_variant:Nn \tl_if_empty_p:n {o}
4272 \cs_generate_variant:Nn \tl_if_empty:nTF {o}
4273 \cs_generate_variant:Nn \tl_if_empty:nT {o}
4274 \cs_generate_variant:Nn \tl_if_empty:nF {o}

```

`\tl_if_blank_p:n` This is based on the answers in “Around the Bend No 2” but is safer as the tests listed
`\tl_if_blank_p:V` there all have one small flaw: If the input in the test is two tokens with the same meaning
`\tl_if_blank_p:o` as the internal delimiter, they will fail since one of them is mistaken for the actual
`\tl_if_blank:nTF` delimiter. In our version below we make sure to pass the input through `\tl_to_str:n`
`\tl_if_blank:VTF` which ensures that all the tokens are converted to catcode 12. However we use an `a` with
`\tl_if_blank:oTF` catcode 11 as delimiter so we can *never* get into the same problem as the solutions in
`\tl_if_blank_p_aux:w` “Around the Bend No 2”.

```

4275 \prg_new_conditional:Npnn \tl_if_blank:n #1 {p,TF,T,F} {
4276   \exp_after:wN \tl_if_blank_p_aux:w \tl_to_str:n {#1} aa..\q_stop
4277 }
4278 \cs_new:Npn \tl_if_blank_p_aux:w #1#2 a #3#4 \q_stop {
4279   \if_meaning:w #3 #4 \prg_return_true: \else: \prg_return_false: \fi:
4280 }
4281 \cs_generate_variant:Nn \tl_if_blank_p:n {V}
4282 \cs_generate_variant:Nn \tl_if_blank:nTF {V}
4283 \cs_generate_variant:Nn \tl_if_blank:nT {V}
4284 \cs_generate_variant:Nn \tl_if_blank:nF {V}
4285 \cs_generate_variant:Nn \tl_if_blank_p:n {o}
4286 \cs_generate_variant:Nn \tl_if_blank:nTF {o}
4287 \cs_generate_variant:Nn \tl_if_blank:nT {o}
4288 \cs_generate_variant:Nn \tl_if_blank:nF {o}

```

`\tl_if_single:nTF` If the argument is a single token. ‘Space’ is considered ‘true’.

`\tl_if_single_p:n`

```

4289 \prg_new_conditional:Nnn \tl_if_single:n {p,TF,T,F} {
4290   \tl_if_empty:nTF {#1}

```

```

4291     {\prg_return_false:}
4292     {
4293       \tl_if_blank:nTF {#1}
4294       {\prg_return_true:}
4295       {
4296         \tl_if_single_aux:w #1 \q_stop
4297       }
4298     }
4299 }

```

Use `\exp_after:wN` below I know what I'm doing. Use `\exp_args:NV` or `\exp_args_unbraced:NV` for more flexibility in your own code.

```

4300 \prg_new_conditional:Nnn \tl_if_single:N {p,TF,T,F} {
4301   \tl_if_empty:NTF #1
4302   {\prg_return_false:}
4303   {
4304     \exp_after:wN \tl_if_blank:nTF #1
4305     {\prg_return_true:}
4306     {
4307       \exp_after:wN \tl_if_single_aux:w #1 \q_stop
4308     }
4309   }
4310 }

4311 \cs_new:Npn \tl_if_single_aux:w #1#2 \q_stop {
4312   \tl_if_empty:nTF {#2} \prg_return_true: \prg_return_false:
4313 }

```

103.4 Working with the contents of token lists

`\tl_to_lowercase:n` Just some names for a few primitives.

`\tl_to_uppercase:n`

```

4314 \cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
4315 \cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D

```

`\tl_to_str:n` Another name for a primitive.

```

4316 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D

```

`\tl_to_str:N` These functions return the replacement text of a token list as a string list with all characters catcoded to ‘other’.

`\tl_to_str:c`

`\tl_to_str_aux:w`

```

4317 \cs_new_nopar:Npn \tl_to_str:N {\exp_after:wN\tl_to_str_aux:w
4318   \token_to_meaning:N}
4319 \cs_new_nopar:Npn \tl_to_str_aux:w #1>{}
4320 \cs_generate_variant:Nn \tl_to_str:N {c}

```


`\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 will be read immediately and the loop terminated.

`\tl_map_function:NN`

`\tl_map_function:cN`

`\tl_map_function_aux:NN`

```

4321 \cs_new:Npn \tl_map_function:nN #1#2{
4322   \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop
4323 }
4324 \cs_new_nopar:Npn \tl_map_function:NN #1#2{
4325   \exp_after:wN \tl_map_function_aux:Nn
4326   \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
4327 }
4328 \cs_new:Npn \tl_map_function_aux:Nn #1#2{
4329   \quark_if_recursion_tail_stop:n{#2}
4330   #1{#2} \tl_map_function_aux:Nn #1
4331 }
4332 \cs_generate_variant:Nn \tl_map_function:NN {cN}

```

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter `\g_tl_inline_level_int` to make them nestable. We can also make use of `\tl_map_function:Nn` from before.

`\tl_map_inline:Nn`

`\tl_map_inline:cn`

`\tl_map_inline_aux:n`

`\g_tl_inline_level_int`

```

4333 \cs_new_protected:Npn \tl_map_inline:nn #1#2{
4334   \int_gincr:N \g_tl_inline_level_int
4335   \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4336   ##1{#2}
4337   \exp_args:Nc \tl_map_function_aux:Nn
4338   {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4339   #1 \q_recursion_tail \q_recursion_stop
4340   \int_gdecr:N \g_tl_inline_level_int
4341 }
4342 \cs_new_protected:Npn \tl_map_inline:Nn #1#2{
4343   \int_gincr:N \g_tl_inline_level_int
4344   \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4345   ##1{#2}
4346   \exp_last_unbraced:NcV \tl_map_function_aux:Nn
4347   {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4348   #1 \q_recursion_tail \q_recursion_stop
4349   \int_gdecr:N \g_tl_inline_level_int
4350 }
4351 \cs_generate_variant:Nn \tl_map_inline:Nn {c}

```

`\tl_map_variable:nNn` `\tl_map_variable:nNn` $\langle token\ list \rangle$ $\langle temp \rangle$ $\langle action \rangle$ assigns $\langle temp \rangle$ to each element and executes $\langle action \rangle$.

`\tl_map_variable:NNn`

`\tl_map_variable:cNn`

```

4352 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3{
4353   \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
4354 }

```

Next really has to be v/V args

```

4355 \cs_new_protected_nopar:Npn \tl_map_variable:NNn {\exp_args:No \tl_map_variable:nNn}
4356 \cs_generate_variant:Nn \tl_map_variable:NNn {c}

```

`\tl_map_variable_aux:NnN` The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```

4357 \cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3{
4358   \tl_set:Nn #1{#3}
4359   \quark_if_recursion_tail_stop:N #1
4360   #2 \tl_map_variable_aux:Nnn #1{#2}
4361 }

```

`\tl_map_break:` The break statement.

```

4362 \cs_new_eq:NN \tl_map_break: \use_none_delimit_by_q_recursion_stop:w

```

`\tl_reverse:n` Reversal of a token list is done by taking one token at a time and putting it in front of the ones before it.

`\tl_reverse:V`

`\tl_reverse:o`

`\tl_reverse_aux:nN`

```

4363 \cs_new:Npn \tl_reverse:n #1{
4364   \tl_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
4365 }
4366 \cs_new:Npn \tl_reverse_aux:nN #1#2{
4367   \quark_if_recursion_tail_stop_do:nn {#2}{ #1 }
4368   \tl_reverse_aux:nN {#2#1}
4369 }
4370 \cs_generate_variant:Nn \tl_reverse:n {V,o}

```

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which in turn is removed by the `f` expansion which comes to a halt.

```

4371 \cs_new_protected_nopar:Npn \tl_reverse:N #1 {
4372   \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } }
4373 }

```

`\tl_elt_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. `\tl_elt_count_aux:n` grabs the element and replaces it by +1. The 0 to ensure it works on an empty list.

`\tl_elt_count:V`

`\tl_elt_count:o`

`\tl_elt_count:N`

```

4374 \cs_new:Npn \tl_elt_count:n #1{
4375   \int_eval:n {
4376     0 \tl_map_function:nN {#1} \tl_elt_count_aux:n
4377   }
4378 }
4379 \cs_generate_variant:Nn \tl_elt_count:n {V,o}
4380 \cs_new_nopar:Npn \tl_elt_count:N #1{
4381   \int_eval:n {
4382     0 \tl_map_function:NN #1 \tl_elt_count_aux:n
4383   }
4384 }

```

`\tl_num_elt_count_aux:n` Helper function for counting elements in a token list.

```
4385 \cs_new:Npn \tl_elt_count_aux:n #1 { + 1 }
```

`\tl_set_rescan:Nnn` These functions store the $\{\langle token\ list\rangle\}$ in $\langle tl\ var.\rangle$ after redefining catcodes, etc., in argument #2.

`\tl_gset_rescan:Nnn`
`\tl_set_rescan:Nno` #1 : $\langle tl\ var.\rangle$
`\tl_gset_rescan:Nno` #2 : $\{\langle catcode\ setup,\ etc.\rangle\}$
#3 : $\{\langle token\ list\rangle\}$

```
4386 \cs_new_protected:Npn \tl_set_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_set:Nn }
4387 \cs_new_protected:Npn \tl_gset_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
4388 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno }
4389 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno }
```

`\tl_set_rescan_aux:NNnn` This macro uses a trick to extract an unexpanded token list after it's rescanned with `\etex_scantokens:D`. This technique was first used (as far as I know) by Heiko Oberdiek in his `catchfile` package, albeit for real files rather than the 'fake' `\scantokens` one.

The basic problem arises because `\etex_scantokens:D` emulates a file read, which inserts an EOF marker into the expansion; the simplistic

```
\exp_args:Nno \cs_set:Npn \tmp:w { \etex_scantokens:D {some text} }
```

unfortunately doesn't work, calling the error:

! File ended while scanning definition of \tmp:w.

(LuaTeX works around this problem with its `\scantextokens` primitive.)

Usually, we'd define `\etex_veryeof:D` to be `\exp_not:N` to gobble the EOF marker, but since we're not expanding the token list, it gets left in there and we have the same basic problem.

Instead, we define `\etex_veryeof:D` to contain a marker that's impossible to occur within the scanned text; that is, the same char twice with different catcodes. (For some reason, we *don't* need to insert a `\exp_not:N` token after it to prevent the EOF marker to expand. Anyone know why?)

A helper function is can be used to save the token list delimited by the special marker, keeping the catcode redefinitions hidden away in a group.

`\c_two_ats_with_two_catcodes_tl` A tl with two @ characters with two different catcodes. Used as a special marker for delimited text.

```
4390 \group_begin:
4391 \tex_lccode:D '\A = '\@ \scan_stop:
4392 \tex_lccode:D '\B = '\@ \scan_stop:
4393 \tex_catcode:D '\A = 8 \scan_stop:
4394 \tex_catcode:D '\B = 3 \scan_stop:
4395 \tl_to_lowercase:n {
4396 \group_end:
4397 \tl_const:Nn \c_two_ats_with_two_catcodes_tl { A B }
4398 }
```

```

#1 : \tl_set function
#2 : <tl var.>
#3 : {\catcode setup, etc.}
#4 : {\token list}

```

Note that if you change `\etex_everyeof:D` in #3 then you'd better do it correctly!

```

4399 \cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4 {
4400   \group_begin:
4401     \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
4402     \tex_endlinechar:D = \c_minus_one
4403     #3
4404     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
4405     \exp_args:NNNV
4406   \group_end:
4407   #1 #2 \l_tmpa_tl
4408 }

```

`\tl_rescan_aux:w`

```

4409 \exp_after:wN \cs_set:Npn
4410 \exp_after:wN \tl_rescan_aux:w
4411 \exp_after:wN #
4412 \exp_after:wN 1 \c_two_ats_with_two_catcodes_tl {
4413   \tl_set:Nn \l_tmpa_tl {#1}
4414 }

```

`\tl_set_rescan:Nnx` These functions store the full expansion of `{\token list}` in `<tl var.>` after redefining
`\tl_gset_rescan:Nnx` catcodes, etc., in argument #2.

```

#1 : <tl var.>
#2 : {\catcode setup, etc.}
#3 : {\token list}

```

The expanded versions are much simpler because the `\etex_scantokens:D` can occur within the expansion.

```

4415 \cs_new_protected:Npn \tl_set_rescan:Nnx #1#2#3 {
4416   \group_begin:
4417     \etex_everyeof:D { \exp_not:N }
4418     \tex_endlinechar:D = \c_minus_one
4419     #2
4420     \tl_set:Nx \l_kernel_tmpa_tl { \etex_scantokens:D {#3} }
4421     \exp_args:NNNV
4422   \group_end:
4423   \tl_set:Nn #1 \l_kernel_tmpa_tl
4424 }

```

Globally is easier again:

```

4425 \cs_new_protected:Npn \tl_gset_rescan:Nnx #1#2#3 {
4426   \group_begin:
4427     \etex_everyeof:D { \exp_not:N }
4428     \tex_endlinechar:D = \c_minus_one
4429     #2
4430     \tl_gset:Nx #1 { \etex_scantokens:D {#3} }
4431   \group_end:
4432 }

```

`\tl_rescan:nn` The inline wrapper for `\etex_scantokens:D`.

#1 : Catcode changes (etc.)
 #2 : Token list to re-tokenise

```

4433 \cs_new_protected:Npn \tl_rescan:nn #1#2 {
4434   \group_begin:
4435     \toks_set:NV \etex_everyeof:D \c_two_at_with_two_catcodes_tl
4436     \tex_endlinechar:D = \c_minus_one
4437     #1
4438     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
4439   \exp_args:NV \group_end:
4440   \l_tmpa_tl
4441 }

```

103.5 Checking for and replacing tokens

`\tl_if_in:NnTF` See the replace functions for further comments. In this part we don't care too much about brace stripping since we are not interested in passing on the tokens which are split off in the process.

`\tl_if_in:cnTF`

```

4442 \prg_new_protected_conditional:Npnn \tl_if_in:Nn #1#2 {TF,T,F} {
4443   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4444     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4445   }
4446   \exp_after:wN \tl_tmp:w #1 #2 \q_no_value \q_stop
4447 }
4448 \cs_generate_variant:Nn \tl_if_in:NnTF {c}
4449 \cs_generate_variant:Nn \tl_if_in:NnT {c}
4450 \cs_generate_variant:Nn \tl_if_in:NnF {c}

```

`\tl_if_in:nnTF`

`\tl_if_in:VnTF`

`\tl_if_in:onTF`

```

4451 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 {TF,T,F} {
4452   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4453     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4454   }
4455   \tl_tmp:w #1 #2 \q_no_value \q_stop
4456 }

```

```

4457 \cs_generate_variant:Nn \tl_if_in:nnTF {V}
4458 \cs_generate_variant:Nn \tl_if_in:nnT {V}
4459 \cs_generate_variant:Nn \tl_if_in:nnF {V}
4460 \cs_generate_variant:Nn \tl_if_in:nnTF {o}
4461 \cs_generate_variant:Nn \tl_if_in:nnT {o}
4462 \cs_generate_variant:Nn \tl_if_in:nnF {o}

```

```

\tl_replace_tl
\tl_replace_in:Nnn
\tl_replace_in:cnn
\tl_greplace_in:Nnn
\tl_greplace_in:cnn
\tl_replace_in_aux:NNnn

```

The concept here is that only the first occurrence should be replaced. The first step is to define an auxiliary which will match the appropriate item, with a trailing marker. If the last token is the marker there is nothing to do, otherwise replace the token and clean up (hence the second use of `\tl_tmp:w`). To prevent losing braces or spaces there are a couple of empty groups and the strange-looking `\use:n`.

```

4463 \tl_new:N \_l_tl_replace_tl
4464 \cs_new_protected_nopar:Npn \tl_replace_in:Nnn {
4465   \tl_replace_in_aux:NNnn \tl_set_eq:NN
4466 }
4467 \cs_new_protected:Npn \_l_tl_replace_in_aux:NNnn #1#2#3#4 {
4468   \cs_set:Npn \_tl_tmp:w ##1 #3 ##2 \q_stop
4469   {
4470     \quark_if_no_value:nF {##2}
4471     {
4472       \tl_set:No \_l_tl_replace_tl { ##1 #4 }
4473       \cs_set:Npn \_tl_tmp:w ####1 #3 \q_no_value {
4474         \tl_put_right:No \_l_tl_replace_tl {####1}
4475       }
4476       \_tl_tmp:w \prg_do_nothing: ##2
4477       #1 #2 \_l_tl_replace_tl
4478     }
4479   }
4480   \use:n
4481   {
4482     \exp_after:wN \_tl_tmp:w \exp_after:wN
4483     \prg_do_nothing:
4484   }
4485   #2 #3 \q_no_value \q_stop
4486 }
4487 \cs_new_protected_nopar:Npn \tl_greplace_in:Nnn {
4488   \tl_replace_in_aux:NNnn \tl_gset_eq:NN
4489 }
4490 \cs_generate_variant:Nn \tl_replace_in:Nnn { c }
4491 \cs_generate_variant:Nn \tl_greplace_in:Nnn { c }

```

```

\tl_replace_all_in:Nnn
\tl_replace_all_in:Nnn
\tl_greplace_all_in:cnn
\tl_greplace_all_in:cnn
\tl_replace_all_in_aux:NNnn

```

A similar approach here but with a loop built in.

```

4492 \cs_new_protected_nopar:Npn \tl_replace_all_in:Nnn {
4493   \tl_replace_all_in_aux:NNnn \tl_set_eq:NN
4494 }
4495 \cs_new_protected_nopar:Npn \tl_greplace_all_in:Nnn {

```

```

4496 \tl_replace_all_in_aux:NNnn \tl_gset_eq:NN
4497 }
4498 \cs_new_protected:Npn \tl_replace_all_in_aux:NNnn #1#2#3#4 {
4499 \tl_clear:N \l_tl_replace_tl
4500 \cs_set:Npn \tl_tmp:w ##1 #3 ##2 \q_stop
4501 {
4502 \quark_if_no_value:nTF {##2}
4503 { \tl_put_right:No \l_tl_replace_tl {##1} }
4504 {
4505 \tl_put_right:No \l_tl_replace_tl { ##1 #4 }
4506 \tl_tmp:w \prg_do_nothing: ##2 \q_stop
4507 }
4508 }
4509 \use:n
4510 {
4511 \exp_after:wN \tl_tmp:w \exp_after:wN
4512 \prg_do_nothing:
4513 }
4514 #2 #3 \q_no_value \q_stop
4515 #1 #2 \l_tl_replace_tl
4516 }
4517 \cs_generate_variant:Nn \tl_replace_all_in:Nnn { c }
4518 \cs_generate_variant:Nn \tl_greplace_all_in:Nnn { c }

```

\tl_remove_in:Nn Next comes a series of removal functions. I have just implemented them as subcases of
\tl_remove_in:cn the replace functions for now (I'm lazy).

```

\tl_gremove_in:Nn
\tl_gremove_in:cn
4519 \cs_new_protected:Npn \tl_remove_in:Nn #1#2{\tl_replace_in:Nnn #1{#2}{}}
4520 \cs_new_protected:Npn \tl_gremove_in:Nn #1#2{\tl_greplace_in:Nnn #1{#2}{}}
4521 \cs_generate_variant:Nn \tl_remove_in:Nn {cn}
4522 \cs_generate_variant:Nn \tl_gremove_in:Nn {cn}

```

\tl_remove_all_in:Nn Same old, same old.

```

\tl_remove_all_in:cn
\tl_gremove_all_in:Nn
\tl_gremove_all_in:cn
4523 \cs_new_protected:Npn \tl_remove_all_in:Nn #1#2{
4524 \tl_replace_all_in:Nnn #1{#2}{}}
4525 }
4526 \cs_new_protected:Npn \tl_gremove_all_in:Nn #1#2{
4527 \tl_greplace_all_in:Nnn #1{#2}{}}
4528 }
4529 \cs_generate_variant:Nn \tl_remove_all_in:Nn {cn}
4530 \cs_generate_variant:Nn \tl_gremove_all_in:Nn {cn}

```

103.6 Heads or tails?

\tl_head:n These functions pick up either the head or the tail of a list. \tl_head_iii:n returns the
\tl_head:V first three items on a list.

```

\tl_head:v
\tl_head_i:n
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f
\tl_head_iii:n
\tl_head_iii:f
\tl_head:w
\tl_head_i:w
\tl_tail:w

```

```

4531 \cs_new:Npn \tl_head:n #1{\tl_head:w #1\q_stop}
4532 \cs_new_eq:NN \tl_head_i:n \tl_head:n
4533 \cs_new:Npn \tl_tail:n #1{\tl_tail:w #1\q_stop}
4534 \cs_generate_variant:Nn \tl_tail:n {f}
4535 \cs_new:Npn \tl_head_iii:n #1{\tl_head_iii:w #1\q_stop}
4536 \cs_generate_variant:Nn \tl_head_iii:n {f}
4537 \cs_new:Npn \tl_head:w #1#2\q_stop{#1}
4538 \cs_new_eq:NN \tl_head_i:w \tl_head:w
4539 \cs_new:Npn \tl_tail:w #1#2\q_stop{#2}
4540 \cs_new:Npn \tl_head_iii:w #1#2#3#4\q_stop{#1#2#3}
4541 \cs_generate_variant:Nn \tl_head:n { V }
4542 \cs_generate_variant:Nn \tl_head:n { v }
4543 \cs_generate_variant:Nn \tl_tail:n { V }
4544 \cs_generate_variant:Nn \tl_tail:n { v }

```

```

\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_p:fN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

```

When we want to check if the first token of a list equals something specific it is usually either to see if it is a control sequence or a character. Hence we make two different functions as the internal test is different. `\tl_if_head_meaning_eq:nNTF` uses `\if_meaning:w` and will consider the tokens `b11` and `b12` different. `\tl_if_head_char_eq:nNTF` on the other hand only compares character codes so would regard `b11` and `b12` as equal but would also regard two primitives as equal.

```

4545 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 {p,TF,T,F} {
4546   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_stop #2
4547   \prg_return_true: \else: \prg_return_false: \fi:
4548 }

```

For the charcode and catcode versions we insert `\exp_not:N` in front of both tokens. If you need them to expand fully as `TEX` does itself with these you can use an `f` type expansion.

```

4549 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 {p,TF,T,F} {
4550   \exp_after:wN \if:w \exp_after:wN \exp_not:N
4551   \tl_head:w #1 \q_stop \exp_not:N #2
4552   \prg_return_true: \else: \prg_return_false: \fi:
4553 }

```

Actually the default is already an `f` type expansion.

```

4554 %% \cs_new:Npn \tl_if_head_eq_charcode_p:fN #1#2{
4555 %%   \exp_after:wN \if_charcode:w \tl_head:w #1\q_stop\exp_not:N#2
4556 %%   \c_true_bool
4557 %%   \else:
4558 %%   \c_false_bool
4559 %%   \fi:
4560 %% }
4561 %% \def_long_test_function_new:npn {tl_if_head_eq_charcode:fN}#1#2{
4562 %%   \if_predicate:w \tl_if_head_eq_charcode_p:fN {#1}#2}

```

These `:fN` variants are broken; temporary patch:


```

4563 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN {f}
4564 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF {f}
4565 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT {f}
4566 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF {f}

```

And now catcodes:

```

4567 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 {p,TF,T,F} {
4568   \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
4569     \tl_head:w #1 \q_stop \exp_not:N #2
4570   \prg_return_true: \else: \prg_return_false: \fi:
4571 }

```

`_tl_check_exists:N` When used as a package, there is an option to be picky and to check definitions exist. The message text for this is created later, as the mechanism is not yet in place.

```

4572 <*package>
4573 \tex_ifodd:D \@l@expl@check@declarations@bool \scan_stop:
4574 \cs_set_protected:Npn \_tl\_check_exists:N #1
4575 {
4576   \cs_if_exist:NF #1
4577   {
4578     \msg_kernel_error:nxx { check } { non-declared-variable }
4579     { \token_to_str:N #1 }
4580   }
4581 }
4582 \cs_set_protected:Npn \tl_set:Nn #1#2
4583 {
4584   \_tl\_check_exists:N #1
4585   \cs_set_nopar:Npx #1 { \exp_not:n {#2} }
4586 }
4587 \cs_set_protected:Npn \tl_set:Nx #1#2
4588 {
4589   \_tl\_check_exists:N #1
4590   \cs_set_nopar:Npx #1 {#2}
4591 }
4592 \cs_set_protected:Npn \tl_gset:Nn #1#2
4593 {
4594   \_tl\_check_exists:N #1
4595   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4596 }
4597 \cs_set_protected:Npn \tl_gset:Nx #1#2
4598 {
4599   \_tl\_check_exists:N #1
4600   \cs_gset_nopar:Npx #1 {#2}
4601 }
4602 \cs_set_protected:Npn \tl_set_eq:NN #1#2
4603 {
4604   \_tl\_check_exists:N #1
4605   \_tl\_check_exists:N #2

```

```

4606     \cs_set_eq:NN #1 #2
4607   }
4608 \cs_set_protected:Npn \tl_gset_eq:NN #1#2
4609 {
4610   \tl_check_exists:N #1
4611   \tl_check_exists:N #2
4612   \cs_gset_eq:NN #1 #2
4613 }
4614 \cs_set_protected:Npn \tl_put_right:Nn #1#2 {
4615   \tl_check_exists:N #1
4616   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4617 }
4618 \cs_set_protected:Npn \tl_put_right:NV #1#2 {
4619   \tl_check_exists:N #1
4620   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4621 }
4622 \cs_set_protected:Npn \tl_put_right:Nv #1#2 {
4623   \tl_check_exists:N #1
4624   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4625 }
4626 \cs_set_protected:Npn \tl_put_right:No #1#2 {
4627   \tl_check_exists:N #1
4628   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4629 }
4630 \cs_set_protected:Npn \tl_put_right:Nx #1#2 {
4631   \tl_check_exists:N #1
4632   \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 }
4633 }
4634 \cs_set_protected:Npn \tl_gput_right:Nn #1#2 {
4635   \tl_check_exists:N #1
4636   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4637 }
4638 \cs_set_protected:Npn \tl_gput_right:NV #1#2 {
4639   \tl_check_exists:N #1
4640   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4641 }
4642 \cs_set_protected:Npn \tl_gput_right:Nv #1#2 {
4643   \tl_check_exists:N #1
4644   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4645 }
4646 \cs_set_protected:Npn \tl_gput_right:Nx #1#2 {
4647   \tl_check_exists:N #1
4648   \cs_gset_nopar:Npx #1 { \exp_not:o #1 #2 }
4649 }
4650 \cs_set_protected:Npn \tl_put_left:Nn #1#2 {
4651   \tl_check_exists:N #1
4652   \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4653 }
4654 \cs_set_protected:Npn \tl_put_left:NV #1#2 {
4655   \tl_check_exists:N #1

```

```

4656 \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4657 }
4658 \cs_set_protected:Npn \tl_put_left:Nv #1#2 {
4659 \tl_check_exists:N #1
4660 \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4661 }
4662 \cs_set_protected:Npn \tl_put_left:No #1#2 {
4663 \tl_check_exists:N #1
4664 \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4665 }
4666 \cs_set_protected:Npn \tl_put_left:Nx #1#2 {
4667 \tl_check_exists:N #1
4668 \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 }
4669 }
4670 \cs_set_protected:Npn \tl_gput_left:Nn #1#2 {
4671 \tl_check_exists:N #1
4672 \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4673 }
4674 \cs_set_protected:Npn \tl_gput_left:NV #1#2 {
4675 \tl_check_exists:N #1
4676 \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4677 }
4678 \cs_set_protected:Npn \tl_gput_left:Nv #1#2 {
4679 \tl_check_exists:N #1
4680 \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4681 }
4682 \cs_set_protected:Npn \tl_gput_left:Nx #1#2 {
4683 \tl_check_exists:N #1
4684 \cs_gset_nopar:Npx #1 { #2 \exp_not:o #1 }
4685 }
4686 \tex_fi:D
4687 \</package>

```

Show token usage:

```

4688 <*showmemory>
4689 \showMemUsage
4690 </showmemory>

```

104 l3toks implementation

We start by ensuring that the required packages are loaded.

```

4691 <*package>
4692 \ProvidesExplPackage
4693 {\filename}{\filedate}{\fileversion}{\filedescription}
4694 \package_check_loaded_expl:
4695 </package>
4696 <*initex | package>

```

104.1 Allocation and use

`\toks_new:N` Allocates a new token register.

```
\toks_new:c
4697 <*initex>
4698 \alloc_new:nnnN {toks} \c_zero \c_max_register_int \tex_toksdef:D
4699 </initex>

4700 <*package>
4701 \cs_new_protected_nopar:Npn \toks_new:N #1 {
4702   \chk_if_free_cs:N #1
4703   \newtoks #1
4704 }
4705 </package>

4706 \cs_generate_variant:Nn \toks_new:N {c}
```

`\toks_use:N` This function returns the contents of a token register.

```
\toks_use:c
4707 \cs_new_eq:NN \toks_use:N \tex_the:D
4708 \cs_generate_variant:Nn \toks_use:N {c}
```

`\toks_set:Nn` `\toks_set:Nn<toks><stuff>` stores `<stuff>` without expansion in `<toks>`. `\toks_set:No` and `\toks_set:Nv` `\toks_set:Nx` expand `<stuff>` once and fully.

```
\toks_set:Nv
4709 <*check>
4710 \cs_new_protected_nopar:Npn \toks_set:Nn #1 { \chk_local:N #1 #1 }
4711 \cs_generate_variant:Nn \toks_set:Nn {No,Nf}
4712 </check>
\toks_set:cn
\toks_set:co If we don't check if <toks> is a local register then the \toks_set:Nn function has nothing
\toks_set:cV to do. We implement \toks_set:No/d/f by hand when not checking because this is going
\toks_set:cv to be used extensively in keyval processing! TODO: (Will) Can we get some numbers
\toks_set:cx published on how necessary this is? On the other hand I'm happy to believe Morten :)
\toks_set:cf
```

```
4713 <!*check>
4714 \cs_new_eq:NN \toks_set:Nn \prg_do_nothing:
4715 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4716   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:N #2 }
4717 }
4718 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4719   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:c {#2} }
4720 }
4721 \cs_new_protected:Npn \toks_set:No #1#2 { #1 \exp_after:wN {#2} }
4722 \cs_new_protected:Npn \toks_set:Nf #1#2 {
4723   #1 \exp_after:wN { \int_to_roman:w -'0#2 }
4724 }
4725 </!check>

4726 \cs_generate_variant:Nn \toks_set:Nn {Nx,cn,cV,cv,co,cx,cf}
```

```

\toks_gset:Nn These functions are the global variants of the above.
\toks_gset:NV
\toks_gset:No 4727 <check>\cs_new_protected_nopar:Npn \toks_gset:Nn #1 { \chk_global:N #1 \pref_global:D #1 }
\toks_gset:Nx 4728 <!check>\cs_new_eq:NN \toks_gset:Nn \pref_global:D
\toks_gset:cn 4729 \cs_generate_variant:Nn \toks_gset:Nn {NV,No,Nx,cn,cV,co,cx}

\toks_set_eq:NN \toks_set_eq:NN<toks1><toks2> copies the contents of <toks2> in <toks1>.
\toks_set_eq:Nc
\toks_set_eq:cn
\toks_set_eq:cc 4730 <*check>
\toks_gset_eq:NN 4731 \cs_new_protected_nopar:Npn \toks_set_eq:NN #1#2 {
\toks_gset_eq:Nc 4732 \chk_local:N #1
\toks_gset_eq:cn 4733 \chk_var_or_const:N #2
\toks_gset_eq:cc 4734 #1 #2
4735 }
4736 \cs_new_protected_nopar:Npn \toks_gset_eq:NN #1#2 {
4737 \chk_global:N #1
4738 \chk_var_or_const:N #2
4739 \pref_global:D #1 #2
4740 }
4741 </check>
4742 <*!check>
4743 \cs_new_eq:NN \toks_set_eq:NN \prg_do_nothing:
4744 \cs_new_eq:NN \toks_gset_eq:NN \pref_global:D
4745 </!check>
4746 \cs_generate_variant:Nn \toks_set_eq:NN {Nc,cN,cc}
4747 \cs_generate_variant:Nn \toks_gset_eq:NN {Nc,cN,cc}

\toks_clear:N These functions clear a token register, either locally or globally.
\toks_gclear:N
\toks_clear:c 4748 \cs_new_protected_nopar:Npn \toks_clear:N #1 {
\toks_gclear:c 4749 #1\c_empty_toks
4750 <check>\chk_local_or_pref_global:N #1
4751 }
4752 \cs_new_protected_nopar:Npn \toks_gclear:N {
4753 <check> \pref_global_chk:
4754 <!check> \pref_global:D
4755 \toks_clear:N
4756 }
4757 \cs_generate_variant:Nn \toks_clear:N {c}
4758 \cs_generate_variant:Nn \toks_gclear:N {c}

\toks_use_clear:N These functions clear a token register (locally or globally) after returning the contents.
\toks_use_clear:c They make sure that clearing the register does not interfere with following tokens. In
\toks_use_gclear:N other words, the contents of the register might operate on what follows in the input
\toks_use_gclear:c stream.
4759 \cs_new_protected_nopar:Npn \toks_use_clear:N #1 {
4760 \exp_last_unbraced:NNV \toks_clear:N #1 #1
4761 }

```

```

4762 \cs_new_protected_nopar:Npn \toks_use_gclear:N {
4763   <check> \pref_global_chk:
4764   <!check> \pref_global:D
4765   \toks_use_clear:N
4766 }

4767 \cs_generate_variant:Nn \toks_use_clear:N {c}
4768 \cs_generate_variant:Nn \toks_use_gclear:N {c}

```

`\toks_show:N` This function shows the contents of a token register on the terminal.

```

\toks_show:c
4769 \cs_new_eq:NN \toks_show:N \kernel_register_show:N
4770 \cs_generate_variant:Nn \toks_show:N {c}

```

104.2 Adding to token registers' contents

```

\toks_put_left:Nn \toks_put_left:Nn <toks><stuff> adds the tokens of stuff on the ‘left-side’ of the token
\toks_put_left:NV register <toks>. \toks_put_left:No does the same, but expands the tokens once. We
\toks_put_left:No need to look out for brace stripping so we add a token, which is then later removed.
\toks_put_left:Nx
\toks_put_left:cn
\toks_put_left:cV
\toks_put_left:co
\toks_gput_left:Nn
\toks_gput_left:NV
\toks_gput_left:No
\toks_gput_left:Nx
\toks_gput_left:cn
\toks_gput_left:cV
\toks_gput_left:co
\toks_put_left_aux:w
4771 \cs_new_protected_nopar:Npn \toks_put_left:Nn #1 {
4772   \exp_after:wN \toks_put_left_aux:w \exp_after:wN \q_nil
4773   \toks_use:N #1 \q_stop #1
4774 }

4775 \cs_generate_variant:Nn \toks_put_left:Nn {NV,No,Nx,cn,co,cV}

4776 \cs_new_protected_nopar:Npn \toks_gput_left:Nn {
4777   <check> \pref_global_chk:
4778   <!check> \pref_global:D
4779   \toks_put_left:Nn
4780 }

4781 \cs_generate_variant:Nn \toks_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

A helper function for `\toks_put_left:Nn`. Its arguments are subsequently the tokens of `<stuff>`, the token register `<toks>` and the current contents of `<toks>`. We make sure to remove the token we inserted earlier.

```

4782 \cs_new:Npn \toks_put_left_aux:w #1\q_stop #2#3 {
4783   #2 \exp_after:wN { \use_i:nn {#3} #1 }
4784   <check> \chk_local_or_pref_global:N #2
4785 }

```

```

\toks_put_right:Nn These macros add a list of tokens to the right of a token register.
\toks_put_right:NV
\toks_put_right:No
\toks_put_right:Nx
\toks_put_right:cn
\toks_put_right:cV
\toks_put_right:co
\toks_gput_right:Nn
\toks_gput_right:NV
\toks_gput_right:No
\toks_gput_right:Nx
\toks_gput_right:cn
\toks_gput_right:cV
\toks_gput_right:co

```

```

4790 \cs_new_protected_nopar:Npn \toks_gput_right:Nn {
4791   <check> \pref_global_chk:
4792   !<check> \pref_global:D
4793   \toks_put_right:Nn
4794 }

```

A couple done by hand for speed.

```

4795 {check} \cs_generate_variant:Nn \toks_put_right:Nn {No}
4796 {*!check}
4797 \cs_new_protected:Npn \toks_put_right:NV #1#2 {
4798   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4799     \exp_after:wN \toks_use:N \exp_after:wN #1
4800     \int_to_roman:w -'0 \exp_eval_register:N #2
4801   }
4802 }
4803 \cs_new_protected:Npn \toks_put_right:No #1#2 {
4804   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4805     \exp_after:wN \toks_use:N \exp_after:wN #1 #2
4806   }
4807 }
4808 </!check>
4809 \cs_generate_variant:Nn \toks_put_right:Nn {Nx,cn,cV,co}
4810 \cs_generate_variant:Nn \toks_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

<code>\toks_put_right:Nf</code>	We implement <code>\toks_put_right:Nf</code> by hand because I think I might use it in the <code>l3keyval</code> module in which case it is going to be used a lot.
---------------------------------	---

```

4811 \check\cs_generate_variant:Nn \toks_put_right:Nn {Nf}
4812 \*!check)
4813 \cs_new_protected:Npn \toks_put_right:Nf #1#2 {
4814   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4815     \exp_after:wN \toks_use:N \exp_after:wN #1 \int_to_roman:w -'0#2
4816   }
4817 }
4818 \</!check>

```

104.3 Predicates and conditionals

<code>\toks_if_empty_p:N</code>	<code>\toks_if_empty:NTF</code>	$\langle toks \rangle$	$\langle true\ code \rangle$	$\langle false\ code \rangle$	tests if a token register is empty and
<code>\toks_if_empty_p:c</code>					executes either $\langle true\ code \rangle$ or $\langle false\ code \rangle$. This test had the advantage of being expand-
<code>\toks_if_empty:NTF</code>					able. Otherwise one has to do an x type expansion in order to prevent problems with
<code>\toks_if_empty:cTF</code>					parameter tokens.

```

4819 \prg_new_conditional:Nnn \toks_if_empty:N {p,TF,T,F} {
4820   \tl_if_empty:VTF #1 {\prg_return_true:} {\prg_return_false:}
4821 }

```

```

4822 \cs_generate_variant:Nn \toks_if_empty_p:N {c}
4823 \cs_generate_variant:Nn \toks_if_empty:NTF {c}
4824 \cs_generate_variant:Nn \toks_if_empty:NT {c}
4825 \cs_generate_variant:Nn \toks_if_empty:NF {c}

```

`\toks_if_eq_p:NN` This function test whether two token registers have the same contents.

```

\toks_if_eq_p:cN
\toks_if_eq_p:Nc
\toks_if_eq_p:cc
\toks_if_eq:NNTF
\toks_if_eq:NcTF
\toks_if_eq:cNTF
\toks_if_eq:ccTF
4826 \prg_new_conditional:Nnn \toks_if_eq:NN {p,TF,T,F} {
4827   \str_if_eq:xxTF {\toks_use:N #1} {\toks_use:N #2}
4828   {\prg_return_true:} {\prg_return_false:}
4829 }
4830 \cs_generate_variant:Nn \toks_if_eq_p:NN {Nc,c,cc}
4831 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4832 \cs_generate_variant:Nn \toks_if_eq:NNT {Nc,c,cc}
4833 \cs_generate_variant:Nn \toks_if_eq:NNF {Nc,c,cc}

```

104.4 Variables and constants

`\l_tmpa_toks` Some scratch registers ...

```

\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks
4834 \tex_toksdef:D \l_tmpa_toks = 255\scan_stop:
4835 \initex\seq_put_right:Nn \g_toks_allocation_seq {255}
4836 \toks_new:N \l_tmpb_toks
4837 \toks_new:N \l_tmpc_toks
4838 \toks_new:N \g_tmpa_toks
4839 \toks_new:N \g_tmpb_toks
4840 \toks_new:N \g_tmpc_toks

```

`\c_empty_toks` And here is a constant, which is a (permanently) empty token register.

```

4841 \toks_new:N \c_empty_toks

```

`\l_tl_replace_toks` And here is one for tl vars. Can't define it there as the allocation isn't set up at that point.

```

4842 \toks_new:N \l_tl_replace_toks
4843 \</initex | package>

```

Show token usage:

```

4844 \*showmemory>
4845 \showMemUsage
4846 \</showmemory>

```


105 l3seq implementation

```

4847 \*package>
4848 \ProvidesExplPackage
4849   {\filename}{\filedate}{\fileversion}{\filedescription}
4850 \package_check_loaded_expl:
4851 \package>

```

A sequence is a control sequence whose top-level expansion is of the form ‘`\seq_elt:w` $\langle text_1 \rangle$ `\seq_elt_end: ... \seq_elt:w` $\langle text_n \rangle$...’. We use explicit delimiters instead of braces around $\langle text \rangle$ to allow efficient searching for an item in the sequence.

`\seq_elt:w` We allocate the delimiters and make them errors if executed.
`\seq_elt_end:`

```

4852 \*initex | package>
4853 \cs_new:Npn \seq_elt:w {\ERROR}
4854 \cs_new:Npn \seq_elt_end: {\ERROR}

```

105.1 Allocating and initialisation

`\seq_new:N` Sequences are implemented using token lists.

```

\seq_new:c
4855 \cs_new_eq:NN \seq_new:N \tl_new:N
4856 \cs_new_eq:NN \seq_new:c \tl_new:c

```

`\seq_clear:N` Clearing a sequence is the same as clearing a token list.

```

\seq_clear:c
4857 \cs_new_eq:NN \seq_clear:N \tl_clear:N
\seq_gclear:N
4858 \cs_new_eq:NN \seq_clear:c \tl_clear:c
\seq_gclear:c
4859 \cs_new_eq:NN \seq_gclear:N \tl_gclear:N
4860 \cs_new_eq:NN \seq_gclear:c \tl_gclear:c

```

`\seq_clear_new:N` Clearing a sequence is the same as clearing a token list.

```

\seq_clear_new:c
4861 \cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N
\seq_gclear_new:N
4862 \cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c
\seq_gclear_new:c
4863 \cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
4864 \cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c

```

`\seq_set_eq:NN` We can set one seq equal to another.

```

\seq_set_eq:Nc
4865 \cs_new_eq:NN \seq_set_eq:NN \cs_set_eq:NN
\seq_set_eq:cN
4866 \cs_new_eq:NN \seq_set_eq:cN \cs_set_eq:cN
\seq_set_eq:cc
4867 \cs_new_eq:NN \seq_set_eq:Nc \cs_set_eq:Nc
4868 \cs_new_eq:NN \seq_set_eq:cc \cs_set_eq:cc

```

`\seq_gset_eq:NN` And of course globally which seems to be needed far more often.¹¹

```

\seq_gset_eq:cN
\seq_gset_eq:Nc
\seq_gset_eq:cc
4869 \cs_new_eq:NN \seq_gset_eq:NN \cs_gset_eq:NN
4870 \cs_new_eq:NN \seq_gset_eq:cN \cs_gset_eq:cN
4871 \cs_new_eq:NN \seq_gset_eq:Nc \cs_gset_eq:Nc
4872 \cs_new_eq:NN \seq_gset_eq:cc \cs_gset_eq:cc

```

`\seq_concat:NNN` `\seq_concat:NNN` $\langle seq\ 1 \rangle$ $\langle seq\ 2 \rangle$ $\langle seq\ 3 \rangle$ will locally assign $\langle seq\ 1 \rangle$ the concatenation of $\langle seq\ 2 \rangle$ and $\langle seq\ 3 \rangle$.

```

4873 \cs_new_protected_nopar:Npn \seq_concat:NNN #1#2#3 {
4874   \tl_set:Nx #1 { \exp_not:V #2 \exp_not:V #3 }
4875 }
4876 \cs_generate_variant:Nn \seq_concat:NNN {ccc}

```

`\seq_gconcat:NNN` `\seq_gconcat:NNN` $\langle seq\ 1 \rangle$ $\langle seq\ 2 \rangle$ $\langle seq\ 3 \rangle$ will globally assign $\langle seq\ 1 \rangle$ the concatenation of $\langle seq\ 2 \rangle$ and $\langle seq\ 3 \rangle$.

```

4877 \cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3 {
4878   \tl_gset:Nx #1 { \exp_not:V #2 \exp_not:V #3 }
4879 }
4880 \cs_generate_variant:Nn \seq_gconcat:NNN {ccc}

```

105.2 Predicates and conditionals

`\seq_if_empty_p:N` A predicate which evaluates to `\c_true_bool` iff the sequence is empty.

```

\seq_if_empty_p:c
\seq_if_empty:NTF
\seq_if_empty:cTF
4881 \prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N {p,TF,T,F}
4882 \prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c {p,TF,T,F}

```

`\seq_if_empty_err:N` Signals an error if the sequence is empty.

```

4883 \cs_new_nopar:Npn \seq_if_empty_err:N #1 {
4884   \if_meaning:w #1 \c_empty_tl

```

As I said before, I don't think we need to provide checks for this kind of error, since it is a severe internal macro package error that can not be produced by the user directly. Can it? So the next line of code should be probably removed. (Will: I have no idea what this comment means.)

```

4885   \tl_clear:N \l_kernel_testa_tl % catch prefixes
4886   \msg_kernel_bug:x {Empty~sequence~'\token_to_str:N#1'}
4887   \fi:
4888 }

```

¹¹To save a bit of space these functions could be made identical to those from the `tl` or `clist` module.

`\seq_if_in:NnTF` `\seq_if_in:NnTF` $\langle seq \rangle \langle item \rangle \langle true\ case \rangle \langle false\ case \rangle$ will check whether $\langle item \rangle$ is in $\langle seq \rangle$
`\seq_if_in:NVTF` and then either execute the $\langle true\ case \rangle$ or the $\langle false\ case \rangle$. $\langle true\ case \rangle$ and $\langle false\ case \rangle$
`\seq_if_in:cnTF` may contain incomplete `\if_charcode:w` statements.
`\seq_if_in:cVTF` Note that ##2 in the definition below for `\seq_tmp:w` contains exactly one token which
`\seq_if_in:coTF` we can compare with `\q_no_value`.
`\seq_if_in:cxTF`

```

4889 \prg_new_protected_conditional:Nnn \seq_if_in:Nn {TF,T,F} {
4890   \cs_set:Npn \seq_tmp:w ##1 \seq_elt:w #2 \seq_elt_end: ##2##3 \q_stop {
4891     \if_meaning:w \q_no_value ##2
4892     \prg_return_false: \else: \prg_return_true: \fi:
4893   }
4894   \exp_after:wN \seq_tmp:w #1 \seq_elt:w #2 \seq_elt_end: \q_no_value \q_stop
4895 }

4896 \cs_generate_variant:Nn \seq_if_in:NnTF { NV, cV, co, c, cx}
4897 \cs_generate_variant:Nn \seq_if_in:NnT { NV, cV, co, c, cx}
4898 \cs_generate_variant:Nn \seq_if_in:NnF { NV, cV, co, c, cx}

```

105.3 Getting data out

`\seq_get:NN` `\seq_get:NN` $\langle sequence \rangle \langle cmd \rangle$ defines $\langle cmd \rangle$ to be the left-most element of $\langle sequence \rangle$.
`\seq_get:cN`
`\seq_get_aux:w`

```

4899 \cs_new_protected_nopar:Npn \seq_get:NN #1 {
4900   \seq_if_empty_err:N #1
4901   \exp_after:wN \seq_get_aux:w #1 \q_stop
4902 }
4903 \cs_new_protected:Npn \seq_get_aux:w \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3 {
4904   \tl_set:Nn #3 {#1}
4905 }
4906 \cs_generate_variant:Nn \seq_get:NN {c}

```

`\seq_pop_aux:nnNN` `\seq_pop_aux:nnNN` $\langle def_1 \rangle \langle def_2 \rangle \langle sequence \rangle \langle cmd \rangle$ assigns the left_most element of
`\seq_pop_aux:w` $\langle sequence \rangle$ to $\langle cmd \rangle$ using $\langle def_2 \rangle$, and assigns the tail of $\langle sequence \rangle$ to $\langle sequence \rangle$ using $\langle def_1 \rangle$.

```

4907 \cs_new_protected:Npn \seq_pop_aux:nnNN #1#2#3 {
4908   \seq_if_empty_err:N #3
4909   \exp_after:wN \seq_pop_aux:w #3 \q_stop #1#2#3
4910 }
4911 \cs_new_protected:Npn \seq_pop_aux:w
4912   \seq_elt:w #1 \seq_elt_end: #2 \q_stop #3#4#5#6 {
4913   #3 #5 {#2}
4914   #4 #6 {#1}
4915 }

```

`\seq_show:N`

`\seq_show:c`

```

4916 \cs_new_eq:NN \seq_show:N \tl_show:N
4917 \cs_new_eq:NN \seq_show:c \tl_show:c

```

\seq_display:N
\seq_display:c

```

4918 \cs_new_protected_nopar:Npn \seq_display:N #1 {
4919   \iow_term:x { Sequence~\token_to_str:N #1~contains~
4920     the~elements~(without~outer~braces): }
4921   \toks_clear:N \l_tmpa_toks
4922   \seq_map_inline:Nn #1 {
4923     \toks_if_empty:NF \l_tmpa_toks {
4924       \toks_put_right:Nx \l_tmpa_toks {\^J>}
4925     }
4926     \toks_put_right:Nx \l_tmpa_toks {
4927       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
4928     }
4929   }
4930   \toks_show:N \l_tmpa_toks
4931 }
4932 \cs_generate_variant:Nn \seq_display:N {c}

```

105.4 Putting data in

\seq_put_aux:Nnn *<sequence>* *<left>* *<right>* adds the elements specified by *<left>* to the left of *<sequence>*, and those specified by *<right>* to the right.

```

4933 \cs_new_protected:Npn \seq_put_aux:Nnn #1 {
4934   \exp_after:wN \seq_put_aux:w #1 \q_stop #1
4935 }
4936 \cs_generate_variant:Nn \seq_put_aux:w #1\q_stop #2#3#4 { \tl_set:Nn #2 {#3#1#4} }

```

\seq_put_left:Nn Here are the usual operations for adding to the left and right.

```

\seq_put_left:NV
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:co
\seq_put_right:Nn
\seq_put_right:No
\seq_put_right:NV
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:co

```

We can't put in a \prg_do_nothing: instead of {} above since this argument is passed literally (and we would end up with many \prg_do_nothing:s inside the sequences).

```

4940 \cs_generate_variant:Nn \seq_put_left:Nn {NV,No,Nx,c,cV,co}

4941 \cs_new_protected:Npn \seq_put_right:Nn #1#2{
4942   \seq_put_aux:Nnn #1{\seq_elt:w #2\seq_elt_end:}{}
4943 \cs_generate_variant:Nn \seq_put_right:Nn {NV,No,Nx,c,cV,co}

```

\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:co
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:co

```

4944 \cs_new_protected:Npn \seq_gput_left:Nn {
4945   <*check>
4946   \pref_global_chk:

```

```

4947 </check>
4948 <!*check>
4949   \pref_global:D
4950 </!check>
4951   \seq_put_left:Nn
4952 }

4953 \cs_new_protected:Npn \seq_gput_right:Nn {
4954 <!*check>
4955   \pref_global_chk:
4956 </check>
4957 <!*check>
4958   \pref_global:D
4959 </!check>
4960   \seq_put_right:Nn
4961 }

4962 \cs_generate_variant:Nn \seq_gput_left:Nn {NV,No,Nx,c,cV,co}

4963 \cs_generate_variant:Nn \seq_gput_right:Nn {NV,No,Nx,c,cV,co}

```

105.5 Mapping

```

\seq_map_variable:NNn Nothing spectacular here.
\seq_map_variable:cNn
\seq_map_variable_aux:Nnw
4964 \cs_new_protected:Npn \seq_map_variable_aux:Nnw #1#2 \seq_elt:w #3 \seq_elt_end: {
4965   \tl_set:Nn #1 {#3}
4966   \quark_if_nil:NT #1 \seq_map_break:
4967   #2
4968   \seq_map_variable_aux:Nnw #1{#2}
4969 }
4970 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3 {
4971   \tl_set:Nn #2 {\seq_map_variable_aux:Nnw #2{#3}}
4972   \exp_after:wN #2 #1 \seq_elt:w \q_nil\seq_elt_end: \q_stop
4973 }
4974 \cs_generate_variant:Nn \seq_map_variable:NNn {c}

```

`\seq_map_break:` Terminate a mapping function at the point of execution. The latter takes an argument to be executed after cleaning up the map.

```

4975 \cs_new_eq:NN \seq_map_break: \use_none_delimit_by_q_stop:w
4976 \cs_new_eq:NN \seq_map_break:n \use_i_delimit_by_q_stop:nw

```

`\seq_map_function:NN` `\seq_map_function:NN` $\langle sequence \rangle$ $\langle cmd \rangle$ applies $\langle cmd \rangle$ to each element of $\langle sequence \rangle$, from left to right. Since we don't have braces, this implementation is not very efficient. It might be better to say that $\langle cmd \rangle$ must be a function with one argument that is delimited by `\seq_elt_end:.`

```

4977 \cs_new_protected_nopar:Npn \seq_map_function:NN #1#2 {
4978   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2{##1}}
4979   #1 \use_none:n \q_stop
4980   \cs_set_eq:NN \seq_elt:w \ERROR
4981 }
4982 \cs_generate_variant:Nn \seq_map_function:NN {c}

```

\seq_map_inline:NN When no braces are used, this version of mapping seems more natural.
 \seq_map_inline:cn

```

4983 \cs_new_protected_nopar:Npn \seq_map_inline:NN #1#2 {
4984   \cs_set:Npn \seq_elt:w ##1 \seq_elt_end: {#2}
4985   #1 \use_none:n \q_stop
4986   \cs_set_eq:NN \seq_elt:w \ERROR
4987 }
4988 \cs_generate_variant:Nn \seq_map_inline:NN {c}

```

105.6 Manipulation

\l_clist_remove_clist A common scratch space for the removal routines.

```

4989 \seq_new:N \l_seq_remove_seq

```

\seq_remove_duplicates_aux:NN Copied from \clist_remove_duplicates.

```

\seq_remove_duplicates_aux:n
  \seq_remove_duplicates:N
  \seq_gremove_duplicates:N
4990 \cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2 {
4991   \seq_clear:N \l_seq_remove_seq
4992   \seq_map_function:NN #2 \seq_remove_duplicates_aux:n
4993   #1 #2 \l_seq_remove_seq
4994 }
4995 \cs_new_protected:Npn \seq_remove_duplicates_aux:n #1 {
4996   \seq_if_in:NnF \l_seq_remove_seq {#1} {
4997     \seq_put_right:Nn \l_seq_remove_seq {#1}
4998   }
4999 }

5000 \cs_new_protected_nopar:Npn \seq_remove_duplicates:N {
5001   \seq_remove_duplicates_aux:NN \seq_set_eq:NN
5002 }
5003 \cs_new_protected_nopar:Npn \seq_gremove_duplicates:N {
5004   \seq_remove_duplicates_aux:NN \seq_gset_eq:NN
5005 }

```

105.7 Sequence stacks

\seq_push:Nn Since sequences can be used as stacks, we ought to have both ‘push’ and ‘pop’. In most cases they are nothing more than new names for old functions.

\seq_push:NV
 \seq_push:No
 \seq_push:cn
 \seq_pop:NN
 \seq_pop:cN

```

5006 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
5007 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
5008 \cs_new_eq:NN \seq_push:No \seq_put_left:No
5009 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
5010 \cs_new_protected_nopar:Npn \seq_pop:NN { \seq_pop_aux:nnNN \tl_set:Nn \tl_set:Nn }
5011 \cs_generate_variant:Nn \seq_pop:NN {c}

```

\seq_gpush:Nn I don't agree with Denys that one needs only local stacks, actually I believe that one
\seq_gpush:Nv will probably need the functions here more often. In case of \seq_gpop:NN the value is
\seq_gpush:No nevertheless returned locally.

```

\seq_gpush:cn
\seq_gpush:Nv
\seq_gpop:NN
\seq_gpop:cN
5012 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
5013 \cs_new_protected_nopar:Npn \seq_gpop:NN { \seq_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn }
5014 \cs_generate_variant:Nn \seq_gpush:Nn {NV,No,c,Nv}
5015 \cs_generate_variant:Nn \seq_gpop:NN {c}

```

\seq_top:NN Looking at the top element of the stack without removing it is done with this operation.
\seq_top:cN

```

5016 \cs_new_eq:NN \seq_top:NN \seq_get:NN
5017 \cs_new_eq:NN \seq_top:cN \seq_get:cN

5018 </initex | package>

```

Show token usage:

```

5019 <*showmemory>
5020 %\showMemUsage
5021 </showmemory>

```

106 l3clist implementation

We start by ensuring that the required packages are loaded.

```

5022 <*package>
5023 \ProvidesExplPackage
5024 { \filename } { \filedate } { \fileversion } { \filedescription }
5025 \package_check_loaded_expl:
5026 </package>
5027 <*initex | package>

```

106.1 Allocation and initialisation

\clist_new:N Comma-Lists are implemented using token lists.
\clist_new:c

```

5028 \cs_new_eq:NN \clist_new:N \tl_new:N
5029 \cs_generate_variant:Nn \clist_new:N {c}

```

`\clist_clear:N` Clearing a comma-list is the same as clearing a token list.

```

\clist_clear:c
\clist_gclear:N
\clist_gclear:c
5030 \cs_new_eq:NN \clist_clear:N \tl_clear:N
5031 \cs_generate_variant:Nn \clist_clear:N {c}
5032 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
5033 \cs_generate_variant:Nn \clist_gclear:N {c}

```

`\clist_clear_new:N` Clearing a comma-list is the same as clearing a token list.

```

\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c
5034 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
5035 \cs_generate_variant:Nn \clist_clear_new:N {c}
5036 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
5037 \cs_generate_variant:Nn \clist_gclear_new:N {c}

```

`\clist_set_eq:NN` We can set one $\langle\textit{clist}\rangle$ equal to another.

```

\clist_set_eq:cN
\clist_set_eq:Nc
\clist_set_eq:cc
5038 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
5039 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
5040 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
5041 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc

```

`\clist_gset_eq:NN` An of course globally which seems to be needed far more often.

```

\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc
5042 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
5043 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
5044 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
5045 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc

```

106.2 Predicates and conditionals

```

\clist_if_empty_p:N
\clist_if_empty_p:c
\clist_if_empty:N $\underline{TF}$ 
\clist_if_empty:c $\underline{TF}$ 
5046 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N {p,TF,T,F}
5047 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c {p,TF,T,F}

```

`\clist_if_empty_err:N` Signals an error if the comma-list is empty.

```

5048 \cs_new_protected_nopar:Npn \clist_if_empty_err:N #1 {
5049   \if_meaning:w #1 \c_empty_tl
5050   \tl_clear:N \l_kernel_testa_tl % catch prefixes
5051   \msg_kernel_bug:x {Empty~comma-list~'\token_to_str:N #1'}
5052   \fi:
5053 }

```

`\clist_if_eq_p:NN` Returns `\c_true` iff the two comma-lists are equal.

```

\clist_if_eq_p:Nc
\clist_if_eq_p:cN
\clist_if_eq_p:cc
\clist_if_eq:NN $\underline{TF}$ 
\clist_if_eq:cN $\underline{TF}$ 
\clist_if_eq:Nc $\underline{TF}$ 
\clist_if_eq:cc $\underline{TF}$ 
5054 \prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN {p,TF,T,F}
5055 \prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN {p,TF,T,F}
5056 \prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc {p,TF,T,F}
5057 \prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc {p,TF,T,F}

```


`\clist_if_in:NnTF` `\clist_if_in:NnTF` $\langle clist \rangle \langle item \rangle$ $\langle true case \rangle$ $\langle false case \rangle$ will check whether $\langle item \rangle$ is in $\langle clist \rangle$ and then either execute the $\langle true case \rangle$ or the $\langle false case \rangle$. $\langle true case \rangle$ and $\langle false case \rangle$ may contain incomplete `\if_charcode:w` statements.

```

\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF
5058 \prg_new_protected_conditional:Nnn \clist_if_in:Nn {TF,T,F} {
5059   \cs_set:Npn \clist_tmp:w ##1,##2,##3 \q_stop {
5060     \if_meaning:w \q_no_value ##2
5061     \prg_return_false: \else: \prg_return_true: \fi:
5062   }
5063   \exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop
5064 }

5065 \cs_generate_variant:Nn \clist_if_in:NnTF {NV,No,cn,cV,co}
5066 \cs_generate_variant:Nn \clist_if_in:NnT {NV,No,cn,cV,co}
5067 \cs_generate_variant:Nn \clist_if_in:NnF {NV,No,cn,cV,co}

```

106.3 Retrieving data

`\clist_use:N` Using a $\langle clist \rangle$ is just executing it but if $\langle clist \rangle$ equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cs_end:` that was created by mistake somewhere.

```

5068 \cs_new_nopar:Npn \clist_use:N #1 {
5069   \if_meaning:w #1 \scan_stop:
5070   \msg_kernel_bug:x {
5071     Comma~list~ '\token_to_str:N #1'~ has~ an~ erroneous~ structure!}
5072   \else:
5073     \exp_after:wN #1
5074   \fi:
5075 }
5076 \cs_generate_variant:Nn \clist_use:N {c}

```

`\clist_get:NN` `\clist_get:NN` $\langle comma-list \rangle \langle cmd \rangle$ defines $\langle cmd \rangle$ to be the left-most element of $\langle comma-list \rangle$.

```

\clist_get:cN
\clist_get_aux:w
5077 \cs_new_protected_nopar:Npn \clist_get:NN #1 {
5078   \clist_if_empty_err:N #1
5079   \exp_after:wN \clist_get_aux:w #1,\q_stop
5080 }

5081 \cs_new_protected:Npn \clist_get_aux:w #1,##2\q_stop #3 { \tl_set:Nn #3{#1} }
5082 \cs_generate_variant:Nn \clist_get:NN {cN}

```

`\clist_pop_aux:nnNN` `\clist_pop_aux:nnNN` $\langle def_1 \rangle \langle def_2 \rangle \langle comma-list \rangle \langle cmd \rangle$ assigns the left-most element of $\langle comma-list \rangle$ to $\langle cmd \rangle$ using $\langle def_2 \rangle$, and assigns the tail of $\langle comma-list \rangle$ to $\langle comma-list \rangle$ using $\langle def_1 \rangle$.

```

5083 \cs_new_protected:Npn \clist_pop_aux:nnNN #1#2#3 {
5084   \clist_if_empty_err:N #3
5085   \exp_after:wN \clist_pop_aux:w #3,\q_nil\q_stop #1#2#3
5086 }

```

After the assignments below, if there was only one element in the original *clist*, it now contains only `\q_nil`.

```

5087 \cs_new_protected:Npn \clist_pop_aux:w #1,#2\q_stop #3#4#5#6 {
5088     #4 #6 {#1}
5089     #3 #5 {#2}
5090     \quark_if_nil:NTF #5 { #3 #5 {} }{ \clist_pop_auxi:w #2 #3#5 }
5091 }

5092 \cs_new:Npn \clist_pop_auxi:w #1,\q_nil #2#3 { #2#3{#1} }

\clist_show:N
\clist_show:c

5093 \cs_new_eq:NN \clist_show:N \tl_show:N
5094 \cs_new_eq:NN \clist_show:c \tl_show:c

\clist_display:N
\clist_display:c

5095 \cs_new_protected_nopar:Npn \clist_display:N #1 {
5096     \iow_term:x { Comma-list~\token_to_str:N #1~contains~
5097         the-elements~(without~outer~braces): }
5098     \toks_clear:N \l_tmpa_toks
5099     \clist_map_inline:Nn #1 {
5100         \toks_if_empty:NF \l_tmpa_toks {
5101             \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5102         }
5103         \toks_put_right:Nx \l_tmpa_toks {
5104             \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
5105         }
5106     }
5107     \toks_show:N \l_tmpa_toks
5108 }
5109 \cs_generate_variant:Nn \clist_display:N {c}

```

106.4 Storing data

`\clist_put_aux:NNnnNn` The generic put function. When adding we have to distinguish between an empty *<clist>* and one that contains at least one item (otherwise we accumulate commas).

MH says: Perhaps we should make sure that empty arguments don't get on the stack as that is probably a mistake. That's what I've implemented here. Since `\tl_if_empty:nF` is expandable prefixes are still allowed.

```

5110 \cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6 {
5111     \clist_if_empty:NTF #5 { #1 #5 {#6} } {
5112         \tl_if_empty:nF {#6} { #2 #5{#3#6#4} }
5113     }
5114 }

```

```

\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_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_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_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

```

The operations for adding to the left.

```

5115 \cs_new_protected_nopar:Npn \clist_put_left:Nn {
5116   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn {} ,
5117 }
5118 \cs_generate_variant:Nn \clist_put_left:Nn {NV,No,Nx,cn,cV,co}

```

Global versions.

```

5119 \cs_new_protected_nopar:Npn \clist_gput_left:Nn {
5120   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn {} ,
5121 }
5122 \cs_generate_variant:Nn \clist_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

Adding something to the right side is almost the same.

```

5123 \cs_new_protected_nopar:Npn \clist_put_right:Nn {
5124   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , {}
5125 }
5126 \cs_generate_variant:Nn \clist_put_right:Nn {NV,No,Nx,cn,cV,co}

```

And here the global variants.

```

5127 \cs_new_protected_nopar:Npn \clist_gput_right:Nn {
5128   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , {}
5129 }
5130 \cs_generate_variant:Nn \clist_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

106.5 Mapping

```

\clist_map_function:NN
\clist_map_function:Nc
\clist_map_function:cN
\clist_map_function:cc
\clist_map_function:nN
\clist_map_function:nc
\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn
\clist_map_variable:nNn
\clist_map_break:
\clist_map_variable:NNn
\clist_map_variable:cNn

```

Using the above creating the comma mappings is easy..

```

5131 \prg_new_map_functions:Nn , { clist }
5132 \cs_generate_variant:Nn \clist_map_function:NN { Nc }
5133 \cs_generate_variant:Nn \clist_map_function:NN { c }
5134 \cs_generate_variant:Nn \clist_map_function:NN { cc }
5135 \cs_generate_variant:Nn \clist_map_inline:Nn { c }
5136 \cs_generate_variant:Nn \clist_map_inline:Nn { nc }

```

`\clist_map_variable:NNn` *<comma-list>* *<temp>* *<action>* assigns *<temp>* to each element and executes *<action>*.

```

5137 \cs_new_protected:Npn \clist_map_variable:nNn #1#2#3 {
5138   \tl_if_empty:nF {#1} {
5139     \clist_map_variable_aux:Nnw #2 {#3} #1
5140     , \q_recursion_tail , \q_recursion_stop
5141   }
5142 }

```

Something for v/V

```
5143 \cs_new_protected_nopar:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
5144 \cs_generate_variant:Nn\clist_map_variable:NNn {cNn}
```

\clist_map_variable_aux:Nnw

The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```
5145 \cs_new_protected:Npn \clist_map_variable_aux:Nnw #1#2#3,{
5146   \cs_set_nopar:Npn #1{#3}
5147   \quark_if_recursion_tail_stop:N #1
5148   #2 \clist_map_variable_aux:Nnw #1{#2}
5149 }
```

106.6 Higher level functions

\clist_concat_aux:NNNN
 \clist_concat:NNN
 \clist_concat:ccc
 \clist_gconcat:NNN
 \clist_gconcat:ccc

\clist_gconcat:NNN $\langle\textit{clist 1}\rangle$ $\langle\textit{clist 2}\rangle$ $\langle\textit{clist 3}\rangle$ will globally assign $\langle\textit{clist 1}\rangle$ the concatenation of $\langle\textit{clist 2}\rangle$ and $\langle\textit{clist 3}\rangle$.

Again the situation is a bit more complicated because of the use of commas between items, so if either list is empty we have to avoid adding a comma.

```
5150 \cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4 {
5151   \tl_set:No \l_tmpa_tl {#3}
5152   \tl_set:No \l_tmpb_tl {#4}
5153   #1 #2 {
5154     \exp_not:V \l_tmpa_tl
5155     \tl_if_empty:NF \l_tmpa_tl { \tl_if_empty:NF \l_tmpb_tl , }
5156     \exp_not:V \l_tmpb_tl
5157   }
5158 }
5159 \cs_new_protected_nopar:Npn \clist_concat:NNN { \clist_concat_aux:NNNN \tl_set:Nx }
5160 \cs_new_protected_nopar:Npn \clist_gconcat:NNN { \clist_concat_aux:NNNN \tl_gset:Nx }
5161 \cs_generate_variant:Nn \clist_concat:NNN {ccc}
5162 \cs_generate_variant:Nn \clist_gconcat:NNN {ccc}
```

\l_clist_remove_clist

A common scratch space for the removal routines.

```
5163 \clist_new:N \l_clist_remove_clist
```

\clist_remove_duplicates_aux:NN
 \clist_remove_duplicates_aux:n
 \clist_remove_duplicates:N
 \clist_gremove_duplicates:N

Removing duplicate entries in a $\langle\textit{clist}\rangle$ is fairly straight forward. We use a temporary variable and then go through the list from left to right. For each element check if the element is already present in the list.

```
5164 \cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2 {
5165   \clist_clear:N \l_clist_remove_clist
5166   \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
5167   #1 #2 \l_clist_remove_clist
```

```

5168 }
5169 \cs_new_protected:Npn \clist_remove_duplicates_aux:n #1 {
5170   \clist_if_in:NnF \l_clist_remove_clist {#1} {
5171     \clist_put_right:Nn \l_clist_remove_clist {#1}
5172   }
5173 }

```

The high level functions are just for telling if it should be a local or global setting.

```

5174 \cs_new_protected_nopar:Npn \clist_remove_duplicates:N {
5175   \clist_remove_duplicates_aux:NN \clist_set_eq:NN
5176 }
5177 \cs_new_protected_nopar:Npn \clist_gremove_duplicates:N {
5178   \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
5179 }

```

```

\clist_remove_element:Nn
\clist_gremove_element:Nn
\clist_remove_element_aux:NNn
\clist_remove_element_aux:n

```

The same general idea is used for removing elements: the parent functions just set things up for the internal ones.

```

5180 \cs_new_protected_nopar:Npn \clist_remove_element:Nn {
5181   \clist_remove_element_aux:NNn \clist_set_eq:NN
5182 }
5183 \cs_new_protected_nopar:Npn \clist_gremove_element:Nn {
5184   \clist_remove_element_aux:NNn \clist_gset_eq:NN
5185 }
5186 \cs_new_protected:Npn \clist_remove_element_aux:NNn #1#2#3 {
5187   \clist_clear:N \l_clist_remove_clist
5188   \cs_set:Npn \clist_remove_element_aux:n ##1 {
5189     \str_if_eq:nnF {#3} {##1} {
5190       \clist_put_right:Nn \l_clist_remove_clist {##1}
5191     }
5192   }
5193   \clist_map_function:NN #2 \clist_remove_element_aux:n
5194   #1 #2 \l_clist_remove_clist
5195 }
5196 \cs_new:Npn \clist_remove_element_aux:n #1 { }

```

106.7 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

```

\clist_push:Nn
\clist_push:No
\clist_push:NV
\clist_push:cn
\clist_pop:NN
\clist_pop:cN

```

Since comma-lists can be used as stacks, we ought to have both ‘push’ and ‘pop’. In most cases they are nothing more then new names for old functions.

```

5197 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
5198 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
5199 \cs_new_eq:NN \clist_push:No \clist_put_left:No
5200 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
5201 \cs_new_protected_nopar:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tl_set:Nn \tl_set:Nn}
5202 \cs_generate_variant:Nn \clist_pop:NN {cN}

```

`\clist_gpush:Nn` I don't agree with Denys that one needs only local stacks, actually I believe that one will
`\clist_gpush:No` probably need the functions here more often. In case of `\clist_gpop:NN` the value is
`\clist_gpush:NV` nevertheless returned locally.
`\clist_gpush:cn`
`\clist_gpop:NN` 5203 `\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn`
`\clist_gpop:cN` 5204 `\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV`
5205 `\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No`
5206 `\cs_generate_variant:Nn \clist_gpush:Nn {cn}`

5207 `\cs_new_protected_nopar:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn}`
5208 `\cs_generate_variant:Nn \clist_gpop:NN {cN}`

`\clist_top:NN` Looking at the top element of the stack without removing it is done with this operation.
`\clist_top:cN`

5209 `\cs_new_eq:NN \clist_top:NN \clist_get:NN`
5210 `\cs_new_eq:NN \clist_top:cN \clist_get:cN`

5211 `\</initex | package>`

Show token usage:

5212 `<*showmemory>`
5213 `%\showMemUsage`
5214 `</showmemory>`

107 l3prop implementation

A property list is a token register whose contents is of the form

`\q_prop <key> \q_prop {\<info>} ... \q_prop <key> \q_prop {\<info>}`

The property `<key>`s and `<info>`s might be arbitrary token lists; each `<info>` is surrounded by braces.

We start by ensuring that the required packages are loaded.

5215 `<*package>`
5216 `\ProvidesExplPackage`
5217 `{\filename}{\filedate}{\fileversion}{\filedescription}`
5218 `\package_check_loaded_expl:`
5219 `</package>`
5220 `<*initex | package>`

`\q_prop` The separator between `<key>`s and `<info>`s and `<key>`s.

5221 `\quark_new:N\q_prop`

To get values from property-lists, token lists should be passed to the appropriate functions.

107.1 Functions

`\prop_new:N` Property lists are implemented as token registers.

`\prop_new:c`

```

5222 \cs_new_eq:NN \prop_new:N \toks_new:N
5223 \cs_new_eq:NN \prop_new:c \toks_new:c

```

`\prop_clear:N` The same goes for clearing a property list, either locally or globally.

`\prop_clear:c`

```

5224 \cs_new_eq:NN \prop_clear:N \toks_clear:N
5225 \cs_new_eq:NN \prop_clear:c \toks_clear:c
5226 \cs_new_eq:NN \prop_gclear:N \toks_gclear:N
5227 \cs_new_eq:NN \prop_gclear:c \toks_gclear:c

```

`\prop_set_eq:NN` This makes two $\langle prop \rangle$ s have the same contents.

`\prop_set_eq:Nc`

```

5228 \cs_new_eq:NN \prop_set_eq:NN \toks_set_eq:NN
5229 \cs_new_eq:NN \prop_set_eq:Nc \toks_set_eq:Nc
5230 \cs_new_eq:NN \prop_set_eq:cN \toks_set_eq:cN
5231 \cs_new_eq:NN \prop_set_eq:cc \toks_set_eq:cc
5232 \cs_new_eq:NN \prop_gset_eq:NN \toks_gset_eq:NN
5233 \cs_new_eq:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
5234 \cs_new_eq:NN \prop_gset_eq:cN \toks_gset_eq:cN
5235 \cs_new_eq:NN \prop_gset_eq:cc \toks_gset_eq:cc

```

`\prop_show:N` Show on the console the raw contents of a property list's token register.

`\prop_show:c`

```

5236 \cs_new_eq:NN \prop_show:N \toks_show:N
5237 \cs_new_eq:NN \prop_show:c \toks_show:c

```

`\prop_display:N` Pretty print the contents of a property list on the console.

`\prop_display:c`

```

5238 \cs_new_protected_nopar:Npn \prop_display:N #1 {
5239   \iow_term:x { Property-list~\token_to_str:N #1~contains~
5240     the~pairs~(without~outer~braces): }
5241   \toks_clear:N \l_tmpa_toks
5242   \prop_map_inline:Nn #1 {
5243     \toks_if_empty:NF \l_tmpa_toks {
5244       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5245     }
5246     \toks_put_right:Nx \l_tmpa_toks {
5247       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \} \c_space_tl
5248       \c_space_tl => \c_space_tl
5249       \c_space_tl \iow_char:N \{ \exp_not:n {##2} \iow_char:N \}
5250     }
5251   }
5252   \toks_show:N \l_tmpa_toks
5253 }
5254 \cs_generate_variant:Nn \prop_display:N {c}

```

`\prop_split_aux:Nnn` `\prop_split_aux:Nnn⟨prop⟩⟨key⟩⟨cmd⟩` invokes `⟨cmd⟩` with 3 arguments: 1st is the beginning of `⟨prop⟩` before `⟨key⟩`, 2nd is the value associated with `⟨key⟩`, 3rd is the rest of `⟨prop⟩` after `⟨key⟩`. If there is no property `⟨key⟩` in `⟨prop⟩`, then the 2nd argument will be `\q_no_value` and the 3rd argument is empty; otherwise the 3rd argument has the extra tokens `\q_prop ⟨key⟩ \q_prop \q_no_value` at the end.

```

5255 \cs_new_protected:Npn \prop_split_aux:Nnn #1#2#3{
5256   \cs_set:Npn \prop_tmp:w ##1 \q_prop #2 \q_prop ##2##3 \q_stop {
5257     #3 {##1}{##2}{##3}
5258   }
5259   \exp_after:wN \prop_tmp:w \toks_use:N #1 \q_prop #2 \q_prop \q_no_value \q_stop
5260 }

```

`\prop_get:NnN` `\prop_get:NnN ⟨prop⟩⟨key⟩⟨tl var.⟩` defines `⟨tl var.⟩` to be the value associated with `⟨key⟩` in `⟨prop⟩`, `\q_no_value` if not found.

```

\prop_get:NnN
\prop_get:NVN
\prop_get:cnN
\prop_get:cVN
\prop_get_aux:w
5261 \cs_new_protected:Npn \prop_get:NnN #1#2 {
5262   \prop_split_aux:Nnn #1{#2}\prop_get_aux:w
5263 }
5264 \cs_new_protected:Npn \prop_get_aux:w #1#2#3#4 { \tl_set:Nn #4 {#2} }
5265 \cs_generate_variant:Nn \prop_get:NnN { NVN, cnN, cVN }

```

`\prop_gget:NnN` The global version of the previous function.

```

\prop_gget:NnN
\prop_gget:NVN
\prop_gget:NnN
\prop_gget:NVN
\prop_gget_aux:w
5266 \cs_new_protected:Npn \prop_gget:NnN #1#2{
5267   \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
5268 \cs_new_protected:Npn \prop_gget_aux:w #1#2#3#4{\tl_gset:Nx#4{\exp_not:n{#2}}}
5269 \cs_generate_variant:Nn \prop_gget:NnN { NVN, cnN, cVN }

```

`\prop_get_gdel:NnN` `\prop_get_gdel:NnN` is the same as `\prop_get:NnN` but the `⟨key⟩` and its value are afterwards globally removed from `⟨property_list⟩`. One probably also needs the local variants or only the local one, or... We decide this later.

```

5270 \cs_new_protected:Npn \prop_get_gdel:NnN #1#2#3{
5271   \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1{#2}}}
5272   \cs_new_protected:Npn \prop_get_del_aux:w #1#2#3#4#5#6{
5273     \tl_set:Nn #1 {#5}
5274     \quark_if_no_value:NF #1 {
5275       \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
5276       \prop_tmp:w #6}
5277   }

```

`\prop_put:Nnn` `\prop_put:Nnn {⟨prop⟩} {⟨key⟩} {⟨info⟩}` adds/changes the value associated with `⟨key⟩` in `⟨prop⟩` to `⟨info⟩`.

```

\prop_put:Nno
\prop_put:NnV
\prop_put:Nnx
\prop_put:NVn
\prop_put:NVV
5278 \cs_new_protected:Npn \prop_put:Nnn #1#2{
5279   \prop_split_aux:Nnn #1{#2}{

```



```

5280     \prop_clear:N #1
5281     \prop_put_aux:w {\toks_put_right:Nn #1}{#2}
5282   }
5283 }

5284 \cs_new_protected:Npn \prop_gput:Nnn #1#2{
5285   \prop_split_aux:Nnn #1{#2}{
5286     \prop_gclear:N #1
5287     \prop_put_aux:w {\toks_gput_right:Nn #1}{#2}
5288   }
5289 }

5290 \cs_new_protected:Npn \prop_put_aux:w #1#2#3#4#5#6{
5291   #1{\q_prop#2\q_prop{#6}{#3}
5292   \tl_if_empty:nF{#5}
5293   {
5294     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
5295     \prop_tmp:w #5
5296   }
5297 }

5298 \cs_generate_variant:Nn \prop_put:Nnn { Nno , NnV, Nnx, NVn, NVV, cnn , cnx }
5299 \cs_generate_variant:Nn \prop_gput:Nnn { NVn,NnV,Nno,Nnx,Nox,cnn,ccx}

```

`\prop_del:Nn` `\prop_del:Nn` $\langle prop \rangle \langle key \rangle$ deletes the entry for $\langle key \rangle$ in $\langle prop \rangle$, if any.
`\prop_del:NV`
`\prop_gdel:NV`
`\prop_gdel:Nn`
`\prop_del_aux:w`

```

5300 \cs_new_protected:Npn \prop_del:Nn #1#2{
5301   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}
5302 \cs_new_protected:Npn \prop_gdel:Nn #1#2{
5303   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}
5304 \cs_new_protected:Npn \prop_del_aux:w #1#2#3#4#5{
5305   \cs_set_nopar:Npn \prop_tmp:w {#4}
5306   \quark_if_no_value:NF \prop_tmp:w {
5307     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{#3##1}}
5308     \prop_tmp:w #5
5309   }
5310 }
5311 \cs_generate_variant:Nn \prop_del:Nn { NV }
5312 \cs_generate_variant:Nn \prop_gdel:Nn { NV }
5313 %

```

`\prop_gput_if_new:Nnn` `\prop_gput_if_new:Nnn` $\langle prop \rangle \langle key \rangle \langle info \rangle$ is equivalent to
`\prop_put_if_new_aux:w`

```

\prop_if_in:NnTF \langle prop \rangle \langle key \rangle
  {}%
  {\prop_gput:Nnn
    \langle property_list \rangle
    \langle key \rangle
    \langle info \rangle}

```

Here we go (listening to Porgy & Bess in a recording with Ella F. and Louis A. which makes writing macros sometimes difficult; I find myself humming instead of working):

```

5314 \cs_new_protected:Npn \prop_gput_if_new:Nnn #1#2{
5315   \prop_split_aux:Nnn #1#{#2}{\prop_put_if_new_aux:w #1#{#2}}
5316 \cs_new_protected:Npn \prop_put_if_new_aux:w #1#2#3#4#5#6{
5317   \tl_if_empty:nT {#5}{#1{\q_prop#2\q_prop#{#6}#3}}

```

107.2 Predicates and conditionals

$\backslash \text{prop_if_empty_p:N}$	This conditional takes a $\langle \text{prop} \rangle$ as its argument and evaluates either the true or the false
$\backslash \text{prop_if_empty_p:c}$	case, depending on whether or not $\langle \text{prop} \rangle$ contains any properties.

```

\prop_if_empty:N $\underline{TF}$ 
\prop_if_empty:c $\underline{TF}$  5318 \prg_new_eq_conditional:NNn \prop_if_empty:N \toks_if_empty:N {p,TF,T,F}
5319 \prg_new_eq_conditional:NNn \prop_if_empty:c \toks_if_empty:c {p,TF,T,F}

```

`\prop_if_eq_p:NN` These functions test whether two property lists are equal.

$\backslash\text{prop_if_eq_p:cN}$	5320	$\backslash\text{prg_new_eq_conditional:Nnn}$	$\backslash\text{prop_if_eq:NN}$	$\backslash\text{toks_if_eq:NN}$	$\{p,TF,T,F\}$
$\backslash\text{prop_if_eq_p:Nc}$	5321	$\backslash\text{prg_new_eq_conditional:Nnn}$	$\backslash\text{prop_if_eq:cN}$	$\backslash\text{toks_if_eq:cN}$	$\{p,TF,T,F\}$
$\backslash\text{prop_if_eq_p:cc}$	5322	$\backslash\text{prg_new_eq_conditional:Nnn}$	$\backslash\text{prop_if_eq:Nc}$	$\backslash\text{toks_if_eq:Nc}$	$\{p,TF,T,F\}$
$\backslash\text{prop_if_eq:NNTF}$	5323	$\backslash\text{prg_new_eq_conditional:Nnn}$	$\backslash\text{prop_if_eq:cc}$	$\backslash\text{toks_if_eq:cc}$	$\{p,TF,T,F\}$

$\backslash\text{prop_if_eq:cNTF}$ $\backslash\text{prop_if_in:NnTF}$ $\langle\text{property_list}\rangle$ $\langle\text{key}\rangle$ $\langle\text{true_case}\rangle$ $\langle\text{false_case}\rangle$ will check whether or
 $\backslash\text{prop_if_eq:ccTF}$ $\backslash\text{prop_if_in:NVTf}$ not $\langle\text{key}\rangle$ is on the $\langle\text{property_list}\rangle$ and then select either the true or false case.

```

\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:ccTF
\prop_if_in_aux:w
5324 \prg_new_protected_conditional:Nnn \prop_if_in:Nn {TF,T,F} {
5325   \prop_split_aux:Nnn #1 {#2} {\prop_if_in_aux:w}
5326 }
5327 \cs_new_nopar:Npn \prop_if_in_aux:w #1#2#3 {
5328   \quark_if_no_value:nTF {#2} {\prg_return_false:} {\prg_return_true:}
5329 }

5330 \cs_generate_variant:Nn \prop_if_in:NnTF {NV,No,cn,cc}
5331 \cs_generate_variant:Nn \prop_if_in:NnT {NV,No,cn,cc}
5332 \cs_generate_variant:Nn \prop_if_in:NnF {NV,No,cn,cc}

```

107.3 Mapping functions

<code>\prop_map_function:NN</code>	Maps a function on every entry in the property list. The function must take 2 arguments:
<code>\prop_map_function:cN</code>	a key and a value.

```
\prop_map_function:Nc
```

First, some failed attempts:

```

\prop_map_function_aux:w
\cs_new_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop{}\q_prop \q_no_value \q_stop

```

```

}
\cs_new_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \if_predicate:w \tl_if_empty_p:n{#2}
    \exp_after:wN \prop_map_break:
  \fi:
  #1{#2}{#3}
  \prop_map_function_aux:w #1
}

```

problem with the above implementation is that an empty key stops the mapping but all other functions in the module allow the use of empty keys (as one value)

```

\cs_set_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop \q_no_value \q_prop \q_no_value
}
\cs_set_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \quark_if_no_value:nF{#2}
  {
    #1{#2}{#3}
    \prop_map_function_aux:w #1
  }
}

```

problem with the above implementation is that `\quark_if_no_value:nF` is fairly slow and if `\quark_if_no_value:NF` is used instead we have to do an assignment thus making the mapping not expandable (is that important?)

Here's the current version of the code:

```

5333 \cs_set_nopar:Npn \prop_map_function:NN #1#2 {
5334   \exp_after:wN \prop_map_function_aux:w
5335   \exp_after:wN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
5336 }
5337 \cs_set:Npn \prop_map_function_aux:w #1 \q_prop #2 \q_prop #3 {
5338   \if_meaning:w \q_nil #2
5339   \exp_after:wN \prop_map_break:
5340   \fi:
5341   #1{#2}{#3}
5342   \prop_map_function_aux:w #1
5343 }

```

(potential) problem with the above implementation is that it will return true if #2 contains more than just `\q_nil` thus executing whatever follows. Claim: this can't happen :-)
so we should be ok

```

5344 \cs_generate_variant:Nn \prop_map_function:NN {c,Nc,cc}

```

`\prop_map_inline:Nn` The inline functions are straight forward. It takes longer to test if the list is empty than to run it on an empty list so we don't waste time doing that.

`\prop_map_inline:cn`

`\g_prop_inline_level_int`

```

5345 \int_new:N \g_prop_inline_level_int
5346 \cs_new_protected_nopar:Npn \prop_map_inline:Nn #1#2 {
5347   \int_gincr:N \g_prop_inline_level_int
5348   \cs_gset:cpn {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5349     ##1##2{#2}
5350   \prop_map_function:Nc #1
5351     {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5352   \int_gdecr:N \g_prop_inline_level_int
5353 }

5354 \cs_generate_variant:Nn\prop_map_inline:Nn {cn}

```

`\prop_map_break:` The break statement.

```

5355 \cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_stop:w

5356 </initex | package>

```

Show token usage:

```

5357 <*showmemory>
5358 %\showMemUsage
5359 </showmemory>

```

108 l3io implementation

We start by ensuring that the required packages are loaded.

```

5360 <*package>
5361 \ProvidesExplPackage
5362   {\filename}{\filedate}{\fileversion}{\filedescription}
5363   \package_check_loaded_expl:
5364 </package>
5365 <*initex | package>

```

108.1 Variables and constants

`\c_iow_term_stream` Here we allocate two output streams for writing to the transcript file only (`\c_iow_log_stream`) and to both the terminal and transcript file (`\c_iow_term_stream`). Both can be used to read from and have equivalent `\c_ior` versions.

`\c_ior_term_stream`

`\c_iow_log_stream`

`\c_ior_log_stream`

```

5366 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
5367 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
5368 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
5369 \cs_new_eq:NN \c_ior_log_stream \c_minus_one

```

`\c_iow_streams_tl` The list of streams available, by number.

```

\c_ior_streams_tl
5370 \tl_const:Nn \c_iow_streams_tl
5371 {
5372   \c_zero
5373   \c_one
5374   \c_two
5375   \c_three
5376   \c_four
5377   \c_five
5378   \c_six
5379   \c_seven
5380   \c_eight
5381   \c_nine
5382   \c_ten
5383   \c_eleven
5384   \c_twelve
5385   \c_thirteen
5386   \c_fourteen
5387   \c_fifteen
5388 }
5389 \cs_new_eq:NN \c_ior_streams_tl \c_iow_streams_tl

```

`\g_iow_streams_prop` The allocations for streams are stored in property lists, which are set up to have a 'full' set of allocations from the start. In package mode, a few slots are always taken, so these are blocked off from use.

```

5390 \prop_new:N \g_iow_streams_prop
5391 \prop_new:N \g_ior_streams_prop
5392 </initex | package>
5393 <*package>
5394 \prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e-reserved }
5395 \prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e-reserved }
5396 \prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e-reserved }
5397 \prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e-reserved }
5398 </package>
5399 <*initex | package>

```

`\l_iow_stream_int` Used to track the number allocated to the stream being created: this is taken from the property list but does alter.

```

5400 \int_new:N \l_iow_stream_int
5401 \cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int

```

108.2 Stream management

`\iow_raw_new:N` The lowest level for stream management is actually creating raw T_EX streams. As these are very limited (even with ϵ -T_EX) this should not be addressed directly.

```

5402 </initex|package>
5403 <*initex>
5404 \alloc_setup_type:nnn { iow } \c_zero \c_sixteen
5405 \cs_new_protected_nopar:Npn \iow_raw_new:N #1 {
5406   \alloc_reg:NnNN g { iow } \tex_chardef:D #1
5407 }
5408 \alloc_setup_type:nnn { ior } \c_zero \c_sixteen
5409 \cs_new_protected_nopar:Npn \ior_raw_new:N #1 {
5410   \alloc_reg:NnNN g { ior } \tex_chardef:D #1
5411 }
5412 </initex>
5413 <*package>
5414 \cs_set_eq:NN \iow_raw_new:N \newwrite
5415 \cs_set_eq:NN \ior_raw_new:N \newread
5416 </package>
5417 <*initex|package>
5418 \cs_generate_variant:Nn \iow_raw_new:N { c }
5419 \cs_generate_variant:Nn \ior_raw_new:N { c }

```

\iow_new:N These are not needed but are included for consistency with other variable types.

```

\iow_new:c
\ior_new:N
\ior_new:c
5420 \cs_new_protected_nopar:Npn \iow_new:N #1 {
5421   \cs_new_eq:NN #1 \c_iow_log_stream
5422 }
5423 \cs_generate_variant:Nn \iow_new:N { c }
5424 \cs_new_protected_nopar:Npn \ior_new:N #1 {
5425   \cs_new_eq:NN #1 \c_ior_log_stream
5426 }
5427 \cs_generate_variant:Nn \ior_new:N { c }

```

\iow_open:Nn In both cases, opening a stream starts with a call to the closing function: this is safest.
\iow_open:cn There is then a loop through the allocation number list to find the first free stream
\ior_open:Nn number. When one is found the allocation can take place, the information can be stored
\ior_open:cn and finally the file can actually be opened.

```

5428 \cs_new_protected_nopar:Npn \iow_open:Nn #1#2 {
5429   \iow_close:N #1
5430   \int_set:Nn \l_iow_stream_int { \c_sixteen }
5431   \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
5432   \int_compare:nTF { \l_iow_stream_int = \c_sixteen }
5433     { \msg_kernel_error:nn { iow } { streams-exhausted } }
5434     {
5435       \iow_stream_alloc:N #1
5436       \prop_gput:Nvn \g_iow_streams_prop \l_iow_stream_int {#2}
5437       \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
5438     }
5439 }
5440 \cs_generate_variant:Nn \iow_open:Nn { c }
5441 \cs_new_protected_nopar:Npn \ior_open:Nn #1#2 {

```

```

5442 \ior_close:N #1
5443 \int_set:Nn \l_ior_stream_int { \c_sixteen }
5444 \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
5445 \int_compare:nTF { \l_ior_stream_int = \c_sixteen }
5446   { \msg_kernel_error:nn { ior } { streams-exhausted } }
5447   {
5448     \ior_stream_alloc:N #1
5449     \prop_gput:NVn \g_ior_streams_prop \l_ior_stream_int {#2}
5450     \tex_openin:D #1#2 \scan_stop:
5451   }
5452 }
5453 \cs_generate_variant:Nn \ior_open:Nn { c }

```

\ior_alloc_write:n These functions are used to see if a particular stream is available. The property list
\ior_alloc_read:n contains file names for streams in use, so any unused ones are for the taking.

```

5454 \cs_new_protected_nopar:Npn \ior_alloc_write:n #1 {
5455   \prop_if_in:NnF \g_ior_streams_prop {#1}
5456   {
5457     \int_set:Nn \l_ior_stream_int {#1}
5458     \tl_map_break:
5459   }
5460 }
5461 \cs_new_protected_nopar:Npn \ior_alloc_read:n #1 {
5462   \prop_if_in:NnF \g_ior_streams_prop {#1}
5463   {
5464     \int_set:Nn \l_ior_stream_int {#1}
5465     \tl_map_break:
5466   }
5467 }

```

\ior_stream_alloc:N Allocating a raw stream is much easier in initex mode than for the package. For the
\ior_stream_alloc:N format, all streams will be allocated by l3io and so there is a simple check to see if a
\ior_stream_alloc_aux: raw stream is actually available. On the other hand, for the package there will be non-
\ior_stream_alloc_aux: managed streams. So if the managed one is not open, a check is made to see if some
\g_ior_tmp_stream other managed stream is available before deciding to open a new one. If a new one is
\g_ior_tmp_stream needed, we get the number allocated by L^AT_EX 2_ε to get ‘back on track’ with allocation.

```

5468 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
5469   \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5470   { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
5471   {
5472     </initex | package>
5473     <*package>
5474     \ior_stream_alloc_aux:
5475     \int_compare:nT { \l_ior_stream_int = \c_sixteen }
5476     {
5477       \ior_raw_new:N \g_ior_tmp_stream
5478       \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }

```

```

5479         \cs_gset_eq:cN
5480         { g_iow_ \int_use:N \l_iow_stream_int _stream }
5481         \g_iow_tmp_stream
5482     }
5483 </package>
5484 <*initex>
5485     \iow_raw_new:c { g_iow_ \int_use:N \l_iow_stream_int _stream }
5486 </initex>
5487 <*initex | package>
5488     \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream }
5489 }
5490 }
5491 </initex | package>
5492 <*package>
5493 \cs_new_protected_nopar:Npn \iow_stream_alloc_aux: {
5494     \int_incr:N \l_iow_stream_int
5495     \int_compare:nT
5496     { \l_iow_stream_int < \c_sixteen }
5497     {
5498         \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
5499         {
5500             \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
5501             { \iow_stream_alloc_aux: }
5502         }
5503         { \iow_stream_alloc_aux: }
5504     }
5505 }
5506 </package>
5507 <*initex | package>
5508 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
5509     \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5510     { \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream } }
5511     {
5512 </initex | package>
5513 <*package>
5514     \ior_stream_alloc_aux:
5515     \int_compare:nT { \l_ior_stream_int = \c_sixteen }
5516     {
5517         \ior_raw_new:N \g_ior_tmp_stream
5518         \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
5519         \cs_gset_eq:cN
5520         { g_ior_ \int_use:N \l_iow_stream_int _stream }
5521         \g_ior_tmp_stream
5522     }
5523 </package>
5524 <*initex>
5525     \ior_raw_new:c { g_ior_ \int_use:N \l_ior_stream_int _stream }
5526 </initex>
5527 <*initex | package>
5528     \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _stream }

```



```

5529     }
5530   }
5531   </initex|package>
5532   <*package>
5533   \cs_new_protected_nopar:Npn \ior_stream_alloc_aux: {
5534     \int_incr:N \l_ior_stream_int
5535     \int_compare:nT
5536       { \l_ior_stream_int < \c_sixteen }
5537     {
5538       \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _stream }
5539       {
5540         \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
5541         { \ior_stream_alloc_aux: }
5542       }
5543       { \ior_stream_alloc_aux: }
5544     }
5545   }
5546   </package>
5547   <*initex|package>

```

\iow_close:N 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.

```

\iow_close:c
\iow_close:N
\iow_close:c
5548   \cs_new_protected_nopar:Npn \iow_close:N #1 {
5549     \cs_if_exist:NT #1
5550     {
5551       \int_compare:nF { #1 = \c_minus_one }
5552       {
5553         \tex_immediate:D \tex_closeout:D #1
5554         \prop_gdel:NV \g_iow_streams_prop #1
5555         \cs_gundefine:N #1
5556       }
5557     }
5558   }
5559   \cs_generate_variant:Nn \iow_close:N { c }
5560   \cs_new_protected_nopar:Npn \ior_close:N #1 {
5561     \cs_if_exist:NT #1
5562     {
5563       \int_compare:nF { #1 = \c_minus_one }
5564       {
5565         \tex_closein:D #1
5566         \prop_gdel:NV \g_ior_streams_prop #1
5567         \cs_gundefine:N #1
5568       }
5569     }
5570   }
5571   \cs_generate_variant:Nn \ior_close:N { c }

```

`\iow_open_streams:` Simply show the property lists.

```
\ior_open_streams:
5572 \cs_new_protected_nopar:Npn \iow_open_streams: {
5573   \prop_display:N \g_iow_streams_prop
5574 }
5575 \cs_new_protected_nopar:Npn \ior_open_streams: {
5576   \prop_display:N \g_ior_streams_prop
5577 }
```

Text for the error messages.

```
5578 \msg_kernel_new:nnnn { iow } { streams-exhausted }
5579 {Output streams exhausted}
5580 {%
5581   TeX can only open up to 16 output streams at one time.\\%
5582   All 16 are currently in use, and something wanted to open
5583   another one.%
5584 }
5585 \msg_kernel_new:nnnn { ior } { streams-exhausted }
5586 {Input streams exhausted}
5587 {%
5588   TeX can only open up to 16 input streams at one time.\\%
5589   All 16 are currently in use, and something wanted to open
5590   another one.%
5591 }
```

108.3 Immediate writing

`\iow_now:Nx` An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```
5592 \cs_new_protected_nopar:Npn \iow_now:Nx { \tex_immediate:D \iow_shipout_x:Nn }
```

`\iow_now:Nn` 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.

```
5593 \cs_new_protected_nopar:Npn \iow_now:Nn #1#2 {
5594   \iow_now:Nx #1 { \exp_not:n {#2} }
5595 }
```

`\iow_log:n` Now we redefine two functions for which we needed a definition very early on.

```
\iow_log:x
5596 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
\iow_term:n
5597 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
\iow_term:x
5598 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
5599 \cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }
```

```

\iow_now_when_avail:Nn
\iow_now_when_avail:cn
\iow_now_when_avail:Nx
\iow_now_when_avail:cx

```

For writing only if the stream requested is open at all.

```

5600 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1 {
5601   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 }
5602 }
5603 \cs_generate_variant:Nn \iow_now_when_avail:Nn { c }
5604 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1 {
5605   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 }
5606 }
5607 \cs_generate_variant:Nn \iow_now_when_avail:Nx { c }

```

```

\iow_now_buffer_safe:Nn
\iow_now_buffer_safe:Nx
\iow_now_buffer_safe_expanded_aux:w

```

Another type of writing onto an output stream is used for potentially long token sequences. We break the output lines at every blank in the second argument. This avoids the problem of buffer overflow when reading back, or badly broken lines on systems with limited file records. The only thing we have to take care of, is the danger of two blanks in succession since these get converted into a `\par` when we read the stuff back. But this can happen only if things like two spaces find their way into the second argument. Usually, multiple spaces are removed by T_EX's scanner.

```

5608 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nn {
5609   \iow_now_buffer_safe_aux:w \iow_now:Nn
5610 }
5611 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nx {
5612   \iow_now_buffer_safe_aux:w \iow_now:Nn
5613 }
5614 \cs_new_protected_nopar:Npn \iow_now_buffer_safe_aux:w #1#2#3 {
5615   \group_begin: \tex_newlinechar:D'\ #1#2 {#3} \group_end:
5616 }

```

108.4 Deferred writing

```

\iow_shipout_x:Nn
\iow_shipout_x:Nx

```

First the easy part, this is the primitive.

```

5617 \cs_set_eq:NN \iow_shipout_x:Nn \tex_write:D
5618 \cs_generate_variant:Nn \iow_shipout_x:Nn {Nx }

```

```

\iow_shipout:Nn
\iow_shipout:Nx

```

With ϵ -T_EX available deferred writing is easy.

```

5619 \cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2 {
5620   \iow_shipout_x:Nn #1 { \exp_not:n {#2} }
5621 }
5622 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }

```

109 Special characters for writing

```

\iow_newline:

```

Global variable holding the character that forces a new line when something is written to an output stream.

```

5623 \cs_new_nopar:Npn \iow_newline: { ^^J }

```

`\iow_char:N` Function to write any escaped char to an output stream.

```
5624 \cs_new:Npn \iow_char:N #1 { \cs_to_str:N #1 }
```

109.1 Reading input

`\if_eof:w` A simple primitive renaming.

```
5625 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` As the pool model means that closed streams are undefined control sequences, the test has two parts.

```
5626 \prg_new_conditional:Nnn \ior_if_eof:N { p , TF , T , F } {
5627   \cs_if_exist:NTF #1
5628   { \tex_ifeof:D #1 \prg_return_true: \else: \prg_return_false: \fi: }
5629   { \prg_return_true: }
5630 }
```

`\ior_to:NN` And here we read from files.

```
\ior_gto:NN
5631 \cs_new_protected_nopar:Npn \ior_to:NN #1#2 {
5632   \tex_read:D #1 to #2
5633 }
5634 \cs_new_protected_nopar:Npn \ior_gto:NN {
5635   \pref_global:D \ior_to:NN
5636 }
5637 </initex | package>
```

110 l3msg implementation

The usual lead-off.

```
5638 <*package>
5639 \ProvidesExplPackage
5640   {\filename}{\filedate}{\fileversion}{\filedescription}
5641   \package_check_loaded_expl:
5642 </package>
5643 <*initex | package>
```

L^AT_EX is handling context, so the T_EX “noise” is turned down.

```
5644 \int_set:Nn \tex_errorcontextlines:D { \c_minus_one }
```

110.1 Variables and constants

<code>\c_msg_error_tl</code>	Header information.
<code>\c_msg_warning_tl</code>	
<code>\c_msg_info_tl</code>	<pre> 5645 \tl_const:Nn \c_msg_error_tl { error } 5646 \tl_const:Nn \c_msg_warning_tl { warning } 5647 \tl_const:Nn \c_msg_info_tl { info } </pre>
<code>\msg_fatal_text:n</code>	Contextual header/footer information.
<code>\msg_see_documentation_text:n</code>	<pre> 5648 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error } 5649 \cs_new:Npn \msg_see_documentation_text:n #1 5650 { See~the~#1~documentation~for~further~information } </pre>
<code>\c_msg_coding_error_text_tl</code>	Simple pieces of text for messages.
<code>\c_msg_fatal_text_tl</code>	
<code>\c_msg_help_text_tl</code>	
<code>\c_msg_kernel_bug_text_tl</code>	
<code>\c_msg_kernel_bug_more_text_tl</code>	
<code>\c_msg_no_info_text_tl</code>	
<code>\c_msg_return_text_tl</code>	<pre> 5651 \tl_const:Nn \c_msg_coding_error_text_tl { 5652 This~is~a~coding~error. 5653 \msg_two_newlines: 5654 } 5655 \tl_const:Nn \c_msg_fatal_text_tl { 5656 This~is~a~fatal~error:~LaTeX~will~abort 5657 } 5658 \tl_const:Nn \c_msg_help_text_tl { 5659 For~immediate~help~type~H~<return> 5660 } 5661 \tl_const:Nn \c_msg_kernel_bug_text_tl { 5662 This~is~a~LaTeX~bug:~check~coding! 5663 } 5664 \tl_const:Nn \c_msg_kernel_bug_more_text_tl { 5665 There~is~a~coding~bug~somewhere~around~here. \\\ 5666 This~probably~needs~examining~by~an~expert. 5667 \c_msg_return_text_tl 5668 } 5669 \tl_const:Nn \c_msg_no_info_text_tl { 5670 LaTeX~does~not~know~anything~more~about~this~error,~sorry. 5671 \c_msg_return_text_tl 5672 } 5673 \tl_const:Nn \c_msg_return_text_tl { 5674 \\\ \\\ 5675 Try~typing~<return>~to~proceed. 5676 \\\ 5677 If~that~doesn't~work,~type~X~<return>~to~quit 5678 } </pre>
<code>\c_msg_hide_tl<spaces></code>	An empty variable with a number of (category code 11) periods at the end of its name. This is used to push the T _E X part of an error message “off the screen”.
	<pre> 5679 \group_begin: </pre>

```

5680 \char_make_letter:N \.
5681 \tl_to_lowercase:n {
5682   \group_end:
5683   \tl_const:Nn \c_msg_hide_tl.....
5684   { }
5685 }

```

`\c_msg_on_line_tl` Text for “on line”.

```

5686 \tl_const:Nn \c_msg_on_line_tl { on~line }

```

`\c_msg_text_prefix_tl` Prefixes for storage areas.
`\c_msg_more_text_prefix_tl`

```

5687 \tl_const:Nn \c_msg_text_prefix_tl { msg_text ~>~ }
5688 \tl_const:Nn \c_msg_more_text_prefix_tl { msg_text_more ~>~ }

```

`\l_msg_class_tl` For holding the current message method and that for redirection.
`\l_msg_current_class_tl`
`\l_msg_current_module_tl`

```

5689 \tl_new:N \l_msg_class_tl
5690 \tl_new:N \l_msg_current_class_tl
5691 \tl_new:N \l_msg_current_module_tl

```

`\l_msg_names_clist` Lists used for filtering.

```

5692 \clist_new:N \l_msg_names_clist

```

`\l_msg_redirect_classes_prop` For filtering messages, a list of all messages and of those which have to be modified is required.
`\l_msg_redirect_names_prop`

```

5693 \prop_new:N \l_msg_redirect_classes_prop
5694 \prop_new:N \l_msg_redirect_names_prop

```

`\l_msg_redirect_classes_clist` To prevent an infinite loop.

```

5695 \clist_new:N \l_msg_redirect_classes_clist

```

`\l_msg_tmp_tl` A scratch variable.

```

5696 \tl_new:N \l_msg_tmp_tl

```

110.2 Output helper functions

`\msg_line_number:` For writing the line number nicely.
`\msg_line_context:`

```

5697 \cs_new_nopar:Npn \msg_line_number: {
5698   \toks_use:N \tex_inputlineno:D
5699 }

```

```

5700 \cs_new_nopar:Npn \msg_line_context: {
5701   \c_msg_on_line_tl
5702   \c_space_tl
5703   \msg_line_number:
5704 }

```

`\msg_newline:` Always forces a new line.

`\msg_two_newlines:`

```

5705 \cs_new_nopar:Npn \msg_newline:      { ^^J }
5706 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }

```

110.3 Generic functions

The lowest level functions make no assumptions about modules, *etc.*

`\msg_generic_new:nnn` Creating a new message is basically the same as the non-checking version, and so after a
`\msg_generic_new:nn` check everything hands over.

```

5707 \cs_new_protected_nopar:Npn \msg_generic_new:nnn #1 {
5708   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5709   \msg_generic_set:nnn {#1}
5710 }
5711 \cs_new_protected_nopar:Npn \msg_generic_new:nn #1 {
5712   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5713   \msg_generic_set:nn {#1}
5714 }

```

`\msg_generic_set:nnn` Creating a message is quite simple. There must be a short text part, while the longer
`\msg_generic_set:nn` text may or may not be available.

`\msg_generic_set_clist:n`

```

5715 \cs_new_protected_nopar:Npn \msg_generic_set:nnn #1#2#3 {
5716   \msg_generic_set_clist:n {#1}
5717   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5718   \cs_set:cpn { \c_msg_more_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#3}
5719 }
5720 \cs_new_protected_nopar:Npn \msg_generic_set:nn #1#2 {
5721   \msg_generic_set_clist:n {#1}
5722   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5723   \cs_set_eq:cN { \c_msg_more_text_prefix_tl #1 :xxxx } \c_undefined
5724 }
5725 \cs_new_protected_nopar:Npn \msg_generic_set_clist:n #1 {
5726   \clist_if_in:NnF \l_msg_names_clist { // #1 / } {
5727     \clist_put_right:Nn \l_msg_names_clist { // #1 / }
5728   }
5729 }

```

`\msg_direct_interrupt:xxxxx` The low-level interruption macro is rather opaque, unfortunately. The idea here is to
`\msg_direct_interrupt:n` create a message which hides all of TeX's own information by filling the output up with

dots. To achieve this, dots have to be letters. The odd `\c_msg_hide_tl<dots>` actually does the hiding: it is the large run of dots in the name that is important here. The meaning of `\\` is altered so that the explanation text is a simple run whilst the initial error has line-continuation shown.

```

5730 \group_begin:
5731 \char_set_lccode:nn {'\&} {'\ } % {
5732 \char_set_lccode:w '\} = '\ \scan_stop:
5733 \char_set_lccode:w '\& = '\!\scan_stop:
5734 \char_make_active:N \&
5735 \char_make_letter:N \.
5736 \tl_to_lowercase:n{
5737 \group_end:
5738 \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 {
5739   \group_begin:
5740     \cs_set_nopar:Npn \\ {\msg_newline:}
5741     \cs_set_eq:NN \ \c_space_tl
5742     \tl_set:Nx \l_tmpa_tl {#5}
5743     \tl_set:Nx \l_tmpb_tl {\c_msg_no_info_text_tl}
5744     \msg_direct_interrupt_aux:n {#5}
5745     \tex_errhelp:D \l_msg_tmp_tl
5746     \cs_set_nopar:Npn \\ {\msg_newline: !~#3}
5747     \iow_term:x
5748       {\msg_newline: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!}
5749     \cs_set:Npn & {
5750       \tex_errmessage:D{
5751         \\ #1 \\ \\ #2 \\ \\ #4
5752         \tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
5753           { . \\ \c_msg_help_text_tl }
5754         \c_msg_hide_tl.....
5755       }
5756     }
5757   &
5758   \group_end:
5759 }
5760 }

5761 \cs_new_protected:Npn \msg_direct_interrupt_aux:n #1 {
5762   \cs_set_nopar:Npn \\ {\msg_newline: |~}
5763   \tl_if_empty:nTF {#1} {
5764     \tl_set:Nx \l_msg_tmp_tl { { \c_msg_no_info_text_tl } }
5765   }{
5766     \tl_set:Nx \l_msg_tmp_tl { {
5767       |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\
5768       #1^^J
5769       |.....^J
5770     } }
5771   }
5772 }

```


`\msg_direct_log:xx` Printing to the log or terminal without a stop is rather easier.
`\msg_direct_term:xx`

```

5773 \cs_new_protected:Npn \msg_direct_log:xx #1#2 {
5774   \group_begin:
5775     \cs_set:Npn \ { \msg_newline: #2 }
5776     \cs_set_eq:NN \ \c_space_tl
5777     \iow_log:x { #1 \msg_newline: }
5778   \group_end:
5779 }
5780 \cs_new_protected:Npn \msg_direct_term:xx #1#2 {
5781   \group_begin:
5782     \cs_set:Npn \ { \msg_newline: #2 }
5783     \cs_set_eq:NN \ \c_space_tl
5784     \iow_term:x { #1 \msg_newline: }
5785   \group_end:
5786 }

```

110.4 General functions

The main functions for messaging are built around the separation of module from the message name. These have short names as they will be widely used.

`\msg_new:nnnn` For making messages: all aliases.
`\msg_new:nnn`
`\msg_set:nnnn`
`\msg_set:nnn`

```

5787 \cs_new_protected_nopar:Npn \msg_new:nnnn #1#2 {
5788   \msg_generic_new:nnn { #1 / #2 }
5789 }
5790 \cs_new_protected_nopar:Npn \msg_new:nnn #1#2 {
5791   \msg_generic_new:nn { #1 / #2 }
5792 }
5793 \cs_new_protected_nopar:Npn \msg_set:nnnn #1#2 {
5794   \msg_generic_set:nnn { #1 / #2 }
5795 }
5796 \cs_new_protected_nopar:Npn \msg_set:nnn #1#2 {
5797   \msg_generic_set:nn { #1 / #2 }
5798 }

```

`\msg_class_new:nn` Creating a new class produces three new functions, with varying numbers of arguments.
`\msg_class_set:nn` The `\msg_class_loop:n` function is set up so that redirection will work as desired.

```

5799 \cs_new_protected_nopar:Npn \msg_class_new:nn #1 {
5800   \chk_if_free_cs:c { msg_ #1 :nnxxxx }
5801   \prop_new:c { l_msg_redirect_ #1 _prop }
5802   \msg_class_set:nn {#1}
5803 }
5804 \cs_new_protected_nopar:Npn \msg_class_set:nn #1#2 {
5805   \prop_clear:c { l_msg_redirect_ #1 _prop }
5806   \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6 {

```

```

5807     \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6}
5808 }
5809 \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5 {
5810     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { }
5811 }
5812 \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4 {
5813     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { }
5814 }
5815 \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3 {
5816     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
5817 }
5818 \cs_set_protected:cpx { msg_ #1 :nn } ##1##2 {
5819     \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
5820 }
5821 }

```

`\msg_use:nnnnxxxx` The main message-using macro creates two auxiliary functions: one containing the code for the message, and the second a loop function. There is then a hand-off to the system for checking if redirection is needed.

```

5822 \cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8 {
5823     \cs_set_nopar:Npn \msg_use_code: {
5824         \clist_clear:N \l_msg_redirect_classes_clist
5825         #2
5826     }
5827     \cs_set:Npn \msg_use_loop:n ##1 {
5828         \clist_if_in:NnTF \l_msg_redirect_classes_clist {#1} {
5829             \msg_kernel_error:nn { msg } { redirect-loop } {#1}
5830         }{
5831             \clist_put_right:Nn \l_msg_redirect_classes_clist {#1}
5832             \cs_if_exist:cTF { msg_ ##1 :nnxxxx } {
5833                 \use:c { msg_ ##1 :nnxxxx } {#3} {#4} {#5} {#6} {#7} {#8}
5834             }{
5835                 \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
5836             }
5837         }
5838     }
5839     \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 :xxxx } {
5840         \msg_use_aux:nnn {#1} {#3} {#4}
5841     }{
5842         \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4}
5843     }
5844 }

```

`\msg_use_code:` Blank definitions are initially created for these functions.

`\msg_use_loop:`

```

5845 \cs_new_nopar:Npn \msg_use_code: { }
5846 \cs_new:Npn \msg_use_loop:n #1 { }

```

`\msg_use_aux:nnn` The first auxiliary macro looks for a match by name: the most restrictive check.

```

5847 \cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3 {
5848   \tl_set:Nn \l_msg_current_class_tl {#1}
5849   \tl_set:Nn \l_msg_current_module_tl {#2}
5850   \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / } {
5851     \msg_use_loop_check:nn { names } { // #2 / #3 / }
5852   }{
5853     \msg_use_aux:nn {#1} {#2}
5854   }
5855 }

```

`\msg_use_aux:nn` The second function checks for general matches by module or for all modules.

```

5856 \cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2 {
5857   \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2} {
5858     \msg_use_loop_check:nn {#1} {#2}
5859   }{
5860     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * } {
5861       \msg_use_loop_check:nn {#1} { * }
5862     }{
5863       \msg_use_code:
5864     }
5865   }
5866 }

```

`\msg_use_loop_check:nn` When checking whether to loop, the same code is needed in a few places.

```

5867 \cs_new_protected:Npn \msg_use_loop_check:nn #1#2 {
5868   \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
5869   \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl {
5870     \msg_use_code:
5871   }{
5872     \msg_use_loop:n { \l_msg_class_tl }
5873   }
5874 }

```

`\msg_fatal:nnxxxx` For fatal errors, after the error message TeX bails out.

```

\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn
5875 \msg_class_new:nn { fatal } {
5876   \msg_direct_interrupt:xxxxx
5877   { \msg_fatal_text:n {#1} : ~ "#2" }
5878   {
5879     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5880   }
5881   {}
5882   { \msg_see_documentation_text:n {#1} }
5883   { \c_msg_fatal_text_tl }
5884   \tex_end:D
5885 }

```

`\msg_error:nnxxxx` For an error, the interrupt routine is called, then any recovery code is tried.

```
\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn

5886 \msg_class_new:nn { error } {
5887   \msg_direct_interrupt:xxxxx
5888   { #1~ \c_msg_error_tl : ~ "#2" }
5889   {
5890     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5891   }
5892   {}
5893   { \msg_see_documentation_text:n {#1} }
5894   {
5895     \cs_if_exist:cTF { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
5896     {
5897       \use:c { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
5898       {#3} {#4} {#5} {#6}
5899     }
5900     { \c_msg_no_info_text_tl }
5901   }
5902 }
```

`\msg_warning:nnxxxx` Warnings are printed to the terminal.

```
\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

5903 \msg_class_new:nn { warning } {
5904   \msg_direct_term:xx {
5905     \c_space_tl #1 ~ \c_msg_warning_tl :~
5906     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5907   }
5908   { ( #1 ) \c_space_tl \c_space_tl }
5909 }
```

`\msg_info:nnxxxx` Information only goes into the log.

```
\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

5910 \msg_class_new:nn { info } {
5911   \msg_direct_log:xx {
5912     \c_space_tl #1~\c_msg_info_tl :~
5913     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5914   }
5915   { ( #1 ) \c_space_tl \c_space_tl }
5916 }
```

`\msg_log:nnxxxx` “Log” data is very similar to information, but with no extras added.

```
\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn

5917 \msg_class_new:nn { log } {
5918   \msg_direct_log:xx {
5919     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5920   }
5921   { }
5922 }
```

```

\msg_trace:nnxxxx Trace data is the same as log data, more or less
\msg_trace:nnxxx
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn
5923 \msg_class_new:nn { trace } {
5924   \msg_direct_log:xx {
5925     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
5926   }
5927   { }
5928 }

```

```

\msg_none:nnxxxx The none message type is needed so that input can be gobbled.
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn
5929 \msg_class_new:nn { none } { }

```

110.5 Redirection functions

```

\msg_redirect_class:nn Converts class one into class two.
5930 \cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2 {
5931   \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2}
5932 }

```

```

\msg_redirect_module:nnn For when all messages of a class should be altered for a given module.
5933 \cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3 {
5934   \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3}
5935 }

```

```

\msg_redirect_name:nnn Named message will always use the given class.
5936 \cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3 {
5937   \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3}
5938 }

```

110.6 Kernel-specific functions

```

\msg_kernel_new:nnnn The kernel needs some messages of its own. These are created using pre-built functions.
\msg_kernel_new:nnn Two functions are provided: one more general and one which only has the short text
\msg_kernel_set:nnnn part.
\msg_kernel_set:nnn

```

```

5939 \cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2 {
5940   \msg_new:nnnn { LaTeX } { #1 / #2 }
5941 }
5942 \cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2 {
5943   \msg_new:nnn { LaTeX } { #1 / #2 }
5944 }
5945 \cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2 {
5946   \msg_set:nnnn { LaTeX } { #1 / #2 }

```

```

5947 }
5948 \cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2 {
5949   \msg_set:nnn { LaTeX } { #1 / #2 }
5950 }

```

\msg_kernel_classes_new:n Quickly make the fewer-arguments versions.

```

5951 \cs_new_protected_nopar:Npn \msg_kernel_classes_new:n #1 {
5952   \cs_new_protected:cpx { msg_kernel_ #1 :nnxxx } ##1##2##3##4##5
5953   {
5954     \exp_not:c { msg_kernel_ #1 :nnxxxx }
5955     {##1} {##2} {##3} {##4} {##5} { }
5956   }
5957   \cs_new_protected:cpx { msg_kernel_ #1 :nnxx } ##1##2##3##4
5958   {
5959     \exp_not:c { msg_kernel_ #1 :nnxxxx }
5960     {##1} {##2} {##3} {##4} { } { }
5961   }
5962   \cs_new_protected:cpx { msg_kernel_ #1 :nnx } ##1##2##3
5963   {
5964     \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
5965   }
5966   \cs_new_protected:cpx { msg_kernel_ #1 :nn } ##1##2
5967   {
5968     \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
5969   }
5970 }

```

\msg_kernel_fatal:nnxxxx Fatal kernel errors cannot be re-defined.

```

\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxx
\msg_kernel_fatal:nnx
\msg_kernel_fatal:nn
5971 \cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6 {
5972   \msg_direct_interrupt:xxxxx
5973   { \msg_fatal_text:n {LaTeX} }
5974   {
5975     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5976     {#3} {#4} {#5} {#6}
5977   }
5978   {}
5979   { \msg_see_documentation_text:n {LaTeX3} }
5980   { \c_msg_fatal_text_tl }
5981   \tex_end:D
5982 }
5983 \msg_kernel_classes_new:n { fatal }

```

\msg_kernel_error:nnxxxx Neither can kernel errors.

```

\msg_kernel_error:nnxxx
\msg_kernel_error:nnxxx
\msg_kernel_error:nnxx
\msg_kernel_error:nnx
\msg_kernel_error:nn
5984 \cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6 {
5985   \msg_direct_interrupt:xxxxx
5986   { LaTeX~\c_msg_error_tl \c_space_tl "#2" }

```

```

5987 {
5988   \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5989   {#3} {#4} {#5} {#6}
5990 }
5991 {}
5992 { \msg_see_documentation_text:n {LaTeX3} }
5993 {
5994   \cs_if_exist:cTF
5995     { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5996     {
5997       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
5998       {#3} {#4} {#5} {#6}
5999     }
6000     { \c_msg_no_info_text_tl }
6001 }
6002 }
6003 \msg_kernel_classes_new:n { error }

```

```

\msg_kernel_warning:nnxxxx
\msg_kernel_warning:nnxxx
\msg_kernel_warning:nnxx
\msg_kernel_warning:nnx
\msg_kernel_warning:nn
\msg_kernel_info:nnxxxx
\msg_kernel_info:nnxxx
\msg_kernel_info:nnxx
\msg_kernel_info:nnx
\msg_kernel_info:nn

```

Life is much more simple for warnings and information messages, as these are just shortcuts to the standard classes.

```

6004 \cs_new_protected_nopar:Npn \msg_kernel_warning:nnxxxx #1#2 {
6005   \msg_warning:nnxxxx { LaTeX } { #1 / #2 }
6006 }
6007 \msg_kernel_classes_new:n { warning }
6008 \cs_new_protected_nopar:Npn \msg_kernel_info:nnxxxx #1#2 {
6009   \msg_info:nnxxxx { LaTeX } { #1 / #2 }
6010 }
6011 \msg_kernel_classes_new:n { info }

```

Error messages needed to actually implement the message system itself.

```

6012 \msg_kernel_new:nnnn { msg } { message-unknown }
6013 { Unknown~message~'#2'~for~module~'#1'. }
6014 {
6015   \c_msg_coding_error_text_tl
6016   LaTeX~was~asked~to~display~a~message~called~'#2'\
6017   by~the~module~'#1'~module:~this~message~does~not~exist.
6018   \c_msg_return_text_tl
6019 }
6020 \msg_kernel_new:nnnn { msg } { message-class-unknown }
6021 { Unknown~message~class~'#1'. }
6022 {
6023   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
6024   this~was~never~defined.
6025 }
6026 \c_msg_return_text_tl
6027 }
6028 \msg_kernel_new:nnnn { msg } { redirect-loop }

```

```

6029 { Message~redirection~loop~for~message~class~'#1'. }
6030 {
6031   LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\\
6032   The~original~message~here~has~been~lost.
6033   \c_msg_return_text_tl
6034 }

```

`\msg_kernel_bug:x` The L^AT_EX coding bug error gets re-visited here.

```

6035 \cs_set_protected:Npn \msg_kernel_bug:x #1 {
6036   \msg_direct_interrupt:xxxxx
6037   { \c_msg_kernel_bug_text_tl }
6038   { #1 }
6039   {}
6040   { \msg_see_documentation_text:n {LaTeX3} }
6041   { \c_msg_kernel_bug_more_text_tl }
6042 }

6043 </initex | package>

```

111 l3box implementation

Announce and ensure that the required packages are loaded.

```

6044 <*package>
6045 \ProvidesExplPackage
6046   {\filename}{\filedate}{\fileversion}{\filedescription}
6047   \package_check_loaded_expl:
6048 </package>
6049 <*initex | package>

```

The code in this module is very straight forward so I'm not going to comment it very extensively.

111.1 Generic boxes

`\box_new:N` Defining a new `<box>` register.
`\box_new:c`

```

6050 <*initex>
6051 \alloc_new:nnnN {box} \c_zero \c_max_register_int \tex_mathchardef:D

```

Now, remember that `\box255` has a special role in T_EX, it shouldn't be allocated...

```

6052 \seq_put_right:Nn \g_box_allocation_seq {255}
6053 </initex>

```


When we run on top of L^AT_EX, we just use its allocation mechanism.

```

6054 <*package>
6055 \cs_new_protected:Npn \box_new:N #1 {
6056   \chk_if_free_cs:N #1
6057   \newbox #1
6058 }
6059 </package>

6060 \cs_generate_variant:Nn \box_new:N {c}

```

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

\if_hbox:N
\if_vbox:N
\if_box_empty:N
6061 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
6062 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
6063 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical:NTF
\box_if_vertical:cTF
6064 \prg_new_conditional:Nnn \box_if_horizontal:N {p,TF,T,F} {
6065   \tex_ifhbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6066 }
6067 \prg_new_conditional:Nnn \box_if_vertical:N {p,TF,T,F} {
6068   \tex_ifvbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6069 }
6070 \cs_generate_variant:Nn \box_if_horizontal_p:N {c}
6071 \cs_generate_variant:Nn \box_if_horizontal:N {NTF} {c}
6072 \cs_generate_variant:Nn \box_if_horizontal:NT {c}
6073 \cs_generate_variant:Nn \box_if_horizontal:NF {c}
6074 \cs_generate_variant:Nn \box_if_vertical_p:N {c}
6075 \cs_generate_variant:Nn \box_if_vertical:N {NTF} {c}
6076 \cs_generate_variant:Nn \box_if_vertical:NT {c}
6077 \cs_generate_variant:Nn \box_if_vertical:NF {c}

```

Testing if a $\langle box \rangle$ is empty/void.

```

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF
6078 \prg_new_conditional:Nnn \box_if_empty:N {p,TF,T,F} {
6079   \tex_ifvoid:D #1 \prg_return_true: \else: \prg_return_false: \fi:
6080 }
6081 \cs_generate_variant:Nn \box_if_empty_p:N {c}
6082 \cs_generate_variant:Nn \box_if_empty:N {NTF} {c}
6083 \cs_generate_variant:Nn \box_if_empty:NT {c}
6084 \cs_generate_variant:Nn \box_if_empty:NF {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
6085 \cs_new_protected_nopar:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_copy:D #2}
6086 \cs_generate_variant:Nn \box_set_eq:NN {cN,Nc,cc}

```

<pre> \box_set_eq_clear:NN \box_set_eq_clear:cN \box_set_eq_clear:Nc \box_set_eq_clear:cc </pre>	<p>Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).</p> <pre> 6087 \cs_new_protected_nopar:Npn \box_set_eq_clear:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2} 6088 \cs_generate_variant:Nn \box_set_eq_clear:NN {cN,Nc,cc} </pre>
<pre> \box_gset_eq:NN \box_gset_eq:cN \box_gset_eq:Nc \box_gset_eq:cc \box_gset_eq_clear:NN \box_gset_eq_clear:cN \box_gset_eq_clear:Nc \box_gset_eq_clear:cc </pre>	<p>Global version of the above.</p> <pre> 6089 \cs_new_protected_nopar:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN} 6090 \cs_generate_variant:Nn \box_gset_eq:NN {cN,Nc,cc} 6091 \cs_new_protected_nopar:Npn \box_gset_eq_clear:NN {\pref_global:D\box_set_eq_clear:NN} 6092 \cs_generate_variant:Nn \box_gset_eq_clear:NN {cN,Nc,cc} </pre>
<pre> \l_last_box \box_gset_eq_clear:cc </pre>	<p>A different name for this read-only primitive.</p> <pre> 6093 \cs_new_eq:NN \l_last_box \tex_lastbox:D </pre>
<pre> \box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c </pre>	<p>Set a box to the previous box.</p> <pre> 6094 \cs_new_protected_nopar:Npn \box_set_to_last:N #1{\tex_setbox:D#1\l_last_box} 6095 \cs_generate_variant:Nn \box_set_to_last:N {c} 6096 \cs_new_protected_nopar:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N} 6097 \cs_generate_variant:Nn \box_gset_to_last:N {c} </pre>
<pre> \box_move_left:nn \box_move_right:nn \box_move_up:nn \box_move_down:nn </pre>	<p>Move box material in different directions.</p> <pre> 6098 \cs_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1} #2} 6099 \cs_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1} #2} 6100 \cs_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1} #2} 6101 \cs_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1} #2} </pre>
<pre> \box_clear:N \box_clear:c \box_gclear:N \box_gclear:c </pre>	<p>Clear a $\langle box \rangle$ register.</p> <pre> 6102 \cs_new_protected_nopar:Npn \box_clear:N #1{\box_set_eq_clear:NN #1 \c_empty_box } 6103 \cs_generate_variant:Nn \box_clear:N {c} 6104 \cs_new_protected_nopar:Npn \box_gclear:N {\pref_global:D\box_clear:N} 6105 \cs_generate_variant:Nn \box_gclear:N {c} </pre>
<pre> \box_ht:N \box_ht:c \box_dp:N \box_dp:c \box_wd:N \box_wd:c </pre>	<p>Accessing the height, depth, and width of a $\langle box \rangle$ register.</p> <pre> 6106 \cs_new_eq:NN \box_ht:N \tex_ht:D 6107 \cs_new_eq:NN \box_dp:N \tex_dp:D 6108 \cs_new_eq:NN \box_wd:N \tex_wd:D 6109 \cs_generate_variant:Nn \box_ht:N {c} 6110 \cs_generate_variant:Nn \box_dp:N {c} 6111 \cs_generate_variant:Nn \box_wd:N {c} </pre>

`\box_set_ht:Nn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived
`\box_set_ht:cn` functions are not either.

`\box_set_dp:Nn`
`\box_set_dp:cn`
`\box_set_wd:Nn`
`\box_set_wd:cn`

```

6112 \cs_new_protected_nopar:Npn \box_set_dp:Nn #1#2 {
6113   \box_dp:N #1 \etex_dimexpr:D #2 \scan_stop:
6114 }
6115 \cs_new_protected_nopar:Npn \box_set_ht:Nn #1#2 {
6116   \box_ht:N #1 \etex_dimexpr:D #2 \scan_stop:
6117 }
6118 \cs_new_protected_nopar:Npn \box_set_wd:Nn #1#2 {
6119   \box_wd:N #1 \etex_dimexpr:D #2 \scan_stop:
6120 }
6121 \cs_generate_variant:Nn \box_set_ht:Nn { c }
6122 \cs_generate_variant:Nn \box_set_dp:Nn { c }
6123 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

`\box_use_clear:N` Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.

`\box_use_clear:c`
`\box_use:N`
`\box_use:c`

```

6124 \cs_new_eq:NN \box_use_clear:N \tex_box:D
6125 \cs_generate_variant:Nn \box_use_clear:N { c }
6126 \cs_new_eq:NN \box_use:N \tex_copy:D
6127 \cs_generate_variant:Nn \box_use:N { c }

```

`\box_show:N` Show the contents of a box and write it into the log file.

`\box_show:c`

```

6128 \cs_set_eq:NN \box_show:N \tex_showbox:D
6129 \cs_generate_variant:Nn \box_show:N { c }

```

`\c_empty_box` We allocate some $\langle box \rangle$ registers here (and borrow a few from L^AT_EX).

`\l_tmpa_box`
`\l_tmpb_box`

```

6130 <package>\cs_set_eq:NN \c_empty_box \voidb@x
6131 <package>\cs_new_eq:NN \l_tmpa_box \@tempboxa
6132 <initex>\box_new:N \c_empty_box
6133 <initex>\box_new:N \l_tmpa_box
6134 \box_new:N \l_tmpb_box

```

111.2 Vertical boxes

`\vbox:n` Put a vertical box directly into the input stream.

`\vbox_top:n`

```

6135 \cs_new_protected_nopar:Npn \vbox:n {\tex_vbox:D \scan_stop:}
6136 \cs_new_protected_nopar:Npn \vbox_top:n {\tex_vtop:D \scan_stop:}

```

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

```

6137 \cs_new_protected:Npn \vbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_vbox:D {#2}}
6138 \cs_generate_variant:Nn \vbox_set:Nn { cn }
6139 \cs_new_protected_nopar:Npn \vbox_gset:Nn {\pref_global:D \vbox_set:Nn}
6140 \cs_generate_variant:Nn \vbox_gset:Nn { cn }

```

<code>\vbox_set_top:Nn</code>	Storing material in a vertical box with a natural height and reference point at the baseline
<code>\vbox_set_top:cn</code>	of the first object in the box.
<code>\vbox_gset_top:Nn</code>	
<code>\vbox_gset_top:cn</code>	<pre> 6141 \cs_new_protected:Npn \vbox_set_top:Nn #1#2 {\tex_setbox:D #1 \tex_vtop:D {#2}} 6142 \cs_generate_variant:Nn \vbox_set_top:Nn {cn} 6143 \cs_new_protected_nopar:Npn \vbox_gset_top:Nn {\pref_global:D \vbox_set_top:Nn} 6144 \cs_generate_variant:Nn \vbox_gset_top:Nn {cn} </pre>
<code>\vbox_set_to_ht:Nnn</code>	Storing material in a vertical box with a specified height.
<code>\vbox_set_to_ht:cnn</code>	
<code>\vbox_gset_to_ht:Nnn</code>	
<code>\vbox_gset_to_ht:cnn</code>	
<code>\vbox_gset_to_ht:ccn</code>	<pre> 6145 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3 { 6146 \tex_setbox:D #1 \tex_vbox:D to #2 {#3} 6147 } 6148 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn {cnn} 6149 \cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn {\pref_global:D \vbox_set_to_ht:Nnn} 6150 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn {cnn,ccn} </pre>
<code>\vbox_set_inline_begin:N</code>	Storing material in a vertical box. This type is useful in environment definitions.
<code>\vbox_set_inline_end:</code>	
<code>\vbox_gset_inline_begin:N</code>	
<code>\vbox_gset_inline_end:</code>	<pre> 6151 \cs_new_protected_nopar:Npn \vbox_set_inline_begin:N #1 { 6152 \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token } 6153 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token 6154 \cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N { 6155 \pref_global:D \vbox_set_inline_begin:N } 6156 \cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token </pre>
<code>\vbox_to_ht:nn</code>	Put a vertical box directly into the input stream.
<code>\vbox_to_zero:n</code>	
	<pre> 6157 \cs_new_protected:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}} 6158 \cs_new_protected:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}} </pre>
<code>\vbox_set_split_to_ht:Nnn</code>	Splitting a vertical box in two.
	<pre> 6159 \cs_new_protected_nopar:Npn \vbox_set_split_to_ht:Nnn #1#2#3{ 6160 \tex_setbox:D #1 \tex_vsplit:D #2 to #3 6161 } </pre>
<code>\vbox_unpack:N</code>	Unpacking a box and if requested also clear it.
<code>\vbox_unpack:c</code>	
<code>\vbox_unpack_clear:N</code>	
<code>\vbox_unpack_clear:c</code>	<pre> 6162 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D 6163 \cs_generate_variant:Nn \vbox_unpack:N {c} 6164 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D 6165 \cs_generate_variant:Nn \vbox_unpack_clear:N {c} </pre>

111.3 Horizontal boxes

`\hbox:n` Put a horizontal box directly into the input stream.

```

6166 \cs_new_protected_nopar:Npn \hbox:n {\tex_hbox:D \scan_stop:}

```

<code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code>	<p>Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).</p> <pre> 6167 \cs_new_protected:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}} 6168 \cs_generate_variant:Nn \hbox_set:Nn {cn} 6169 \cs_new_protected_nopar:Npn \hbox_gset:Nn {\pref_global:D \hbox_set:Nn} 6170 \cs_generate_variant:Nn \hbox_gset:Nn {cn} </pre>
<code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code>	<p>Storing material in a horizontal box with a specified width.</p> <pre> 6171 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3 { 6172 \tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3} 6173 } 6174 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn {cnn} 6175 \cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn } 6176 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn} </pre>
<code>\hbox_set_inline_begin:N</code> <code>\hbox_set_inline_begin:c</code> <code>\hbox_set_inline_end:</code> <code>\hbox_gset_inline_begin:N</code> <code>\hbox_gset_inline_begin:c</code> <code>\hbox_gset_inline_end:</code>	<p>Storing material in a horizontal box. This type is useful in environment definitions.</p> <pre> 6177 \cs_new_protected_nopar:Npn \hbox_set_inline_begin:N #1 { 6178 \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token 6179 } 6180 \cs_generate_variant:Nn \hbox_set_inline_begin:N {c} 6181 \cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token 6182 \cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N { 6183 \pref_global:D \hbox_set_inline_begin:N 6184 } 6185 \cs_generate_variant:Nn \hbox_gset_inline_begin:N {c} 6186 \cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token </pre>
<code>\hbox_to_wd:nn</code> <code>\hbox_to_zero:n</code>	<p>Put a horizontal box directly into the input stream.</p> <pre> 6187 \cs_new_protected:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}} 6188 \cs_new_protected:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}} </pre>
<code>\hbox_overlap_left:n</code> <code>\hbox_overlap_right:n</code>	<p>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.</p> <pre> 6189 \cs_new_protected:Npn \hbox_overlap_left:n #1 {\hbox_to_zero:n {\tex_hss:D #1}} 6190 \cs_new_protected:Npn \hbox_overlap_right:n #1 {\hbox_to_zero:n {#1 \tex_hss:D}} </pre>
<code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code>	<p>Unpacking a box and if requested also clear it.</p> <pre> 6191 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D 6192 \cs_generate_variant:Nn \hbox_unpack:N {c} 6193 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D 6194 \cs_generate_variant:Nn \hbox_unpack_clear:N {c} 6195 </initex package> 6196 <*showmemory> 6197 \showMemUsage 6198 </showmemory> </pre>

112 l3xref implementation

112.1 Internal functions and variables

<code>\g_xref_all_curr_immediate_fields_prop</code> <code>\g_xref_all_curr_deferred_fields_prop</code>	What they say they are :)
---	---------------------------

<code>\xref_write</code>	A stream for writing cross references, although they are not required to be in a separate file.
--------------------------	---

<code>\xref_define_label:nn</code>	<code>\xref_define_label:nn {<name>} {<plist contents>}</code>
------------------------------------	--

Define the property list for each label; used internally by `\xref_set_label:n`.

112.2 Module code

We start by ensuring that the required packages are loaded.

```

6199 <*package>
6200 \ProvidesExplPackage
6201   {\filename}{\filedate}{\fileversion}{\filedescription}
6202 \package_check_loaded_expl:
6203 </package>
6204 <*initex | package>

```

There are two kinds of information, namely information which is *immediate* like a section title and then there's *deferred* information like page numbers. Each reference type belong to one of these categories, which we save internally as the property lists `\g_xref_all_curr_immediate_fields_prop` and `\g_xref_all_curr_deferred_fields_prop` and the reference type `<xyz>` exists as the key-info pair `\xref_<xyz>_key {\l_xref_curr_<xyz>_tl}` on one of these lists. This way each new entry type is just added as another key-info pair.

When the cross references are generated at the beginning of the document each will turn into a control sequence. Thus `\label{mylab}` will internally refer to the property list `\g_xref_mylab_prop`.

The extraction of values from this property list can be done in several different ways but we want to keep the operation expandable. Therefore we use a dedicated function for each type of cross reference, which looks like this:

```
\xref_get_value_xyz_aux:w -> #1 \xref_xyz_key #2#3\q_nil{#2}
```

This will throw away all the bits we don't need. In case `xyz` is the first on the `mylab` property list `#1` is empty, if it's the last key-info pair `#3` is empty. The value of the field can be extracted with the function `\xref_get_value:nn` where the first argument is the type and the second the label name so here it would be `\xref_get_value:nn {xyz} {mylab}`.

`_all_curr_immediate_fields_prop` `_all_curr_deferred_fields_prop` The two main property lists for storing information. They contain key-info pairs for all known types.

```
6205 \prop_new:N \g_xref_all_curr_immediate_fields_prop
6206 \prop_new:N \g_xref_all_curr_deferred_fields_prop
```

`\xref_new:nn` `\xref_deferred_new:nn` `\xref_new_aux:nnn` Setting up a new cross reference type is fairly straight forward when we follow the game plan mentioned earlier.

```
6207 \cs_new_nopar:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
6208 \cs_new_nopar:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
6209 \cs_new_nopar:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
6210 \prop_gput:ccx {g_xref_all_curr_ #1 _fields_prop}
6211 { xref_ #2 _key }
6212 { \exp_not:c {l_xref_curr_#2_tl } }
```

Then define the key to be a protected macro.¹²

```
6213 \cs_set_protected_nopar:cpn { xref_#2_key }{ }
6214 \tl_new:cn{l_xref_curr_#2_tl}{#3}
```

Now for the function extracting the value of a reference. We could do this with a simple `\prop_if_in` thing put since we want to do things in an expandable way we make a separate grabber for each type—this is also faster. The grabber function can be defined by using an intricate construction of `\exp_after:wN` and other goodies but I prefer readable code. The end result for the input `xyz` is

```
\cs_set_nopar:Npn\xref_get_value_xyz_aux:w #1\xref_xyz_key #2#3\q_nil{#2}
```

```
6215 \toks_set:Nx \l_tmpa_toks {
6216   \exp_not:n { \cs_set_nopar:cpn {xref_get_value_#2_aux:w} ##1 }
6217   \exp_not:N \q_prop
6218   \exp_not:c { xref_#2_key }
6219   \exp_not:N \q_prop
6220 }
6221 \toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
6222 }
```

¹²We could also set it equal to `\scan_stop:` but this just feels “cleaner”.

`\xref_get_value:nn` Getting the correct value for a given label-type pair is a matter of connecting the correct grabber functions and property list.

```

6223 \cs_new_nopar:Npn \xref_get_value:nn #1#2 {
6224   \cs_if_exist:CTF{g_xref_#2_prop}
6225   {

```

This next expansion may look a little weird but it isn't if you think about it!

```

6226     \exp_args:NcNc \exp_after:wN {xref_get_value_#1_aux:w}
6227     \toks_use:N {g_xref_#2_prop}

```

Better put in the stop marker.

```

6228     \q_nil
6229   }
6230   {??}
6231 }

```

Temporary! We expand the property list and so we can't have the `\q_prop` marker just expand!

```

6232 \cs_set_nopar:Npn \exp_after:cc #1#2 {
6233   \exp_after:wN \exp_after:wN
6234   \cs:w #1\exp_after:wN\cs_end: \cs:w #2\cs_end:
6235 }
6236 \cs_set_protected:Npn \q_prop {\q_prop}

```

`\xref_define_label:nn` Define the property list for each label. We better do this in two steps because the special catcode regime is in effect and since some of the info fields are very likely to contain actual text, we better make sure spaces aren't ignored! As for the meaning of other characters then it is a possibility to also have a field containing catcode instructions which can then be activated with `\etex_scantokens:D`.

```

6237 \cs_new_protected_nopar:Npn \xref_define_label:nn {
6238   \group_begin:
6239   \char_set_catcode:nn {'\ }c_ten
6240   \xref_define_label_aux:nn
6241 }

```

If the label is already taken we have a multiply defined label and we should do something about it. For now we don't do anything spectacular.

```

6242 \cs_new_nopar:Npn \xref_define_label_aux:nn #1#2 {
6243   \cs_if_free:CTF{g_xref_#1_prop}
6244   {\prop_new:c{g_xref_#1_prop}}{\WARNING}
6245   \toks_gset:cn{g_xref_#1_prop}{#2}
6246   \group_end:
6247 }

```


`\xref_set_label:n` Then the generic command for setting a label. We expand the immediate labels fully before calling the write function but make sure the deferred fields aren't expanded just yet. Due to property lists being implemented as token list registers we must expand the 'immediate' fields twice.

```

6248 \cs_set_nopar:Npn \xref_set_label:n #1{
6249   \cs_set_nopar:Npx \xref_tmp:w{\toks_use:N\g_xref_all_curr_immediate_fields_prop}
6250   \exp_args:NNx\iow_shipout_x:Nn \xref_write{
6251     \xref_define_label:nn {#1} {
6252       \xref_tmp:w
6253       \toks_use:N \g_xref_all_curr_deferred_fields_prop
6254     }
6255   }
6256 }

```

That's it (for now).

```

6257 </initex | package>

6258 <*showmemory>
6259 \showMemUsage
6260 </showmemory>

```

113 l3xref test file

```

6261 <*testfile>
6262 \documentclass{article}
6263 \usepackage{l3xref}
6264 \ExplSyntaxOn
6265 \cs_set_nopar:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
6266 \cs_set_nopar:Npn \DefineCrossReferences {
6267   \group_begin:
6268     \ExplSyntaxNamesOn
6269     \InputIfFileExists{\jobname.xref}{\{}}
6270   \group_end:
6271 }
6272 \AtBeginDocument{\DefineCrossReferences\startrecording}
6273
6274 \xref_new:nn {name}{\{}}
6275 \cs_set_nopar:Npn \setname{\tl_set:Nn\l_xref_curr_name_tl}
6276 \cs_set_nopar:Npn \getname{\xref_get_value:nn{name}}
6277
6278 \xref_deferred_new:nn {page}{\thepage}
6279 \cs_set_nopar:Npn \getpage{\xref_get_value:nn{page}}
6280
6281 \xref_deferred_new:nn {valuepage}{\number\value{page}}
6282 \cs_set_nopar:Npn \getvaluepage{\xref_get_value:nn{valuepage}}
6283

```

```

6284 \cs_set_eq:NN \setlabel \xref_set_label:n
6285
6286 \ExplSyntaxOff
6287 \begin{document}
6288 \pagenumbering{roman}
6289
6290 Text\setname{This is a name}\setlabel{testlabel1}. More
6291 text\setname{This is another name}\setlabel{testlabel2}. \clearpage
6292
6293 Text\setname{This is a third name}\setlabel{testlabel3}. More
6294 text\setname{Hello World!}\setlabel{testlabel4}. \clearpage
6295
6296 \pagenumbering{arabic}
6297
6298 Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
6299 6}\setlabel{testlabel6}. \clearpage
6300
6301 Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
6302 8}\setlabel{testlabel8}. \clearpage
6303
6304 Now let's extract some values. \getname{testlabel1} on page
6305 \getpage{testlabel1} with value \getvaluepage{testlabel1}.
6306
6307 Now let's extract some values. \getname{testlabel 7} on page
6308 \getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
6309 \end{document}
6310 </testfile>

```

114 l3keyval implementation

\KV_sanitize_outerlevel_active_equals:N \KV_sanitize_outerlevel_active_commas:N	\KV_sanitize_outerlevel_active_equals:N <tl var.>
--	---

Replaces catcode other = and , within a <tl var.> with active characters.

\KV_remove_surrounding_spaces:nw \KV_remove_surrounding_spaces_auxi:w ★	\KV_remove_surrounding_spaces:nw <tl> <token list> \q_nil \KV_remove_surrounding_spaces_auxi:w <token list> \q_3
--	---

Removes a possible leading space plus a possible ending space from a <token list>. The first version (which is not used in the code) stores it in <tl>.

\KV_add_value_element:w \KV_set_key_element:w	\KV_set_key_element:w <token list> \q_nil \KV_add_value_element:w \q_stop <token list> \q_nil
--	--

Specialised functions to strip spaces from their input and set the token registers `\l_KV_currkey_tl` or `\l_KV_currval_tl` respectively.

<pre> \KV_split_key_value_current:w \KV_split_key_value_space_removal:w \KV_split_key_value_space_removal_detect_error:wTF \KV_split_key_value_no_space_removal:w </pre>	<pre> \KV_split_key_value_current:w ... </pre>
--	--

These functions split keyval lists into chunks depending which sanitising method is being used. `\KV_split_key_value_current:w` is `\cs_set_eq:NN` to whichever is appropriate.

114.1 Module code

We start by ensuring that the required packages are loaded.

```

6311 <*package>
6312 \ProvidesExplPackage
6313   {\filename}{\filedate}{\fileversion}{\filedescription}
6314 \package_check_loaded_expl:
6315 </package>
6316 <*initex | package>

```

<pre> \l_KV_tmpa_tl \l_KV_tmpp_tl \c_KV_single_equal_sign_tl </pre>	<p>Various useful things.</p> <pre> 6317 \tl_new:N \l_KV_tmpa_tl 6318 \tl_new:N \l_KV_tmpp_tl 6319 \tl_const:Nn \c_KV_single_equal_sign_tl { = } </pre>
---	---

<pre> \l_KV_parse_tl \l_KV_currkey_tl \l_KV_currval_tl </pre>	<p>Some more useful things.</p> <pre> 6320 \tl_new:N \l_KV_parse_tl 6321 \tl_new:N \l_KV_currkey_tl 6322 \tl_new:N \l_KV_currval_tl </pre>
---	--

<pre> \l_KV_level_int </pre>	<p>This is used to track how deeply nested calls to the keyval processor are, so that the correct functions are always in use.</p> <pre> 6323 \int_new:N \l_KV_level_int </pre>
------------------------------	---

<pre> remove_one_level_of_braces_bool </pre>	<p>A boolean to control</p> <pre> 6324 \bool_new:N \l_KV_remove_one_level_of_braces_bool 6325 \bool_set_true:N \l_KV_remove_one_level_of_braces_bool </pre>
--	---

```

process_space_removal_sanitiz:NNn
s_space_removal_no_sanitiz:NNn
o_space_removal_no_sanitiz:NNn
\KV_process_aux:NNNn

```

The wrapper function takes care of assigning the appropriate `elt` functions before and after the parsing step. In that way there is no danger of a mistake with the wrong functions being used.

```

6326 \cs_new_protected_nopar:Npn \KV_process_space_removal_sanitiz:NNn {
6327   \KV_process_aux:NNNn \KV_parse_space_removal_sanitiz:n
6328 }
6329 \cs_new_protected_nopar:Npn \KV_process_space_removal_no_sanitiz:NNn {
6330   \KV_process_aux:NNNn \KV_parse_space_removal_no_sanitiz:n
6331 }
6332 \cs_new_protected_nopar:Npn \KV_process_no_space_removal_no_sanitiz:NNn {
6333   \KV_process_aux:NNNn \KV_parse_no_space_removal_no_sanitiz:n
6334 }
6335 \cs_new_protected:Npn \KV_process_aux:NNNn #1#2#3#4 {
6336   \cs_set_eq:cN
6337     { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6338     \KV_key_no_value_elt:n
6339   \cs_set_eq:cN
6340     { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6341     \KV_key_value_elt:nn
6342   \cs_set_eq:NN \KV_key_no_value_elt:n #2
6343   \cs_set_eq:NN \KV_key_value_elt:nn #3
6344   \int_incr:N \l_KV_level_int
6345   #1 {#4}
6346   \int_decr:N \l_KV_level_int
6347   \cs_set_eq:Nc \KV_key_no_value_elt:n
6348     { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6349   \cs_set_eq:Nc \KV_key_value_elt:nn
6350     { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6351 }

```

```

size_outerlevel_active_equals:N
size_outerlevel_active_commas:N

```

Some functions for sanitizing top level equals and commas. Replace `=13` and `,13` with `=12` and `,12` resp.

```

6352 \group_begin:
6353 \char_set_catcode:nn{'\}{13}
6354 \char_set_catcode:nn{'\,}{13}
6355 \char_set_lccode:nn{'\8}{'\=}
6356 \char_set_lccode:nn{'\9}{'\,}
6357 \tl_to_lowercase:n{\group_end:
6358 \cs_new_protected_nopar:Npn \KV_sanitiz_outerlevel_active_equals:N #1{
6359   \tl_replace_all_in:Nnn #1 = 8
6360 }
6361 \cs_new_nopar:Npn \KV_sanitiz_outerlevel_active_commas:N #1{
6362   \tl_replace_all_in:Nnn #1 , 9
6363 }
6364 }

```

```

KV_remove_surrounding_spaces:nw
remove_surrounding_spaces_auxi:w
move_surrounding_spaces_auxii:w
\KV_set_key_element:w
\KV_add_value_element:w

```

The macro `\KV_remove_surrounding_spaces:nw` removes a possible leading space plus a possible ending space from its second argument and stores it in the token register `#1`.

Based on Around the Bend No. 15 but with some enhancements. For instance, this definition is purely expandable.

We use a funny token Q_3 as a delimiter.

```

6365 \group_begin:
6366 \char_set_catcode:nn{'\Q}{3}

6367 \cs_new:Npn\KV_remove_surrounding_spaces:nw#1#2\q_nil{

```

The idea in this processing is to use a Q with strange catcode to remove a trailing space. But first, how to get this expansion going?

If you have read the fine print in the `l3expan` module, you'll know that the `f` type expansion will expand until the first non-expandable token is seen and if this token is a space, it will be gobbled. Sounds useful for removing a leading space but we also need to make sure that it does nothing but removing that space! Therefore we prepend the argument to be trimmed with an `\exp_not:N`. Now why is that? `\exp_not:N` in itself is an expandable command so will allow the `f` expansion to continue. If the first token in the argument to be trimmed is a space, it will be gobbled and the expansion stop. If the first token isn't a space, the `\exp_not:N` turns it temporarily into `\scan_stop:` which is unexpandable. The processing stops but the token following directly after `\exp_not:N` is now back to normal.

The function here allows you to insert arbitrary functions in the first argument but they should all be with an `f` type expansion. For the application in this module, we use `\tl_set:Nf`.

Once the expansion has been kick-started, we apply `\KV_remove_surrounding_spaces_auxi:w` to the replacement text of `#2`, adding a leading `\exp_not:N`. Note that no braces are stripped off of the original argument.

```

6368 #1{\KV_remove_surrounding_spaces_auxi:w \exp_not:N#2Q~Q}
6369 }

```

`\KV_remove_surrounding_spaces_auxi:w` removes a trailing space if present, then calls `\KV_remove_surrounding_spaces_auxii:w` to clean up any leftover bizarre Q s. In order for `\KV_remove_surrounding_spaces_auxii:w` to work properly we need to put back a Q first.

```

6370 \cs_new:Npn\KV_remove_surrounding_spaces_auxi:w#1~Q{
6371   \KV_remove_surrounding_spaces_auxii:w #1 Q
6372 }

```

Now all that is left to do is remove a leading space which should be taken care of by the function used to initiate the expansion. Simply return the argument before the funny Q .

```

6373 \cs_new:Npn\KV_remove_surrounding_spaces_auxii:w#1Q#2{#1}

```

Here are some specialized versions of the above. They do exactly what we want in one go. First trim spaces from the value and then put the result surrounded in braces onto `\l_KV_parse_tl`.

```

6374 \cs_new_protected:Npn\KV_add_value_element:w\q_stop#1\q_nil{
6375   \tl_set:Nf\l_KV_currval_tl {
6376     \KV_remove_surrounding_spaces_auxi:w \exp_not:N#1Q~Q
6377   }
6378   \tl_put_right:No\l_KV_parse_tl{
6379     \exp_after:wN { \l_KV_currval_tl }
6380   }
6381 }

```

When storing the key we firstly remove spaces plus the prepended `\q_no_value`.

```

6382 \cs_new_protected:Npn\KV_set_key_element:w#1\q_nil{
6383   \tl_set:Nf\l_KV_currkey_tl
6384   {
6385     \exp_last_unbraced:NNo \KV_remove_surrounding_spaces_auxi:w
6386     \exp_not:N \use_none:n #1Q~Q
6387   }

```

Afterwards we gobble an extra level of braces if that's what we are asked to do.

```

6388   \bool_if:NT \l_KV_remove_one_level_of_braces_bool
6389   {
6390     \exp_args:NNo \tl_set:No \l_KV_currkey_tl {
6391       \exp_after:wN \KV_add_element_aux:w \l_KV_currkey_tl \q_nil
6392     }
6393   }
6394 }
6395 \group_end:

```

`\KV_add_element_aux:w` A helper function for fixing braces around keys and values.

```

6396 \cs_new:Npn \KV_add_element_aux:w#1\q_nil{#1}

```

Parse a list of keyvals, put them into list form with entries like `\KV_key_no_value_elt:n{key1}` and `\KV_key_value_elt:nn{key2}{val2}`.

`\KV_parse_sanitize_aux:n` The slow parsing algorithm sanitizes active commas and equal signs at the top level first. Then uses `#1` as inspector of each element in the comma list.

```

6397 \cs_new_protected:Npn \KV_parse_sanitize_aux:n #1 {
6398   \group_begin:
6399   \tl_clear:N \l_KV_parse_tl
6400   \tl_set:Nn \l_KV_tmpa_tl {#1}
6401   \KV_sanitize_outerlevel_active_equals:N \l_KV_tmpa_tl
6402   \KV_sanitize_outerlevel_active_commas:N \l_KV_tmpa_tl
6403   \exp_last_unbraced:NNV \KV_parse_elt:w \q_no_value
6404   \l_KV_tmpa_tl , \q_nil ,

```

We evaluate the parsed keys and values outside the group so the token register is restored to its previous value.

```

6405 \exp_after:wN \group_end:
6406 \l_KV_parse_tl
6407 }

```

`\KV_parse_no_sanitize_aux:n` Like above but we don't waste time sanitizing. This is probably the one we will use for preamble parsing where catcodes of = and , are as expected!

```

6408 \cs_new_protected:Npn \KV_parse_no_sanitize_aux:n #1{
6409 \group_begin:
6410 \tl_clear:N \l_KV_parse_tl
6411 \KV_parse_elt:w \q_no_value #1 , \q_nil ,
6412 \exp_after:wN \group_end:
6413 \l_KV_parse_tl
6414 }

```

`\KV_parse_elt:w` This function will always have a `\q_no_value` stuffed in as the rightmost token in #1. In case there was a blank entry in the comma separated list we just run it again. The `\use_none:n` makes sure to gobble the quark `\q_no_value`. A similar test is made to check if we hit the end of the recursion.

```

6415 \cs_set:Npn \KV_parse_elt:w #1,{
6416 \tl_if_blank:OTF{\use_none:n #1}
6417 { \KV_parse_elt:w \q_no_value }
6418 {
6419 \quark_if_nil:OF {\use_ii:nn #1 }

```

If we made it to here we can start parsing the key and value. When done try, try again.

```

6420 {
6421 \KV_split_key_value_current:w #1==\q_nil
6422 \KV_parse_elt:w \q_no_value
6423 }
6424 }
6425 }

```

`\KV_split_key_value_current:w` The function called to split the keys and values.

```

6426 \cs_new:Npn \KV_split_key_value_current:w {\ERROR}

```

We provide two functions for splitting keys and values. The reason being that most of the time, we should probably be in the special coding regime where spaces are ignored. Hence it makes no sense to spend time searching for extra space tokens and we can do the settings directly. When comparing these two versions (neither doing any sanitizing) the `no_space_removal` version is more than 40% faster than `space_removal`.

It is up to functions like `\DeclareTemplate` to check which catcode regime is active and then pick up the version best suited for it.

```
split_key_value_space_removal:w
space_removal_detect_error:wTF
split_key_value_space_removal_aux:w
```

The code below removes extraneous spaces around the keys and values plus one set of braces around the entire value.

Unlike the version to be used when spaces are ignored, this one only grabs the key which is everything up to the first = and save the rest for closer inspection. Reason is that if a user has entered `mykey={{myval}}`, then the outer braces have already been removed before we even look at what might come after the key. So this is slightly more tedious (but only slightly) but at least it always removes only one level of braces.

```
6427 \cs_new_protected:Npn \KV_split_key_value_space_removal:w #1 = #2\q_nil{
```

First grab the key.

```
6428 \KV_set_key_element:w#1\q_nil
```

Then we start checking. If only a key was entered, #2 contains = and nothing else, so we test for that first.

```
6429 \tl_set:Nn\l_KV_tmpa_tl{#2}
6430 \tl_if_eq:NNTF\l_KV_tmpa_tl\c_KV_single_equal_sign_tl
```

Then we just insert the default key.

```
6431 {
6432   \tl_put_right:Nn\l_KV_parse_tl{
6433     \exp_after:wN \KV_key_no_value_elt:n
6434     \exp_after:wN {\l_KV_currkey_tl}
6435   }
6436 }
```

Otherwise we must take a closer look at what is left. The remainder of the original list up to the comma is now stored in #2 plus an additional ==, which wasn't gobbled during the initial reading of arguments. If there is an error then we can see at least one more = so we call an auxiliary function to check for this.

```
6437 {
6438   \KV_split_key_value_space_removal_detect_error:wTF#2\q_no_value\q_nil
6439   {\KV_split_key_value_space_removal_aux:w \q_stop #2}
6440   {\msg_kernel_error:nn {keyval} {misplaced-equals-sign} } }
6441 }
6442 }
```

The error test.

```
6443 \cs_new_protected:Npn
6444 \KV_split_key_value_space_removal_detect_error:wTF#1=#2#3\q_nil{
6445   \tl_if_head_eq_meaning:NNTF{#3}\q_no_value
6446 }
```

Now we can start extracting the value. Recall that #1 here starts with `\q_stop` so all braces are still there! First we try to see how much is left if we gobble three brace groups

from #1. If #1 is empty or blank, all three quarks are gobbled. If #1 consists of exactly one token or brace group, only the latter quark is left.

```

6447 \cs_new:Npn \KV_val_preserve_braces:NnN #1#2#3{{#2}}
6448 \cs_new_protected:Npn \KV_split_key_value_space_removal_aux:w #1=={
6449   \tl_set:Nx\l_KV_tmpa_tl{\exp_not:o{\use_none:nnn#1\q_nil\q_nil}}
6450   \tl_put_right:No\l_KV_parse_tl{
6451     \exp_after:wN \KV_key_value_elt:nn
6452     \exp_after:wN {\l_KV_currkey_tl}
6453   }

```

If there a blank space or nothing at all, \l_KV_tmpa_tl is now completely empty.

```

6454   \tl_if_empty:NTF\l_KV_tmpa_tl

```

We just put an empty value on the stack.

```

6455   { \tl_put_right:Nn\l_KV_parse_tl{{}} }
6456   {

```

If there was exactly one brace group or token in #1, \l_KV_tmpa_tl is now equal to \q_nil. Then we can just pick it up as the second argument of #1. This will also take care of any spaces which might surround it.

```

6457   \quark_if_nil:NTF\l_KV_tmpa_tl
6458   {
6459     \bool_if:NTF \l_KV_remove_one_level_of_braces_bool
6460     {
6461       \tl_put_right:No\l_KV_parse_tl{
6462         \exp_after:wN{\use_ii:nnn #1\q_nil}
6463       }
6464     }
6465     {
6466       \tl_put_right:No\l_KV_parse_tl{
6467         \exp_after:wN{\KV_val_preserve_braces:NnN #1\q_nil}
6468       }
6469     }
6470   }

```

Otherwise we grab the value.

```

6471   { \KV_add_value_element:w #1\q_nil }
6472   }
6473 }

```

it_key_value_no_space_removal:w This version is for when in the special coding regime where spaces are ignored so there is no need to do any fancy space hacks, however fun they may be. Since there are no spaces, a set of braces around a value is automatically stripped by T_EX.

```

6474 \cs_new_protected:Npn \KV_split_key_value_no_space_removal:w #1#2=#3=#4\q_nil{
6475   \tl_set:Nn\l_KV_tmpa_tl{#4}

```

```

6476 \tl_if_empty:NTF \l_KV_tmpa_tl
6477 {
6478   \tl_put_right:Nn\l_KV_parse_tl{\KV_key_no_value_elt:n{#2}}
6479 }
6480 {
6481   \tl_if_eq:NNTF\c_KV_single_equal_sign_tl\l_KV_tmpa_tl
6482   {
6483     \tl_put_right:Nn\l_KV_parse_tl{\KV_key_value_elt:nn{#2}{#3}}
6484   }
6485   { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
6486 }
6487 }

```

\KV_key_no_value_elt:n
\KV_key_value_elt:nn

```

6488 \cs_new:Npn \KV_key_no_value_elt:n #1{\ERROR}
6489 \cs_new:Npn \KV_key_value_elt:nn #1#2{\ERROR}

```

no_space_removal_no_sanitize:n Finally we can put all the things together. \KV_parse_no_space_removal_no_sanitize:n is the version that disallows unmatched conditional and does no space removal.

```

6490 \cs_new_protected_nopar:Npn \KV_parse_no_space_removal_no_sanitize:n {
6491   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_no_space_removal:w
6492   \KV_parse_no_sanitize_aux:n
6493 }

```

parse_space_removal_sanitize:n The other varieties can be defined in a similar manner. For the version needed at the document level, we can use this one.

```

6494 \cs_new_protected_nopar:Npn \KV_parse_space_removal_sanitize:n {
6495   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6496   \KV_parse_sanitize_aux:n
6497 }

```

For preamble use by the non-programmer this is probably best.

```

6498 \cs_new_protected_nopar:Npn \KV_parse_space_removal_no_sanitize:n {
6499   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6500   \KV_parse_no_sanitize_aux:n
6501 }

6502 \msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
6503 {Misplaced-equals-sign-in-key--value-input~\msg_line_context:}
6504 {
6505   I-am-trying-to-read-some-key--value-input-but-found-two-equals-
6506   signs\\%
6507   without~a-comma-between-them.
6508 }

6509 </initex | package>

```

```

6510 <*showmemory>
6511 \showMemUsage
6512 </showmemory>

```

The usual preliminaries.

```

6513 <*package>
6514 \ProvidesExplPackage
6515   {\filename}{\filedate}{\fileversion}{\filedescription}
6516 \package_check_loaded_expl:
6517 </package>
6518 <*initex | package>

```

114.1.1 Variables and constants

\c_keys_root_tl	Where the keys are really stored.
\c_keys_properties_root_tl	<pre> 6519 \tl_const:Nn \c_keys_root_tl { keys~>~ } 6520 \tl_const:Nn \c_keys_properties_root_tl { keys_properties } </pre>
\c_keys_value_forbidden_tl	Two marker token lists.
\c_keys_value_required_tl	<pre> 6521 \tl_const:Nn \c_keys_value_forbidden_tl { forbidden } 6522 \tl_const:Nn \c_keys_value_required_tl { required } </pre>
\l_keys_choice_int	Used for the multiple choice system.
\l_keys_choice_tl	<pre> 6523 \int_new:N \l_keys_choice_int 6524 \tl_new:N \l_keys_choice_tl </pre>
\l_keys_choice_code_tl	When creating multiple choices, the code is stored here.
	<pre> 6525 \tl_new:N \l_keys_choice_code_tl </pre>
\l_keys_key_tl	Storage for the current key name and the path of the key (key name plus module name).
\l_keys_path_tl	
\l_keys_property_tl	<pre> 6526 \tl_new:N \l_keys_key_tl 6527 \tl_new:N \l_keys_path_tl 6528 \tl_new:N \l_keys_property_tl </pre>
\l_keys_module_tl	The module for an entire set of keys.
	<pre> 6529 \tl_new:N \l_keys_module_tl </pre>
\l_keys_no_value_bool	To indicate that no value has been given.
	<pre> 6530 \bool_new:N \l_keys_no_value_bool </pre>
\l_keys_value_tl	A token variable for the given value.
	<pre> 6531 \tl_new:N \l_keys_value_tl </pre>

114.1.2 Internal functions

`\keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand.

```
6532 \cs_new_protected_nopar:Npn \keys_bool_set:Nn #1#2 {  
6533   \keys_cmd_set:nx { \l_keys_path_tl / true } {  
6534     \exp_not:c { bool_ #2 set_true:N }  
6535     \exp_not:N #1  
6536   }  
6537   \keys_cmd_set:nx { \l_keys_path_tl / false } {  
6538     \exp_not:N \use:c  
6539     { bool_ #2 set_false:N }  
6540     \exp_not:N #1  
6541   }  
6542   \keys_choice_make:  
6543   \cs_if_exist:NF #1 {  
6544     \bool_new:N #1  
6545   }  
6546   \keys_default_set:n { true }  
6547 }
```

`\keys_choice_code_store:x` The code for making multiple choices is stored in a token list as there should not be any # tokens.

```
6548 \cs_new_protected:Npn \keys_choice_code_store:x #1 {  
6549   \tl_set:cx { \c_keys_root_tl \l_keys_path_tl .choice_code_tl } {#1}  
6550 }
```

`\keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```
6551 \cs_new_protected_nopar:Npn \keys_choice_find:n #1 {  
6552   \keys_execute_aux:nn { \l_keys_path_tl / \tl_to_str:n {#1} } {  
6553     \keys_execute_aux:nn { \l_keys_path_tl / unknown } { }  
6554   }  
6555 }
```

`\keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```
6556 \cs_new_protected_nopar:Npn \keys_choice_make: {  
6557   \keys_cmd_set:nn { \l_keys_path_tl } {  
6558     \keys_choice_find:n {##1}  
6559   }  
6560   \keys_cmd_set:nn { \l_keys_path_tl / unknown } {  
6561     \msg_kernel_error:nnxx { keys } { choice-unknown }  
6562     { \l_keys_path_tl } {##1}  
6563   }  
6564 }
```

`\keys_choices_generate:n` Creating multiple-choices means setting up the “indicator” code, then applying whatever
`\keys_choices_generate_aux:n` the user wanted.

```

6565 \cs_new_protected:Npn \keys_choices_generate:n #1 {
6566   \keys_choice_make:
6567   \int_zero:N \l_keys_choice_int
6568   \cs_if_exist:cTF {
6569     \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6570   } {
6571     \tl_set:Nv \l_keys_choice_code_tl {
6572       \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6573     }
6574   }{
6575     \msg_kernel_error:nxx { keys } { generate-choices-before-code }
6576     { \l_keys_path_tl }
6577   }
6578   \clist_map_function:nN {#1} \keys_choices_generate_aux:n
6579 }
6580 \cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1 {
6581   \keys_cmd_set:nx { \l_keys_path_tl / #1 } {
6582     \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {#1}
6583     \exp_not:n { \int_set:Nn \l_keys_choice_int }
6584     { \int_use:N \l_keys_choice_int }
6585     \exp_not:V \l_keys_choice_code_tl
6586   }
6587   \int_incr:N \l_keys_choice_int
6588 }

```

`\keys_cmd_set:nn` Creating a new command means setting properties and then creating a function with the
`\keys_cmd_set:nx` correct number of arguments.

```

\keys_cmd_set_aux:n
6589 \cs_new_protected:Npn \keys_cmd_set:nn #1#2 {
6590   \keys_cmd_set_aux:n {#1}
6591   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6592   \cs_set:Npn 1 {#2}
6593 }
6594 \cs_new_protected:Npn \keys_cmd_set:nx #1#2 {
6595   \keys_cmd_set_aux:n {#1}
6596   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6597   \cs_set:Npx 1 {#2}
6598 }
6599 \cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1 {
6600   \keys_property_undefine:n { #1 .default_tl }
6601   \cs_if_free:cT { \c_keys_root_tl #1 .req_tl }
6602   { \tl_new:c { \c_keys_root_tl #1 .req_tl } }
6603   \tl_clear:c { \c_keys_root_tl #1 .req_tl }
6604 }

```

`\keys_default_set:n` Setting a default value is easy.
`\keys_default_set:V`

```

6605 \cs_new_protected:Npn \keys_default_set:n #1 {
6606   \cs_if_free:cT { \c_keys_root_tl \l_keys_path_tl .default_tl }
6607   { \tl_new:c { \c_keys_root_tl \l_keys_path_tl .default_tl } }
6608   \tl_set:cn { \c_keys_root_tl \l_keys_path_tl .default_tl } {#1}
6609 }
6610 \cs_generate_variant:Nn \keys_default_set:n { V }

```

`\keys_define:nn` The main key-defining function mainly sets up things for `l3keyval` to use.

```

\keys_define_aux:nnn
\keys_define_aux:onn
6611 \cs_new_protected:Npn \keys_define:nn {
6612   \keys_define_aux:onn { \l_keys_module_tl }
6613 }
6614 \cs_new_protected:Npn \keys_define_aux:nnn #1#2#3 {
6615   \tl_set:Nn \l_keys_module_tl {#2}
6616   \KV_process_no_space_removal_no_sanitise:NNn
6617   \keys_define_elt:n \keys_define_elt:nn {#3}
6618   \tl_set:Nn \l_keys_module_tl {#1}
6619 }
6620 \cs_generate_variant:Nn \keys_define_aux:nnn { o }

```

`\keys_define_elt:n` The element processors for defining keys.

```

\keys_define_elt:nn
6621 \cs_new_protected_nopar:Npn \keys_define_elt:n #1 {
6622   \bool_set_true:N \l_keys_no_value_bool
6623   \keys_define_elt_aux:nn {#1} { }
6624 }
6625 \cs_new_protected:Npn \keys_define_elt:nn #1#2 {
6626   \bool_set_false:N \l_keys_no_value_bool
6627   \keys_define_elt_aux:nn {#1} {#2}
6628 }

```

`\keys_define_elt_aux:nn` The auxiliary function does most of the work.

```

6629 \cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
6630   \keys_property_find:n {#1}
6631   \cs_set_eq:Nc \keys_tmp:w
6632   { \c_keys_properties_root_tl \l_keys_property_tl }
6633   \cs_if_exist:NTF \keys_tmp:w {
6634     \keys_define_key:n {#2}
6635   }{
6636     \msg_kernel_error:nnxx { keys } { property-unknown }
6637     { \l_keys_property_tl } { \l_keys_path_tl }
6638   }
6639 }

```

`\keys_define_key:n` Defining a new key means finding the code for the appropriate property then running it. As properties have signatures, a check can be made for required values without needing anything set explicitly.

```

6640 \cs_new_protected:Npn \keys_define_key:n #1 {
6641   \bool_if:NTF \l_keys_no_value_bool {
6642     \int_compare:nTF {
6643       \exp_args:Nc \cs_get_arg_count_from_signature:N
6644         { \l_keys_property_tl } = \c_zero
6645     } {
6646       \keys_tmp:w
6647     }{
6648       \msg_kernel_error:nxxx { key } { property-requires-value }
6649       { \l_keys_property_tl } { \l_keys_path_tl }
6650     }
6651   }{
6652     \keys_tmp:w {#1}
6653   }
6654 }

```

`\keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for
`\keys_execute_unknown:` the unknown key with the same path. If both of these fail, complain!
`\keys_execute_aux:nn`

```

6655 \cs_new_protected_nopar:Npn \keys_execute: {
6656   \keys_execute_aux:nn { \l_keys_path_tl } {
6657     \keys_execute_unknown:
6658   }
6659 }
6660 \cs_new_protected_nopar:Npn \keys_execute_unknown: {
6661   \keys_execute_aux:nn { \l_keys_module_tl / unknown } {
6662     \msg_kernel_error:nxxx { keys } { key-unknown } { \l_keys_path_tl }
6663     { \l_keys_module_tl }
6664   }
6665 }

```

If there is only one argument required, it is wrapped in braces so that everything is passed through properly. On the other hand, if more than one is needed it is down to the user to have put things in correctly! The use of `\q_keys_stop` here means that arguments do not run away (hence the nine empty groups), but that the module can clean up the spare groups at the end of executing the key.

```

6666 \cs_new_protected_nopar:Npn \keys_execute_aux:nn #1#2 {
6667   \cs_set_eq:Nc \keys_tmp:w { \c_keys_root_tl #1 .cmd:n }
6668   \cs_if_exist:NTF \keys_tmp:w {
6669     \exp_args:NV \keys_tmp:w \l_keys_value_tl
6670   }{
6671     #2
6672   }
6673 }

```

`\keys_if_exist:nnTF` A check for the existence of a key. This works by looking for the command function for the key (which ends `.cmd:n`).

```

6674 \prg_set_conditional:Nnn \keys_if_exist:nn {TF,T,F} {
6675   \cs_if_exist:cTF { \c_keys_root_tl #1 / #2 .cmd:n } {
6676     \prg_return_true:
6677   }{
6678     \prg_return_false:
6679   }
6680 }

```

`\keys_if_value_requirement:nTF` To test if a value is required or forbidden. Only one version is needed, so done by hand.

```

6681 \cs_new_nopar:Npn \keys_if_value_requirement:nTF #1 {
6682   \tl_if_eq:ccTF { c_keys_value_ #1 _tl } {
6683     \c_keys_root_tl \l_keys_path_tl .req_tl
6684   }
6685 }

```

`\keys_meta_make:n` To create a met-key, simply set up to pass data through.

`\keys_meta_make:x`

```

6686 \cs_new_protected_nopar:Npn \keys_meta_make:n #1 {
6687   \exp_last_unbraced:NNo \keys_cmd_set:nn \l_keys_path_tl
6688   \exp_after:wN { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
6689 }
6690 \cs_new_protected_nopar:Npn \keys_meta_make:x #1 {
6691   \keys_cmd_set:nx { \l_keys_path_tl } {
6692     \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1}
6693   }
6694 }

```

`\keys_property_find:n` Searching for a property means finding the last “.” in the input, and storing the text before and after it.

`\keys_property_find_aux:n`

`\keys_property_find_aux:w`

```

6695 \cs_new_protected_nopar:Npn \keys_property_find:n #1 {
6696   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
6697   \tl_if_in:nnTF {#1} {.} {
6698     \keys_property_find_aux:n {#1}
6699   }{
6700     \msg_kernel_error:nnx { keys } { key-no-property } {#1}
6701   }
6702 }
6703 \cs_new_protected_nopar:Npn \keys_property_find_aux:n #1 {
6704   \keys_property_find_aux:w #1 \q_stop
6705 }
6706 \cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop {
6707   \tl_if_in:nnTF {#2} { . } {
6708     \tl_set:Nx \l_keys_path_tl {
6709       \l_keys_path_tl \tl_to_str:n {#1} .
6710     }
6711     \keys_property_find_aux:w #2 \q_stop
6712   }{

```



```

6713     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
6714     \tl_set:Nn \l_keys_property_tl { . #2 }
6715   }
6716 }

```

`\keys_property_new:nn` Creating a new property is simply a case of making the correctly-named function.
`\keys_property_new_arg:nn`

```

6717 \cs_new_nopar:Npn \keys_property_new:nn #1#2 {
6718   \cs_new:cpn { \c_keys_properties_root_tl #1 } {#2}
6719 }
6720 \cs_new_protected_nopar:Npn \keys_property_new_arg:nn #1#2 {
6721   \cs_new:cpn { \c_keys_properties_root_tl #1 } ##1 {#2}
6722 }

```

`\keys_property_undefine:n` Removing a property means undefining it.

```

6723 \cs_new_protected_nopar:Npn \keys_property_undefine:n #1 {
6724   \cs_set_eq:cN { \c_keys_root_tl #1 } \c_undefined
6725 }

```

`\keys_set:nn` The main setting function just does the set up to get `l3keyval` to do the hard work.
`\keys_set:nV`
`\keys_set:nv`
`\keys_set_aux:nnn`
`\keys_set_aux:onn`

```

6726 \cs_new_protected:Npn \keys_set:nn {
6727   \keys_set_aux:onn { \l_keys_module_tl }
6728 }
6729 \cs_generate_variant:Nn \keys_set:nn { nV, nv }
6730 \cs_new_protected:Npn \keys_set_aux:nnn #1#2#3 {
6731   \tl_set:Nn \l_keys_module_tl {#2}
6732   \KV_process_space_removal_sanitization:NNn
6733     \keys_set_elt:n \keys_set_elt:nn {#3}
6734   \tl_set:Nn \l_keys_module_tl {#1}
6735 }
6736 \cs_generate_variant:Nn \keys_set_aux:nnn { o }

```

`\keys_set_elt:n` The two element processors are almost identical, and pass the data through to the underlying auxiliary, which does the work.
`\keys_set_elt:nn`

```

6737 \cs_new_protected_nopar:Npn \keys_set_elt:n #1 {
6738   \bool_set_true:N \l_keys_no_value_bool
6739   \keys_set_elt_aux:nn {#1} { }
6740 }
6741 \cs_new_protected:Npn \keys_set_elt:nn #1#2 {
6742   \bool_set_false:N \l_keys_no_value_bool
6743   \keys_set_elt_aux:nn {#1} {#2}
6744 }

```

`\keys_set_elt_aux:nn` First, set the current path and add a default if needed. There are then checks to see if
`\keys_set_elt_aux:` the a value is required or forbidden. If everything passes, move on to execute the code.

```

6745 \cs_new_protected:Npn \keys_set_elt_aux:nn #1#2 {
6746   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
6747   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
6748   \keys_value_or_default:n {#2}
6749   \keys_if_value_requirement:NTF { required } {
6750     \bool_if:NTF \l_keys_no_value_bool {
6751       \msg_kernel_error:nnx { keys } { value-required }
6752       { \l_keys_path_tl }
6753     }{
6754       \keys_set_elt_aux:
6755     }
6756   }{
6757     \keys_set_elt_aux:
6758   }
6759 }
6760 \cs_new_protected_nopar:Npn \keys_set_elt_aux: {
6761   \keys_if_value_requirement:NTF { forbidden } {
6762     \bool_if:NTF \l_keys_no_value_bool {
6763       \keys_execute:
6764     }{
6765       \msg_kernel_error:nnxx { keys } { value-forbidden }
6766       { \l_keys_path_tl } { \tl_use:N \l_keys_value_tl }
6767     }
6768   }{
6769     \keys_execute:
6770   }
6771 }

```

\keys_show:nn Showing a key is just a question of using the correct name.

```

6772 \cs_new_nopar:Npn \keys_show:nn #1#2 {
6773   \cs_show:c { \c_keys_root_tl #1 / \tl_to_str:n {#2} .cmd:n }
6774 }

```

\keys_tmp:w This scratch function is used to actually execute keys.

```

6775 \cs_new:Npn \keys_tmp:w {}

```

\keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

6776 \cs_new_protected:Npn \keys_value_or_default:n #1 {
6777   \tl_set:Nn \l_keys_value_tl {#1}
6778   \bool_if:NT \l_keys_no_value_bool {
6779     \cs_if_exist:cT { \c_keys_root_tl \l_keys_path_tl .default_tl } {
6780       \tl_set:Nv \l_keys_value_tl {
6781         \c_keys_root_tl \l_keys_path_tl .default_tl
6782       }
6783     }
6784   }
6785 }

```

`\keys_value_requirement:n` Values can be required or forbidden by having the appropriate marker set.

```

6786 \cs_new_protected_nopar:Npn \keys_value_requirement:n #1 {
6787   \tl_set_eq:cc { \c_keys_root_tl \l_keys_path_tl .req_tl }
6788   { c_keys_value_ #1 _tl }
6789 }

```

`\keys_variable_set:NnNN` Setting a variable takes the type and scope separately so that it is easy to make a new
`\keys_variable_set:cnNN` variable if needed.

```

6790 \cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4 {
6791   \cs_if_exist:NF #1 {
6792     \use:c { #2 _new:N } #1
6793   }
6794   \keys_cmd_set:nx { \l_keys_path_tl } {
6795     \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1}
6796   }
6797 }
6798 \cs_generate_variant:Nn \keys_variable_set:NnNN { c }

```

114.1.3 Properties

`.bool_set:N` One function for this.

`.bool_gset:N`

```

6799 \keys_property_new_arg:nn { .bool_set:N } {
6800   \keys_bool_set:Nn #1 { }
6801 }
6802 \keys_property_new_arg:nn { .bool_gset:N } {
6803   \keys_bool_set:Nn #1 { g }
6804 }

```

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

6805 \keys_property_new:nn { .choice: } {
6806   \keys_choice_make:
6807 }

```

`.choice_code:n` Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

`.choice_code:x`

```

6808 \keys_property_new_arg:nn { .choice_code:n } {
6809   \keys_choice_code_store:x { \exp_not:n {#1} }
6810 }
6811 \keys_property_new_arg:nn { .choice_code:x } {
6812   \keys_choice_code_store:x {#1}
6813 }

```

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

`.code:x`

```

6814 \keys_property_new_arg:nn { .code:n } {
6815   \keys_cmd_set:nn { \l_keys_path_tl } {#1}
6816 }
6817 \keys_property_new_arg:nn { .code:x } {
6818   \keys_cmd_set:nx { \l_keys_path_tl } {#1}
6819 }

```

`.default:n` Expansion is left to the internal functions.

```

.default:V
6820 \keys_property_new_arg:nn { .default:n } {
6821   \keys_default_set:n {#1}
6822 }
6823 \keys_property_new_arg:nn { .default:V } {
6824   \keys_default_set:V #1
6825 }

```

`.dim_set:N` Setting a variable is very easy: just pass the data along.

```

.dim_set:c
6826 \keys_property_new_arg:nn { .dim_set:N } {
6827   \keys_variable_set:NnNN #1 { dim } { } n
6828 }
6829 \keys_property_new_arg:nn { .dim_set:c } {
6830   \keys_variable_set:cnNN {#1} { dim } { } n
6831 }
6832 \keys_property_new_arg:nn { .dim_gset:N } {
6833   \keys_variable_set:NnNN #1 { dim } g n
6834 }
6835 \keys_property_new_arg:nn { .dim_gset:c } {
6836   \keys_variable_set:cnNN {#1} { dim } g n
6837 }

```

`.fp_set:N` Setting a variable is very easy: just pass the data along.

```

.fp_set:c
6838 \keys_property_new_arg:nn { .fp_set:N } {
6839   \keys_variable_set:NnNN #1 { fp } { } n
6840 }
6841 \keys_property_new_arg:nn { .fp_set:c } {
6842   \keys_variable_set:cnNN {#1} { fp } { } n
6843 }
6844 \keys_property_new_arg:nn { .fp_gset:N } {
6845   \keys_variable_set:NnNN #1 { fp } g n
6846 }
6847 \keys_property_new_arg:nn { .fp_gset:c } {
6848   \keys_variable_set:cnNN {#1} { fp } g n
6849 }

```

`.generate_choices:n` Making choices is easy.

```

6850 \keys_property_new_arg:nn { .generate_choices:n } {
6851   \keys_choices_generate:n {#1}
6852 }

```

.int_set:N Setting a variable is very easy: just pass the data along.

```
.int_set:c
.int_gset:N
.int_gset:c
6853 \keys_property_new_arg:nn { .int_set:N } {
6854 \keys_variable_set:NnNN #1 { int } { } n
6855 }
6856 \keys_property_new_arg:nn { .int_set:c } {
6857 \keys_variable_set:cnNN {#1} { int } { } n
6858 }
6859 \keys_property_new_arg:nn { .int_gset:N } {
6860 \keys_variable_set:NnNN #1 { int } g n
6861 }
6862 \keys_property_new_arg:nn { .int_gset:c } {
6863 \keys_variable_set:cnNN {#1} { int } g n
6864 }
```

.meta:n Making a meta is handled internally.

```
.meta:x
6865 \keys_property_new_arg:nn { .meta:n } {
6866 \keys_meta_make:n {#1}
6867 }
6868 \keys_property_new_arg:nn { .meta:x } {
6869 \keys_meta_make:x {#1}
6870 }
```

.skip_set:N Setting a variable is very easy: just pass the data along.

```
.skip_set:c
.skip_gset:N
.skip_gset:c
6871 \keys_property_new_arg:nn { .skip_set:N } {
6872 \keys_variable_set:NnNN #1 { skip } { } n
6873 }
6874 \keys_property_new_arg:nn { .skip_set:c } {
6875 \keys_variable_set:cnNN {#1} { skip } { } n
6876 }
6877 \keys_property_new_arg:nn { .skip_gset:N } {
6878 \keys_variable_set:NnNN #1 { skip } g n
6879 }
6880 \keys_property_new_arg:nn { .skip_gset:c } {
6881 \keys_variable_set:cnNN {#1} { skip } g n
6882 }
```

.tl_set:N Setting a variable is very easy: just pass the data along.

```
.tl_set:c
.tl_set_x:N
.tl_set_x:c
.tl_gset:N
.tl_gset:c
.tl_gset_x:N
.tl_gset_x:c
6883 \keys_property_new_arg:nn { .tl_set:N } {
6884 \keys_variable_set:NnNN #1 { tl } { } n
6885 }
6886 \keys_property_new_arg:nn { .tl_set:c } {
6887 \keys_variable_set:cnNN {#1} { tl } { } n
6888 }
6889 \keys_property_new_arg:nn { .tl_set_x:N } {
6890 \keys_variable_set:NnNN #1 { tl } { } x
6891 }
```

```

6892 \keys_property_new_arg:nn { .tl_set_x:c } {
6893   \keys_variable_set:cnNN {#1} { tl } { } x
6894 }
6895 \keys_property_new_arg:nn { .tl_gset:N } {
6896   \keys_variable_set:NnNN #1 { tl } g n
6897 }
6898 \keys_property_new_arg:nn { .tl_gset:c } {
6899   \keys_variable_set:cnNN {#1} { tl } g n
6900 }
6901 \keys_property_new_arg:nn { .tl_gset_x:N } {
6902   \keys_variable_set:NnNN #1 { tl } g x
6903 }
6904 \keys_property_new_arg:nn { .tl_gset_x:c } {
6905   \keys_variable_set:cnNN {#1} { tl } g x
6906 }

```

.value_forbidden: These are very similar, so both call the same function.

```

.value_required:
6907 \keys_property_new:nn { .value_forbidden: } {
6908   \keys_value_requirement:n { forbidden }
6909 }
6910 \keys_property_new:nn { .value_required: } {
6911   \keys_value_requirement:n { required }
6912 }

```

114.1.4 Messages

For when there is a need to complain.

```

6913 \msg_kernel_new:nnnn { keys } { choice-unknown }
6914 { Choice~'#2'~unknown~for~key~'#1'. }
6915 {
6916   The~key~'#1'~takes~a~limited~number~of~values.\\
6917   The~input~given,~'#2',~is~not~on~the~list~accepted.
6918 }
6919 \msg_kernel_new:nnnn { keys } { generate-choices-before-code }
6920 { No~code~available~to~generate~choices~for~key~'#1'. }
6921 {
6922   \l_msg_coding_error_text_tl
6923   Before~using~.generate_choices:n~the~code~should~be~defined\\%
6924   with~.choice_code:n~or~.choice_code:x.
6925 }
6926 \msg_kernel_new:nnnn { keys } { key-no-property }
6927 { No~property~given~in~definition~of~key~'#1'. }
6928 {
6929   \c_msg_coding_error_text_tl
6930   Inside~\token_to_str:N \keys_define:nn \c_space_tl each~key~name
6931   needs~a~property: \\
6932   ~ ~ #1 .<property> \\

```

```

6933 LaTeX~did~not~find~a~'. '~to~indicate~the~start~of~a~property.
6934 }
6935 \msg_kernel_new:nnnn { keys } { key-unknown }
6936 { The~key~'#1'~is~unknown~and~is~being~ignored. }
6937 {
6938   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
6939   Check~that~you~have~spelled~the~key~name~correctly.
6940 }
6941 \msg_kernel_new:nnnn { keys } { property-requires-value }
6942 { The~property~'#1'~requires~a~value. }
6943 {
6944   \l_msg_coding_error_text_tl
6945   LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
6946   No~value~was~given~for~the~property,~and~one~is~required.
6947 }
6948 \msg_kernel_new:nnnn { keys } { property-unknown }
6949 { The~key~property~'#1'~is~unknown. }
6950 {
6951   \l_msg_coding_error_text_tl
6952   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':\\
6953   this~property~is~not~defined.
6954 }
6955 \msg_kernel_new:nnnn { keys } { value-forbidden }
6956 { The~key~'#1'~does~not~taken~a~value. }
6957 {
6958   The~key~'#1'~should~be~given~without~a~value.\\
6959   LaTeX~will~ignore~the~given~value~'#2'.
6960 }
6961 \msg_kernel_new:nnnn { keys } { value-required }
6962 { The~key~'#1'~requires~a~value. }
6963 {
6964   The~key~'#1'~must~have~a~value.\\
6965   No~value~was~present:~the~key~will~be~ignored.
6966 }
6967 </initex | package>

```

115 l3file implementation

The usual lead-off.

```

6968 <*package>
6969 \ProvidesExplPackage
6970 { \filename } { \filedate } { \fileversion } { \filedescription }
6971 \package_check_loaded_expl:
6972 </package>
6973 <*initex | package>

```

`\g_file_current_name_tl` The name of the current file should be available at all times.

`\g_file_stack_seq`

```
6974 \tl_new:N \g_file_current_name_tl
6975 \seq_new:N \g_file_stack_seq
```

For the format the file name needs to be picked up at the start of the file. In package mode the current file name is collected from LaTeX2e.

```
6976 </initex | package>
6977 <*initex>
6978 \toks_put_right:Nn \tex_everyjob:D {
6979   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
6980 }
6981 </initex>
6982 <*package>
6983 \tl_gset_eq:NN \g_file_current_name_tl \@currname
6984 </package>
6985 <*initex | package>
```

`\g_file_record_seq` The total list of files used is recorded separately from the stack.

```
6986 \seq_new:N \g_file_record_seq
```

The current file name should be included in the file list!

```
6987 </initex | package>
6988 <*initex>
6989 \toks_put_right:Nn \tex_everyjob:D {
6990   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
6991 }
6992 </initex>
6993 <*initex | package>
```

`\l_file_search_path_seq` The current search path.

```
6994 \seq_new:N \l_file_search_path_seq
```

`\l_file_search_path_saved_seq` The current search path has to be saved for package use.

```
6995 </initex | package>
6996 <*package>
6997 \seq_new:N \l_file_search_path_saved_seq
6998 </package>
6999 <*initex | package>
```

`\l_file_name_tl`

`\g_file_test_stream`

`\file_if_exist:nTF`

`\file_if_exist:VTF`

`\file_if_exist_aux:n`

Checking if a file exists takes place in two parts. First, look on the TeX path, then look on the LaTeX path. The token list `\l_file_name_tl` is used as a marker for finding the file, and is also needed by `\file_input:n`.

```
7000 \tl_new:N \l_file_name_tl
```



```

7001 \prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF } {
7002   \ior_open:Nn \g_file_test_stream {#1}
7003   \ior_if_eof:NTF \g_file_test_stream
7004   { \file_if_exist_path_aux:n {#1} }
7005   {
7006     \ior_close:N \g_file_test_stream
7007     \tl_set:Nn \l_file_name_tl {#1}
7008     \prg_return_true:
7009   }
7010 }
7011 \cs_new_protected_nopar:Npn \file_if_exist_path_aux:n #1 {
7012   \tl_clear:N \l_file_name_tl
7013   </initex | package>
7014   <*package>
7015   \cs_if_exist:NT \input@path
7016   {
7017     \seq_set_eq:NN \l_file_search_path_saved_seq
7018     \l_file_search_path_seq
7019     \clist_map_inline:Nn \input@path
7020     { \seq_put_right:Nn \l_file_search_path_seq {##1} }
7021   }
7022   </package>
7023   <*initex | package>
7024   \seq_map_inline:Nn \l_file_search_path_seq
7025   {
7026     \ior_open:Nn \g_file_test_stream { ##1 #1 }
7027     \ior_if_eof:NF \g_file_test_stream
7028     {
7029       \tl_set:Nn \l_file_name_tl { ##1 #1 }
7030       \seq_map_break:
7031     }
7032   }
7033   </initex | package>
7034   <*package>
7035   \cs_if_exist:NT \input@path
7036   {
7037     \seq_set_eq:NN \l_file_search_path_seq
7038     \l_file_search_path_saved_seq
7039   }
7040   </package>
7041   <*initex | package>
7042   \ior_close:N \g_file_test_stream
7043   \tl_if_empty:NTF \l_file_name_tl
7044   { \prg_return_false: }
7045   { \prg_return_true: }
7046 }
7047 \cs_generate_variant:Nn \file_if_exist:nT { V }
7048 \cs_generate_variant:Nn \file_if_exist:nF { V }
7049 \cs_generate_variant:Nn \file_if_exist:nTF { V }

```

`\file_input:n` Most of the work is done by the file test above.

```

\file_input:V
7050 \cs_new_protected_nopar:Npn \file_input:n #1 {
7051   \file_if_exist:nT {#1}
7052   {
7053     </initex | package>
7054     <*package>
7055     \@addtofilelist {#1}
7056     </package>
7057     <*initex | package>
7058     \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
7059     \tl_gset:Nn \g_file_current_name_tl {#1}
7060     \tex_expandafter:D \tex_input:D \l_file_name_tl ~
7061     \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
7062   }
7063 }
7064 \cs_generate_variant:Nn \file_input:n { V }

```

`\file_path_include:n` Wrapper functions to manage the search path.

```

\file_path_remove:n
7065 \cs_new_protected_nopar:Npn \file_path_include:n #1 {
7066   \seq_put_right:Nn \l_file_search_path_seq {#1}
7067   \seq_remove_duplicates:N \l_file_search_path_seq
7068 }
7069 \cs_new_protected_nopar:Npn \file_path_remove:n #1 {
7070   \seq_remove_element:Nn \l_file_search_path_seq {#1}
7071 }

```

`\file_list:` A function to list all files used to the log.

```

\file_list_aux:n
7072 \cs_new_protected_nopar:Npn \file_list: {
7073   \seq_remove_duplicates:N \g_file_record_seq
7074   \iow_log:n { ~File~List~* }
7075   \seq_map_function:NN \g_file_record_seq \file_list_aux:n
7076   \iow_log:n { ***** }
7077 }
7078 \cs_new_protected_nopar:Npn \file_list_aux:n #1 { \iow_log:n {#1} }

```

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

7079 </initex | package>
7080 <*package>
7081 \AtBeginDocument{
7082   \clist_map_inline:Nn \@filelist
7083   { \seq_put_right:Nn \g_file_record_seq {#1} }
7084 }
7085 </package>

```

116 Implementation

We start by ensuring that the required packages are loaded.

```

7086 \<package>
7087 \ProvidesExplPackage
7088   {\filename}{\filedate}{\fileversion}{\filedescription}
7089 \package_check_loaded_expl:
7090 \</package>
7091 \*initex | package>

```

116.1 Constants

\c_forty_four

There is some speed to gain by moving numbers into fixed positions.

\c_one_hundred

\c_one_thousand

\c_one_million

\c_one_hundred_million

\c_five_hundred_million

\c_one_thousand_million

```

7092 \int_const:Nn \c_forty_four { 44 }
7093 \int_const:Nn \c_one_hundred { 100 }
7094 \int_const:Nn \c_one_thousand { 1000 }
7095 \int_const:Nn \c_one_million { 1 000 000 }
7096 \int_const:Nn \c_one_hundred_million { 100 000 000 }
7097 \int_const:Nn \c_five_hundred_million { 500 000 000 }
7098 \int_const:Nn \c_one_thousand_million { 1 000 000 000 }

```

\c_fp_pi_by_four_decimal_int

\c_fp_pi_by_four_extended_int

\c_fp_pi_decimal_int

\c_fp_pi_extended_int

\c_fp_two_pi_decimal_int

\c_fp_two_pi_extended_int

Parts of π for trigonometric range reduction, implemented as int variables for speed.

```

7099 \int_new:N \c_fp_pi_by_four_decimal_int
7100 \int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
7101 \int_new:N \c_fp_pi_by_four_extended_int
7102 \int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
7103 \int_new:N \c_fp_pi_decimal_int
7104 \int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
7105 \int_new:N \c_fp_pi_extended_int
7106 \int_set:Nn \c_fp_pi_extended_int { 589 793 238 }
7107 \int_new:N \c_fp_two_pi_decimal_int
7108 \int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
7109 \int_new:N \c_fp_two_pi_extended_int
7110 \int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }

```

\c_e_fp

The value e as a ‘machine number’.

```

7111 \tl_new:N \c_e_fp
7112 \tl_set:Nn \c_e_fp { + 2.718281828 e 0 }

```

\c_one_fp

The constant value 1: used for fast comparisons.

```

7113 \tl_new:N \c_one_fp
7114 \tl_set:Nn \c_one_fp { + 1.000000000 e 0 }

```

`\c_pi_fp` The value π as a ‘machine number’.

```
7115 \tl_new:N \c_pi_fp
7116 \tl_set:Nn \c_pi_fp { + 3.141592654 e 0 }
```

`\c_undefined_fp` A marker for undefined values.

```
7117 \tl_new:N \c_undefined_fp
7118 \tl_set:Nn \c_undefined_fp { X 0.000000000 e 0 }
```

`\c_zero_fp` The constant zero value.

```
7119 \tl_new:N \c_zero_fp
7120 \tl_set:Nn \c_zero_fp { + 0.000000000 e 0 }
```

116.2 Variables

`\l_fp_arg_tl` A token list to store the formalised representation of the input for transcendental functions.

```
7121 \tl_new:N \l_fp_arg_tl
```

`\l_fp_count_int` A counter for things like the number of divisions possible.

```
7122 \int_new:N \l_fp_count_int
```

`\l_fp_div_offset_int` When carrying out division, an offset is used for the results to get the decimal part correct.

```
7123 \int_new:N \l_fp_div_offset_int
```

`\l_fp_exp_integer_int` Used for the calculation of exponent values.

`\l_fp_exp_decimal_int`

`\l_fp_exp_extended_int`

`\l_fp_exp_exponent_int`

```
7124 \int_new:N \l_fp_exp_integer_int
```

```
7125 \int_new:N \l_fp_exp_decimal_int
```

```
7126 \int_new:N \l_fp_exp_extended_int
```

```
7127 \int_new:N \l_fp_exp_exponent_int
```

`\l_fp_input_a_sign_int`

Storage for the input: two storage areas as there are at most two inputs.

`\l_fp_input_a_integer_int`

`\l_fp_input_a_decimal_int`

`\l_fp_input_a_exponent_int`

`\l_fp_input_b_sign_int`

`\l_fp_input_b_integer_int`

`\l_fp_input_b_decimal_int`

`\l_fp_input_b_exponent_int`

```
7128 \int_new:N \l_fp_input_a_sign_int
```

```
7129 \int_new:N \l_fp_input_a_integer_int
```

```
7130 \int_new:N \l_fp_input_a_decimal_int
```

```
7131 \int_new:N \l_fp_input_a_exponent_int
```

```
7132 \int_new:N \l_fp_input_b_sign_int
```

```
7133 \int_new:N \l_fp_input_b_integer_int
```

```
7134 \int_new:N \l_fp_input_b_decimal_int
```

```
7135 \int_new:N \l_fp_input_b_exponent_int
```

<code>\l_fp_input_a_extended_int</code>	For internal use, ‘extended’ floating point numbers are needed.
<code>\l_fp_input_b_extended_int</code>	<pre> 7136 \int_new:N \l_fp_input_a_extended_int 7137 \int_new:N \l_fp_input_b_extended_int </pre>
<pre> \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int </pre>	<p>Multiplication requires that the decimal part is split into parts so that there are no overflows.</p> <pre> 7138 \int_new:N \l_fp_mul_a_i_int 7139 \int_new:N \l_fp_mul_a_ii_int 7140 \int_new:N \l_fp_mul_a_iii_int 7141 \int_new:N \l_fp_mul_a_iv_int 7142 \int_new:N \l_fp_mul_a_v_int 7143 \int_new:N \l_fp_mul_a_vi_int 7144 \int_new:N \l_fp_mul_b_i_int 7145 \int_new:N \l_fp_mul_b_ii_int 7146 \int_new:N \l_fp_mul_b_iii_int 7147 \int_new:N \l_fp_mul_b_iv_int 7148 \int_new:N \l_fp_mul_b_v_int 7149 \int_new:N \l_fp_mul_b_vi_int </pre>
<pre> \l_fp_mul_output_int \l_fp_mul_output_tl </pre>	<p>Space for multiplication results.</p> <pre> 7150 \int_new:N \l_fp_mul_output_int 7151 \tl_new:N \l_fp_mul_output_tl </pre>
<pre> \l_fp_output_sign_int \l_fp_output_integer_int \l_fp_output_decimal_int \l_fp_output_exponent_int </pre>	<p>Output is stored in the same way as input.</p> <pre> 7152 \int_new:N \l_fp_output_sign_int 7153 \int_new:N \l_fp_output_integer_int 7154 \int_new:N \l_fp_output_decimal_int 7155 \int_new:N \l_fp_output_exponent_int </pre>
<code>\l_fp_output_extended_int</code>	<p>Again, for calculations an extended part.</p> <pre> 7156 \int_new:N \l_fp_output_extended_int </pre>
<code>\l_fp_round_carry_bool</code>	<p>To indicate that a digit needs to be carried forward.</p> <pre> 7157 \bool_new:N \l_fp_round_carry_bool </pre>
<code>\l_fp_round_decimal_tl</code>	<p>A temporary store when rounding, to build up the decimal part without needing to do any maths.</p> <pre> 7158 \tl_new:N \l_fp_round_decimal_tl </pre>
<pre> \l_fp_round_position_int \l_fp_round_target_int </pre>	<p>Used to check the position for rounding.</p> <pre> 7159 \int_new:N \l_fp_round_position_int 7160 \int_new:N \l_fp_round_target_int </pre>

`\l_fp_sign_tl` There are places where the sign needs to be set up ‘early’, so that the registers can be re-used.

```
7161 \tl_new:N \l_fp_sign_tl
```

`\l_fp_split_sign_int` When splitting the input it is fastest to use a fixed name for the sign part, and to transfer it after the split is complete.

```
7162 \int_new:N \l_fp_split_sign_int
```

`\l_fp_tmp_int` A scratch int: used only where the value is not carried forward.

```
7163 \int_new:N \l_fp_tmp_int
```

`\l_fp_tmp_tl` A scratch token list variable for expanding material.

```
7164 \tl_new:N \l_fp_tmp_tl
```

`\l_fp_trig_octant_int` To track which octant the trigonometric input is in.

```
7165 \int_new:N \l_fp_trig_octant_int
```

`\l_fp_trig_sign_int` Used for the calculation of trigonometric values.

`\l_fp_trig_decimal_int`
`\l_fp_trig_extended_int`

```
7166 \int_new:N \l_fp_trig_sign_int
7167 \int_new:N \l_fp_trig_decimal_int
7168 \int_new:N \l_fp_trig_extended_int
```

116.3 Parsing numbers

`\fp_read:N` Reading a stored value is made easier as the format is designed to match the delimited
`\fp_read_aux:w` function. This is always used to read the first value (register a).

```
7169 \cs_new_protected_nopar:Npn \fp_read:N #1 {
7170   \tex_expandafter:D \fp_read_aux:w #1 \q_stop
7171 }
7172 \cs_new_protected_nopar:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop {
7173   \tex_if:D #1 -
7174     \l_fp_input_a_sign_int \c_minus_one
7175   \tex_else:D
7176     \l_fp_input_a_sign_int \c_one
7177   \tex_fi:D
7178   \l_fp_input_a_integer_int #2 \scan_stop:
7179   \l_fp_input_a_decimal_int #3 \scan_stop:
7180   \l_fp_input_a_exponent_int #4 \scan_stop:
7181 }
```

\backslash fp_split:Nn \backslash fp_split_sign: \backslash fp_split_exponent: \backslash fp_split_aux_i:w \backslash fp_split_aux_ii:w \backslash fp_split_aux_iii:w \backslash fp_split_decimal:w \backslash fp_split_decimal_aux:w	<p>The aim here is to use as much of T_EX's mechanism as possible to pick up the numerical input without any mistakes. In particular, negative numbers have to be filtered out first in case the integer part is 0 (in which case T_EX would drop the - sign). That process has to be done in a loop for cases where the sign is repeated. Finding an exponent is relatively easy, after which the next phase is to find the integer part, which will terminate with a ., and trigger the decimal-finding code. The later will allow the decimal to be too long, truncating the result.</p> <pre> 7182 \cs_new_protected_nopar:Npn \fp_split:Nn #1#2 { 7183 \tl_set:Nx \l_fp_tmp_tl {#2} 7184 \tl_set_rescan:Nno \l_fp_tmp_tl { \char_make_ignore:n { 32 } } 7185 { \l_fp_tmp_tl } 7186 \l_fp_split_sign_int \c_one 7187 \fp_split_sign: 7188 \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int 7189 \tex_expandafter:D \fp_split_exponent:w \l_fp_tmp_tl e e \q_stop #1 7190 } 7191 \cs_new_protected_nopar:Npn \fp_split_sign: { 7192 \tex_ifnum:D \pdf_strcmp:D 7193 { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { - } 7194 = \c_zero 7195 \tl_set:Nx \l_fp_tmp_tl 7196 { 7197 \tex_expandafter:D 7198 \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop 7199 } 7200 \l_fp_split_sign_int -\l_fp_split_sign_int 7201 \tex_expandafter:D \fp_split_sign: 7202 \tex_else:D 7203 \tex_ifnum:D \pdf_strcmp:D 7204 { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { + } 7205 = \c_zero 7206 \tl_set:Nx \l_fp_tmp_tl 7207 { 7208 \tex_expandafter:D 7209 \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop 7210 } 7211 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D 7212 \fp_split_sign: 7213 \tex_fi:D 7214 \tex_fi:D 7215 } 7216 \cs_new_protected_nopar:Npn 7217 \fp_split_exponent:w #1 e #2 e #3 \q_stop #4 { 7218 \use:c { l_fp_input_ #4 _exponent_int } 7219 \etex_numexpr:D 0 #2 \scan_stop: 7220 \tex_afterassignment:D \fp_split_aux_i:w 7221 \use:c { l_fp_input_ #4 _integer_int } 7222 \etex_numexpr:D 0 #1 . . \q_stop #4 7223 } </pre>
--	---

```

7224 \cs_new_protected_nopar:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop {
7225   \fp_split_aux_ii:w #2 000000000 \q_stop
7226 }
7227 \cs_new_protected_nopar:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9 {
7228   \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9}
7229 }
7230 \cs_new_protected_nopar:Npn \fp_split_aux_iii:w #1#2 \q_stop {
7231   \l_fp_tmp_int 1 #1 \scan_stop:
7232   \tex_expandafter:D \fp_split_decimal:w
7233   \int_use:N \l_fp_tmp_int 000000000 \q_stop
7234 }
7235 \cs_new_protected_nopar:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9 {
7236   \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9}
7237 }
7238 \cs_new_protected_nopar:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4 {
7239   \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
7240   \tex_ifnum:D
7241     \etex_numexpr:D
7242       \use:c { l_fp_input_ #4 _integer_int } +
7243       \use:c { l_fp_input_ #4 _decimal_int }
7244     \scan_stop:
7245     = \c_zero
7246     \use:c { l_fp_input_ #4 _sign_int } \c_one
7247   \tex_fi:D
7248   \tex_ifnum:D
7249     \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
7250   \tex_else:D
7251     \tex_expandafter:D \fp_overflow_msg:
7252   \tex_fi:D
7253 }

```

`\fp_standardise:NNNN`
`\fp_standardise_aux:NNNN`
`\fp_standardise_aux:`
`\fp_standardise_aux:w`

The idea here is to shift the input into a known exponent range. This is done using \TeX tokens where possible, as this is faster than arithmetic.

```

7254 \cs_new_protected_nopar:Npn \fp_standardise:NNNN #1#2#3#4 {
7255   \tex_ifnum:D
7256     \etex_numexpr:D #2 + #3 = \c_zero
7257     #1 \c_one
7258     #4 \c_zero
7259     \tex_expandafter:D \use_none:nnnn
7260   \tex_else:D
7261     \tex_expandafter:D \fp_standardise_aux:NNNN
7262   \tex_fi:D
7263   #1#2#3#4
7264 }
7265 \cs_new_protected_nopar:Npn \fp_standardise_aux:NNNN #1#2#3#4 {
7266   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7267     {
7268       \tex_ifnum:D #2 = \c_zero
7269       \tex_advance:D #3 \c_one_thousand_million

```



```

7270         \tex_expandafter:D \fp_standardise_aux:w
7271         \int_use:N #3 \q_stop
7272         \tex_expandafter:D \fp_standardise_aux:
7273         \tex_fi:D
7274     }
7275 \cs_set_protected_nopar:Npn
7276   \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
7277   {
7278     #2 ##2 \scan_stop:
7279     #3 ##3##4##5##6##7##8##9 0 \scan_stop:
7280     \tex_advance:D #4 \c_minus_one
7281   }
7282 \fp_standardise_aux:
7283 \cs_set_protected_nopar:Npn \fp_standardise_aux:
7284   {
7285     \tex_ifnum:D #2 > \c_nine
7286     \tex_advance:D #2 \c_one_thousand_million
7287     \tex_expandafter:D \use_i:nn \tex_expandafter:D
7288     \fp_standardise_aux:w \int_use:N #2
7289     \tex_expandafter:D \fp_standardise_aux:
7290     \tex_fi:D
7291   }
7292 \cs_set_protected_nopar:Npn
7293   \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9
7294   {
7295     #2 ##1##2##3##4##5##6##7##8 \scan_stop:
7296     \tex_advance:D #3 \c_one_thousand_million
7297     \tex_divide:D #3 \c_ten
7298     \tl_set:Nx \l_fp_tmp_tl
7299     {
7300       ##9
7301       \tex_expandafter:D \use_none:n \int_use:N #3
7302     }
7303     #3 \l_fp_tmp_tl \scan_stop:
7304     \tex_advance:D #4 \c_one
7305   }
7306 \fp_standardise_aux:
7307 \tex_ifnum:D #4 < \c_one_hundred
7308 \tex_ifnum:D #4 > -\c_one_hundred
7309 \tex_else:D
7310   #1 \c_one
7311   #2 \c_zero
7312   #3 \c_zero
7313   #4 \c_zero
7314 \tex_fi:D
7315 \tex_else:D
7316   \tex_expandafter:D \fp_overflow_msg:
7317   \tex_fi:D
7318 }
7319 \cs_new_protected_nopar:Npn \fp_standardise_aux: { }

```

```
7320 \cs_new_protected_nopar:Npn \fp_standardise_aux:w { }
```

116.4 Internal utilities

`\fp_level_input_exponents:` The routines here are similar to those used to standardise the exponent. However, the aim here is different: the two exponents need to end up the same.

```
\fp_level_input_exponents_a:NNNNNNNNN
\fp_level_input_exponents_b:NNNNNNNNN
\fp_level_input_exponents_a:NNNNNNNNN
\fp_level_input_exponents_b:NNNNNNNNN

7321 \cs_new_protected_nopar:Npn \fp_level_input_exponents: {
7322   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7323     \tex_expandafter:D \fp_level_input_exponents_a:
7324   \tex_else:D
7325     \tex_expandafter:D \fp_level_input_exponents_b:
7326   \tex_fi:D
7327 }
7328 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a: {
7329   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7330     \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
7331     \tex_expandafter:D \use_i:nn \tex_expandafter:D
7332       \fp_level_input_exponents_a:NNNNNNNNN
7333       \int_use:N \l_fp_input_b_integer_int
7334     \tex_expandafter:D \fp_level_input_exponents_a:
7335   \tex_fi:D
7336 }
7337 \cs_new_protected_nopar:Npn
7338   \fp_level_input_exponents_a:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7339   \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7340   \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
7341   \tex_divide:D \l_fp_input_b_decimal_int \c_ten
7342   \tl_set:Nx \l_fp_tmp_tl
7343     {
7344       #9
7345       \tex_expandafter:D \use_none:n
7346       \int_use:N \l_fp_input_b_decimal_int
7347     }
7348   \l_fp_input_b_decimal_int \l_fp_tmp_tl \scan_stop:
7349   \tex_advance:D \l_fp_input_b_exponent_int \c_one
7350 }
7351 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b: {
7352   \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
7353     \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
7354     \tex_expandafter:D \use_i:nn \tex_expandafter:D
7355       \fp_level_input_exponents_b:NNNNNNNNN
7356       \int_use:N \l_fp_input_a_integer_int
7357     \tex_expandafter:D \fp_level_input_exponents_b:
7358   \tex_fi:D
7359 }
7360 \cs_new_protected_nopar:Npn
7361   \fp_level_input_exponents_b:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7362   \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
```

```

7363 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7364 \tex_divide:D \l_fp_input_a_decimal_int \c_ten
7365 \tl_set:Nx \l_fp_tmp_tl
7366 {
7367   #9
7368   \tex_expandafter:D \use_none:n
7369   \int_use:N \l_fp_input_a_decimal_int
7370 }
7371 \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
7372 \tex_advance:D \l_fp_input_a_exponent_int \c_one
7373 }

```

`\fp_tmp:w` Used for output of results, cutting down on `\tex_expandafter:D`. This is just a placeholder definition.

```

7374 \cs_new_protected_nopar:Npn \fp_tmp:w #1#2 { }

```

116.5 Operations for fp variables

The format of `fp` variables is tightly defined, so that they can be read quickly by the internal code. The format is a single sign token, a single number, the decimal point, nine decimal numbers, an `e` and finally the exponent. This final part may vary in length. When stored, floating points will always be stored with a value in the integer position unless the number is zero.

`\fp_new:N` Fixed-points always have a value, and of course this has to be initialised globally.

```

\fp_new:c
7375 \cs_new_protected_nopar:Npn \fp_new:N #1 {
7376   \tl_new:N #1
7377   \tl_gset_eq:NN #1 \c_zero_fp
7378 }
7379 \cs_generate_variant:Nn \fp_new:N { c }

```

`\fp_const:Nn` A simple wrapper.

```

\fp_const:cn
7380 \cs_new_protected_nopar:Npn \fp_const:Nn #1#2 {
7381   \cs_if_free:NTF #1
7382   {
7383     \fp_new:N #1
7384     \fp_gset:Nn #1 {#2}
7385   }
7386   {
7387     \msg_kernel_error:nx { variable-already-defined }
7388     { \token_to_str:N #1 }
7389   }
7390 }
7391 \cs_generate_variant:Nn \fp_const:Nn { c }

```

```

\fp_zero:N Zeroing fixed-points is pretty obvious.
\fp_zero:c
\fp_gzero:N
\fp_gzero:c
7392 \cs_new_protected_nopar:Npn \fp_zero:N #1 {
7393   \tl_set_eq:NN #1 \c_zero_fp
7394 }
7395 \cs_new_protected_nopar:Npn \fp_gzero:N #1 {
7396   \tl_gset_eq:NN #1 \c_zero_fp
7397 }
7398 \cs_generate_variant:Nn \fp_zero:N { c }
7399 \cs_generate_variant:Nn \fp_gzero:N { c }

\fp_set:Nn To trap any input errors, a very simple version of the parser is run here. This will pick
\fp_set:cn up any invalid characters at this stage, saving issues later. The splitting approach is the
\fp_gset:Nn same as the more advanced function later.
\fp_gset:cn
\fp_set_aux:NNn
7400 \cs_new_protected_nopar:Npn \fp_set:Nn {
7401   \fp_set_aux:NNn \tl_set:Nn
7402 }
7403 \cs_new_protected_nopar:Npn \fp_gset:Nn {
7404   \fp_set_aux:NNn \tl_gset:Nn
7405 }
7406 \cs_new_protected_nopar:Npn \fp_set_aux:NNn #1#2#3 {
7407   \group_begin:
7408     \fp_split:Nn a {#3}
7409     \fp_standardise:NNNN
7410     \l_fp_input_a_sign_int
7411     \l_fp_input_a_integer_int
7412     \l_fp_input_a_decimal_int
7413     \l_fp_input_a_exponent_int
7414     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7415     \cs_set_protected_nopar:Npx \fp_tmp:w
7416     {
7417       \group_end:
7418       #1 \exp_not:N #2
7419       {
7420         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7421         -
7422         \tex_else:D
7423         +
7424         \tex_fi:D
7425         \int_use:N \l_fp_input_a_integer_int
7426         .
7427         \tex_expandafter:D \use_none:n
7428         \int_use:N \l_fp_input_a_decimal_int
7429         e
7430         \int_use:N \l_fp_input_a_exponent_int
7431       }
7432     }
7433     \fp_tmp:w
7434 }

```

```

7435 \cs_generate_variant:Nn \fp_set:Nn { c }
7436 \cs_generate_variant:Nn \fp_gset:Nn { c }

```

`\fp_set_from_dim:Nn` Here, dimensions are converted to fixed-points *via* a temporary variable. This ensures that they always convert as points. The code is then essentially the same as for `\fp_set:Nn`, but with the dimension passed so that it will be striped of the `pt` on the way through. The passage through a skip is used to remove any rubber part.

```

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn
\fp_set_from_dim_aux:NNn
\fp_set_from_dim_aux:w
    \l_fp_tmp_dim
    \l_fp_tmp_skip
7437 \cs_new_protected_nopar:Npn \fp_set_from_dim:Nn {
7438     \fp_set_from_dim_aux:NNn \tl_set:Nx
7439 }
7440 \cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn {
7441     \fp_set_from_dim_aux:NNn \tl_gset:Nx
7442 }
7443 \cs_new_protected_nopar:Npn \fp_set_from_dim_aux:NNn #1#2#3 {
7444     \group_begin:
7445     \l_fp_tmp_skip \etex_glueexpr:D #3 \scan_stop:
7446     \l_fp_tmp_dim \l_fp_tmp_skip
7447     \fp_split:Nn a
7448     {
7449         \tex_expandafter:D \fp_set_from_dim_aux:w
7450         \dim_use:N \l_fp_tmp_dim
7451     }
7452     \fp_standardise:NNNN
7453     \l_fp_input_a_sign_int
7454     \l_fp_input_a_integer_int
7455     \l_fp_input_a_decimal_int
7456     \l_fp_input_a_exponent_int
7457     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7458     \cs_set_protected_nopar:Npx \fp_tmp:w
7459     {
7460         \group_end:
7461         #1 \exp_not:N #2
7462         {
7463             \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7464             -
7465             \tex_else:D
7466             +
7467             \tex_fi:D
7468             \int_use:N \l_fp_input_a_integer_int
7469             .
7470             \tex_expandafter:D \use_none:n
7471             \int_use:N \l_fp_input_a_decimal_int
7472             e
7473             \int_use:N \l_fp_input_a_exponent_int
7474         }
7475     }
7476     \fp_tmp:w
7477 }
7478 \cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w {

```

```

7479 \cs_set_nopar:Npn \exp_not:N \fp_set_from_dim_aux:w
7480     ##1 \tl_to_str:n { pt } {##1}
7481 }
7482 \fp_set_from_dim_aux:w
7483 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
7484 \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
7485 \dim_new:N \l_fp_tmp_dim
7486 \skip_new:N \l_fp_tmp_skip

```

\fp_set_eq:NN Pretty simple, really.

```

\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
7487 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
7488 \cs_new_eq:NN \fp_set_eq:cN \tl_set_eq:cN
7489 \cs_new_eq:NN \fp_set_eq:Nc \tl_set_eq:Nc
7490 \cs_new_eq:NN \fp_set_eq:cc \tl_set_eq:cc
7491 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
7492 \cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
7493 \cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
7494 \cs_new_eq:NN \fp_gset_eq:cc \tl_gset_eq:cc

```

\fp_show:N Simple showing of the underlying variable.

```

\fp_show:c
7495 \cs_new_eq:NN \fp_show:N \tl_show:N
7496 \cs_new_eq:NN \fp_show:c \tl_show:c

```

\fp_use:N The idea of the \fp_use:N function to convert the stored value into something suitable for T_EX to use as a number in an expandable manner. The first step is to deal with the sign, then work out how big the input is.

```

\fp_use:c
\fp_use_aux:w
\fp_use_none:w
\fp_use_small:w
\fp_use_large:w
7497 \cs_new_nopar:Npn \fp_use:N #1 {
7498   \tex_expandafter:D \fp_use_aux:w #1 \q_stop
7499 }
7500 \cs_generate_variant:Nn \fp_use:N { c }
7501 \cs_new_nopar:Npn \fp_use_aux:w #1#2 e #3 \q_stop {
7502   \tex_if:D #1 -
7503   -
7504   \tex_fi:D
7505   \tex_ifnum:D #3 > \c_zero
7506   \tex_expandafter:D \fp_use_large:w
7507   \tex_else:D
7508   \tex_ifnum:D #3 < \c_zero
7509   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7510   \fp_use_small:w
7511   \tex_else:D
7512   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7513   \fp_use_none:w
7514   \tex_fi:D
7515   \tex_fi:D
7516   #2 e #3 \q_stop
7517 }

```

When the exponent is zero, the input is simply returned as output.

```
7518 \cs_new_nopar:Npn \fp_use_none:w #1 e #2 \q_stop {#1}
```

For small numbers (less than 1) the correct number of zeros have to be inserted, but the decimal point is easy.

```
7519 \cs_new_nopar:Npn \fp_use_small:w #1 . #2 e #3 \q_stop {
7520   0 .
7521   \prg_replicate:nn { -#3 - 1 } { 0 }
7522   #1#2
7523 }
```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```
7524 \cs_new_nopar:Npn \fp_use_large:w #1 . #2 e #3 \q_stop {
7525   \tex_ifnum:D #3 < \c_ten
7526     \tex_expandafter:D \fp_use_large_aux_i:w
7527   \tex_else:D
7528     \tex_expandafter:D \fp_use_large_aux_ii:w
7529   \tex_fi:D
7530   #1#2 e #3 \q_stop
7531 }
7532 \cs_new_nopar:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop {
7533   #1
7534   \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
7535 }
7536 \cs_new_nopar:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
7537 \cs_new_nopar:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop {
7538   #1#2 . #3
7539 }
7540 \cs_new_nopar:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop {
7541   #1#2#3 . #4
7542 }
7543 \cs_new_nopar:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop {
7544   #1#2#3#4 . #5
7545 }
7546 \cs_new_nopar:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7547   #1#2#3#4#5 . #6
7548 }
7549 \cs_new_nopar:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7550   #1#2#3#4#5#6 . #7
7551 }
7552 \cs_new_nopar:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7553   #1#2#3#4#6#7 . #8
7554 }
7555 \cs_new_nopar:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7556   #1#2#3#4#5#6#7#8 . #9
7557 }
7558 \cs_new_nopar:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
```

```

7559 \cs_new_nopar:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop {
7560   #1
7561   \prg_replicate:nn { #2 - 9 } { 0 }
7562   .
7563 }

```

116.6 Transferring to other types

The `\fp_use:N` function converts a floating point variable to a form that can be used by \TeX . Here, the functions are slightly different, as some information may be discarded.

`\fp_to_dim:N` A very simple wrapper.

`\fp_to_dim:c`

```

7564 \cs_new_nopar:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
7565 \cs_generate_variant:Nn \fp_to_dim:N { c }

```

`\fp_to_int:N` Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

`\fp_to_int:c`

```

\fp_to_int_aux:w
\fp_to_int_none:w
\fp_to_int_small:w
\fp_to_int_large:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux_1:w
\fp_to_int_large_aux_2:w
\fp_to_int_large_aux_3:w
\fp_to_int_large_aux_4:w
\fp_to_int_large_aux_5:w
\fp_to_int_large_aux_6:w
\fp_to_int_large_aux_7:w
\fp_to_int_large_aux_8:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux:nnn
\fp_to_int_large_aux_ii:w

```

```

7566 \cs_new_nopar:Npn \fp_to_int:N #1 {
7567   \tex_expandafter:D \fp_to_int_aux:w #1 \q_stop
7568 }
7569 \cs_generate_variant:Nn \fp_to_int:N { c }
7570 \cs_new_nopar:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop {
7571   \tex_if:D #1 -
7572   -
7573   \tex_fi:D
7574   \tex_ifnum:D #3 < \c_zero
7575   \tex_expandafter:D \fp_to_int_small:w
7576   \tex_else:D
7577   \tex_expandafter:D \fp_to_int_large:w
7578   \tex_fi:D
7579   #2 e #3 \q_stop
7580 }

```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```

7581 \cs_new_nopar:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop {
7582   \tex_ifnum:D #3 > \c_one
7583   \tex_else:D
7584     \tex_ifnum:D #1 < \c_five
7585     0
7586   \tex_else:D
7587     1
7588   \tex_fi:D
7589   \tex_fi:D
7590 }

```


For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```

7591 \cs_new_nopar:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop {
7592   \tex_ifnum:D #3 < \c_ten
7593     \tex_expandafter:D \fp_to_int_large_aux_i:w
7594   \tex_else:D
7595     \tex_expandafter:D \fp_to_int_large_aux_ii:w
7596   \tex_fi:D
7597   #1#2 e #3 \q_stop
7598 }
7599 \cs_new_nopar:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop {
7600   \use:c { fp_to_int_large_aux_ #3 :w } #2 \q_stop {#1}
7601 }
7602 \cs_new_nopar:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop {
7603   \fp_to_int_large_aux:nnn { #2 0 } {#1}
7604 }
7605 \cs_new_nopar:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop {
7606   \fp_to_int_large_aux:nnn { #3 00 } {#1#2}
7607 }
7608 \cs_new_nopar:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop {
7609   \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3}
7610 }
7611 \cs_new_nopar:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop {
7612   \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4}
7613 }
7614 \cs_new_nopar:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7615   \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5}
7616 }
7617 \cs_new_nopar:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7618   \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6}
7619 }
7620 \cs_new_nopar:cpn
7621 { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7622   \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7}
7623 }
7624 \cs_new_nopar:cpn
7625 { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7626   \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8}
7627 }
7628 \cs_new_nopar:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
7629 \cs_new_nopar:Npn \fp_to_int_large_aux:nnn #1#2#3 {
7630   \tex_ifnum:D #1 < \c_five_hundred_million
7631     #3#2
7632   \tex_else:D
7633     \tex_number:D \etex_numexpr:D #3#2 + 1 \scan_stop:
7634   \tex_fi:D
7635 }
7636 \cs_new_nopar:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop {
7637   #1

```

```

7638 \prg_replicate:nn { #2 - 9 } { 0 }
7639 }

```

`\fp_to_tl:N` Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

```

\fp_to_tl:c
\fp_to_tl_aux:w
\fp_to_tl_large:w
\fp_to_tl_large_aux_i:w
\fp_to_tl_large_aux_ii:w
\fp_to_tl_large_0:w
\fp_to_tl_large_1:w
\fp_to_tl_large_2:w
\fp_to_tl_large_3:w
\fp_to_tl_large_4:w
\fp_to_tl_large_5:w
\fp_to_tl_large_6:w
\fp_to_tl_large_7:w
\fp_to_tl_large_8:w
\fp_to_tl_large_8_aux:w
\fp_to_tl_large_9:w
\fp_to_tl_small:w
\fp_to_tl_small_one:w
\fp_to_tl_small_two:w
\fp_to_tl_small_aux:w
\fp_to_tl_large_zeros:NNNNNNNNN
\fp_to_tl_small_zeros:NNNNNNNNN
\fp_use_iix_ix:NNNNNNNNN
\fp_use_ix:NNNNNNNNN
\fp_use_i_to_vii:NNNNNNNNN
\fp_use_i_to_iix:NNNNNNNNN

```

```

7640 \cs_new_nopar:Npn \fp_to_tl:N #1 {
7641   \tex_expandafter:D \fp_to_tl_aux:w #1 \q_stop
7642 }
7643 \cs_generate_variant:Nn \fp_to_tl:N { c }
7644 \cs_new_nopar:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop {
7645   \tex_if:D #1 -
7646   -
7647   \tex_fi:D
7648   \tex_ifnum:D #3 < \c_zero
7649   \tex_expandafter:D \fp_to_tl_small:w
7650   \tex_else:D
7651   \tex_expandafter:D \fp_to_tl_large:w
7652   \tex_fi:D
7653   #2 e #3 \q_stop
7654 }

```

For ‘large’ numbers (exponent ≥ 0) there are two cases. For very large exponents (≥ 10) life is easy: apart from dropping extra zeros there is no work to do. On the other hand, for intermediate exponent values the decimal needs to be moved, then zeros can be dropped.

```

\fp_to_tl_large_zeros:NNNNNNNNN
\fp_to_tl_small_zeros:NNNNNNNNN
\fp_use_iix_ix:NNNNNNNNN
\fp_use_ix:NNNNNNNNN
\fp_use_i_to_vii:NNNNNNNNN
\fp_use_i_to_iix:NNNNNNNNN

```

```

7655 \cs_new_nopar:Npn \fp_to_tl_large:w #1 e #2 \q_stop {
7656   \tex_ifnum:D #2 < \c_ten
7657   \tex_expandafter:D \fp_to_tl_large_aux_i:w
7658   \tex_else:D
7659   \tex_expandafter:D \fp_to_tl_large_aux_ii:w
7660   \tex_fi:D
7661   #1 e #2 \q_stop
7662 }
7663 \cs_new_nopar:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop {
7664   \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop
7665 }
7666 \cs_new_nopar:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop {
7667   #1
7668   \fp_to_tl_large_zeros:NNNNNNNNN #2
7669   e #3
7670 }
7671 \cs_new_nopar:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop {
7672   #1
7673   \fp_to_tl_large_zeros:NNNNNNNNN #2
7674 }
7675 \cs_new_nopar:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop {
7676   #1#2
7677   \fp_to_tl_large_zeros:NNNNNNNNN #3 0
7678 }

```

```

7679 \cs_new_nopar:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop {
7680   #1#2#3
7681   \fp_to_tl_large_zeros:NNNNNNNN #4 00
7682 }
7683 \cs_new_nopar:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop {
7684   #1#2#3#4
7685   \fp_to_tl_large_zeros:NNNNNNNN #5 000
7686 }
7687 \cs_new_nopar:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop {
7688   #1#2#3#4#5
7689   \fp_to_tl_large_zeros:NNNNNNNN #6 0000
7690 }
7691 \cs_new_nopar:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop {
7692   #1#2#3#4#5#6
7693   \fp_to_tl_large_zeros:NNNNNNNN #7 00000
7694 }
7695 \cs_new_nopar:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop {
7696   #1#2#3#4#5#6#7
7697   \fp_to_tl_large_zeros:NNNNNNNN #8 000000
7698 }
7699 \cs_new_nopar:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop {
7700   #1#2#3#4#5#6#7#8
7701   \fp_to_tl_large_zeros:NNNNNNNN #9 0000000
7702 }
7703 \cs_new_nopar:cpn { fp_to_tl_large_8:w } #1 . {
7704   #1
7705   \use:c { fp_to_tl_large_8_aux:w }
7706 }
7707 \cs_new_nopar:cpn
7708 { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7709   #1#2#3#4#5#6#7#8
7710   \fp_to_tl_large_zeros:NNNNNNNN #9 00000000
7711 }
7712 \cs_new_nopar:cpn { fp_to_tl_large_9:w } #1 . #2 \q_stop {#1#2}

```

Dealing with small numbers is a bit more complex as there has to be rounding. This makes life rather awkward, as there need to be a series of tests and calculations, as things cannot be stored in an expandable system.

```

7713 \cs_new_nopar:Npn \fp_to_tl_small:w #1 e #2 \q_stop {
7714   \tex_ifnum:D #2 = \c_minus_one
7715   \tex_expandafter:D \fp_to_tl_small_one:w
7716   \tex_else:D
7717     \tex_ifnum:D #2 = -\c_two
7718     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7719     \fp_to_tl_small_two:w
7720     \tex_else:D
7721     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7722     \fp_to_tl_small_aux:w
7723   \tex_fi:D

```

```

7724 \tex_fi:D
7725 #1 e #2 \q_stop
7726 }
7727 \cs_new_nopar:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop {
7728 \tex_ifnum:D \fp_use_ix:NNNNNNNN #2 > \c_four
7729 \tex_ifnum:D
7730 \etex_numexpr:D #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
7731 < \c_one_thousand_million
7732 0.
7733 \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNN
7734 \tex_number:D
7735 \etex_numexpr:D
7736 #1 \fp_use_i_to_iix:NNNNNNNN #2 + 1
7737 \scan_stop:
7738 \tex_else:D
7739 1
7740 \tex_fi:D
7741 \tex_else:D
7742 0. #1
7743 \fp_to_tl_small_zeros:NNNNNNNN #2
7744 \tex_fi:D
7745 }
7746 \cs_new_nopar:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop {
7747 \tex_ifnum:D \fp_use_iix_ix:NNNNNNNN #2 > \c_forty_four
7748 \tex_ifnum:D
7749 \etex_numexpr:D #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
7750 < \c_one_thousand_million
7751 0.0
7752 \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNN
7753 \tex_number:D
7754 \etex_numexpr:D
7755 #1 \fp_use_i_to_vii:NNNNNNNN #2 0 + \c_ten
7756 \scan_stop:
7757 \tex_else:D
7758 0.1
7759 \tex_fi:D
7760 \tex_else:D
7761 0.0
7762 #1
7763 \fp_to_tl_small_zeros:NNNNNNNN #2
7764 \tex_fi:D
7765 }
7766 \cs_new_nopar:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop {
7767 #1
7768 \fp_to_tl_large_zeros:NNNNNNNN #2
7769 e #3
7770 }

```

Rather than a complex recursion, the tests for finding trailing zeros are written out

long-hand. The difference between the two is only the need for a decimal marker.

```

7771 \cs_new_nopar:Npn \fp_to_tl_large_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7772   \tex_ifnum:D #9 = \c_zero
7773   \tex_ifnum:D #8 = \c_zero
7774   \tex_ifnum:D #7 = \c_zero
7775   \tex_ifnum:D #6 = \c_zero
7776   \tex_ifnum:D #5 = \c_zero
7777   \tex_ifnum:D #4 = \c_zero
7778   \tex_ifnum:D #3 = \c_zero
7779   \tex_ifnum:D #2 = \c_zero
7780   \tex_ifnum:D #1 = \c_zero
7781   \tex_else:D
7782   . #1
7783   \tex_fi:D
7784   \tex_else:D
7785   . #1#2
7786   \tex_fi:D
7787   \tex_else:D
7788   . #1#2#3
7789   \tex_fi:D
7790   \tex_else:D
7791   . #1#2#3#4
7792   \tex_fi:D
7793   \tex_else:D
7794   . #1#2#3#4#5
7795   \tex_fi:D
7796   \tex_else:D
7797   . #1#2#3#4#5#6
7798   \tex_fi:D
7799   \tex_else:D
7800   . #1#2#3#4#5#6#7
7801   \tex_fi:D
7802   \tex_else:D
7803   . #1#2#3#4#5#6#7#8
7804   \tex_fi:D
7805   \tex_else:D
7806   . #1#2#3#4#5#6#7#8#9
7807   \tex_fi:D
7808 }
7809 \cs_new_nopar:Npn \fp_to_tl_small_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7810   \tex_ifnum:D #9 = \c_zero
7811   \tex_ifnum:D #8 = \c_zero
7812   \tex_ifnum:D #7 = \c_zero
7813   \tex_ifnum:D #6 = \c_zero
7814   \tex_ifnum:D #5 = \c_zero
7815   \tex_ifnum:D #4 = \c_zero
7816   \tex_ifnum:D #3 = \c_zero
7817   \tex_ifnum:D #2 = \c_zero
7818   \tex_ifnum:D #1 = \c_zero

```

```

7819         \tex_else:D
7820         #1
7821         \tex_fi:D
7822         \tex_else:D
7823         #1#2
7824         \tex_fi:D
7825         \tex_else:D
7826         #1#2#3
7827         \tex_fi:D
7828         \tex_else:D
7829         #1#2#3#4
7830         \tex_fi:D
7831         \tex_else:D
7832         #1#2#3#4#5
7833         \tex_fi:D
7834         \tex_else:D
7835         #1#2#3#4#5#6
7836         \tex_fi:D
7837         \tex_else:D
7838         #1#2#3#4#5#6#7
7839         \tex_fi:D
7840         \tex_else:D
7841         #1#2#3#4#5#6#7#8
7842         \tex_fi:D
7843         \tex_else:D
7844         #1#2#3#4#5#6#7#8#9
7845         \tex_fi:D
7846     }

```

Some quick ‘return a few’ functions.

```

7847 \cs_new_nopar:Npn \fp_use_iix_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
7848 \cs_new_nopar:Npn \fp_use_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
7849 \cs_new_nopar:Npn \fp_use_i_to_vii:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7850     #1#2#3#4#5#6#7
7851 }
7852 \cs_new_nopar:Npn \fp_use_i_to_iix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7853     #1#2#3#4#5#6#7#8
7854 }

```

116.7 Rounding numbers

The results may well need to be rounded. A couple of related functions to do this for a stored value.

<pre> \fp_round_figures:Nn \fp_round_figures:cn \fp_ground_figures:Nn \fp_ground_figures:cn \fp_round_figures_aux:NNn </pre>	<p>Rounding to figures needs only an adjustment to the target by one (as the target is in decimal places).</p>
--	--

```

7855 \cs_new_protected_nopar:Npn \fp_round_figures:Nn {
7856   \fp_round_figures_aux:NNn \tl_set:Nn
7857 }
7858 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
7859 \cs_new_protected_nopar:Npn \fp_ground_figures:Nn {
7860   \fp_round_figures_aux:NNn \tl_gset:Nn
7861 }
7862 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
7863 \cs_new_protected_nopar:Npn \fp_round_figures_aux:NNn #1#2#3 {
7864   \group_begin:
7865     \fp_read:N #2
7866     \int_set:Nn \l_fp_round_target_int { #3 - 1 }
7867     \tex_ifnum:D \l_fp_round_target_int < \c_ten
7868       \tex_expandafter:D \fp_round:
7869     \tex_fi:D
7870     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7871     \cs_set_protected_nopar:Npx \fp_tmp:w
7872     {
7873       \group_end:
7874       #1 \exp_not:N #2
7875       {
7876         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7877         -
7878         \tex_else:D
7879         +
7880         \tex_fi:D
7881         \int_use:N \l_fp_input_a_integer_int
7882         .
7883         \tex_expandafter:D \use_none:n
7884         \int_use:N \l_fp_input_a_decimal_int
7885         e
7886         \int_use:N \l_fp_input_a_exponent_int
7887       }
7888     }
7889     \fp_tmp:w
7890 }

```

\fp_round_places:Nn

\fp_round_places:cn

\fp_ground_places:Nn

\fp_ground_places:cn

\fp_round_places_aux:NNn

Rounding to places needs an adjustment for the exponent value, which will mean that everything should be correct.

```

7891 \cs_new_protected_nopar:Npn \fp_round_places:Nn {
7892   \fp_round_places_aux:NNn \tl_set:Nn
7893 }
7894 \cs_generate_variant:Nn \fp_round_places:Nn { c }
7895 \cs_new_protected_nopar:Npn \fp_ground_places:Nn {
7896   \fp_round_places_aux:NNn \tl_gset:Nn
7897 }
7898 \cs_generate_variant:Nn \fp_ground_places:Nn { c }
7899 \cs_new_protected_nopar:Npn \fp_round_places_aux:NNn #1#2#3 {
7900   \group_begin:

```

```

7901 \fp_read:N #2
7902 \int_set:Nn \l_fp_round_target_int
7903 { #3 + \l_fp_input_a_exponent_int }
7904 \tex_ifnum:D \l_fp_round_target_int < \c_ten
7905 \tex_expandafter:D \fp_round:
7906 \tex_fi:D
7907 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7908 \cs_set_protected_nopar:Npx \fp_tmp:w
7909 {
7910 \group_end:
7911 #1 \exp_not:N #2
7912 {
7913 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7914 -
7915 \tex_else:D
7916 +
7917 \tex_fi:D
7918 \int_use:N \l_fp_input_a_integer_int
7919 .
7920 \tex_expandafter:D \use_none:n
7921 \int_use:N \l_fp_input_a_decimal_int
7922 e
7923 \int_use:N \l_fp_input_a_exponent_int
7924 }
7925 }
7926 \fp_tmp:w
7927 }

```

`\fp_round:` The rounding approach is the same for decimal places and significant figures. There are
`\fp_round_aux:NNNNNNNN` always nine decimal digits to round, so the code can be written to account for this. The
`\fp_round_loop:N` basic logic is simply to find the rounding, track any carry digit and move along. At the
end of the loop there is a possible shuffle if the integer part has become 10.

```

7928 \cs_new_protected_nopar:Npn \fp_round: {
7929 \bool_set_false:N \l_fp_round_carry_bool
7930 \l_fp_round_position_int \c_eight
7931 \tl_clear:N \l_fp_round_decimal_tl
7932 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7933 \tex_expandafter:D \use_i:nn \tex_expandafter:D
7934 \fp_round_aux:NNNNNNNN \int_use:N \l_fp_input_a_decimal_int
7935 }
7936 \cs_new_protected_nopar:Npn \fp_round_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7937 \fp_round_loop:N #9#8#7#6#5#4#3#2#1
7938 \bool_if:NT \l_fp_round_carry_bool
7939 { \tex_advance:D \l_fp_input_a_integer_int \c_one }
7940 \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
7941 \tex_ifnum:D \l_fp_input_a_integer_int < \c_ten
7942 \tex_else:D
7943 \l_fp_input_a_integer_int \c_one
7944 \tex_divide:D \l_fp_input_a_decimal_int \c_ten

```



```

7945     \tex_advance:D \l_fp_input_a_exponent_int \c_one
7946   \tex_fi:D
7947 }
7948 \cs_new_protected_nopar:Npn \fp_round_loop:N #1 {
7949   \tex_ifnum:D \l_fp_round_position_int < \l_fp_round_target_int
7950     \bool_if:NTF \l_fp_round_carry_bool
7951       { \l_fp_tmp_int \etex_numexpr:D #1 + \c_one \scan_stop: }
7952       { \l_fp_tmp_int \etex_numexpr:D #1 \scan_stop: }
7953   \tex_ifnum:D \l_fp_tmp_int = \c_ten
7954     \l_fp_tmp_int \c_zero
7955   \tex_else:D
7956     \bool_set_false:N \l_fp_round_carry_bool
7957   \tex_fi:D
7958   \tl_set:Nx \l_fp_round_decimal_tl
7959     { \int_use:N \l_fp_tmp_int \l_fp_round_decimal_tl }
7960 \tex_else:D
7961   \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
7962   \tex_ifnum:D \l_fp_round_position_int = \l_fp_round_target_int
7963     \tex_ifnum:D #1 > \c_four
7964       \bool_set_true:N \l_fp_round_carry_bool
7965   \tex_fi:D
7966   \tex_fi:D
7967 \tex_fi:D
7968 \tex_advance:D \l_fp_round_position_int \c_minus_one
7969 \tex_ifnum:D \l_fp_round_position_int > \c_minus_one
7970   \tex_expandafter:D \fp_round_loop:N
7971 \tex_fi:D
7972 }

```

116.8 Unary functions

`\fp_abs:N` Setting the absolute value is easy: read the value, ignore the sign, return the result.

```

\fp_abs:c
\fp_gabs:N
\fp_gabs:c
\fp_abs_aux:NN
7973 \cs_new_protected_nopar:Npn \fp_abs:N {
7974   \fp_abs_aux:NN \tl_set:Nn
7975 }
7976 \cs_new_protected_nopar:Npn \fp_gabs:N {
7977   \fp_abs_aux:NN \tl_gset:Nn
7978 }
7979 \cs_generate_variant:Nn \fp_abs:N { c }
7980 \cs_generate_variant:Nn \fp_gabs:N { c }
7981 \cs_new_protected_nopar:Npn \fp_abs_aux:NN #1#2 {
7982   \group_begin:
7983     \fp_read:N #2
7984     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7985     \cs_set_protected_nopar:Npx \fp_tmp:w
7986     {
7987       \group_end:
7988       #1 \exp_not:N #2

```

```

7989         {
7990         +
7991         \int_use:N \l_fp_input_a_integer_int
7992         .
7993         \tex_expandafter:D \use_none:n
7994         \int_use:N \l_fp_input_a_decimal_int
7995         e
7996         \int_use:N \l_fp_input_a_exponent_int
7997         }
7998     }
7999     \fp_tmp:w
8000 }

```

\fp_neg:N Just a bit more complex: read the input, reverse the sign and output the result.

```

\fp_neg:c
\fp_gneg:N 8001 \cs_new_protected_nopar:Npn \fp_neg:N {
\fp_gneg:c 8002 \fp_neg_aux:NN \tl_set:Nn
\fp_neg:NN 8003 }
8004 \cs_new_protected_nopar:Npn \fp_gneg:N {
8005 \fp_neg_aux:NN \tl_gset:Nn
8006 }
8007 \cs_generate_variant:Nn \fp_neg:N { c }
8008 \cs_generate_variant:Nn \fp_gneg:N { c }
8009 \cs_new_protected_nopar:Npn \fp_neg_aux:NN #1#2 {
8010 \group_begin:
8011 \fp_read:N #2
8012 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8013 \tl_set:Nx \l_fp_tmp_tl
8014 {
8015 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8016 +
8017 \tex_else:D
8018 -
8019 \tex_fi:D
8020 \int_use:N \l_fp_input_a_integer_int
8021 .
8022 \tex_expandafter:D \use_none:n
8023 \int_use:N \l_fp_input_a_decimal_int
8024 e
8025 \int_use:N \l_fp_input_a_exponent_int
8026 }
8027 \tex_expandafter:D \group_end: \tex_expandafter:D
8028 #1 \tex_expandafter:D #2 \tex_expandafter:D { \l_fp_tmp_tl }
8029 }

```

116.9 Basic arithmetic

`\fp_add:Nn` The various addition functions are simply different ways to call the single master function
`\fp_add:cn` below. This pattern is repeated for the other arithmetic functions.
`\fp_gadd:Nn`
`\fp_gadd:cn`

```
8030 \cs_new_protected_nopar:Npn \fp_add:Nn {
8031   \fp_add_aux:NNn \tl_set:Nn
8032 }
8033 \cs_new_protected_nopar:Npn \fp_gadd:Nn {
8034   \fp_add_aux:NNn \tl_gset:Nn
8035 }
8036 \cs_generate_variant:Nn \fp_add:Nn { c }
8037 \cs_generate_variant:Nn \fp_gadd:Nn { c }
```

`\fp_add_aux:NNn`
`\fp_add_core:`
`\fp_add_sum:`
`\fp_add_difference:`

Addition takes place using one of two paths. If the signs of the two parts are the same, they are simply combined. On the other hand, if the signs are different the calculation finds this difference.

```
8038 \cs_new_protected_nopar:Npn \fp_add_aux:NNn #1#2#3 {
8039   \group_begin:
8040     \fp_read:N #2
8041     \fp_split:Nn b {#3}
8042     \fp_standardise:NNNN
8043     \l_fp_input_b_sign_int
8044     \l_fp_input_b_integer_int
8045     \l_fp_input_b_decimal_int
8046     \l_fp_input_b_exponent_int
8047     \fp_add_core:
8048     \fp_tmp:w #1#2
8049   }
8050   \cs_new_protected_nopar:Npn \fp_add_core: {
8051     \fp_level_input_exponents:
8052     \tex_ifnum:D
8053       \etex_numexpr:D
8054         \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8055       \scan_stop:
8056       > \c_zero
8057       \tex_expandafter:D \fp_add_sum:
8058     \tex_else:D
8059       \tex_expandafter:D \fp_add_difference:
8060     \tex_fi:D
8061     \l_fp_output_exponent_int \l_fp_input_a_exponent_int
8062     \fp_standardise:NNNN
8063     \l_fp_output_sign_int
8064     \l_fp_output_integer_int
8065     \l_fp_output_decimal_int
8066     \l_fp_output_exponent_int
8067     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8068     {
8069       \group_end:
```

```

8070     ##1 ##2
8071     {
8072         \tex_ifnum:D \l_fp_output_sign_int < \c_zero
8073         -
8074         \tex_else:D
8075         +
8076         \tex_fi:D
8077         \int_use:N \l_fp_output_integer_int
8078         .
8079         \tex_expandafter:D \use_none:n
8080         \tex_number:D \etex_numexpr:D
8081             \l_fp_output_decimal_int + \c_one_thousand_million
8082         e
8083         \int_use:N \l_fp_output_exponent_int
8084     }
8085 }
8086 }

```

Finding the sum of two numbers is trivially easy.

```

8087 \cs_new_protected_nopar:Npn \fp_add_sum: {
8088     \l_fp_output_sign_int \l_fp_input_a_sign_int
8089     \l_fp_output_integer_int
8090     \etex_numexpr:D
8091         \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
8092     \scan_stop:
8093     \l_fp_output_decimal_int
8094     \etex_numexpr:D
8095         \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
8096     \scan_stop:
8097     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
8098     \tex_else:D
8099         \tex_advance:D \l_fp_output_integer_int \c_one
8100         \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
8101     \tex_fi:D
8102 }

```

When the signs of the two parts of the input are different, the absolute difference is worked out first. There is then a calculation to see which way around everything has worked out, so that the final sign is correct. The difference might also give a zero result with a negative sign, which is reversed as zero is regarded as positive.

```

8103 \cs_new_protected_nopar:Npn \fp_add_difference: {
8104     \l_fp_output_integer_int
8105     \etex_numexpr:D
8106         \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
8107     \scan_stop:
8108     \l_fp_output_decimal_int
8109     \etex_numexpr:D
8110         \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int

```

```

8111 \scan_stop:
8112 \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
8113 \tex_advance:D \l_fp_output_integer_int \c_minus_one
8114 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8115 \tex_fi:D
8116 \tex_ifnum:D \l_fp_output_integer_int < \c_zero
8117 \l_fp_output_sign_int \l_fp_input_b_sign_int
8118 \tex_ifnum:D \l_fp_output_decimal_int = \c_zero
8119 \l_fp_output_integer_int -\l_fp_output_integer_int
8120 \tex_else:D
8121 \l_fp_output_decimal_int
8122 \etex_numexpr:D
8123 \c_one_thousand_million - \l_fp_output_decimal_int
8124 \scan_stop:
8125 \l_fp_output_integer_int
8126 \etex_numexpr:D
8127 - \l_fp_output_integer_int - \c_one
8128 \scan_stop:
8129 \tex_fi:D
8130 \tex_else:D
8131 \l_fp_output_sign_int \l_fp_input_a_sign_int
8132 \tex_fi:D
8133 }

```

`\fp_sub:Nn` Subtraction is essentially the same as addition, but with the sign of the second component
`\fp_sub:cn` reversed. Thus the core of the two function groups is the same, with just a little set up
`\fp_gsub:Nn` here.
`\fp_gsub:cn`
`\fp_sub_aux:NNn`

```

8134 \cs_new_protected_nopar:Npn \fp_sub:Nn {
8135 \fp_sub_aux:NNn \tl_set:Nn
8136 }
8137 \cs_new_protected_nopar:Npn \fp_gsub:Nn {
8138 \fp_sub_aux:NNn \tl_gset:Nn
8139 }
8140 \cs_generate_variant:Nn \fp_sub:Nn { c }
8141 \cs_generate_variant:Nn \fp_gsub:Nn { c }
8142 \cs_new_protected_nopar:Npn \fp_sub_aux:NNn #1#2#3 {
8143 \group_begin:
8144 \fp_read:N #2
8145 \fp_split:Nn b {#3}
8146 \fp_standardise:NNNN
8147 \l_fp_input_b_sign_int
8148 \l_fp_input_b_integer_int
8149 \l_fp_input_b_decimal_int
8150 \l_fp_input_b_exponent_int
8151 \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
8152 \fp_add_core:
8153 \fp_tmp:w #1#2
8154 }

```

```

\fp_mul:Nn The pattern is much the same for multiplication.
\fp_mul:cn
\fp_gmul:Nn 8155 \cs_new_protected_nopar:Npn \fp_mul:Nn {
\fp_gmul:cn 8156 \fp_mul_aux:NNn \tl_set:Nn
8157 }
\fp_mul_aux:NNn 8158 \cs_new_protected_nopar:Npn \fp_gmul:Nn {
\fp_mul_internal: 8159 \fp_mul_aux:NNn \tl_gset:Nn
\fp_mul_split:NNNN 8160 }
\fp_mul_split:w 8161 \cs_generate_variant:Nn \fp_mul:Nn { c }
\fp_mul_end_level: 8162 \cs_generate_variant:Nn \fp_gmul:Nn { c }
\fp_mul_end_level:NNNNNNNN

```

The approach to multiplication is as follows. First, the two numbers are split into blocks of three digits. These are then multiplied together to find products for each group of three output digits. This is all written out in full for speed reasons. Between each block of three digits in the output, there is a carry step. The very lowest digits are not calculated, while

```

8163 \cs_new_protected_nopar:Npn \fp_mul_aux:NNn #1#2#3 {
8164 \group_begin:
8165 \fp_read:N #2
8166 \fp_split:Nn b {#3}
8167 \fp_standardise:NNNN
8168 \l_fp_input_b_sign_int
8169 \l_fp_input_b_integer_int
8170 \l_fp_input_b_decimal_int
8171 \l_fp_input_b_exponent_int
8172 \fp_mul_internal:
8173 \l_fp_output_exponent_int
8174 \etex_numexpr:D
8175 \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
8176 \scan_stop:
8177 \fp_standardise:NNNN
8178 \l_fp_output_sign_int
8179 \l_fp_output_integer_int
8180 \l_fp_output_decimal_int
8181 \l_fp_output_exponent_int
8182 \cs_set_protected_nopar:Npx \fp_tmp:w
8183 {
8184 \group_end:
8185 #1 \exp_not:N #2
8186 {
8187 \tex_ifnum:D
8188 \etex_numexpr:D
8189 \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8190 < \c_zero
8191 \tex_ifnum:D
8192 \etex_numexpr:D
8193 \l_fp_output_integer_int + \l_fp_output_decimal_int
8194 = \c_zero
8195 +
8196 \tex_else:D

```

```

8197         -
8198         \tex_fi:D
8199         \tex_else:D
8200         +
8201         \tex_fi:D
8202         \int_use:N \l_fp_output_integer_int
8203         .
8204         \tex_expandafter:D \use_none:n
8205         \tex_number:D \etex_numexpr:D
8206         \l_fp_output_decimal_int + \c_one_thousand_million
8207         e
8208         \int_use:N \l_fp_output_exponent_int
8209     }
8210 }
8211 \fp_tmp:w
8212 }

```

Done separately so that the internal use is a bit easier.

```

8213 \cs_new_protected_nopar:Npn \fp_mul_internal: {
8214   \fp_mul_split:NNNN \l_fp_input_a_decimal_int
8215     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8216   \fp_mul_split:NNNN \l_fp_input_b_decimal_int
8217     \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8218   \l_fp_mul_output_int \c_zero
8219   \tl_clear:N \l_fp_mul_output_tl
8220   \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_iii_int
8221   \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_ii_int
8222   \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_mul_b_i_int
8223   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8224   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
8225   \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_ii_int
8226   \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_mul_b_i_int
8227   \fp_mul_product:NN \l_fp_mul_a_iii_int    \l_fp_input_b_integer_int
8228   \fp_mul_end_level:
8229   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
8230   \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_mul_b_i_int
8231   \fp_mul_product:NN \l_fp_mul_a_ii_int     \l_fp_input_b_integer_int
8232   \fp_mul_end_level:
8233   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
8234   \fp_mul_product:NN \l_fp_mul_a_i_int      \l_fp_input_b_integer_int
8235   \fp_mul_end_level:
8236   \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
8237   \tl_clear:N \l_fp_mul_output_tl
8238   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
8239   \fp_mul_end_level:
8240   \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
8241 }

```

The split works by making a 10 digit number, from which the first digit can then be dropped using a delimited argument. The groups of three digits are then assigned to the

various parts of the input: notice that ##9 contains the last two digits of the smallest part of the input.

```

8242 \cs_new_protected_nopar:Npn \fp_mul_split:NNNN #1#2#3#4 {
8243   \tex_advance:D #1 \c_one_thousand_million
8244   \cs_set_protected_nopar:Npn \fp_mul_split_aux:w
8245     ##1##2##3##4##5##6##7##8##9 \q_stop {
8246     #2 ##2##3##4 \scan_stop:
8247     #3 ##5##6##7 \scan_stop:
8248     #4 ##8##9 \scan_stop:
8249   }
8250   \tex_expandafter:D \fp_mul_split_aux:w \int_use:N #1 \q_stop
8251   \tex_advance:D #1 -\c_one_thousand_million
8252 }
8253 \cs_new_protected_nopar:Npn \fp_mul_product:NN #1#2 {
8254   \l_fp_mul_output_int
8255   \etex_numexpr:D \l_fp_mul_output_int + #1 * #2 \scan_stop:
8256 }

```

At the end of each output group of three, there is a transfer of information so that there is no danger of an overflow. This is done by expansion to keep the number of calculations down.

```

8257 \cs_new_protected_nopar:Npn \fp_mul_end_level: {
8258   \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
8259   \tex_expandafter:D \use_i:nn \tex_expandafter:D
8260   \fp_mul_end_level:NNNNNNNNN \int_use:N \l_fp_mul_output_int
8261 }
8262 \cs_new_protected_nopar:Npn \fp_mul_end_level:NNNNNNNNN
8263   #1#2#3#4#5#6#7#8#9 {
8264   \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
8265   \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
8266 }

```

\fp_div:Nn The pattern is much the same for multiplication.

```

\fp_div:cn
\fp_gdiv:Nn
\fp_gdiv:cn
\fp_div_aux:NNn
\fp_div_internal:
  \fp_div_loop:
\fp_div_divide:
\fp_div_divide_aux:
  \fp_div_store:
\fp_div_store_integer:
\fp_div_store_decimal:

```

```

8267 \cs_new_protected_nopar:Npn \fp_div:Nn {
8268   \fp_div_aux:NNn \tl_set:Nn
8269 }
8270 \cs_new_protected_nopar:Npn \fp_gdiv:Nn {
8271   \fp_div_aux:NNn \tl_gset:Nn
8272 }
8273 \cs_generate_variant:Nn \fp_div:Nn { c }
8274 \cs_generate_variant:Nn \fp_gdiv:Nn { c }

```

Division proper starts with a couple of tests. If the denominator is zero then a error is issued. On the other hand, if the numerator is zero then the result must be 0.0 and can be given with no further work.

```

8275 \cs_new_protected_nopar:Npn \fp_div_aux:NNn #1#2#3 {

```



```

8276 \group_begin:
8277   \fp_read:N #2
8278   \fp_split:Nn b {#3}
8279   \fp_standardise:NNNN
8280   \l_fp_input_b_sign_int
8281   \l_fp_input_b_integer_int
8282   \l_fp_input_b_decimal_int
8283   \l_fp_input_b_exponent_int
8284   \tex_ifnum:D
8285   \etex_numexpr:D
8286     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
8287     = \c_zero
8288   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8289   {
8290     \group_end:
8291     #1 \exp_not:N #2 { \c_undefined_fp }
8292   }
8293   \tex_else:D
8294   \tex_ifnum:D
8295   \etex_numexpr:D
8296     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8297     = \c_zero
8298   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8299   {
8300     \group_end:
8301     #1 \exp_not:N #2 { \c_zero_fp }
8302   }
8303   \tex_else:D
8304   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8305   \fp_div_internal:
8306   \tex_fi:D
8307   \tex_fi:D
8308   \fp_tmp:w #1#2
8309 }

```

The main division algorithm works by finding how many times **b** can be removed from **a**, storing the result and doing the subtraction. Input **a** is then multiplied by 10, and the process is repeated. The looping ends either when there is nothing left of **a** (*i.e.* an exact result) or when the code reaches the ninth decimal place. Most of the process takes place in the loop function below.

```

8310 \cs_new_protected_nopar:Npn \fp_div_internal: {
8311   \l_fp_output_integer_int \c_zero
8312   \l_fp_output_decimal_int \c_zero
8313   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
8314   \l_fp_div_offset_int \c_one_hundred_million
8315   \fp_div_loop:
8316   \l_fp_output_exponent_int
8317   \etex_numexpr:D
8318     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int

```

```

8319 \scan_stop:
8320 \fp_standardise:NNNN
8321 \l_fp_output_sign_int
8322 \l_fp_output_integer_int
8323 \l_fp_output_decimal_int
8324 \l_fp_output_exponent_int
8325 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8326 {
8327 \group_end:
8328 ##1 ##2
8329 {
8330 \tex_ifnum:D
8331 \etex_numexpr:D
8332 \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8333 < \c_zero
8334 \tex_ifnum:D
8335 \etex_numexpr:D
8336 \l_fp_output_integer_int + \l_fp_output_decimal_int
8337 = \c_zero
8338 +
8339 \tex_else:D
8340 -
8341 \tex_if:D
8342 \tex_else:D
8343 +
8344 \tex_if:D
8345 \int_use:N \l_fp_output_integer_int
8346 .
8347 \tex_expandafter:D \use_none:n
8348 \tex_number:D \etex_numexpr:D
8349 \l_fp_output_decimal_int + \c_one_thousand_million
8350 \scan_stop:
8351 e
8352 \int_use:N \l_fp_output_exponent_int
8353 }
8354 }
8355 }

```

The main loop implements the approach described above. The storing function is done as a function so that the integer and decimal parts can be done separately but rapidly.

```

8356 \cs_new_protected_nopar:Npn \fp_div_loop: {
8357 \l_fp_count_int \c_zero
8358 \fp_div_divide:
8359 \fp_div_store:
8360 \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8361 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8362 \tex_expandafter:D \fp_div_loop_step:w
8363 \int_use:N \l_fp_input_a_decimal_int \q_stop
8364 \tex_ifnum:D

```

```

8365 \etex_numexpr:D
8366 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8367 > \c_zero
8368 \tex_ifnum:D \l_fp_div_offset_int > \c_zero
8369 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8370 \fp_div_loop:
8371 \tex_fi:D
8372 \tex_fi:D
8373 }

```

Checking to see if the numerator can be divided needs quite an involved check. Either the integer part has to be bigger for the numerator or, if it is not smaller then the decimal part of the numerator must not be smaller than that of the denominator. Once the test is right the rest is much as elsewhere.

```

8374 \cs_new_protected_nopar:Npn \fp_div_divide: {
8375 \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
8376 \tex_expandafter:D \fp_div_divide_aux:
8377 \tex_else:D
8378 \tex_ifnum:D \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
8379 \tex_else:D
8380 \tex_ifnum:D
8381 \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
8382 \tex_else:D
8383 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8384 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8385 \tex_expandafter:D \fp_div_divide_aux:
8386 \tex_fi:D
8387 \tex_fi:D
8388 \tex_fi:D
8389 }
8390 \cs_new_protected_nopar:Npn \fp_div_divide_aux: {
8391 \tex_advance:D \l_fp_count_int \c_one
8392 \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
8393 \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int
8394 \tex_ifnum:D \l_fp_input_a_decimal_int < \c_zero
8395 \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8396 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8397 \tex_fi:D
8398 \fp_div_divide:
8399 }

```

Storing the number of each division is done differently for the integer and decimal. The integer is easy and a one-off, while the decimal also needs to account for the position of the digit to store.

```

8400 \cs_new_protected_nopar:Npn \fp_div_store: { }
8401 \cs_new_protected_nopar:Npn \fp_div_store_integer: {
8402 \l_fp_output_integer_int \l_fp_count_int
8403 \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:

```

```

8404 }
8405 \cs_new_protected_nopar:Npn \fp_div_store_decimal: {
8406   \l_fp_output_decimal_int
8407   \etex_numexpr:D
8408     \l_fp_output_decimal_int +
8409     \l_fp_count_int * \l_fp_div_offset_int
8410   \scan_stop:
8411   \tex_divide:D \l_fp_div_offset_int \c_ten
8412 }
8413 \cs_new_protected_nopar:Npn
8414   \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop {
8415   \l_fp_input_a_integer_int
8416   \etex_numexpr:D
8417     #2 + \l_fp_input_a_integer_int
8418   \scan_stop:
8419   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
8420 }

```

116.10 Arithmetic for internal use

For the more complex functions, it is only possible to deliver reliable 10 digit accuracy if the internal calculations are carried out to a higher degree of precision. This is done using a second set of functions so that the ‘user’ versions are not slowed down. These versions are also focussed on the needs of internal calculations. No error checking, sign checking or exponent levelling is done. For addition and subtraction, the arguments are:

- Integer part of input a.
- Decimal part of input a.
- Additional decimal part of input a.
- Integer part of input b.
- Decimal part of input b.
- Additional decimal part of input b.
- Integer part of output.
- Decimal part of output.
- Additional decimal part of output.

The situation for multiplication and division is a little different as they only deal with the decimal part.

`\fp_add:NNNNNNNN` The internal sum is always exactly that: it is always a sum and there is no sign check.

```

8421 \cs_new_protected_nopar:Npn \fp_add:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8422   #7 \etex_numexpr:D #1 + #4 \scan_stop:
8423   #8 \etex_numexpr:D #2 + #5 \scan_stop:
8424   #9 \etex_numexpr:D #3 + #6 \scan_stop:
8425   \tex_ifnum:D #9 < \c_one_thousand_million
8426   \tex_else:D
8427     \tex_advance:D #8 \c_one
8428     \tex_advance:D #9 -\c_one_thousand_million
8429   \tex_fi:D
8430   \tex_ifnum:D #8 < \c_one_thousand_million
8431   \tex_else:D
8432     \tex_advance:D #7 \c_one
8433     \tex_advance:D #8 -\c_one_thousand_million
8434   \tex_fi:D
8435 }

```

`\fp_sub:NNNNNNNN` Internal subtraction is needed only when the first number is bigger than the second, so there is no need to worry about the sign. This is a good job as there are no arguments left. The flipping flag is used in the rare case where a sign change is possible.

```

8436 \cs_new_protected_nopar:Npn \fp_sub:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8437   #7 \etex_numexpr:D #1 - #4 \scan_stop:
8438   #8 \etex_numexpr:D #2 - #5 \scan_stop:
8439   #9 \etex_numexpr:D #3 - #6 \scan_stop:
8440   \tex_ifnum:D #9 < \c_zero
8441     \tex_advance:D #8 \c_minus_one
8442     \tex_advance:D #9 \c_one_thousand_million
8443   \tex_fi:D
8444   \tex_ifnum:D #8 < \c_zero
8445     \tex_advance:D #7 \c_minus_one
8446     \tex_advance:D #8 \c_one_thousand_million
8447   \tex_fi:D
8448   \tex_ifnum:D #7 < \c_zero
8449     \tex_ifnum:D \etex_numexpr:D #8 + #9 = \c_zero
8450       #7 -#7
8451     \tex_else:D
8452       \tex_advance:D #7 \c_one
8453       #8 \etex_numexpr:D \c_one_thousand_million - #8 \scan_stop:
8454       #9 \etex_numexpr:D \c_one_thousand_million - #9 \scan_stop:
8455     \tex_fi:D
8456   \tex_fi:D
8457 }

```

`\fp_mul:NNNNNN` Decimal-part only multiplication but with higher accuracy than the user version.

```

8458 \cs_new_protected_nopar:Npn \fp_mul:NNNNNN #1#2#3#4#5#6 {
8459   \fp_mul_split:NNNN #1
8460     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int

```

```

8461 \fp_mul_split:NNNN #2
8462   \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8463 \fp_mul_split:NNNN #3
8464   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8465 \fp_mul_split:NNNN #4
8466   \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8467 \l_fp_mul_output_int \c_zero
8468 \tl_clear:N \l_fp_mul_output_tl
8469 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
8470 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
8471 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
8472 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
8473 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
8474 \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
8475 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8476 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
8477 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
8478 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
8479 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
8480 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
8481 \fp_mul_end_level:
8482 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
8483 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
8484 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
8485 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
8486 \fp_mul_end_level:
8487 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8488 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8489 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8490 \fp_mul_end_level:
8491 #6 0 \l_fp_mul_output_tl \scan_stop:
8492 \tl_clear:N \l_fp_mul_output_tl
8493 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8494 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8495 \fp_mul_end_level:
8496 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8497 \fp_mul_end_level:
8498 \fp_mul_end_level:
8499 #5 0 \l_fp_mul_output_tl \scan_stop:
8500 }

```

`\fp_mul:NNNNNNNN` For internal multiplication where the integer does need to be retained. This means of course that this code is quite slow, and so is only used when necessary.

```

8501 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8502   \fp_mul_split:NNNN #2
8503     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8504 \fp_mul_split:NNNN #3
8505     \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8506 \fp_mul_split:NNNN #5

```

```

8507 \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8508 \fp_mul_split:NNNN #6
8509 \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8510 \l_fp_mul_output_int \c_zero
8511 \tl_clear:N \l_fp_mul_output_tl
8512 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
8513 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
8514 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
8515 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
8516 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
8517 \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
8518 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8519 \fp_mul_product:NN #1 \l_fp_mul_b_vi_int
8520 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
8521 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
8522 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
8523 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
8524 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
8525 \fp_mul_product:NN \l_fp_mul_a_vi_int #4
8526 \fp_mul_end_level:
8527 \fp_mul_product:NN #1 \l_fp_mul_b_v_int
8528 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
8529 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
8530 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
8531 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
8532 \fp_mul_product:NN \l_fp_mul_a_v_int #4
8533 \fp_mul_end_level:
8534 \fp_mul_product:NN #1 \l_fp_mul_b_iv_int
8535 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8536 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8537 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8538 \fp_mul_product:NN \l_fp_mul_a_iv_int #4
8539 \fp_mul_end_level:
8540 #9 0 \l_fp_mul_output_tl \scan_stop:
8541 \tl_clear:N \l_fp_mul_output_tl
8542 \fp_mul_product:NN #1 \l_fp_mul_b_iii_int
8543 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8544 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8545 \fp_mul_product:NN \l_fp_mul_a_iii_int #4
8546 \fp_mul_end_level:
8547 \fp_mul_product:NN #1 \l_fp_mul_b_ii_int
8548 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8549 \fp_mul_product:NN \l_fp_mul_a_ii_int #4
8550 \fp_mul_end_level:
8551 \fp_mul_product:NN #1 \l_fp_mul_b_i_int
8552 \fp_mul_product:NN \l_fp_mul_a_i_int #4
8553 \fp_mul_end_level:
8554 #8 0 \l_fp_mul_output_tl \scan_stop:
8555 \tl_clear:N \l_fp_mul_output_tl
8556 \fp_mul_product:NN #1 #4

```

```

8557 \fp_mul_end_level:
8558 #7 0 \l_fp_mul_output_tl \scan_stop:
8559 }

```

`\fp_div_integer:NNNNN` Here, division is always by an integer, and so it is possible to use TeX's native calculations rather than doing it in macros. The idea here is to divide the decimal part, find any remainder, then do the real division of the two parts before adding in what is needed for the remainder.

```

8560 \cs_new_protected_nopar:Npn \fp_div_integer:NNNNN #1#2#3#4#5 {
8561   \l_fp_tmp_int #1
8562   \tex_divide:D \l_fp_tmp_int #3
8563   \l_fp_tmp_int \etex_numexpr:D #1 - \l_fp_tmp_int * #3 \scan_stop:
8564   #4 #1
8565   \tex_divide:D #4 #3
8566   #5 #2
8567   \tex_divide:D #5 #3
8568   \tex_multiply:D \l_fp_tmp_int \c_one_thousand
8569   \tex_divide:D \l_fp_tmp_int #3
8570   #5 \etex_numexpr:D #5 + \l_fp_tmp_int * \c_one_million \scan_stop:
8571   \tex_ifnum:D #5 > \c_one_thousand_million
8572   \tex_advance:D #4 \c_one
8573   \tex_advancd:D #5 -\c_one_thousand_million
8574   \tex_fi:D
8575 }

```

`\fp_extended_normalise:` The ‘extended’ integers for internal use are mainly used in fixed-point mode. This comes up in a few places, so a generalised utility is made available to carry out the change. This function simply calls the two loops to shift the input to the point of having a zero exponent.

```

\fp_extended_normalise_aux_i:
\fp_extended_normalise_aux_i:w
\fp_extended_normalise_aux_ii:w
\fp_extended_normalise_aux_ii:
\fp_extended_normalise_aux:NNNNNNNNN
8576 \cs_new_protected_nopar:Npn \fp_extended_normalise: {
8577   \fp_extended_normalise_aux_i:
8578   \fp_extended_normalise_aux_ii:
8579 }
8580 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i: {
8581   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
8582   \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8583   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8584   \tex_expandafter:D \fp_extended_normalise_aux_i:w
8585   \int_use:N \l_fp_input_a_decimal_int \q_stop
8586   \tex_expandafter:D \fp_extended_normalise_aux_i:
8587   \tex_fi:D
8588 }
8589 \cs_new_protected_nopar:Npn
8590 \fp_extended_normalise_aux_i:w #1#2#3#4#5#6#7#8#9 \q_stop {
8591   \l_fp_input_a_integer_int
8592   \etex_numexpr:D \l_fp_input_a_integer_int + #2 \scan_stop:
8593   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:

```



```

8594 \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
8595 \tex_expandafter:D \fp_extended_normalise_aux_ii:w
8596 \int_use:N \l_fp_input_a_extended_int \q_stop
8597 }
8598 \cs_new_protected_nopar:Npn
8599 \fp_extended_normalise_aux_ii:w #1#2#3#4#5#6#7#8#9 \q_stop {
8600 \l_fp_input_a_decimal_int
8601 \tex_numexpr:D \l_fp_input_a_decimal_int + #2 \scan_stop:
8602 \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
8603 \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
8604 }
8605 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii: {
8606 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
8607 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8608 \tex_expandafter:D \use_i:nn \tex_expandafter:D
8609 \fp_extended_normalise_ii_aux:NNNNNNNNN
8610 \int_use:N \l_fp_input_a_decimal_int
8611 \tex_expandafter:D \fp_extended_normalise_aux_ii:
8612 \tex_fi:D
8613 }
8614 \cs_new_protected_nopar:Npn
8615 \fp_extended_normalise_ii_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8616 \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
8617 \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
8618 \tex_else:D
8619 \tl_set:Nx \l_fp_tmp_tl
8620 {
8621 \int_use:N \l_fp_input_a_integer_int
8622 #1#2#3#4#5#6#7#8
8623 }
8624 \l_fp_input_a_integer_int \c_zero
8625 \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
8626 \tex_fi:D
8627 \tex_divide:D \l_fp_input_a_extended_int \c_ten
8628 \tl_set:Nx \l_fp_tmp_tl
8629 {
8630 #9
8631 \int_use:N \l_fp_input_a_extended_int
8632 }
8633 \l_fp_input_a_extended_int \l_fp_tmp_tl \scan_stop:
8634 \tex_advance:D \l_fp_input_a_exponent_int \c_one
8635 }

```

\fp_extended_normalise_output:
ormalise_output_aux_i:NNNNNNNNN
ormalise_output_aux_ii:NNNNNNNNN
extended_normalise_output_aux:N

At some stages in working out extended output, it is possible for the value to need shifting to keep the integer part in range. This only ever happens such that the integer needs to be made smaller.

```

8636 \cs_new_protected_nopar:Npn \fp_extended_normalise_output: {
8637 \tex_ifnum:D \l_fp_output_integer_int > \c_nine
8638 \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million

```

```

8639 \tex_expandafter:D \use_i:nn \tex_expandafter:D
8640 \fp_extended_normalise_output_aux_i:NNNNNNNNN
8641 \int_use:N \l_fp_output_integer_int
8642 \tex_expandafter:D \fp_extended_normalise_output:
8643 \tex_fi:D
8644 }
8645 \cs_new_protected_nopar:Npn
8646 \fp_extended_normalise_output_aux_i:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8647 \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
8648 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8649 \tl_set:Nx \l_fp_tmp_tl
8650 {
8651 #9
8652 \tex_expandafter:D \use_none:n
8653 \int_use:N \l_fp_output_decimal_int
8654 }
8655 \tex_expandafter:D \fp_extended_normalise_output_aux_ii:NNNNNNNNN
8656 \l_fp_tmp_tl
8657 }
8658 \cs_new_protected_nopar:Npn
8659 \fp_extended_normalise_output_aux_ii:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8660 \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
8661 \fp_extended_normalise_output_aux:N
8662 }
8663 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux:N #1 {
8664 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
8665 \tex_divide:D \l_fp_output_extended_int \c_ten
8666 \tl_set:Nx \l_fp_tmp_tl
8667 {
8668 #1
8669 \tex_expandafter:D \use_none:n
8670 \int_use:N \l_fp_output_extended_int
8671 }
8672 \l_fp_output_extended_int \l_fp_tmp_tl \scan_stop:
8673 \tex_advance:D \l_fp_output_exponent_int \c_one
8674 }

```

116.11 Trigonometric functions

`\fp_trig_normalise:` For normalisation, the code essentially switches to fixed-point arithmetic. There is a shift of the exponent, then repeated subtractions. The end result is a number in the range $-\pi < x \leq \pi$.

`\fp_trig_normalise_aux:`

`\fp_trig_sub:NNN`

```

8675 \cs_new_protected_nopar:Npn \fp_trig_normalise: {
8676 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
8677 \l_fp_input_a_extended_int \c_zero
8678 \fp_extended_normalise:
8679 \fp_trig_normalise_aux:
8680 \tex_ifnum:D \l_fp_input_a_integer_int < \c_zero

```

```

8681     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
8682     \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
8683     \tex_fi:D
8684     \tex_expandafter:D \fp_trig_octant:
8685 \tex_else:D
8686     \l_fp_input_a_sign_int     \c_one
8687     \l_fp_output_integer_int   \c_zero
8688     \l_fp_output_decimal_int   \c_zero
8689     \l_fp_output_exponent_int  \c_zero
8690     \tex_expandafter:D \fp_trig_overflow_msg:
8691     \tex_fi:D
8692 }
8693 \cs_new_protected_nopar:Npn \fp_trig_normalise_aux: {
8694     \tex_ifnum:D \l_fp_input_a_integer_int > \c_three
8695     \fp_trig_sub:NNN
8696     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8697     \tex_expandafter:D \fp_trig_normalise_aux:
8698 \tex_else:D
8699     \tex_ifnum:D \l_fp_input_a_integer_int > \c_two
8700     \tex_ifnum:D \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
8701     \fp_trig_sub:NNN
8702     \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8703     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8704     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8705     \tex_expandafter:D \fp_trig_normalise_aux:
8706     \tex_fi:D
8707     \tex_fi:D
8708 \tex_fi:D
8709 }

```

Here, there may be a sign change but there will never be any variation in the input. So a dedicated function can be used.

```

8710 \cs_new_protected_nopar:Npn \fp_trig_sub:NNN #1#2#3 {
8711     \l_fp_input_a_integer_int
8712     \etex_numexpr:D \l_fp_input_a_integer_int - #1 \scan_stop:
8713     \l_fp_input_a_decimal_int
8714     \etex_numexpr:D \l_fp_input_a_decimal_int - #2 \scan_stop:
8715     \l_fp_input_a_extended_int
8716     \etex_numexpr:D \l_fp_input_a_extended_int - #3 \scan_stop:
8717     \tex_ifnum:D \l_fp_input_a_extended_int < \c_zero
8718     \tex_advance:D \l_fp_input_a_decimal_int \c_minus_one
8719     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
8720     \tex_fi:D
8721     \tex_ifnum:D \l_fp_input_a_decimal_int < \c_zero
8722     \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8723     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8724     \tex_fi:D
8725     \tex_ifnum:D \l_fp_input_a_integer_int < \c_zero
8726     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int

```

```

8727 \tex_ifnum:D
8728 \etex_numexpr:D
8729 \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
8730 = \c_zero
8731 \l_fp_input_a_integer_int - \l_fp_input_a_integer_int
8732 \tex_else:D
8733 \l_fp_input_a_integer_int
8734 \etex_numexpr:D
8735 - \l_fp_input_a_integer_int - \c_one
8736 \scan_stop:
8737 \l_fp_input_a_decimal_int
8738 \etex_numexpr:D
8739 \c_one_thousand_million - \l_fp_input_a_decimal_int
8740 \scan_stop:
8741 \l_fp_input_a_extended_int
8742 \etex_numexpr:D
8743 \c_one_thousand_million - \l_fp_input_a_extended_int
8744 \scan_stop:
8745 \tex_fi:D
8746 \tex_fi:D
8747 }

```

`\fp_trig_octant:` Here, the input is further reduced into the range $0 \leq x < \pi/4$. This is pretty simple:
`\fp_trig_octant_aux:` check if $\pi/4$ can be taken off and if it can do it and loop. The check at the end is to ‘mop up’ values which are so close to $\pi/4$ that they should be treated as such. The test for an even octant is needed as the ‘remainder’ needed is from the nearest $\pi/2$.

```

8748 \cs_new_protected_nopar:Npn \fp_trig_octant: {
8749 \l_fp_trig_octant_int \c_one
8750 \fp_trig_octant_aux:
8751 \tex_ifnum:D \l_fp_input_a_decimal_int < \c_ten
8752 \l_fp_input_a_decimal_int \c_zero
8753 \l_fp_input_a_extended_int \c_zero
8754 \tex_fi:D
8755 \tex_ifodd:D \l_fp_trig_octant_int
8756 \tex_else:D
8757 \fp_sub:NNNNNNNNN
8758 \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8759 \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8760 \l_fp_input_a_extended_int
8761 \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8762 \l_fp_input_a_extended_int
8763 \tex_fi:D
8764 }
8765 \cs_new_protected_nopar:Npn \fp_trig_octant_aux: {
8766 \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
8767 \fp_sub:NNNNNNNNN
8768 \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8769 \l_fp_input_a_extended_int
8770 \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8771 }

```

```

8771     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8772     \l_fp_input_a_extended_int
8773     \tex_advance:D \l_fp_trig_octant_int \c_one
8774     \tex_expandafter:D \fp_trig_octant_aux:
8775 \tex_else:D
8776     \tex_ifnum:D
8777     \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
8778     \fp_sub:NNNNNNNNN
8779     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8780     \l_fp_input_a_extended_int
8781     \c_zero \c_fp_pi_by_four_decimal_int
8782     \c_fp_pi_by_four_extended_int
8783     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8784     \l_fp_input_a_extended_int
8785     \tex_advance:D \l_fp_trig_octant_int \c_one
8786     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8787     \fp_trig_octant_aux:
8788     \tex_fi:D
8789 \tex_fi:D
8790 }

```

`\fp_sin:Nn` Calculating the sine starts off in the usual way. There is a check to see if the value has
`\fp_sin:cn` already been worked out before proceeding further.
`\fp_gsin:Nn`
`\fp_gsin:cn` 8791 `\cs_new_protected_nopar:Npn \fp_sin:Nn {`
8792 `\fp_sin_aux:NNn \tl_set:Nn`
`\fp_sin_aux:NNn` 8793 `}`
8794 `\cs_new_protected_nopar:Npn \fp_gsin:Nn {`
`\fp_sin_aux:i:` 8795 `\fp_sin_aux:NNn \tl_gset:Nn`
`\fp_sin_aux:ii:` 8796 `}`
8797 `\cs_generate_variant:Nn \fp_sin:Nn { c }`
8798 `\cs_generate_variant:Nn \fp_gsin:Nn { c }`

The internal routine for sines does a check to see if the value is already known. This saves a lot of repetition when doing rotations. For very small values it is best to simply return the input as the sine: the cut-off is 1×10^{-5} .

```

8799 \cs_new_protected_nopar:Npn \fp_sin_aux:NNn #1#2#3 {
8800   \group_begin:
8801     \fp_split:Nn a {#3}
8802     \fp_standardise:NNNN
8803     \l_fp_input_a_sign_int
8804     \l_fp_input_a_integer_int
8805     \l_fp_input_a_decimal_int
8806     \l_fp_input_a_exponent_int
8807     \tl_set:Nx \l_fp_arg_tl
8808     {
8809       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8810       -
8811       \tex_else:D

```

```

8812         +
8813         \tex_fi:D
8814         \int_use:N \l_fp_input_a_integer_int
8815         .
8816         \tex_expandafter:D \use_none:n
8817         \tex_number:D \etex_numexpr:D
8818         \l_fp_input_a_decimal_int + \c_one_thousand_million
8819         e
8820         \int_use:N \l_fp_input_a_exponent_int
8821     }
8822     \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
8823     \cs_set_protected_nopar:Npx \fp_tmp:w
8824     {
8825         \group_end:
8826         #1 \exp_not:N #2 { \l_fp_arg_tl }
8827     }
8828     \tex_else:D
8829     \etex_ifcsname:D
8830     c_fp_sin ( \l_fp_arg_tl ) _fp
8831     \tex_endcsname:D
8832     \tex_else:D
8833     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8834     \fp_sin_aux_i:
8835     \tex_fi:D
8836     \cs_set_protected_nopar:Npx \fp_tmp:w
8837     {
8838         \group_end:
8839         #1 \exp_not:N #2
8840         { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
8841     }
8842     \tex_fi:D
8843     \fp_tmp:w
8844 }

```

The internals for sine first normalise the input into an octant, then choose the correct set up for the Taylor series. The sign for the sine function is easy, so there is no worry about it. So the only thing to do is to get the output standardised.

```

8845 \cs_new_protected_nopar:Npn \fp_sin_aux_i: {
8846   \fp_trig_normalise:
8847   \fp_sin_aux_ii:
8848   \tex_ifnum:D \l_fp_output_integer_int = \c_one
8849   \l_fp_output_exponent_int \c_zero
8850   \tex_else:D
8851   \l_fp_output_integer_int \l_fp_output_decimal_int
8852   \l_fp_output_decimal_int \l_fp_output_extended_int
8853   \l_fp_output_exponent_int -\c_nine
8854   \tex_fi:D
8855   \fp_standardise:NNNN
8856   \l_fp_input_a_sign_int

```

```

8857 \l_fp_output_integer_int
8858 \l_fp_output_decimal_int
8859 \l_fp_output_exponent_int
8860 \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
8861 \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
8862 {
8863   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8864   +
8865   \tex_else:D
8866   -
8867   \tex_fi:D
8868   \int_use:N \l_fp_output_integer_int
8869   .
8870   \tex_expandafter:D \use_none:n
8871   \tex_number:D \etex_numexpr:D
8872   \l_fp_output_decimal_int + \c_one_thousand_million
8873   \scan_stop:
8874   e
8875   \int_use:N \l_fp_output_exponent_int
8876 }
8877 }
8878 \cs_new_protected_nopar:Npn \fp_sin_aux_ii: {
8879   \tex_ifcase:D \l_fp_trig_octant_int
8880   \tex_or:D
8881   \tex_expandafter:D \fp_trig_calc_sin:
8882   \tex_or:D
8883   \tex_expandafter:D \fp_trig_calc_cos:
8884   \tex_or:D
8885   \tex_expandafter:D \fp_trig_calc_cos:
8886   \tex_or:D
8887   \tex_expandafter:D \fp_trig_calc_sin:
8888   \tex_fi:D
8889 }

```

\fp_cos:Nn Cosine is almost identical, but there is no short cut code here.

```

\fp_cos:cn
\fp_gcos:Nn
\fp_gcos:cn
\fp_cos_aux:NNn
\fp_cos_aux_i:
\fp_cos_aux_ii:
8890 \cs_new_protected_nopar:Npn \fp_cos:Nn {
8891   \fp_cos_aux:NNn \tl_set:Nn
8892 }
8893 \cs_new_protected_nopar:Npn \fp_gcos:Nn {
8894   \fp_cos_aux:NNn \tl_gset:Nn
8895 }
8896 \cs_generate_variant:Nn \fp_cos:Nn { c }
8897 \cs_generate_variant:Nn \fp_gcos:Nn { c }
8898 \cs_new_protected_nopar:Npn \fp_cos_aux:NNn #1#2#3 {
8899   \group_begin:
8900   \fp_split:Nn a {#3}
8901   \fp_standardise:NNNN
8902   \l_fp_input_a_sign_int
8903   \l_fp_input_a_integer_int

```

```

8904     \l_fp_input_a_decimal_int
8905     \l_fp_input_a_exponent_int
8906     \tl_set:Nx \l_fp_arg_tl
8907     {
8908         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8909         -
8910         \tex_else:D
8911         +
8912         \tex_fi:D
8913         \int_use:N \l_fp_input_a_integer_int
8914         .
8915         \tex_expandafter:D \use_none:n
8916         \tex_number:D \etex_numexpr:D
8917             \l_fp_input_a_decimal_int + \c_one_thousand_million
8918         e
8919         \int_use:N \l_fp_input_a_exponent_int
8920     }
8921     \etex_ifcsname:D c_fp_cos ( \l_fp_arg_tl ) _fp \tex_endcsname:D
8922     \tex_else:D
8923         \tex_expandafter:D \fp_cos_aux_i:
8924     \tex_fi:D
8925     \cs_set_protected_nopar:Npx \fp_tmp:w
8926     {
8927         \group_end:
8928         #1 \exp_not:N #2
8929         { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
8930     }
8931     \fp_tmp:w
8932 }

```

Almost the same as for sine: just a bit of correction for the sign of the output.

```

8933 \cs_new_protected_nopar:Npn \fp_cos_aux_i: {
8934     \fp_trig_normalise:
8935     \fp_cos_aux_ii:
8936     \tex_ifnum:D \l_fp_output_integer_int = \c_one
8937         \l_fp_output_exponent_int \c_zero
8938     \tex_else:D
8939         \l_fp_output_integer_int \l_fp_output_decimal_int
8940         \l_fp_output_decimal_int \l_fp_output_extended_int
8941         \l_fp_output_exponent_int -\c_nine
8942     \tex_fi:D
8943     \fp_standardise:NNNN
8944         \l_fp_input_a_sign_int
8945         \l_fp_output_integer_int
8946         \l_fp_output_decimal_int
8947         \l_fp_output_exponent_int
8948     \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
8949     \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
8950     {

```



```

8951 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8952 +
8953 \tex_else:D
8954 -
8955 \tex_fi:D
8956 \int_use:N \l_fp_output_integer_int
8957 .
8958 \tex_expandafter:D \use_none:n
8959 \tex_number:D \etex_numexpr:D
8960 \l_fp_output_decimal_int + \c_one_thousand_million
8961 \scan_stop:
8962 e
8963 \int_use:N \l_fp_output_exponent_int
8964 }
8965 }
8966 \cs_new_protected_nopar:Npn \fp_cos_aux_ii: {
8967 \tex_ifcase:D \l_fp_trig_octant_int
8968 \tex_or:D
8969 \tex_expandafter:D \fp_trig_calc_cos:
8970 \tex_or:D
8971 \tex_expandafter:D \fp_trig_calc_sin:
8972 \tex_or:D
8973 \tex_expandafter:D \fp_trig_calc_sin:
8974 \tex_or:D
8975 \tex_expandafter:D \fp_trig_calc_cos:
8976 \tex_fi:D
8977 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8978 \tex_ifnum:D \l_fp_trig_octant_int > \c_two
8979 \l_fp_input_a_sign_int \c_minus_one
8980 \tex_fi:D
8981 \tex_else:D
8982 \tex_ifnum:D \l_fp_trig_octant_int > \c_two
8983 \tex_else:D
8984 \l_fp_input_a_sign_int \c_one
8985 \tex_fi:D
8986 \tex_fi:D
8987 }

```

`\fp_trig_calc_cos:` These functions actually do the calculation for sine and cosine.
`\fp_trig_calc_sin:`
`\fp_trig_calc_Taylor:`

```

8988 \cs_new_protected_nopar:Npn \fp_trig_calc_cos: {
8989 \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
8990 \l_fp_output_integer_int \c_one
8991 \l_fp_output_decimal_int \c_zero
8992 \tex_else:D
8993 \l_fp_trig_sign_int \c_minus_one
8994 \fp_mul:NNNNNN
8995 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
8996 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
8997 \l_fp_trig_decimal_int \l_fp_trig_extended_int

```

```

8998 \fp_div_integer:NNNNN
8999 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9000 \c_two
9001 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9002 \l_fp_count_int \c_three
9003 \tex_ifnum:D \l_fp_trig_extended_int = \c_zero
9004 \tex_ifnum:D \l_fp_trig_decimal_int = \c_zero
9005 \l_fp_output_integer_int \c_one
9006 \l_fp_output_decimal_int \c_zero
9007 \l_fp_output_extended_int \c_zero
9008 \tex_else:D
9009 \l_fp_output_integer_int \c_zero
9010 \l_fp_output_decimal_int \c_one_thousand_million
9011 \l_fp_output_extended_int \c_zero
9012 \tex_fi:D
9013 \tex_else:D
9014 \l_fp_output_integer_int \c_zero
9015 \l_fp_output_decimal_int 999999999 \scan_stop:
9016 \l_fp_output_extended_int \c_one_thousand_million
9017 \tex_fi:D
9018 \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
9019 \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9020 \tex_expandafter:D \fp_trig_calc_Taylor:
9021 \tex_fi:D
9022 }
9023 \cs_new_protected_nopar:Npn \fp_trig_calc_sin: {
9024 \l_fp_output_integer_int \c_zero
9025 \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9026 \l_fp_output_decimal_int \c_zero
9027 \tex_else:D
9028 \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9029 \l_fp_output_extended_int \l_fp_input_a_extended_int
9030 \l_fp_trig_sign_int \c_one
9031 \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
9032 \l_fp_trig_extended_int \l_fp_input_a_extended_int
9033 \l_fp_count_int \c_two
9034 \tex_expandafter:D \fp_trig_calc_Taylor:
9035 \tex_fi:D
9036 }

```

This implements a Taylor series calculation for the trigonometric functions. Lots of shuffling about as TeX is not exactly a natural choice for this sort of thing.

```

9037 \cs_new_protected_nopar:Npn \fp_trig_calc_Taylor: {
9038 \l_fp_trig_sign_int -\l_fp_trig_sign_int
9039 \fp_mul:NNNNNN
9040 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9041 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9042 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9043 \fp_mul:NNNNNN

```

```

9044 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9045 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9046 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9047 \fp_div_integer:NNNNN
9048 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9049 \l_fp_count_int
9050 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9051 \tex_advance:D \l_fp_count_int \c_one
9052 \fp_div_integer:NNNNN
9053 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9054 \l_fp_count_int
9055 \l_fp_trig_decimal_int \l_fp_trig_extended_int
9056 \tex_advance:D \l_fp_count_int \c_one
9057 \tex_ifnum:D \l_fp_trig_decimal_int > \c_zero
9058 \tex_ifnum:D \l_fp_trig_sign_int > \c_zero
9059 \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
9060 \tex_advance:D \l_fp_output_extended_int
9061 \l_fp_trig_extended_int
9062 \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
9063 \tex_else:D
9064 \tex_advance:D \l_fp_output_decimal_int \c_one
9065 \tex_advance:D \l_fp_output_extended_int
9066 -\c_one_thousand_million
9067 \tex_fi:D
9068 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9069 \tex_else:D
9070 \tex_advance:D \l_fp_output_integer_int \c_one
9071 \tex_advance:D \l_fp_output_decimal_int
9072 -\c_one_thousand_million
9073 \tex_fi:D
9074 \tex_else:D
9075 \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9076 \tex_advance:D \l_fp_output_extended_int
9077 -\l_fp_input_a_extended_int
9078 \tex_ifnum:D \l_fp_output_extended_int < \c_zero
9079 \tex_advance:D \l_fp_output_decimal_int \c_minus_one
9080 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9081 \tex_fi:D
9082 \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
9083 \tex_advance:D \l_fp_output_integer_int \c_minus_one
9084 \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9085 \tex_fi:D
9086 \tex_fi:D
9087 \tex_expandafter:D \fp_trig_calc_Taylor:
9088 \tex_fi:D
9089 }

```

`\fp_tan:Nn` As might be expected, tangents are calculated from the sine and cosine by division. So
`\fp_tan:cn` there is a bit of set up, the two subsidiary pieces of work are done and then a division
`\fp_gtan:Nn`
`\fp_gtan:cn`
`\fp_tan_aux:NNn` 429
`\fp_tan_aux_i:`
`\fp_tan_aux_ii:`
`\fp_tan_aux_iii:`
`\fp_tan_aux_iv:`

takes place. For small numbers, the same approach is used as for sines, with the input value simply returned as is.

```

9090 \cs_new_protected_nopar:Npn \fp_tan:Nn {
9091   \fp_tan_aux:NNn \tl_set:Nn
9092 }
9093 \cs_new_protected_nopar:Npn \fp_gtan:Nn {
9094   \fp_tan_aux:NNn \tl_gset:Nn
9095 }
9096 \cs_generate_variant:Nn \fp_tan:Nn { c }
9097 \cs_generate_variant:Nn \fp_gtan:Nn { c }
9098 \cs_new_protected_nopar:Npn \fp_tan_aux:NNn #1#2#3 {
9099   \group_begin:
9100     \fp_split:Nn a {#3}
9101     \fp_standardise:NNNN
9102     \l_fp_input_a_sign_int
9103     \l_fp_input_a_integer_int
9104     \l_fp_input_a_decimal_int
9105     \l_fp_input_a_exponent_int
9106     \tl_set:Nx \l_fp_arg_tl
9107     {
9108       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9109       -
9110       \tex_else:D
9111       +
9112       \tex_fi:D
9113       \int_use:N \l_fp_input_a_integer_int
9114       .
9115       \tex_expandafter:D \use_none:n
9116       \tex_number:D \etex_numexpr:D
9117         \l_fp_input_a_decimal_int + \c_one_thousand_million
9118       e
9119       \int_use:N \l_fp_input_a_exponent_int
9120     }
9121     \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
9122     \cs_set_protected_nopar:Npx \fp_tmp:w
9123     {
9124       \group_end:
9125       #1 \exp_not:N #2 { \l_fp_arg_tl }
9126     }
9127     \tex_else:D
9128     \etex_ifcsname:D
9129       c_fp_tan ( \l_fp_arg_tl ) _fp
9130     \tex_endcsname:D
9131     \tex_else:D
9132     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9133     \fp_tan_aux_i:
9134     \tex_fi:D
9135     \cs_set_protected_nopar:Npx \fp_tmp:w
9136     {

```

```

9137         \group_end:
9138         #1 \exp_not:N #2
9139         { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
9140     }
9141     \tex_fi:D
9142     \fp_tmp:w
9143 }

```

The business of the calculation does not check for stored sines or cosines as there would then be an overhead to reading them back in. There is also no need to worry about ‘small’ sine values as these will have been dealt with earlier. There is a two-step lead off so that undefined division is not even attempted.

```

9144 \cs_new_protected_nopar:Npn \fp_tan_aux_i: {
9145     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
9146     \tex_expandafter:D \fp_tan_aux_ii:
9147     \tex_else:D
9148     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9149     \c_zero_fp
9150     \tex_expandafter:D \fp_trig_overflow_msg:
9151     \tex_fi:D
9152 }
9153 \cs_new_protected_nopar:Npn \fp_tan_aux_ii: {
9154     \fp_trig_normalise:
9155     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9156     \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9157     \l_fp_output_sign_int \c_minus_one
9158     \tex_else:D
9159     \l_fp_output_sign_int \c_one
9160     \tex_fi:D
9161     \tex_else:D
9162     \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9163     \l_fp_output_sign_int \c_one
9164     \tex_else:D
9165     \l_fp_output_sign_int \c_minus_one
9166     \tex_fi:D
9167     \tex_fi:D
9168     \fp_cos_aux_ii:
9169     \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9170     \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9171     \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9172     \c_undefined_fp
9173     \tex_else:D
9174     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9175     \fp_tan_aux_iii:
9176     \tex_fi:D
9177     \tex_else:D
9178     \tex_expandafter:D \fp_tan_aux_iii:
9179     \tex_fi:D
9180 }

```

The division is done here using the same code as the standard division unit, shifting the digits in the calculated sine and cosine to maintain accuracy.

```

9181 \cs_new_protected_nopar:Npn \fp_tan_aux_iii: {
9182   \l_fp_input_b_integer_int \l_fp_output_decimal_int
9183   \l_fp_input_b_decimal_int \l_fp_output_extended_int
9184   \l_fp_input_b_exponent_int -\c_nine
9185   \fp_standardise:NNNN
9186   \l_fp_input_b_sign_int
9187   \l_fp_input_b_integer_int
9188   \l_fp_input_b_decimal_int
9189   \l_fp_input_b_exponent_int
9190   \fp_sin_aux_ii:
9191   \l_fp_input_a_integer_int \l_fp_output_decimal_int
9192   \l_fp_input_a_decimal_int \l_fp_output_extended_int
9193   \l_fp_input_a_exponent_int -\c_nine
9194   \fp_standardise:NNNN
9195   \l_fp_input_a_sign_int
9196   \l_fp_input_a_integer_int
9197   \l_fp_input_a_decimal_int
9198   \l_fp_input_a_exponent_int
9199   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9200   \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9201   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9202     \c_zero_fp
9203   \tex_else:D
9204     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9205     \fp_tan_aux_iv:
9206   \tex_fi:D
9207   \tex_else:D
9208     \tex_expandafter:D \fp_tan_aux_iv:
9209   \tex_fi:D
9210 }
9211 \cs_new_protected_nopar:Npn \fp_tan_aux_iv: {
9212   \l_fp_output_integer_int \c_zero
9213   \l_fp_output_decimal_int \c_zero
9214   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
9215   \l_fp_div_offset_int \c_one_hundred_million
9216   \fp_div_loop:
9217   \l_fp_output_exponent_int
9218   \etex_numexpr:D
9219     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
9220   \scan_stop:
9221   \fp_standardise:NNNN
9222   \l_fp_output_sign_int
9223   \l_fp_output_integer_int
9224   \l_fp_output_decimal_int
9225   \l_fp_output_exponent_int
9226   \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
9227   \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }

```

```

9228 {
9229   \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9230   +
9231   \tex_else:D
9232   -
9233   \tex_fi:D
9234   \int_use:N \l_fp_output_integer_int
9235   .
9236   \tex_expandafter:D \use_none:n
9237   \tex_number:D \etex_numexpr:D
9238     \l_fp_output_decimal_int + \c_one_thousand_million
9239   \scan_stop:
9240   e
9241   \int_use:N \l_fp_output_exponent_int
9242 }
9243 }

```

116.12 Exponent and logarithm functions

Calculation of exponentials requires a number of precomputed values: first the positive integers.

```

\c_fp_exp_1_tl
\c_fp_exp_2_tl
\c_fp_exp_3_tl
\c_fp_exp_4_tl
\c_fp_exp_5_tl
\c_fp_exp_6_tl
\c_fp_exp_7_tl
\c_fp_exp_8_tl
\c_fp_exp_9_tl
\c_fp_exp_10_tl
\c_fp_exp_20_tl
\c_fp_exp_30_tl
\c_fp_exp_40_tl
\c_fp_exp_50_tl
\c_fp_exp_60_tl
\c_fp_exp_70_tl
\c_fp_exp_80_tl
\c_fp_exp_90_tl
\c_fp_exp_100_tl
\c_fp_exp_200_tl

```

```

9244 \tl_new:c { c_fp_exp_1_tl }
9245 \tl_set:cn { c_fp_exp_1_tl }
9246   { { 2 } { 718281828 } { 459045235 } { 0 } }
9247 \tl_new:c { c_fp_exp_2_tl }
9248 \tl_set:cn { c_fp_exp_2_tl }
9249   { { 7 } { 389056098 } { 930650227 } { 0 } }
9250 \tl_new:c { c_fp_exp_3_tl }
9251 \tl_set:cn { c_fp_exp_3_tl }
9252   { { 2 } { 008553692 } { 318766774 } { 1 } }
9253 \tl_new:c { c_fp_exp_4_tl }
9254 \tl_set:cn { c_fp_exp_4_tl }
9255   { { 5 } { 459815003 } { 314423908 } { 1 } }
9256 \tl_new:c { c_fp_exp_5_tl }
9257 \tl_set:cn { c_fp_exp_5_tl }
9258   { { 1 } { 484131591 } { 025766034 } { 2 } }
9259 \tl_new:c { c_fp_exp_6_tl }
9260 \tl_set:cn { c_fp_exp_6_tl }
9261   { { 4 } { 034287934 } { 927351226 } { 2 } }
9262 \tl_new:c { c_fp_exp_7_tl }
9263 \tl_set:cn { c_fp_exp_7_tl }
9264   { { 1 } { 096633158 } { 428458599 } { 3 } }
9265 \tl_new:c { c_fp_exp_8_tl }
9266 \tl_set:cn { c_fp_exp_8_tl }
9267   { { 2 } { 980957987 } { 041728275 } { 3 } }
9268 \tl_new:c { c_fp_exp_9_tl }
9269 \tl_set:cn { c_fp_exp_9_tl }
9270   { { 8 } { 103083927 } { 575384008 } { 3 } }

```

```

9271 \tl_new:c { c_fp_exp_10_tl }
9272 \tl_set:cn { c_fp_exp_10_tl }
9273 { { 2 } { 202646579 } { 480671652 } { 4 } }
9274 \tl_new:c { c_fp_exp_20_tl }
9275 \tl_set:cn { c_fp_exp_20_tl }
9276 { { 4 } { 851651954 } { 097902280 } { 8 } }
9277 \tl_new:c { c_fp_exp_30_tl }
9278 \tl_set:cn { c_fp_exp_30_tl }
9279 { { 1 } { 068647458 } { 152446215 } { 13 } }
9280 \tl_new:c { c_fp_exp_40_tl }
9281 \tl_set:cn { c_fp_exp_40_tl }
9282 { { 2 } { 353852668 } { 370199854 } { 17 } }
9283 \tl_new:c { c_fp_exp_50_tl }
9284 \tl_set:cn { c_fp_exp_50_tl }
9285 { { 5 } { 184705528 } { 587072464 } { 21 } }
9286 \tl_new:c { c_fp_exp_60_tl }
9287 \tl_set:cn { c_fp_exp_60_tl }
9288 { { 1 } { 142007389 } { 815684284 } { 26 } }
9289 \tl_new:c { c_fp_exp_70_tl }
9290 \tl_set:cn { c_fp_exp_70_tl }
9291 { { 2 } { 515438670 } { 919167006 } { 30 } }
9292 \tl_new:c { c_fp_exp_80_tl }
9293 \tl_set:cn { c_fp_exp_80_tl }
9294 { { 5 } { 540622384 } { 393510053 } { 34 } }
9295 \tl_new:c { c_fp_exp_90_tl }
9296 \tl_set:cn { c_fp_exp_90_tl }
9297 { { 1 } { 220403294 } { 317840802 } { 39 } }
9298 \tl_new:c { c_fp_exp_100_tl }
9299 \tl_set:cn { c_fp_exp_100_tl }
9300 { { 2 } { 688117141 } { 816135448 } { 43 } }
9301 \tl_new:c { c_fp_exp_200_tl }
9302 \tl_set:cn { c_fp_exp_200_tl }
9303 { { 7 } { 225973768 } { 125749258 } { 86 } }

```

\c_fp_exp_-1_tl Now the negative integers.

```

\c_fp_exp_-2_tl
\c_fp_exp_-3_tl
\c_fp_exp_-4_tl
\c_fp_exp_-5_tl
\c_fp_exp_-6_tl
\c_fp_exp_-7_tl
\c_fp_exp_-8_tl
\c_fp_exp_-9_tl
\c_fp_exp_-10_tl
\c_fp_exp_-20_tl
\c_fp_exp_-30_tl
\c_fp_exp_-40_tl
\c_fp_exp_-50_tl
\c_fp_exp_-60_tl
\c_fp_exp_-70_tl
\c_fp_exp_-80_tl
\c_fp_exp_-90_tl
\c_fp_exp_-100_tl
\c_fp_exp_-200_tl

```

```

9304 \tl_new:c { c_fp_exp_-1_tl }
9305 \tl_set:cn { c_fp_exp_-1_tl }
9306 { { 3 } { 678794411 } { 71442322 } { -1 } }
9307 \tl_new:c { c_fp_exp_-2_tl }
9308 \tl_set:cn { c_fp_exp_-2_tl }
9309 { { 1 } { 353352832 } { 366132692 } { -1 } }
9310 \tl_new:c { c_fp_exp_-3_tl }
9311 \tl_set:cn { c_fp_exp_-3_tl }
9312 { { 4 } { 978706836 } { 786394298 } { -2 } }
9313 \tl_new:c { c_fp_exp_-4_tl }
9314 \tl_set:cn { c_fp_exp_-4_tl }
9315 { { 1 } { 831563888 } { 873418029 } { -2 } }
9316 \tl_new:c { c_fp_exp_-5_tl }
9317 \tl_set:cn { c_fp_exp_-5_tl }

```



```

9318 { { 6 } { 737946999 } { 085467097 } { -3 } }
9319 \tl_new:c { c_fp_exp_-6_tl }
9320 \tl_set:cn { c_fp_exp_-6_tl }
9321 { { 2 } { 478752176 } { 666358423 } { -3 } }
9322 \tl_new:c { c_fp_exp_-7_tl }
9323 \tl_set:cn { c_fp_exp_-7_tl }
9324 { { 9 } { 118819655 } { 545162080 } { -4 } }
9325 \tl_new:c { c_fp_exp_-8_tl }
9326 \tl_set:cn { c_fp_exp_-8_tl }
9327 { { 3 } { 354626279 } { 025118388 } { -4 } }
9328 \tl_new:c { c_fp_exp_-9_tl }
9329 \tl_set:cn { c_fp_exp_-9_tl }
9330 { { 1 } { 234098040 } { 866795495 } { -4 } }
9331 \tl_new:c { c_fp_exp_-10_tl }
9332 \tl_set:cn { c_fp_exp_-10_tl }
9333 { { 4 } { 539992976 } { 248451536 } { -5 } }
9334 \tl_new:c { c_fp_exp_-20_tl }
9335 \tl_set:cn { c_fp_exp_-20_tl }
9336 { { 2 } { 061153622 } { 438557828 } { -9 } }
9337 \tl_new:c { c_fp_exp_-30_tl }
9338 \tl_set:cn { c_fp_exp_-30_tl }
9339 { { 9 } { 357622968 } { 840174605 } { -14 } }
9340 \tl_new:c { c_fp_exp_-40_tl }
9341 \tl_set:cn { c_fp_exp_-40_tl }
9342 { { 4 } { 248354255 } { 291588995 } { -18 } }
9343 \tl_new:c { c_fp_exp_-50_tl }
9344 \tl_set:cn { c_fp_exp_-50_tl }
9345 { { 1 } { 928749847 } { 963917783 } { -22 } }
9346 \tl_new:c { c_fp_exp_-60_tl }
9347 \tl_set:cn { c_fp_exp_-60_tl }
9348 { { 8 } { 756510762 } { 696520338 } { -27 } }
9349 \tl_new:c { c_fp_exp_-70_tl }
9350 \tl_set:cn { c_fp_exp_-70_tl }
9351 { { 3 } { 975449735 } { 908646808 } { -31 } }
9352 \tl_new:c { c_fp_exp_-80_tl }
9353 \tl_set:cn { c_fp_exp_-80_tl }
9354 { { 1 } { 804851387 } { 845415172 } { -35 } }
9355 \tl_new:c { c_fp_exp_-90_tl }
9356 \tl_set:cn { c_fp_exp_-90_tl }
9357 { { 8 } { 194012623 } { 990515430 } { -40 } }
9358 \tl_new:c { c_fp_exp_-100_tl }
9359 \tl_set:cn { c_fp_exp_-100_tl }
9360 { { 3 } { 720075976 } { 020835963 } { -44 } }
9361 \tl_new:c { c_fp_exp_-200_tl }
9362 \tl_set:cn { c_fp_exp_-200_tl }
9363 { { 1 } { 383896526 } { 736737530 } { -87 } }

```

\fp_exp:Nn The calculation of an exponent starts off starts in much the same way as the trigonometric
 \fp_exp:cn functions: normalise the input, look for a pre-defined value and if one is not found hand
 \fp_gexp:Nn
 \fp_gexp:cn

\fp_exp_aux:NNn
 \fp_exp_internal:
 \fp_exp_aux:
 \fp_exp_integer:
 \fp_exp_integer_tens:
 \fp_exp_integer_units:
 \fp_exp_integer_const:n
 \fp_exp_integer_const:nnnn

off to the real workhorse function. The test for a definition of the result is used so that overflows do not result in any outcome being defined.

```

9364 \cs_new_protected_nopar:Npn \fp_exp:Nn {
9365   \fp_exp_aux:NNn \tl_set:Nn
9366 }
9367 \cs_new_protected_nopar:Npn \fp_gexp:Nn {
9368   \fp_exp_aux:NNn \tl_gset:Nn
9369 }
9370 \cs_generate_variant:Nn \fp_exp:Nn { c }
9371 \cs_generate_variant:Nn \fp_gexp:Nn { c }
9372 \cs_new_protected_nopar:Npn \fp_exp_aux:NNn #1#2#3 {
9373   \group_begin:
9374     \fp_split:Nn a {#3}
9375     \fp_standardise:NNNN
9376     \l_fp_input_a_sign_int
9377     \l_fp_input_a_integer_int
9378     \l_fp_input_a_decimal_int
9379     \l_fp_input_a_exponent_int
9380     \l_fp_input_a_extended_int \c_zero
9381     \tl_set:Nx \l_fp_arg_tl
9382     {
9383       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9384       -
9385       \tex_else:D
9386       +
9387       \tex_fi:D
9388       \int_use:N \l_fp_input_a_integer_int
9389       .
9390       \tex_expandafter:D \use_none:n
9391       \tex_number:D \etex_numexpr:D
9392         \l_fp_input_a_decimal_int + \c_one_thousand_million
9393       e
9394       \int_use:N \l_fp_input_a_exponent_int
9395     }
9396     \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9397     \tex_else:D
9398       \tex_expandafter:D \fp_exp_internal:
9399     \tex_fi:D
9400     \cs_set_protected_nopar:Npx \fp_tmp:w
9401     {
9402       \group_end:
9403       #1 \exp_not:N #2
9404       {
9405         \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp
9406         \tex_endcsname:D
9407         \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
9408       \tex_else:D
9409         \c_zero_fp
9410       \tex_fi:D

```

```

9411     }
9412   }
9413   \fp_tmp:w
9414 }

```

The first real step is to convert the input into a fixed-point representation for further calculation: anything which is dropped here as too small would not influence the output in any case. There are a couple of overflow tests: the maximum

```

9415 \cs_new_protected_nopar:Npn \fp_exp_internal: {
9416   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_three
9417   \fp_extended_normalise:
9418   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9419   \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9420     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9421     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9422     \tex_expandafter:D \fp_exp_aux:
9423   \tex_else:D
9424     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9425     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9426     \tex_expandafter:D \fp_exp_overflow_msg:
9427     \tex_fi:D
9428   \tex_else:D
9429     \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9430       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9431       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9432       \tex_expandafter:D \fp_exp_aux:
9433     \tex_else:D
9434       \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9435       { \c_zero_fp }
9436     \tex_fi:D
9437   \tex_fi:D
9438 \tex_else:D
9439   \tex_expandafter:D \fp_exp_overflow_msg:
9440 \tex_fi:D
9441 }

```

The main algorithm makes use of the fact that

$$e^{nmp.q} = e^n e^m e^p e^{0.q}$$

and that there is a Taylor series that can be used to calculate $e^{0.q}$. Thus the approach needed is in three parts. First, the exponent of the integer part of the input is found using the pre-calculated constants. Second, the Taylor series is used to find the exponent for the decimal part of the input. Finally, the two parts are multiplied together to give the result. As the normalisation code will already have dealt with any overflowing values, there are no further checks needed.

```

9442 \cs_new_protected_nopar:Npn \fp_exp_aux: {
9443   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero

```

```

9444 \tex_expandafter:D \fp_exp_integer:
9445 \tex_else:D
9446 \l_fp_output_integer_int \c_one
9447 \l_fp_output_decimal_int \c_zero
9448 \l_fp_output_extended_int \c_zero
9449 \l_fp_output_exponent_int \c_zero
9450 \tex_expandafter:D \fp_exp_decimal:
9451 \tex_fi:D
9452 }

```

The integer part calculation starts with the hundreds. This is set up such that very large negative numbers can short-cut the entire procedure and simply return zero. In other cases, the code either recovers the exponent of the hundreds value or sets the appropriate storage to one (so that multiplication works correctly).

```

9453 \cs_new_protected_nopar:Npn \fp_exp_integer: {
9454 \tex_ifnum:D \l_fp_input_a_integer_int < \c_one_hundred
9455 \l_fp_exp_integer_int \c_one
9456 \l_fp_exp_decimal_int \c_zero
9457 \l_fp_exp_extended_int \c_zero
9458 \l_fp_exp_exponent_int \c_zero
9459 \tex_expandafter:D \fp_exp_integer_tens:
9460 \tex_else:D
9461 \tl_set:Nx \l_fp_tmp_tl
9462 {
9463 \tex_expandafter:D \use_i:nnn
9464 \int_use:N \l_fp_input_a_integer_int
9465 }
9466 \l_fp_input_a_integer_int
9467 \etex_numexpr:D
9468 \l_fp_input_a_integer_int - \l_fp_tmp_tl 00
9469 \scan_stop:
9470 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9471 \tex_ifnum:D \l_fp_output_integer_int > 200 \scan_stop:
9472 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9473 { \c_zero_fp }
9474 \tex_else:D
9475 \fp_exp_integer_const:n { - \l_fp_tmp_tl 00 }
9476 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9477 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9478 \tex_expandafter:D \fp_exp_integer_tens:
9479 \tex_fi:D
9480 \tex_else:D
9481 \fp_exp_integer_const:n { \l_fp_tmp_tl 00 }
9482 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9483 \tex_expandafter:D \fp_exp_integer_tens:
9484 \tex_fi:D
9485 \tex_fi:D
9486 }

```

The tens and units parts are handled in a similar way, with a multiplication step to build

up the final value. That also includes a correction step to avoid an overflow of the integer part.

```

9487 \cs_new_protected_nopar:Npn \fp_exp_integer_tens: {
9488   \l_fp_output_integer_int \l_fp_exp_integer_int
9489   \l_fp_output_decimal_int \l_fp_exp_decimal_int
9490   \l_fp_output_extended_int \l_fp_exp_extended_int
9491   \l_fp_output_exponent_int \l_fp_exp_exponent_int
9492   \tex_ifnum:D \l_fp_input_a_integer_int > \c_nine
9493   \tl_set:Nx \l_fp_tmp_tl
9494   {
9495     \tex_expandafter:D \use_i:nn
9496     \int_use:N \l_fp_input_a_integer_int
9497   }
9498   \l_fp_input_a_integer_int
9499   \etex_numexpr:D
9500     \l_fp_input_a_integer_int - \l_fp_tmp_tl 0
9501   \scan_stop:
9502   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9503   \fp_exp_integer_const:n { \l_fp_tmp_tl 0 }
9504   \tex_else:D
9505   \fp_exp_integer_const:n { - \l_fp_tmp_tl 0 }
9506   \tex_fi:D
9507   \fp_mul:NNNNNNNNN
9508     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9509     \l_fp_output_integer_int \l_fp_output_decimal_int
9510     \l_fp_output_extended_int
9511     \l_fp_output_integer_int \l_fp_output_decimal_int
9512     \l_fp_output_extended_int
9513   \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9514   \fp_extended_normalise_output:
9515   \tex_fi:D
9516   \fp_exp_integer_units:
9517 }
9518 \cs_new_protected_nopar:Npn \fp_exp_integer_units: {
9519   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
9520   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9521   \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
9522   \tex_else:D
9523   \fp_exp_integer_const:n
9524     { - \int_use:N \l_fp_input_a_integer_int }
9525   \tex_fi:D
9526   \fp_mul:NNNNNNNNN
9527     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9528     \l_fp_output_integer_int \l_fp_output_decimal_int
9529     \l_fp_output_extended_int
9530     \l_fp_output_integer_int \l_fp_output_decimal_int
9531     \l_fp_output_extended_int
9532   \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9533   \fp_extended_normalise_output:

```

```

9534 \tex_fi:D
9535 \fp_exp_decimal:
9536 }

```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```

9537 \cs_new_protected_nopar:Npn \fp_exp_integer_const:n #1 {
9538 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9539 \fp_exp_integer_const:nnnn
9540 \tex_csname:D c_fp_exp_ #1 _tl \tex_endcsname:D
9541 }
9542 \cs_new_protected_nopar:Npn \fp_exp_integer_const:nnnn #1#2#3#4 {
9543 \l_fp_exp_integer_int #1 \scan_stop:
9544 \l_fp_exp_decimal_int #2 \scan_stop:
9545 \l_fp_exp_extended_int #3 \scan_stop:
9546 \l_fp_exp_exponent_int #4 \scan_stop:
9547 }

```

Finding the exponential for the decimal part of the number requires a Taylor series calculation. The set up is done here with the loop itself a separate function. Once the decimal part is available this is multiplied by the integer part already worked out to give the final result.

```

9548 \cs_new_protected_nopar:Npn \fp_exp_decimal: {
9549 \tex_ifnum:D \l_fp_input_a_decimal_int > \c_zero
9550 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9551 \l_fp_exp_integer_int \c_one
9552 \l_fp_exp_decimal_int \l_fp_input_a_decimal_int
9553 \l_fp_exp_extended_int \l_fp_input_a_extended_int
9554 \tex_else:D
9555 \l_fp_exp_integer_int \c_zero
9556 \tex_ifnum:D \l_fp_exp_extended_int = \c_zero
9557 \l_fp_exp_decimal_int
9558 \etex_numexpr:D
9559 \c_one_thousand_million - \l_fp_input_a_decimal_int
9560 \scan_stop:
9561 \l_fp_exp_extended_int \c_zero
9562 \tex_else:D
9563 \l_fp_exp_decimal_int
9564 \etex_numexpr:D
9565 999999999 - \l_fp_input_a_decimal_int
9566 \scan_stop:
9567 \l_fp_exp_extended_int
9568 \etex_numexpr:D
9569 \c_one_thousand_million - \l_fp_input_a_extended_int
9570 \scan_stop:
9571 \tex_fi:D
9572 \tex_fi:D
9573 \l_fp_input_b_sign_int \l_fp_input_a_sign_int

```

```

9574 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9575 \l_fp_input_b_extended_int \l_fp_input_a_extended_int
9576 \l_fp_count_int \c_one
9577 \fp_exp_Taylor:
9578 \fp_mul:NNNNNNNN
9579 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9580 \l_fp_output_integer_int \l_fp_output_decimal_int
9581 \l_fp_output_extended_int
9582 \l_fp_output_integer_int \l_fp_output_decimal_int
9583 \l_fp_output_extended_int
9584 \tex_fi:D
9585 \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9586 \tex_else:D
9587 \tex_advance:D \l_fp_output_decimal_int \c_one
9588 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9589 \tex_else:D
9590 \l_fp_output_decimal_int \c_zero
9591 \tex_advance:D \l_fp_output_integer_int \c_one
9592 \tex_fi:D
9593 \tex_fi:D
9594 \fp_standardise:NNNN
9595 \l_fp_output_sign_int
9596 \l_fp_output_integer_int
9597 \l_fp_output_decimal_int
9598 \l_fp_output_exponent_int
9599 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9600 {
9601 +
9602 \int_use:N \l_fp_output_integer_int
9603 .
9604 \tex_expandafter:D \use_none:n
9605 \tex_number:D \etex_numexpr:D
9606 \l_fp_output_decimal_int + \c_one_thousand_million
9607 \scan_stop:
9608 e
9609 \int_use:N \l_fp_output_exponent_int
9610 }
9611 }

```

The Taylor series for $\exp(x)$ is

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

which converges for $-1 < x < 1$. The code above sets up the x part, leaving the loop to multiply the running value by x/n and add it onto the sum. The way that this is done is that the running total is stored in the `exp` set of registers, while the current item is stored as `input_b`.

```

9612 \cs_new_protected_nopar:Npn \fp_exp_Taylor: {

```

```

9613 \tex_advance:D \l_fp_count_int \c_one
9614 \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
9615 \fp_mul:NNNNNN
9616   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9617   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9618   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9619 \fp_div_integer:NNNNN
9620   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9621   \l_fp_count_int
9622   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9623 \tex_ifnum:D
9624   \etex_numexpr:D
9625     \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
9626     > \c_zero
9627 \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
9628   \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
9629   \tex_advance:D \l_fp_exp_extended_int
9630     \l_fp_input_b_extended_int
9631   \tex_ifnum:D \l_fp_exp_extended_int < \c_one_thousand_million
9632   \tex_else:D
9633     \tex_advance:D \l_fp_exp_decimal_int \c_one
9634     \tex_advance:D \l_fp_exp_extended_int
9635       -\c_one_thousand_million
9636   \tex_fi:D
9637 \tex_ifnum:D \l_fp_exp_decimal_int < \c_one_thousand_million
9638 \tex_else:D
9639   \tex_advance:D \l_fp_exp_integer_int \c_one
9640   \tex_advance:D \l_fp_exp_decimal_int
9641     -\c_one_thousand_million
9642   \tex_fi:D
9643 \tex_else:D
9644   \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
9645   \tex_advance:D \l_fp_exp_extended_int
9646     -\l_fp_input_a_extended_int
9647   \tex_ifnum:D \l_fp_exp_extended_int < \c_zero
9648     \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
9649     \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
9650   \tex_fi:D
9651   \tex_ifnum:D \l_fp_exp_decimal_int < \c_zero
9652     \tex_advance:D \l_fp_exp_integer_int \c_minus_one
9653     \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9654   \tex_fi:D
9655   \tex_fi:D
9656   \tex_expandafter:D \fp_exp_Taylor:
9657 \tex_fi:D
9658 }

```

This is set up as a function so that the power code can redirect the effect.

```

9659 \cs_new_protected_nopar:Npn \fp_exp_const:Nx #1#2 {

```



```

9660 \tl_new:N #1
9661 \tl_gset:Nx #1 {#2}
9662 }
9663 \cs_generate_variant:Nn \fp_exp_const:Nx { c }

```

\c_fp_ln_10_1_tl Constants for working out logarithms: first those for the powers of ten.

```

\c_fp_ln_10_2_tl
\c_fp_ln_10_3_tl
\c_fp_ln_10_4_tl
\c_fp_ln_10_5_tl
\c_fp_ln_10_6_tl
\c_fp_ln_10_7_tl
\c_fp_ln_10_8_tl
\c_fp_ln_10_9_tl
9664 \tl_new:c { c_fp_ln_10_1_tl }
9665 \tl_set:cn { c_fp_ln_10_1_tl }
9666 { { 2 } { 302585092 } { 994045684 } { 0 } }
9667 \tl_new:c { c_fp_ln_10_2_tl }
9668 \tl_set:cn { c_fp_ln_10_2_tl }
9669 { { 4 } { 605170185 } { 988091368 } { 0 } }
9670 \tl_new:c { c_fp_ln_10_3_tl }
9671 \tl_set:cn { c_fp_ln_10_3_tl }
9672 { { 6 } { 907755278 } { 982137052 } { 0 } }
9673 \tl_new:c { c_fp_ln_10_4_tl }
9674 \tl_set:cn { c_fp_ln_10_4_tl }
9675 { { 9 } { 210340371 } { 976182736 } { 0 } }
9676 \tl_new:c { c_fp_ln_10_5_tl }
9677 \tl_set:cn { c_fp_ln_10_5_tl }
9678 { { 1 } { 151292546 } { 497022842 } { 1 } }
9679 \tl_new:c { c_fp_ln_10_6_tl }
9680 \tl_set:cn { c_fp_ln_10_6_tl }
9681 { { 1 } { 381551055 } { 796427410 } { 1 } }
9682 \tl_new:c { c_fp_ln_10_7_tl }
9683 \tl_set:cn { c_fp_ln_10_7_tl }
9684 { { 1 } { 611809565 } { 095831979 } { 1 } }
9685 \tl_new:c { c_fp_ln_10_8_tl }
9686 \tl_set:cn { c_fp_ln_10_8_tl }
9687 { { 1 } { 842068074 } { 395226547 } { 1 } }
9688 \tl_new:c { c_fp_ln_10_9_tl }
9689 \tl_set:cn { c_fp_ln_10_9_tl }
9690 { { 2 } { 072326583 } { 694641116 } { 1 } }

```

\c_fp_ln_2_1_tl The smaller set for powers of two.

```

\c_fp_ln_2_2_tl
\c_fp_ln_2_3_tl
9691 \tl_new:c { c_fp_ln_2_1_tl }
9692 \tl_set:cn { c_fp_ln_2_1_tl }
9693 { { 0 } { 693147180 } { 559945309 } { 0 } }
9694 \tl_new:c { c_fp_ln_2_2_tl }
9695 \tl_set:cn { c_fp_ln_2_2_tl }
9696 { { 1 } { 386294361 } { 119890618 } { 0 } }
9697 \tl_new:c { c_fp_ln_2_3_tl }
9698 \tl_set:cn { c_fp_ln_2_3_tl }
9699 { { 2 } { 079441541 } { 679835928 } { 0 } }

```

\fp_ln:Nn The approach for logarithms is again based on a mix of tables and Taylor series. Here,
\fp_ln:cn the initial validation is a bit easier and so it is set up earlier, meaning less need to escape
\fp_gln:Nn later on.
\fp_gln:cn

\fp_ln_aux:NNn

\fp_ln_aux:

\fp_ln_exponent:

\fp_ln_internal:

\fp_ln_exponent_units:

\fp_ln_normalise:

\fp_ln_normalise_aux:NNNNNNNN

\fp_ln_mantissa:

\fp_ln_mantissa_aux:

```

9700 \cs_new_protected_nopar:Npn \fp_ln:Nn {
9701   \fp_ln_aux:NNn \tl_set:Nn
9702 }
9703 \cs_new_protected_nopar:Npn \fp_gln:Nn {
9704   \fp_ln_aux:NNn \tl_gset:Nn
9705 }
9706 \cs_generate_variant:Nn \fp_ln:Nn { c }
9707 \cs_generate_variant:Nn \fp_gln:Nn { c }
9708 \cs_new_protected_nopar:Npn \fp_ln_aux:NNn #1#2#3 {
9709   \group_begin:
9710     \fp_split:Nn a {#3}
9711     \fp_standardise:NNNN
9712     \l_fp_input_a_sign_int
9713     \l_fp_input_a_integer_int
9714     \l_fp_input_a_decimal_int
9715     \l_fp_input_a_exponent_int
9716     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9717     \tex_ifnum:D
9718       \etex_numexpr:D
9719         \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
9720         > \c_zero
9721       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9722         \fp_ln_aux:
9723     \tex_else:D
9724       \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9725       {
9726         \group_end:
9727         ##1 \exp_not:N ##2 { \c_zero_fp }
9728       }
9729       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9730       \fp_ln_error_msg:
9731       \tex_fi:D
9732     \tex_else:D
9733       \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9734       {
9735         \group_end:
9736         ##1 \exp_not:N ##2 { \c_zero_fp }
9737       }
9738       \tex_expandafter:D \fp_ln_error_msg:
9739       \tex_fi:D
9740     \fp_tmp:w #1 #2
9741 }

```

As the input at this stage meets the validity criteria above, the argument can now be saved for further processing. There is no need to look at the sign of the input as it must be positive. The function here simply sets up to either do the full calculation or recover the stored value, as appropriate.

```

9742 \cs_new_protected_nopar:Npn \fp_ln_aux: {
9743   \tl_set:Nx \l_fp_arg_tl

```

```

9744 {
9745 +
9746 \int_use:N \l_fp_input_a_integer_int
9747 .
9748 \tex_expandafter:D \use_none:n
9749 \tex_number:D \etex_numexpr:D
9750 \l_fp_input_a_decimal_int + \c_one_thousand_million
9751 e
9752 \int_use:N \l_fp_input_a_exponent_int
9753 }
9754 \etex_ifcsname:D c_fp_ln ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9755 \tex_else:D
9756 \tex_expandafter:D \fp_ln_exponent:
9757 \tex_fi:D
9758 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9759 {
9760 \group_end:
9761 ##1 \exp_not:N ##2
9762 { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
9763 }
9764 }

```

The main algorithm here uses the fact the logarithm can be divided up, first taking out the powers of ten, then powers of two and finally using a Taylor series for the remainder.

$$\ln(10^n \times 2^m \times x) = \ln(10^n) \times \ln(2^m) \times \ln(x)$$

The second point to remember is that

$$\ln(x^{-1}) = -\ln(x)$$

which means that for the powers of 10 and 2 constants are only needed for positive powers.

The first step is to set up the sign for the output functions and work out the powers of ten in the exponent. First the larger powers are sorted out. The values for the constants are the same as those for the smaller ones, just with a shift in the exponent.

```

9765 \cs_new_protected_nopar:Npn \fp_ln_exponent: {
9766 \fp_ln_internal:
9767 \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9768 \tex_else:D
9769 \tex_advance:D \l_fp_output_decimal_int \c_one
9770 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9771 \tex_else:D
9772 \l_fp_output_decimal_int \c_zero
9773 \tex_advance:D \l_fp_output_integer_int \c_one
9774 \tex_fi:D
9775 \tex_fi:D
9776 \fp_standardise:N

```

```

9777 \l_fp_output_sign_int
9778 \l_fp_output_integer_int
9779 \l_fp_output_decimal_int
9780 \l_fp_output_exponent_int
9781 \tl_new:c { c_fp_ln ( \l_fp_arg_tl ) _fp }
9782 \tl_gset:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
9783 {
9784   \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9785   +
9786   \tex_else:D
9787   -
9788   \tex_fi:D
9789   \int_use:N \l_fp_output_integer_int
9790   .
9791   \tex_expandafter:D \use_none:n
9792   \tex_number:D \etex_numexpr:D
9793     \l_fp_output_decimal_int + \c_one_thousand_million
9794   \scan_stop:
9795   e
9796   \int_use:N \l_fp_output_exponent_int
9797 }
9798 }
9799 \cs_new_protected_nopar:Npn \fp_ln_internal: {
9800   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
9801     \l_fp_input_a_exponent_int - \l_fp_input_a_exponent_int
9802     \l_fp_output_sign_int \c_minus_one
9803   \tex_else:D
9804     \l_fp_output_sign_int \c_one
9805   \tex_fi:D
9806   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_nine
9807     \tl_set:Nx \l_fp_tmp_tl
9808     {
9809       \tex_expandafter:D \use_i:nn
9810       \int_use:N \l_fp_input_a_exponent_int
9811     }
9812     \l_fp_input_a_exponent_int
9813     \etex_numexpr:D
9814       \l_fp_input_a_exponent_int - \l_fp_tmp_tl 0
9815     \scan_stop:
9816     \fp_ln_const:nn { 10 } { \l_fp_tmp_tl }
9817     \tex_advance:D \l_fp_exp_exponent_int \c_one
9818     \l_fp_output_integer_int \l_fp_exp_integer_int
9819     \l_fp_output_decimal_int \l_fp_exp_decimal_int
9820     \l_fp_output_extended_int \l_fp_exp_extended_int
9821     \l_fp_output_exponent_int \l_fp_exp_exponent_int
9822   \tex_else:D
9823     \l_fp_output_integer_int \c_zero
9824     \l_fp_output_decimal_int \c_zero
9825     \l_fp_output_extended_int \c_zero
9826     \l_fp_output_exponent_int \c_zero

```

```

9827 \tex_fi:D
9828 \fp_ln_exponent_units:
9829 }

```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```

9830 \cs_new_protected_nopar:Npn \fp_ln_exponent_units: {
9831 \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
9832 \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
9833 \fp_ln_normalise:
9834 \fp_add:NNNNNNNNN
9835 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9836 \l_fp_output_integer_int \l_fp_output_decimal_int
9837 \l_fp_output_extended_int
9838 \l_fp_output_integer_int \l_fp_output_decimal_int
9839 \l_fp_output_extended_int
9840 \tex_fi:D
9841 \fp_ln_mantissa:
9842 }

```

The smaller table-based parts may need to be exponent shifted so that they stay in line with the larger parts. This is similar to the approach in other places, but here there is a need to watch the extended part of the number.

```

9843 \cs_new_protected_nopar:Npn \fp_ln_normalise: {
9844 \tex_ifnum:D \l_fp_exp_exponent_int < \l_fp_output_exponent_int
9845 \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9846 \tex_expandafter:D \use_i:nn \tex_expandafter:D
9847 \fp_ln_normalise_aux:NNNNNNNNN
9848 \int_use:N \l_fp_exp_decimal_int
9849 \tex_expandafter:D \fp_ln_normalise:
9850 \tex_fi:D
9851 }
9852 \cs_new_protected_nopar:Npn
9853 \fp_ln_normalise_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
9854 \tex_ifnum:D \l_fp_exp_integer_int = \c_zero
9855 \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9856 \tex_else:D
9857 \tl_set:Nx \l_fp_tmp_tl
9858 {
9859 \int_use:N \l_fp_exp_integer_int
9860 #1#2#3#4#5#6#7#8
9861 }
9862 \l_fp_exp_integer_int \c_zero
9863 \l_fp_exp_decimal_int \l_fp_tmp_tl \scan_stop:
9864 \tex_fi:D
9865 \tex_divide:D \l_fp_exp_extended_int \c_ten
9866 \tl_set:Nx \l_fp_tmp_tl
9867 {

```

```

9868      #9
9869      \int_use:N \l_fp_exp_extended_int
9870    }
9871    \l_fp_exp_extended_int \l_fp_tmp_tl \scan_stop:
9872    \tex_advance:D \l_fp_exp_exponent_int \c_one
9873  }

```

The next phase is to decompose the mantissa by division by two to leave a value which is in the range $1 \leq x < 2$. The sum of the two powers needs to take account of the sign of the output: if it is negative then the result gets *smaller* as the mantissa gets *bigger*.

```

9874 \cs_new_protected_nopar:Npn \fp_ln_mantissa: {
9875   \l_fp_count_int \c_zero
9876   \l_fp_input_a_extended_int \c_zero
9877   \fp_ln_mantissa_aux:
9878   \tex_ifnum:D \l_fp_count_int > \c_zero
9879     \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
9880     \fp_ln_normalise:
9881     \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9882       \tex_expandafter:D \fp_add:NNNNNNNNN
9883     \tex_else:D
9884       \tex_expandafter:D \fp_sub:NNNNNNNNN
9885     \tex_fi:D
9886     \l_fp_output_integer_int \l_fp_output_decimal_int
9887     \l_fp_output_extended_int
9888     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9889     \l_fp_output_integer_int \l_fp_output_decimal_int
9890     \l_fp_output_extended_int
9891   \tex_fi:D
9892   \tex_ifnum:D
9893     \etex_numexpr:D
9894       \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
9895     \scan_stop:
9896     \tex_expandafter:D \fp_ln_Taylor:
9897   \tex_fi:D
9898 }
9899 \cs_new_protected_nopar:Npn \fp_ln_mantissa_aux: {
9900   \tex_ifnum:D \l_fp_input_a_integer_int > \c_one
9901     \tex_advance:D \l_fp_count_int \c_one
9902     \fp_ln_mantissa_divide_two:
9903     \tex_expandafter:D \fp_ln_mantissa_aux:
9904   \tex_fi:D
9905 }

```

A fast one-shot division by two.

```

9906 \cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two: {
9907   \tex_ifodd:D \l_fp_input_a_decimal_int
9908     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9909   \tex_fi:D

```

```

9910 \tex_ifodd:D \l_fp_input_a_integer_int
9911 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
9912 \tex_fi:D
9913 \tex_divide:D \l_fp_input_a_integer_int \c_two
9914 \tex_divide:D \l_fp_input_a_decimal_int \c_two
9915 \tex_divide:D \l_fp_input_a_extended_int \c_two
9916 }

```

Recovering constants makes use of the same auxiliary code as for exponents.

```

9917 \cs_new_protected_nopar:Npn \fp_ln_const:nn #1#2 {
9918 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9919 \fp_exp_integer_const:nnnn
9920 \tex_csname:D c_fp_ln_ #1 _ #2 _tl \tex_endcsname:D
9921 }

```

The Taylor series for the logarithm function is best implemented using the identity

$$\ln(x) = \ln\left(\frac{y+1}{y-1}\right)$$

with

$$y = \frac{x-1}{x+1}$$

This leads to the series

$$\ln(x) = 2y \left(1 + y^2 \left(\frac{1}{3} + y^2 \left(\frac{1}{5} + y^2 \left(\frac{1}{7} + y^2 \left(\frac{1}{9} + \cdots \right) \right) \right) \right) \right)$$

This expansion has the advantage that a lot of the work can be loaded up early by finding y^2 before the loop itself starts. (In practice, the implementation does the multiplication by two at the end of the loop, and expands out the brackets as this is an overall more efficient approach.)

At the implementation level, the code starts by calculating y and storing that in input **a** (which is no longer needed for other purposes). That is done using the full division system avoiding the parsing step. The value is then switched to a fixed-point representation. There is then some shuffling to get all of the working space set up. At this stage, a lot of registers are in use and so the Taylor series is calculated within a group so that the **output** variables can be used to hold the result. The value of y^2 is held in input **b** (there are a few assignments saved by choosing this over **a**), while input **a** is used for the ‘loop value’.

```

9922 \cs_new_protected_nopar:Npn \fp_ln_Taylor: {
9923 \group_begin:
9924 \l_fp_input_a_integer_int \c_zero
9925 \l_fp_input_a_exponent_int \c_zero
9926 \l_fp_input_b_integer_int \c_two
9927 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9928 \l_fp_input_b_exponent_int \c_zero

```

```

9929 \fp_div_internal:
9930 \fp_ln_fixed:
9931 \l_fp_input_a_integer_int \l_fp_output_integer_int
9932 \l_fp_input_a_decimal_int \l_fp_output_decimal_int
9933 \l_fp_input_a_exponent_int \l_fp_output_exponent_int
9934 \l_fp_input_a_extended_int \c_zero
9935 \l_fp_output_decimal_int \c_zero
9936 \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9937 \l_fp_output_extended_int \l_fp_input_a_extended_int
9938 \fp_mul:NNNNNN
9939 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9940 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9941 \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9942 \l_fp_count_int \c_one
9943 \fp_ln_Taylor_aux:
9944 \cs_set_protected_nopar:Npx \fp_tmp:w
9945 {
9946 \group_end:
9947 \exp_not:N \l_fp_exp_decimal_int
9948 \int_use:N \l_fp_output_decimal_int \scan_stop:
9949 \exp_not:N \l_fp_exp_extended_int
9950 \int_use:N \l_fp_output_extended_int \scan_stop:
9951 \exp_not:N \l_fp_exp_exponent_int
9952 \int_use:N \l_fp_output_exponent_int \scan_stop:
9953 }
9954 \fp_tmp:w

```

After the loop part of the Taylor series, the factor of 2 needs to be included. The total for the result can then be constructed.

```

9955 \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
9956 \tex_ifnum:D \l_fp_exp_extended_int < \c_five_hundred_million
9957 \tex_else:D
9958 \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
9959 \tex_advance:D \l_fp_exp_decimal_int \c_one
9960 \tex_fi:D
9961 \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int
9962 \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9963 \tex_expandafter:D \fp_add:NNNNNNNNN
9964 \tex_else:D
9965 \tex_expandafter:D \fp_sub:NNNNNNNNN
9966 \tex_fi:D
9967 \l_fp_output_integer_int \l_fp_output_decimal_int
9968 \l_fp_output_extended_int
9969 \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
9970 \l_fp_output_integer_int \l_fp_output_decimal_int
9971 \l_fp_output_extended_int
9972 }

```

The usual shifts to move to fixed-point working. This is done using the `output` registers

as this saves a reassignment here.

```

9973 \cs_new_protected_nopar:Npn \fp_ln_fixed: {
9974   \tex_ifnum:D \l_fp_output_exponent_int < \c_zero
9975   \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9976   \tex_expandafter:D \use_i:nn \tex_expandafter:D
9977   \fp_ln_fixed_aux:NNNNNNNNN
9978   \int_use:N \l_fp_output_decimal_int
9979   \tex_expandafter:D \fp_ln_fixed:
9980   \tex_fi:D
9981 }
9982 \cs_new_protected_nopar:Npn
9983   \fp_ln_fixed_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
9984   \tex_ifnum:D \l_fp_output_integer_int = \c_zero
9985   \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9986   \tex_else:D
9987     \tl_set:Nx \l_fp_tmp_tl
9988     {
9989       \int_use:N \l_fp_output_integer_int
9990       #1#2#3#4#5#6#7#8
9991     }
9992     \l_fp_output_integer_int \c_zero
9993     \l_fp_output_decimal_int \l_fp_tmp_tl \scan_stop:
9994   \tex_fi:D
9995   \tex_advance:D \l_fp_output_exponent_int \c_one
9996 }

```

The main loop for the Taylor series: unlike some of the other similar functions, the result here is not the final value and is therefore subject to further manipulation outside of the loop.

```

9997 \cs_new_protected_nopar:Npn \fp_ln_Taylor_aux: {
9998   \tex_advance:D \l_fp_count_int \c_two
9999   \fp_mul:NNNNNN
10000   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10001   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10002   \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10003   \tex_ifnum:D
10004     \etex_numexpr:D
10005       \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10006       > \c_zero
10007     \fp_div_integer:NNNNN
10008     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10009     \l_fp_count_int
10010     \l_fp_exp_decimal_int \l_fp_exp_extended_int
10011     \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
10012     \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
10013     \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
10014     \tex_else:D
10015     \tex_advance:D \l_fp_output_decimal_int \c_one

```

```

10016         \tex_advance:D \l_fp_output_extended_int
10017         -\c_one_thousand_million
10018     \tex_fi:D
10019     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
10020     \tex_else:D
10021         \tex_advance:D \l_fp_output_integer_int \c_one
10022         \tex_advance:D \l_fp_output_decimal_int
10023         -\c_one_thousand_million
10024     \tex_fi:D
10025     \tex_expandafter:D \fp_ln_Taylor_aux:
10026 \tex_fi:D
10027 }

```

`\fp_pow:Nn` The approach used for working out powers is to first filter out the various special cases and
`\fp_pow:cn` then do most of the work using the logarithm and exponent functions. The two storage
`\fp_gpow:Nn` areas are used in the reverse of the ‘natural’ logic as this avoids some re-assignment in
`\fp_gpow:cn` the sanity checking code.

```

\fp_pow_aux:NNn
\fp_pow_aux_i:
\fp_pow_positive:
\fp_pow_negative:
\fp_pow_aux_ii:
\fp_pow_aux_iii:
\fp_pow_aux_iv:
10028 \cs_new_protected_nopar:Npn \fp_pow:Nn {
10029     \fp_pow_aux:NNn \tl_set:Nn
10030 }
10031 \cs_new_protected_nopar:Npn \fp_gpow:Nn {
10032     \fp_pow_aux:NNn \tl_gset:Nn
10033 }
10034 \cs_generate_variant:Nn \fp_pow:Nn { c }
10035 \cs_generate_variant:Nn \fp_gpow:Nn { c }
10036 \cs_new_protected_nopar:Npn \fp_pow_aux:NNn #1#2#3 {
10037     \group_begin:
10038         \fp_read:N #2
10039         \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10040         \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
10041         \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
10042         \l_fp_input_b_exponent_int  \l_fp_input_a_exponent_int
10043         \fp_split:Nn a {#3}
10044         \fp_standardise:NNNN
10045         \l_fp_input_a_sign_int
10046         \l_fp_input_a_integer_int
10047         \l_fp_input_a_decimal_int
10048         \l_fp_input_a_exponent_int
10049     \tex_ifnum:D
10050     \etex_numexpr:D
10051         \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10052         = \c_zero
10053     \tex_ifnum:D
10054     \etex_numexpr:D
10055         \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10056         = \c_zero
10057     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10058     {
10059         \group_end:

```

```

10060         ##1 ##2 { \c_undefined_fp }
10061     }
10062     \tex_else:D
10063     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10064     {
10065         \group_end:
10066         ##1 ##2 { \c_zero_fp }
10067     }
10068     \tex_fi:D
10069     \tex_else:D
10070     \tex_ifnum:D
10071     \etex_numexpr:D
10072     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10073     = \c_zero
10074     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10075     {
10076         \group_end:
10077         ##1 ##2 { \c_one_fp }
10078     }
10079     \tex_else:D
10080     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10081     \fp_pow_aux_i:
10082     \tex_fi:D
10083     \tex_fi:D
10084     \fp_tmp:w #1 #2
10085 }

```

Simply using the logarithm function directly will fail when negative numbers are raised to integer powers, which is a mathematically valid operation. So there are some more tests to make, after forcing the power into an integer and decimal parts, if necessary.

```

10086 \cs_new_protected_nopar:Npn \fp_pow_aux_i: {
10087     \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
10088     \tl_set:Nn \l_fp_sign_tl { + }
10089     \tex_expandafter:D \fp_pow_aux_ii:
10090     \tex_else:D
10091     \l_fp_input_a_extended_int \c_zero
10092     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10093     \group_begin:
10094     \fp_extended_normalise:
10095     \tex_ifnum:D
10096     \etex_numexpr:D
10097     \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10098     = \c_zero
10099     \group_end:
10100     \tl_set:Nn \l_fp_sign_tl { - }
10101     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10102     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10103     \tex_expandafter:D \fp_pow_aux_ii:
10104     \tex_else:D

```

```

10105     \group_end:
10106     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10107     {
10108         \group_end:
10109         ##1 ##2 { \c_undefined_fp }
10110     }
10111     \tex_fi:D
10112 \tex_else:D
10113     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10114     {
10115         \group_end:
10116         ##1 ##2 { \c_undefined_fp }
10117     }
10118     \tex_fi:D
10119 \tex_fi:D
10120 }

```

The approach used here for powers works well in most cases but gives poorer results for negative integer powers, which often have exact values. So there is some filtering to do. For negative powers where the power is small, an alternative approach is used in which the positive value is worked out and the reciprocal is then taken. The filtering is unfortunately rather long.

```

10121 \cs_new_protected_nopar:Npn \fp_pow_aux_ii: {
10122     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10123     \tex_expandafter:D \fp_pow_aux_iv:
10124 \tex_else:D
10125     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10126     \group_begin:
10127     \l_fp_input_a_extended_int \c_zero
10128     \fp_extended_normalise:
10129     \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
10130     \tex_ifnum:D \l_fp_input_a_integer_int > \c_ten
10131     \group_end:
10132     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10133     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10134     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10135     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10136     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10137     \fp_pow_aux_iv:
10138 \tex_else:D
10139     \group_end:
10140     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10141     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10142     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10143     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10144     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10145     \tex_expandafter:D \fp_pow_aux_iii:
10146     \tex_fi:D
10147 \tex_else:D

```

```

10148         \group_end:
10149         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10150         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10151         \tex_expandafter:D \fp_pow_aux_iv:
10152     \tex_fi:D
10153 \tex_else:D
10154     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10155     \fp_pow_aux_iv:
10156 \tex_fi:D
10157 \tex_fi:D
10158 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10159 {
10160     \group_end:
10161     ##1 ##2
10162     {
10163         \l_fp_sign_tl
10164         \int_use:N \l_fp_output_integer_int
10165         .
10166         \tex_expandafter:D \use_none:n
10167         \tex_number:D \etex_numexpr:D
10168         \l_fp_output_decimal_int + \c_one_thousand_million
10169         \scan_stop:
10170     e
10171     \int_use:N \l_fp_output_exponent_int
10172     }
10173 }
10174 }

```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```

10175 \cs_new_protected_nopar:Npn \fp_pow_aux_iii: {
10176     \l_fp_input_a_sign_int \c_one
10177     \fp_pow_aux_iv:
10178     \l_fp_input_a_integer_int \c_one
10179     \l_fp_input_a_decimal_int \c_zero
10180     \l_fp_input_a_exponent_int \c_zero
10181     \l_fp_input_b_integer_int \l_fp_output_integer_int
10182     \l_fp_input_b_decimal_int \l_fp_output_decimal_int
10183     \l_fp_input_b_exponent_int \l_fp_output_exponent_int
10184     \fp_div_internal:
10185 }

```

The business end of the code starts by finding the logarithm of the given base. There is a bit of a shuffle so that this does not have to be re-parsed and so that the output ends up in the correct place. There is also a need to enable using the short-cut for a pre-calculated result. The internal part of the multiplication function can then be used to do the second part of the calculation directly. There is some more set up before doing the exponential: the idea here is to deactivate some internals so that everything works

smoothly.

```

10186 \cs_new_protected_nopar:Npn \fp_pow_aux_iv: {
10187   \group_begin:
10188     \l_fp_input_a_integer_int \l_fp_input_b_integer_int
10189     \l_fp_input_a_decimal_int \l_fp_input_b_decimal_int
10190     \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10191     \fp_ln_internal:
10192     \cs_set_protected_nopar:Npx \fp_tmp:w
10193     {
10194       \group_end:
10195       \exp_not:N \l_fp_input_b_sign_int
10196       \int_use:N \l_fp_output_sign_int \scan_stop:
10197       \exp_not:N \l_fp_input_b_integer_int
10198       \int_use:N \l_fp_output_integer_int \scan_stop:
10199       \exp_not:N \l_fp_input_b_decimal_int
10200       \int_use:N \l_fp_output_decimal_int \scan_stop:
10201       \exp_not:N \l_fp_input_b_extended_int
10202       \int_use:N \l_fp_output_extended_int \scan_stop:
10203       \exp_not:N \l_fp_input_b_exponent_int
10204       \int_use:N \l_fp_output_exponent_int \scan_stop:
10205     }
10206     \fp_tmp:w
10207     \l_fp_input_a_extended_int \c_zero
10208     \fp_mul:NNNNNNNNN
10209     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10210     \l_fp_input_a_extended_int
10211     \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
10212     \l_fp_input_b_extended_int
10213     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10214     \l_fp_input_a_extended_int
10215     \tex_advance:D \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10216     \l_fp_output_integer_int \c_zero
10217     \l_fp_output_decimal_int \c_zero
10218     \l_fp_output_exponent_int \c_zero
10219     \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
10220     \fp_exp_internal:
10221   }

```

116.13 Tests for special values

`\fp_if_undefined_p:N` Testing for an undefined value is easy.

`\fp_if_undefined:N \overline{TF}`

```

10222 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { T , F , TF , p } {
10223   \tex_ifx:D #1 \c_undefined_fp
10224   \prg_return_true:
10225   \tex_else:D
10226     \prg_return_false:
10227   \tex_fi:D

```

```
10228 }
```

`\fp_if_zero_p:N` Testing for a zero fixed-point is also easy.

`\fp_if_zero:N \underline{TF}`

```
10229 \prg_new_conditional:Npnn \fp_if_zero:N #1 { T , F , TF , p } {
10230   \tex_ifx:D #1 \c_zero_fp
10231   \prg_return_true:
10232   \tex_else:D
10233   \prg_return_false:
10234   \tex_fi:D
10235 }
```

116.14 Floating-point conditionals

`\fp_compare:nNn \underline{TF}`

`\fp_compare:NNN \underline{TF}`

`\fp_compare_aux:N`

`\fp_compare_=:`

`\fp_compare_<:`

`\fp_compare_<_aux:`

`\fp_compare_absolute_a>b:`

`\fp_compare_absolute_a<b:`

`\fp_compare_>:`

The idea for the comparisons is to provide two versions: slower and faster. The lead off for both is the same: get the two numbers read and then look for a function to handle the comparison.

```
10236 \prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3
10237 { T , F , TF }
10238 {
10239   \group_begin:
10240     \fp_split:Nn a {#1}
10241     \fp_standardise:NNNN
10242     \l_fp_input_a_sign_int
10243     \l_fp_input_a_integer_int
10244     \l_fp_input_a_decimal_int
10245     \l_fp_input_a_exponent_int
10246     \fp_split:Nn b {#2}
10247     \fp_standardise:NNNN
10248     \l_fp_input_b_sign_int
10249     \l_fp_input_b_integer_int
10250     \l_fp_input_b_decimal_int
10251     \l_fp_input_b_exponent_int
10252     \fp_compare_aux:N #2
10253   }
10254   \prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3
10255   { T , F , TF }
10256   {
10257     \group_begin:
10258     \fp_read:N #3
10259     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10260     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int
10261     \l_fp_input_b_decimal_int   \l_fp_input_a_decimal_int
10262     \l_fp_input_b_exponent_int  \l_fp_input_a_exponent_int
10263     \fp_read:N #1
10264     \fp_compare_aux:N #2
10265   }
```

```

10266 \cs_new_protected_nopar:Npn \fp_compare_aux:N #1 {
10267   \cs_if_exist:cTF { fp_compare_#1: }
10268   { \use:c { fp_compare_#1: } }
10269   {
10270     \group_end:
10271     \prg_return_false:
10272   }
10273 }

```

For equality, the test is pretty easy as things are either equal or they are not.

```

10274 \cs_new_protected_nopar:cpn { fp_compare_=: } {
10275   \tex_ifnum:D \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
10276   \tex_ifnum:D \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
10277   \tex_ifnum:D \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
10278   \tex_ifnum:D
10279     \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
10280   \group_end:
10281   \prg_return_true:
10282   \tex_else:D
10283     \group_end:
10284     \prg_return_false:
10285   \tex_fi:D
10286   \tex_else:D
10287     \group_end:
10288     \prg_return_false:
10289   \tex_fi:D
10290   \tex_else:D
10291     \group_end:
10292     \prg_return_false:
10293   \tex_fi:D
10294   \tex_else:D
10295     \group_end:
10296     \prg_return_false:
10297   \tex_fi:D
10298 }

```

Comparing two values is quite complex. First, there is a filter step to check if one or other of the given values is zero. If it is then the result is relatively easy to determine.

```

10299 \cs_new_protected_nopar:cpn { fp_compare_>: } {
10300   \tex_ifnum:D \etex_numexpr:D
10301     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10302     = \c_zero
10303   \tex_ifnum:D \etex_numexpr:D
10304     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10305     = \c_zero
10306   \group_end:
10307   \prg_return_false:
10308   \tex_else:D
10309     \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero

```



```

10310         \group_end:
10311         \prg_return_false:
10312     \tex_else:D
10313         \group_end:
10314         \prg_return_true:
10315     \tex_fi:D
10316 \tex_fi:D
10317 \tex_else:D
10318     \tex_ifnum:D \etex_numexpr:D
10319         \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10320     = \c_zero
10321     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10322         \group_end:
10323         \prg_return_true:
10324     \tex_else:D
10325         \group_end:
10326         \prg_return_false:
10327     \tex_fi:D
10328 \tex_else:D
10329     \use:c { fp_compare_>_aux: }
10330 \tex_fi:D
10331 \tex_fi:D
10332 }

```

Next, check the sign of the input: this again may give an obvious result. If both signs are the same, then hand off to comparing the absolute values.

```

10333 \cs_new_protected_nopar:cpn { fp_compare_>_aux: } {
10334     \tex_ifnum:D \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
10335     \group_end:
10336     \prg_return_true:
10337 \tex_else:D
10338     \tex_ifnum:D \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
10339     \group_end:
10340     \prg_return_false:
10341 \tex_else:D
10342     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10343         \use:c { fp_compare_absolute_a>b: }
10344     \tex_else:D
10345         \use:c { fp_compare_absolute_a<b: }
10346     \tex_fi:D
10347 \tex_fi:D
10348 \tex_fi:D
10349 }

```

Rather long runs of checks, as there is the need to go through each layer of the input and do the comparison. There is also the need to avoid messing up with equal inputs at each stage.

```

10350 \cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: } {

```

```

10351 \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
10352 \group_end:
10353 \prg_return_true:
10354 \tex_else:D
10355 \tex_ifnum:D \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
10356 \group_end:
10357 \prg_return_false:
10358 \tex_else:D
10359 \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
10360 \group_end:
10361 \prg_return_true:
10362 \tex_else:D
10363 \tex_ifnum:D
10364 \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
10365 \group_end:
10366 \prg_return_false:
10367 \tex_else:D
10368 \tex_ifnum:D
10369 \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
10370 \group_end:
10371 \prg_return_true:
10372 \tex_else:D
10373 \group_end:
10374 \prg_return_false:
10375 \tex_fi:D
10376 \tex_fi:D
10377 \tex_fi:D
10378 \tex_fi:D
10379 \tex_fi:D
10380 }
10381 \cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: } {
10382 \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
10383 \group_end:
10384 \prg_return_true:
10385 \tex_else:D
10386 \tex_ifnum:D \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
10387 \group_end:
10388 \prg_return_false:
10389 \tex_else:D
10390 \tex_ifnum:D \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
10391 \group_end:
10392 \prg_return_true:
10393 \tex_else:D
10394 \tex_ifnum:D
10395 \l_fp_input_b_integer_int < \l_fp_input_a_integer_int
10396 \group_end:
10397 \prg_return_false:
10398 \tex_else:D
10399 \tex_ifnum:D
10400 \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int

```

```

10401         \group_end:
10402         \prg_return_true:
10403     \tex_else:D
10404         \group_end:
10405         \prg_return_false:
10406     \tex_fi:D
10407     \tex_fi:D
10408     \tex_fi:D
10409     \tex_fi:D
10410     \tex_fi:D
10411 }

```

This is just a case of reversing the two input values and then running the tests already defined.

```

10412 \cs_new_protected_nopar:cpn { fp_compare_<: } {
10413     \tl_set:Nx \l_fp_tmp_tl
10414     {
10415         \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
10416         { \int_use:N \l_fp_input_b_sign_int }
10417         \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
10418         { \int_use:N \l_fp_input_b_integer_int }
10419         \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
10420         { \int_use:N \l_fp_input_b_decimal_int }
10421         \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
10422         { \int_use:N \l_fp_input_b_exponent_int }
10423         \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
10424         { \int_use:N \l_fp_input_a_sign_int }
10425         \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
10426         { \int_use:N \l_fp_input_a_integer_int }
10427         \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
10428         { \int_use:N \l_fp_input_a_decimal_int }
10429         \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
10430         { \int_use:N \l_fp_input_a_exponent_int }
10431     }
10432     \l_fp_tmp_tl
10433     \use:c { fp_compare_>: }
10434 }

```

116.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```

10435 \msg_kernel_new:nnnn { fpu } { overflow }
10436 { Number~too~big. }
10437 {
10438     The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \\
10439     Further~errors~may~well~occur!
10440 }

```

```

10441 \cs_new_protected_nopar:Npn \fp_overflow_msg: {
10442   \msg_kernel_error:nn { fpu } { overflow }
10443 }

```

`\fp_exp_overflow_msg:` A slightly more helpful message for exponent overflows.

```

10444 \msg_kernel_new:nnnn { fpu } { exponent-overflow }
10445 { Number~too~big~for~exponent~unit. }
10446 {
10447   The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
10448   unit:~the~maximum~input~value~for~an~exponent~is~230.
10449 }
10450 \cs_new_protected_nopar:Npn \fp_exp_overflow_msg: {
10451   \msg_kernel_error:nn { fpu } { exponent-overflow }
10452 }

```

`\fp_ln_error_msg:` Logarithms are only valid for positive number

```

10453 \msg_kernel_new:nnnn { fpu } { logarithm-input-error }
10454 { Invalid~input~to~ln~function. }
10455 { Logarithms~can~only~be~calculated~for~positive~numbers. }
10456 \cs_new_protected_nopar:Npn \fp_ln_error_msg: {
10457   \msg_kernel_error:nn { fpu } { logarithm-input-error }
10458 }

```

`\fp_trig_overflow_msg:` A slightly more helpful message for trigonometric overflows.

```

10459 \msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
10460 { Number~too~big~for~trigonometry~unit. }
10461 {
10462   The~trigonometry~code~can~only~work~with~numbers~smaller~
10463   than~1000000000.
10464 }
10465 \cs_new_protected_nopar:Npn \fp_trig_overflow_msg: {
10466   \msg_kernel_error:nn { fpu } { trigonometric-overflow }
10467 }
10468 </initex | package>

```

117 Implementation

Announce and ensure that the required packages are loaded.

```

10469 <*package>
10470 \ProvidesExplPackage
10471   {\filename}{\filedate}{\fileversion}{\filedescription}
10472   \package_check_loaded_expl:
10473 </package>
10474 <*initex | package>

```

`\lua_now:n` When LuaTeX is in use, this is all a question of primitives with new names. On the other
`\lua_now:x` hand, for pdfTeX and XeTeX the argument should be removed from the input stream
`\lua_shipout_x:n` before issuing an error. This needs to be expandable, so the same idea is used as for
`\lua_shipout_x:x` V-type expansion, with an appropriately-named but undefined function.
`\lua_shipout:n`
`\lua_shipout:x`
`\lua_wrong_engine:`

```

10475 \luatex_if_engine:TF
10476 {
10477   \cs_new_eq:NN \lua_now:x      \luatex_directlua:D
10478   \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
10479 }
10480 {
10481   \cs_new:Npn \lua_now:x #1      { \lua_wrong_engine: }
10482   \cs_new_protected:Npn \lua_shipout_x:n #1 { \lua_wrong_engine: }
10483 }
10484 \cs_new:Npn \lua_now:n #1 {
10485   \lua_now:x { \exp_not:n {#1} }
10486 }
10487 \cs_generate_variant:Nn \lua_shipout_x:n { x }
10488 \cs_new_protected:Npn \lua_shipout:n #1 {
10489   \lua_shipout_x:n { \exp_not:n {#1} }
10490 }
10491 \cs_generate_variant:Nn \lua_shipout:n { x }
10492 \group_begin:
10493 \char_make_letter:N\!
10494 \char_make_letter:N\%
10495 \cs_gset:Npn \lua_wrong_engine:{%
10496   \LuaTeX engine not in use!%
10497 }%
10498 \group_end:%

```

117.1 Category code tables

`\g_cctab_allocate_int` To allocate category code tables, both the read-only and stack tables need to be followed.
`\g_cctab_stack_int` There is also a sequence stack for the dynamic tables themselves.
`\g_cctab_stack_seq`

```

10499 \int_new:N \g_cctab_allocate_int
10500 \int_set:Nn \g_cctab_allocate_int { -1 }
10501 \int_new:N \g_cctab_stack_int
10502 \seq_new:N \g_cctab_stack_seq

```

`\cctab_new:N` Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```

10503 \cs_new_protected_nopar:Npn \cctab_new:N #1 {
10504   \cs_if_free:NTF #1

```

```

10505 {
10506   \int_gadd:Nn \g_cctab_allocate_int { 2 }
10507   \int_compare:nNnTF
10508     { \g_cctab_allocate_int } < { \c_allocate_max_tl + 1 }
10509     {
10510       \tex_global:D \tex_mathchardef:D #1 \g_cctab_allocate_int
10511       \luatex_initcatcodetable:D #1
10512     }
10513     {
10514       \msg_kernel_error:nxx { code } { out-of-registers } { cctab }
10515     }
10516   }
10517   {
10518     \msg_kernel_error:nxx { code } { variable-already-defined }
10519     { \token_to_str:N #1 }
10520   }
10521 }
10522 \luatex_if_engine:F {
10523   \cs_set_protected_nopar:Npn \cctab_new:N #1 { \lua_wrong_engine: }
10524 }
10525 <*package>
10526 \luatex_if_engine:T {
10527   \cs_set_protected_nopar:Npn \cctab_new:N #1
10528   {
10529     \newcatcodetable #1
10530     \luatex_initcatcodetable:D #1
10531   }
10532 }
10533 </package>

```

`\cctab_begin:N` The aim here is to ensure that the saved tables are read-only. This is done by using a
`\cctab_end:` stack of tables which are not read only, and actually having them as 'in use' copies.
`\l_cctab_tmp_tl`

```

10534 \cs_new_protected_nopar:Npn \cctab_begin:N #1 {
10535   \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
10536   \luatex_catcodetable:D #1
10537   \int_gadd:Nn \g_cctab_stack_int { 2 }
10538   \int_compare:nNnT { \g_cctab_stack_int } > { 268435453 }
10539     { \msg_kernel_error:nn { code } { cctab-stack-full } }
10540   \luatex_savecatcodetable:D \g_cctab_stack_int
10541   \luatex_catcodetable:D \g_cctab_stack_int
10542 }
10543 \cs_new_protected_nopar:Npn \cctab_end: {
10544   \int_gsub:Nn \g_cctab_stack_int { 2 }
10545   \seq_gpop:NN \g_cctab_stack_seq \l_cctab_tmp_tl
10546   \quark_if_no_value:NT \l_cctab_tmp_tl
10547     { \tl_set:Nn \l_cctab_tmp_tl { 0 } }
10548   \luatex_catcodetable:D \l_cctab_tmp_tl \scan_stop:
10549 }
10550 \luatex_if_engine:F {

```

```

10551 \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \lua_wrong_engine: }
10552 \cs_set_protected_nopar:Npn \cctab_end: { \lua_wrong_engine: }
10553 }
10554 <*package>
10555 \luatex_if_engine:T {
10556   \cs_set_protected_nopar:Npn \cctab_begin:N #1
10557     { \BeginCatcodeRegime #1 }
10558   \cs_set_protected_nopar:Npn \cctab_end:
10559     { \EndCatcodeRegime }
10560 }
10561 </package>
10562 \tl_new:N \l_cctab_tmp_tl

```

`\cctab_gset:Nn` Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```

10563 \cs_new_protected:Npn \cctab_gset:Nn #1#2 {
10564   \group_begin:
10565     #2
10566     \luatex_savecatcodetable:D #1
10567   \group_end:
10568 }
10569 \luatex_if_engine:F {
10570   \cs_set_protected_nopar:Npn \cctab_gset:Nn #1#2 { \lua_wrong_engine: }
10571 }

```

`\c_code_cctab` Creating category code tables is easy using the function above. The `other` and `string`
`\c_document_cctab` ones are done by completely ignoring the existing codes as this makes life a lot less
`\c_initex_cctab` complex. The table for `expl3` category codes is always needed, whereas when in package
`\c_other_cctab` mode the rest can be copied from the existing L^AT_EX 2_ε package `luatex`.
`\c_string_cctab`

```

10572 \luatex_if_engine:T {
10573   \cctab_new:N \c_code_cctab
10574   \cctab_gset:Nn \c_code_cctab { }
10575 }
10576 <*package>
10577 \luatex_if_engine:T {
10578   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
10579   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
10580   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
10581   \cs_new_eq:NN \c_string_cctab \CatcodeTableString
10582 }
10583 </package>
10584 <!*package>
10585 \luatex_if_engine:T {
10586   \cctab_new:N \c_document_cctab
10587   \cctab_new:N \c_other_cctab
10588   \cctab_new:N \c_string_cctab

```

```

10589 \cctab_gset:Nn \c_document_cctab
10590 {
10591   \char_make_space:n      { 9 }
10592   \char_make_space:n      { 32 }
10593   \char_make_other:n      { 58 }
10594   \char_make_subscript:n  { 95 }
10595   \char_make_active:n     { 126 }
10596 }
10597 \cctab_gset:Nn \c_other_cctab
10598 {
10599   \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10600   { \char_make_other:n {#1} }
10601 }
10602 \cctab_gset:Nn \c_string_cctab
10603 {
10604   \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10605   { \char_make_other:n {#1} }
10606   \char_make_space:n { 32 }
10607 }
10608 }
10609 <!/package>

10610 </initex | package>

```