The LaTeX3 Sources

The LaTeX3 Project*

2006/08/21 Patch level

Contents

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

Abstract

This package sets up an experimental naming scheme for LaTeX commands. It allows the LaTeX programmer to systematically name functions and variables, and specify the argument types of functions.
The TeX primitives are all given a new name according to these conventions.
Warning: This package, and all packages using it should be regarded as *expermiental*!
The names of these packages, and the names and syntax of any commands defined in them might change at any time.
These conventions are being distributed in this form to encourage discussion and experimentation. It is *not* intentended that these packages be used in 'real' documents at this stage.

1 Conventions

This section gives an overview of the syntax for LaTeX commands that is set up for use in these 'experimental' pacages.

Commands in LaTeX3 are either functions or parameters. All primitive commands of TeX have private names.

1.1 Functions

Functions have the following general syntax:

$$\langle module\rangle\_\langle description\rangle:\langle arg\text{-}spec\rangle$$

where $\langle module\rangle$ is one of the (to be) chosen module names and $\langle description\rangle$ is a verbal description of the functionality. $\langle arg\text{-}spec\rangle$ finally describes the type of arguments that the function takes and is left empty if it is a function without arguments.

All three parts consists of letters only $\langle description\rangle$ is allowed to take further _ characters to separate words is necessary.

Currently there exists some functions which don't have a proper $\langle module\rangle$ name.

As a semi-formalized concept the letter `g` is sometimes used to prefix the $\langle module\rangle$ name and certain parts of the $\langle description\rangle$ to mark the function as "globally acting".

The $\langle arg\text{-}spec\rangle$ currently supports the following types of arguments:

n Unexpanded token (or token-list if in braces) braces.

o One time expanded token or token-list. In the latter case, effectively only the first token in the list gets expanded. Since the expansion might result in more than one token, the result is surrounded for further processing with braces.

x Fully expanded token or token-list. Like `o` but the argument is expanded using `\def:Npx` before it is passed on.

c A character string or a token-list that expands to characters of catcode 11 or 12. This string (after expansion) is used to construct a command name that is eventually passed on.

N,O,X Like `n`, `o`, `x` but the argument must be a single token without any braces around it.

w One or more arguments with "weird" syntax that one has to know by heart or better leave it alone.

p Denotes parameter text specification part, e.g. `#1#2\q_stop#3`.

T,F denotes the "true" or the "false" case in a functional predicate.

Especially for the new names of TEX primitives there are is one more character to denote arguments. It implies that these functions should not be used outside this bootstrapping file.

D Zero or more arguments with "weird" syntax. Uppercase "D" means (DON'T USE IT), i.e., that this is a primitive TEX command that should not show up in code except in the very basic functions of LATEX3 that provide a more sensible interface.

One could perhaps envisage an extended system which allocated letters to denote the various primitive argument types available in TEX, however it seems that this just complicates the system without adding any real benefit, as these primitives would never be used in production code, as higher level packages should offer a better interface. Thus the following letters, although they were considered have not been used. "D" is used in most cases in preference.

i Denotes an integer in TEX notation (which might be a register or ...).

d Denotes a dimension in TEX notation.

g Denotes a glue in TEX notation.

m Denotes an muglue or mukern in TEX notation.

b Denotes a box specification in TEX notation (again something pretty arbitrary).

r Denotes a rule specification in TEX notation.

Some of the primitive functions below are flagged "D" even if they actually might be useful in average code. So certainly there are some adjustments necessary. It all depends whether or not we provide some safer interface or leave them alone.

1.2 Parameters

Parameter names have the following general syntax:

\⟨*access*⟩_⟨*module*⟩_⟨*description*⟩_⟨*type*⟩

⟨*module*⟩ and ⟨*description*⟩ is as above. ⟨*type*⟩ should denote the type of parameter if this helps in using it. The currently used types are:

int Integer valued.

factor Another integer value type. Used for things where the parameter is used as a factor for something else.

status The sort of boolean stuff TEX provides. Essentially an integer with the meaning `0` = 'off' and other values may or may not have sensible meanings.

pen Another integer describing penalties.

dem The demerits.

dim A dimension.

skip A glue value.

toks A toks register (sort of).

char An integer denoting a character.

muskip A math unit.

⟨*access*⟩ describes how the parameter can be accessed. The following characters are possible:

c A constant. Should not be set in the code except with special functions to define the value for the whole processing.

C A constant according to TEX's rules. Can not be changed at all.

l A local variable which therefore should not be changed globally.

L A local variable that is usually set (and/or reset) by TEX itself.

g A global variable.

G A global variable that is usually set (and/or reset by TEX.

R A variable that is set (and changed) by TEX and can not be changed by in the code (read-only).

2 Modules

Nearly all operations of LaTeX3 are carried out by calling control sequences. For better programming concepts many types of functions are identified and gathered in modules. Functions in such modules starts with special prefixes, for example `\tlp_` is the prefix for functions dealing with token list pointers.

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

3 Basics

Here we describe those functions that 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.

3.1 Predicates and conditionals

3.1.1 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_`.

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

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

`\if_true:` always executes ⟨*true code*⟩, while `\if_false:` always executes ⟨*false code*⟩. `\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional.

\if_meaning:NN ⟨*cs1*⟩ ⟨*cs2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_cs_meaning_eq:NN ⟨*cs1*⟩ ⟨*cs2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_token_eq:NN ⟨*token1*⟩ ⟨*token2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

\if_meaning:NN
\if_cs_meaning_eq:NN
\if_token_eq:NN

\if_meaning:NN executes ⟨*true code*⟩ when the replacement text, i.e., the expansion of ⟨*cs1*⟩ and ⟨*cs2*⟩ are the same, otherwise it executes ⟨*false code*⟩. However this name isn't really that good. What the TEX primitive does is compare two tokens to see if they are equal. Hence this is actually a `token` functions. A similar argument applies to the situation where it is used to compare control sequences, where it is the meaning being compared. Something to be cleaned up at some point.

\if:w ⟨*token1*⟩ ⟨*token2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_charcode:w ⟨*token1*⟩ ⟨*token2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_catcode:w ⟨*token1*⟩ ⟨*token2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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

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.

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

\if_cs_exist:N
\if_cs_exist:w

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

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

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

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

3.1.2 Non-primitive conditionals

\cs_if_eq_p:NN

\cs_if_eq_p:NN ⟨*cs1*⟩ ⟨*cs2*⟩

Returns 'true' if ⟨*cs1*⟩ and ⟨*cs2*⟩ are textually the same, i.e. have the same name, otherwise it returns 'false'.

\cs_if_eq:NNTF
\cs_if_eq:NNT
\cs_if_eq:NNF
\cs_if_eq:cNTF
\cs_if_eq:cNT
\cs_if_eq:cNF
\cs_if_eq:NcTF
\cs_if_eq:NcT
\cs_if_eq:NcF
\cs_if_eq:ccTF
\cs_if_eq:ccT
\cs_if_eq:ccF
\cs_if_eq:NNTF ⟨*cs1*⟩ ⟨*cs2*⟩ {⟨*true code*⟩}{⟨*false code*⟩}

These functions check if ⟨*cs1*⟩ and ⟨*cs2*⟩ have same meaning and then execute either ⟨*true code*⟩ or ⟨*false code*⟩.

\cs_if_free_p:N
\cs_if_free_p:N ⟨*cs*⟩

Returns 'true' if ⟨*cs*⟩ is either undefined or equal to `\scan_stop:`. However, it returns 'false' if ⟨*cs*⟩ is textually `\c_undefined` (the constantly undefined function), or textually `\scan_stop:`.

\cs_if_free:NTF
\cs_if_free:NT
\cs_if_free:NF
\cs_if_free:cTF
\cs_if_free:cT
\cs_if_free:cF
\cs_if_free:NTF ⟨*cs*⟩ {⟨*true code*⟩}{⟨*false code*⟩}

These functions check if ⟨*cs*⟩ is free and then execute either ⟨*true code*⟩ or ⟨*false code*⟩.

TEXhackers note: The conditional `\cs_if_free:cTF` is the LaTeX3 implementation of the LaTeX2 function `\@ifundefined`. The other functions haven't been around before.

\cs_if_really_free:cTF
\cs_if_really_free:cF
\cs_if_really_free:cT
\cs_if_really_free:cTF {⟨*tokens*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Similar to `\cs_if_free:cTF` but does not put anything previously undefined into the hash table. Useful for special control sequences like `\foo/bar` which cannot be entered as one token.

\cs_if_exist_p:N
\cs_if_exist_p:N ⟨*cs*⟩

This function does the opposite of `\cs_if_free_p:N`.

\cs_if_exist:NTF
\cs_if_exist:NT
\cs_if_exist:NF
\cs_if_exist:cTF
\cs_if_exist:cT
\cs_if_exist:cF
\cs_if_exist:NTF ⟨*cs*⟩ {⟨*true code*⟩}{⟨*false code*⟩}
These functions check if ⟨*cs*⟩ exists and then execute either ⟨*true code*⟩ or ⟨*false code*⟩. Exactly the opposite of \cs_if_free:NTF.

\cs_if_really_exist:cTF
\cs_if_really_exist:cF
\cs_if_really_exist:cT
\cs_if_really_exist:cTF {⟨*tokens*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

The opposite of \cs_if_really_free:cTF.

\chk_new_cs:N
\chk_new_cs:N ⟨*cs*⟩
This function checks that ⟨*cs*⟩ is so far either undefined or equals \scan_stop: (the function that is assigned to newly created control sequences by TEX when \cs:w ... \cs_end: is used).

\chk_exist_cs:N
\chk_exist_cs:c
\chk_exist_cs:N ⟨*cs*⟩
This function checks that ⟨*cs*⟩ is defined. If it is not an error is generated.

\c_true
\c_false
Constants that represent 'true' or 'false', respectively. Used to implement predicates.

3.2 Selecting and discarding tokens from the input stream

The conditional processing could not have been implemented without being able to gobble and select which tokens to use from the input stream.

\use_none:n
\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn
\use_none:nnnnnnnnn
\use_none:n {⟨*arg1*⟩}
\use_none:nn {⟨*arg1*⟩}{⟨*arg2*⟩}
These functions gobble the tokens or brace groups from the input stream.

`\use_arg_i:n` `\use_arg_i:n { ⟨code1⟩ }`
Function that executes the next argument after removing the surrounding braces. Used to implement conditionals.

`\use_arg_i:nn`
`\use_arg_ii:nn` `\use_arg_i:nn { ⟨code1⟩ }{ ⟨code2⟩ }`
Functions that execute the first or second argument respectively, after removing the surrounding braces. Primarily used to implement conditionals.

`\use_arg_i:nnn`
`\use_arg_ii:nnn`
`\use_arg_iii:nnn` `\use_arg_i:nnn { ⟨arg1⟩ }{ ⟨arg2⟩ }{ ⟨arg3⟩ }`
Functions that pick up one of three arguments and execute them after removing the surrounding braces. Should be described somewhere else.

`\use_arg_i:nnnn`
`\use_arg_ii:nnnn`
`\use_arg_iii:nnnn`
`\use_arg_iv:nnnn` `\use_arg_i:nnnn { ⟨arg1⟩ }{ ⟨arg2⟩ }{ ⟨arg3⟩ }{ ⟨arg4⟩ }`
Functions that pick up one of four arguments and execute them after removing the surrounding braces.

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

`\use_none_delimit_by_q_nil:w`
`\use_none_delimit_by_q_stop:w` `\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil`

Gobbles ⟨*balanced text*⟩. Useful in gobbling the remainder in a list structure.

`\use_arg_i_delimit_by_q_nil:nw`
`\use_arg_i_delimit_by_q_stop:nw` `\use_arg_i_delimit_by_q_nil:nw {⟨arg⟩} ⟨balanced text⟩ \q_nil`

Gobbles ⟨*balanced text*⟩ and executes ⟨*arg*⟩ afterwards. This can also be used to get the first item in a token list.

`\use_arg_i_after_fi:nw`
`\use_arg_i_after_else:nw`
`\use_arg_i_after_or:nw`

```
\use_arg_i_after_fi:nw {⟨arg⟩} \fi:
\use_arg_i_after_else:nw {⟨arg⟩} \else: ⟨balanced text⟩ \fi:
\use_arg_i_after_or:nw {⟨arg⟩} \or: ⟨balanced text⟩ \fi:
```

Executes ⟨*arg*⟩ after executing closing out `\fi:`.

3.3 Internal functions

`\cs:w`
`\cs_end:`

`\cs:w` ⟨*tokens*⟩ `\cs_end:`

This is the TEX internal way of generating a control sequence from some token list. ⟨*tokens*⟩ get expanded and must ultimately result in a sequence of characters.

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

`\pref_global:D`
`\pref_long:D`
`\pref_protected:D`

`\pref_global:D \def:Npn`

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 `\write`s etc.

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

TEXhackers note: These prefixes are the primitives `\global`, `\long`, and `\protected`. The `\outer` isn't used at all within LATEX3 because . . .

`\io_put_log:x`
`\io_put_term:x`
`\io_put_deferred:Nx`

`\io_put_log:x {`⟨*message*⟩`}`
`\io_put_deferred:Nx` ⟨*write_stream*⟩ `{`⟨*message*⟩`}`

Writes ⟨*message*⟩ to either to log or the terminal.

3.4 Defining functions

There are two types of function definitions in LATEX3: versions that check if the function name is still unused, and versions that simply make the definition. The later 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, so in most cases the programmer just want to input the number of arguments, which is basically how `\newcommand` in LATEX 2$_\varepsilon$ works. Therefore we provide functions that expect a number

as the primary type and later on in this module you can find the ones with the more primitive syntax.

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.

TEXhackers note: While TEX makes all definition functions directly available to the user LATEX3 hides them very carefully to avoid the problems with definitions that are overwritten accidentally. Many functions that are in TEX a combination of prefixes and definition functions are provided as individual functions.

3.4.1 Defining new functions

Firstly comes to variants most used namely those taking a number to denote the number of arguments.

`\def_new:NNn`
`\def_new:NNx`
`\def_new:cNn`
`\def_new:cNx`
`\def_new:NNn` ⟨*cs*⟩ ⟨*num*⟩ `{` ⟨*code*⟩ `}`
Defines a new function, making sure that ⟨*cs*⟩ is unused so far. ⟨*num*⟩ is the number of arguments which is in the interval $[0, 9]$ otherwise 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).

`\gdef_new:NNn`
`\gdef_new:cNn`
`\gdef_new:NNx`
`\gdef_new:cNx`
`\gdef_new:NNn` ⟨*cs*⟩ ⟨*num*⟩ `{` ⟨*code*⟩ `}`
Like `\def_new:NNn` but defines the new function globally.

`\def_long_new:NNn`
`\def_long_new:NNx`
`\def_long_new:cNn`
`\def_long_new:cNx`
`\def_long_new:NNn` ⟨*cs*⟩ ⟨*num*⟩ `{` ⟨*code*⟩ `}`
Defines a function that may contain `\par` tokens in the argument(s) when called. This is not allowed for normal functions.

\gdef_long_new:NNn
\gdef_long_new:NNx
\gdef_long_new:cNn
\gdef_long_new:cNx
\gdef_long_new:NNn ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Global versions of the above functions.

\def_protected_new:NNn
\def_protected_new:NNx
\def_protected_new:cNn
\def_protected_new:cNx
\def_protected_new:NNn ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }

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

\gdef_protected_new:NNn
\gdef_protected_new:NNx
\gdef_protected_new:cNn
\gdef_protected_new:cNx
\gdef_protected_new:NNn ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }

Global versions of the above functions.

\def_protected_long_new:NNn
\def_protected_long_new:NNx
\def_protected_long_new:cNn
\def_protected_long_new:cNx
\def_protected_long_new:NNn ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }

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

\gdef_protected_long_new:NNn
\gdef_protected_long_new:NNx
\gdef_protected_long_new:cNn
\gdef_protected_long_new:cNx
\gdef_protected_long_new:NNn ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }

Global versions of the above functions.

Secondly comes the ones where the programmer can use delimited arguments. Rarely needed outside the kernel.

\def_new:Npn
\def_new:Npx
\def_new:cpn
\def_new:cpx
\def_new:Npn ⟨*cs*⟩ ⟨*parms*⟩ { ⟨*code*⟩ }
Defines a new function, making sure that ⟨*cs*⟩ is unused so far. ⟨*parms*⟩ may consist of arbitrary parameter specification in TEX 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).

`\gdef_new:Npn`
`\gdef_new:cpn`
`\gdef_new:Npx`
`\gdef_new:cpx`
`\gdef_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`
Like `\def_new:Npn` but defines the new function globally. See comments above.

`\def_long_new:Npn`
`\def_long_new:Npx`
`\def_long_new:cpn`
`\def_long_new:cpx`
`\def_long_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.

`\gdef_long_new:Npn`
`\gdef_long_new:Npx`
`\gdef_long_new:cpn`
`\gdef_long_new:cpx`
`\gdef_long_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`
Global versions of the above functions.

`\def_protected_new:Npn`
`\def_protected_new:Npx`
`\def_protected_new:cpn`
`\def_protected_new:cpx`
`\def_protected_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

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

`\gdef_protected_new:Npn`
`\gdef_protected_new:Npx`
`\gdef_protected_new:cpn`
`\gdef_protected_new:cpx`
`\gdef_protected_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

Global versions of the above functions.

`\def_protected_long_new:Npn`
`\def_protected_long_new:Npx`
`\def_protected_long_new:cpn`
`\def_protected_long_new:cpx`
`\def_protected_long_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

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

`\gdef_protected_long_new:Npn`
`\gdef_protected_long_new:Npx`
`\gdef_protected_long_new:cpn`
`\gdef_protected_long_new:cpx`
`\gdef_protected_long_new:Npn` ⟨*cs*⟩ ⟨*parms*⟩ { ⟨*code*⟩ }

Global versions of the above functions.

`\let_new:NN`
`\let_new:cN`
`\let_new:Nc`
`\let_new:cc`
`\glet_new:NN`
`\glet_new:cN`
`\glet_new:Nc`
`\glet_new:cc`
`\let_new:NN` ⟨*cs1*⟩ ⟨*cs2*⟩
Gives the function ⟨*cs1*⟩ the current meaning of ⟨*cs2*⟩. Again, we may do this always globally.

3.4.2 Undefining functions

`\cs_gundefine:N` `\cs_gundefine:N` ⟨*cs*⟩
Undefines the control sequence.

3.4.3 Defining internal functions (no checks)

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.

First comes the versions expecting a number to denote the number of arguments.

`\def:NNn`
`\def:NNx`
`\def:cNn`
`\def:cNx`
`\def:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Like `\def_new:NNn` etc. but does not check the ⟨*cs*⟩ name.

\gdef:NNn
\gdef:NNx
\gdef:cNn
\gdef:cNx
`\gdef:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Like `\def:NNn` but defines the ⟨*cs*⟩ globally.

\def_long:NNn
\def_long:NNx
\def_long:cNn
\def_long:cNx
`\def_long:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Like `\def:NNn` but allows `\par` tokens in the arguments of the function being defined.

\gdef_long:NNn
\gdef_long:NNx
\gdef_long:cNn
\gdef_long:cNx
`\gdef_long:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Global variant of `\def_long:NNn`.

\def_protected:NNn
\def_protected:cNn
\def_protected:NNx
\def_protected:cNx
`\def_protected:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*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:NN \use_noop:`.

\gdef_protected:NNn
\gdef_protected:cNn
\gdef_protected:NNx
\gdef_protected:cNx
`\gdef_protected:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }
Global versions of the above functions.

\def_protected_long:NNn
\def_protected_long:cNn
\def_protected_long:NNx
\def_protected_long:cNx
`\def_protected_long:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*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.

\gdef_protected_long:NNn
\gdef_protected_long:cNn
\gdef_protected_long:NNx
\gdef_protected_long:cNx
`\gdef_protected_long:NNn` ⟨*cs*⟩ ⟨*num*⟩ { ⟨*code*⟩ }

Global versions of the above functions.

Secondly the ones that use the primitive parameter build-up:

\def:Npn
\def:Npx
\def:cpn
\def:cpx

`\def:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

Like `\def_new:Npn` etc. but does not check the ⟨*cs*⟩ name.

TEXhackers note: `\def:Npn` is the LATEX3 name for TEX's `\def` and `\def:Npx` corresponds to the primitive `\edef`. The `\def:cpn` function was known in LATEX2 as `\@namedef`. `\def:cpx` has no equivalent.

\gdef:Npn
\gdef:Npx
\gdef:cpn
\gdef:cpx

`\gdef:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

Like `\def:Npn` but defines the ⟨*cs*⟩ globally.

TEXhackers note: `\gdef:Npn` and `\gdef:Npx` are known to TEXhackers as `\gdef` and `\xdef`.

\def_long:Npn
\def_long:Npx
\def_long:cpn
\def_long:cpx

`\def_long:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

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

\gdef_long:Npn
\gdef_long:Npx
\gdef_long:cpn
\gdef_long:cpx

`\gdef_long:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

Global variant of `\def_long:Npn`.

\def_protected:Npn
\def_protected:cpn
\def_protected:Npx
\def_protected:cpx

`\def_protected:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*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:NN \use_noop:`.

`\gdef_protected:Npn`
`\gdef_protected:cpn`
`\gdef_protected:Npx`
`\gdef_protected:cpx`
`\gdef_protected:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`
Global versions of the above functions.

`\def_protected_long:Npn`
`\def_protected_long:cpn`
`\def_protected_long:Npx`
`\def_protected_long:cpx`
`\def_protected_long:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*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.

`\gdef_protected_long:Npn`
`\gdef_protected_long:cpn`
`\gdef_protected_long:Npx`
`\gdef_protected_long:cpx`
`\gdef_protected_long:Npn` ⟨*cs*⟩ ⟨*parms*⟩ `{` ⟨*code*⟩ `}`

Global versions of the above functions.

`\let:NN`
`\let:cN`
`\let:Nc`
`\let:cc`
`\glet:NN`
`\glet:cN`
`\glet:Nc`
`\glet:cc`
`\let:cN` ⟨*cs1*⟩ ⟨*cs2*⟩
Gives the function ⟨*cs1*⟩ the current meaning of ⟨*cs2*⟩. Again, we may always do this globally.

`\let:NwN` ⟨*cs1*⟩ ⟨*cs2*⟩
`\let:NwN`
`\let:NwN` ⟨*cs1*⟩ `=` ⟨*cs2*⟩
These functions assign the meaning of ⟨*cs2*⟩ locally or globally to the function ⟨*cs1*⟩. Because the TEX primitive operation is being used which may have an equal sign and (a certain number of) spaces between ⟨*cs1*⟩ and ⟨*cs2*⟩ the name contains a `w`. (Not happy about this convention!).

TEXhackers note: `\let:NwN` is the LATEX3 name for TEX's `\let`.

3.5 Defining test functions

\def_test_function:npn
\def_long_test_function:npn
\def_test_function_new:npn
\def_long_test_function_new:npn `\def_test_function_new:npn {name} ⟨parms⟩ {⟨test⟩}`

Define all the common test cases for a simple test to reduce the risk of typos. As an example here's how we defined the functions `\cs_free:cTF`, `\cs_free:cT` and `\cs_free:cF`. You just have to fill in the test.

```
\def_test_function:npn{cs_free:c} #1 {
  \exp_after:NN \if_meaning:NN \cs:w#1\cs_end: \scan_stop:}
```

Be careful not to use this function inside some primitive conditional as TEX will most likely get confused because of the unmatched conditionals.

3.6 The innards of a function

\cs_to_str:N `\cs_to_str:N ⟨cs⟩`
This function return the name of ⟨*cs*⟩ as a sequence of letters with the escape character removed.

\token_to_string:N `\token_to_string:N ⟨arg⟩`
This function return the name of ⟨*arg*⟩ as a sequence of letters including the escape character.

\token_to_meaning:N `\token_to_meaning:N ⟨arg⟩`
This function returns the type and definition of ⟨*arg*⟩ as a sequence of letters.

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

3.7 Grouping and scanning

\scan_stop: `\scan_stop:`
This function stops TEX's scanning ahead when ending a number.

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

`\group_begin:` `\group_end:` `\group_begin:` ⟨...⟩ `\group_end:`
Encloses ⟨...⟩ inside a group.

TEXhackers note: These are the TEX primitives `\begingroup` and `\endgroup` renamed.

3.8 Engine specific definitions

`\engine_aleph:TF` `\engine_aleph:TF {⟨true code⟩} {⟨false code⟩}`
This function detects if we're running an Aleph based format. This is particularly useful when allocating registers.

4 The chk module

To ensure that functions and variables are properly used certain checking functions are implemented that may or may not be compiled into the final format.

4.1 Functions

`\chk_var_or_const:N` `\chk_var_or_const:c` `\chk_var_or_const:N` ⟨*cs*⟩
Checks that ⟨*cs*⟩ is a proper variable or constant which means that its name starts out with `\L`, `\l`, `\G`, `\g`, `\R`, `\C`, `\c`, or `\q`.

`\chk_local:N` `\chk_local:c` `\chk_global:N` `\chk_global:c` `\chk_local:N` ⟨*cs*⟩
Checks that ⟨*cs*⟩ is a proper local or global variable. This means that its name starts out with `\L`, `\l`, or `\G`, `\g` respectively.

`\chk_local_or_pref_global:N` `\pref_global_chk:` To allow implementations where we precede some function with `\pref_global:D` without loosing the possibility to check for the correct variable type the following helper functions can be used: `\chk_local_or_pref_global:N` ⟨*cs*⟩ is the variable check which is usually let to `\chk_local:N`, i.e. it will check that its argument is a local variable. This behavior will be changed by `\pref_global_chk:`. This function first changes `\chk_local_or_pref_global:N` to check for global variables then

it issues a \pref_global:D. After use \chk_local_or_pref_global:N will restore itself to \chk_local:N. So, if we use \chk_local_or_pref_global:N inside some function \foo_bar:n we can implement a global version \foo_gbar:n by defining

```
\def_new:Npn \foo_gbar:n {\pref_global_chk: \foo_bar:n }
```

provided that \foo_bar:n is built in a way that prefixing it with \pref_global:D turns its operation into a global one. See implementation for details.

4.2 Constants

\c_undefined This constant is always undefined and therefore can be used to check for free function names.

4.3 Internal functions

\chk_global_aux:w
\chk_local_aux:w
\chk_var_or_const_aux:w Helper functions that implement the checking.

5 Token list pointers

LaTeX3 stores token lists in so called 'token list pointers'. Variables of this type get the suffix tlp and functions of this type have the prefix tlp. To use a token list pointer you simply call the corresponding variable.

Often you find yourself with not a token list pointer but an arbitrary token list which has to undergo certain tests. We will prefix these functions with tlist. While token list pointers are always single tokens, token lists are always surrounded by braces. Perhaps these token lists should have their own module but for now I decided to put them here because there is quite a bit of overlap with token list pointers.

5.1 Functions

\tlp_new:Nn
\tlp_new:cn
\tlp_new:Nx \tlp_new:Nn ⟨*tlp*⟩ { ⟨*initial token list*⟩ }
Defines ⟨*tlp*⟩ to be a new variable (or constant) of type token list pointer. ⟨*initial token*

list⟩ is the initial value of ⟨*tlp*⟩. This makes it possible to assign values to a constant token list pointer.

\tlp_use:N
\tlp_use:c
\tlp_use:N ⟨*tlp*⟩
Function that inserts the ⟨*tlp*⟩ into the processing stream.

\tlp_set:Nn
\tlp_set:Nc
\tlp_set:No
\tlp_set:Nf
\tlp_set:Nx
\tlp_gset:Nn
\tlp_gset:Nc
\tlp_gset:No
\tlp_gset:Nx
\tlp_gset:cn
\tlp_gset:cx
\tlp_set:Nn ⟨*tlp*⟩ { ⟨*token list*⟩ }
Defines ⟨*tlp*⟩ to hold the token list ⟨*token list*⟩. Global variants of this command assign the value globally the other variants expand the ⟨*token list*⟩ up to a certain level before the assignment or interpret the ⟨*token list*⟩ as a character list and form a control sequence out of it.

\tlp_clear:N
\tlp_clear:c
\tlp_gclear:N
\tlp_gclear:c
\tlp_clear:N ⟨*tlp*⟩
The ⟨*tlp*⟩ is locally or globally cleared. The `c` variants will generate a control sequence name which is then interpreted as ⟨*tlp*⟩ before clearing.

\tlp_clear_new:N
\tlp_clear_new:c
\tlp_gclear_new:N
\tlp_gclear_new:c
\tlp_clear_new:N ⟨*tlp*⟩
These functions check if ⟨*tlp*⟩ exists. If it does it will be cleared; if it doesn't it will be allocated.

\tlp_put_left:Nn
\tlp_put_left:No
\tlp_gput_left:Nn
\tlp_gput_left:No
\tlp_gput_left:Nx
\tlp_put_right:Nn
\tlp_put_right:cc
\tlp_gput_right:Nn
\tlp_gput_right:No
\tlp_gput_right:cn
\tlp_gput_right:co

`\tlp_put_left:Nn` ⟨*tlp*⟩ `{` ⟨*token list*⟩ `}`

These functions will append ⟨*token list*⟩ to the left or right of ⟨*tlp*⟩. Assignment is done either locally or globally and ⟨*token list*⟩ might be subject to expansion before assignment.

\tlp_set_eq:NN
\tlp_set_eq:Nc
\tlp_set_eq:cN
\tlp_set_eq:cc
\tlp_gset_eq:NN
\tlp_gset_eq:Nc
\tlp_gset_eq:cN
\tlp_gset_eq:cc

`\tlp_set_eq:NN` ⟨*tlp1*⟩ ⟨*tlp2*⟩

Fast form for `\tlp_set:No` ⟨*tlp1*⟩ `{` ⟨*tlp2*⟩ `}`
when ⟨*tlp2*⟩ is known to be a variable of type token list pointer.

\tlp_to_str:N
\tlp_to_str:c

`\tlp_to_str:N` ⟨*tlp*⟩

This function returns the token list kept in ⟨*tlp*⟩ as a string list with all characters catcoded to 'other'.

5.2 Predicates and conditionals

\tlp_if_empty_p:N
\tlp_if_empty_p:c

`\tlp_if_empty_p:N` ⟨*tlp*⟩

This predicate returns 'true' if ⟨*tlp*⟩ is 'empty' i.e., doesn't contain any tokens.

\tlp_if_empty:NTF
\tlp_if_empty:NT
\tlp_if_empty:NF
\tlp_if_empty:cTF
\tlp_if_empty:cT
\tlp_if_empty:cF

`\tlp_if_empty:NTF` ⟨*tlp*⟩ {⟨*true code*⟩}{⟨*false code*⟩}

Execute ⟨*true code*⟩ if ⟨*tlp*⟩ is empty and ⟨*false code*⟩ if it contains any tokens.

\tlp_if_eq_p:NN
\tlp_if_eq_p:cN
\tlp_if_eq_p:Nc
\tlp_if_eq_p:cc

`\tlp_if_eq_p:NN` ⟨*tlp1*⟩ ⟨*tlp2*⟩
Predicate function which returns 'true' if the two token list pointers are identical and 'false' otherwise.

\tlp_if_eq:NNTF
\tlp_if_eq:NNT
\tlp_if_eq:NNF
\tlp_if_eq:cNTF
\tlp_if_eq:cNT
\tlp_if_eq:cNF
\tlp_if_eq:NcTF
\tlp_if_eq:NcT
\tlp_if_eq:NcF
\tlp_if_eq:ccTF
\tlp_if_eq:ccT
\tlp_if_eq:ccF

`\tlp_if_eq:NNTF` ⟨*tlp1*⟩ ⟨*tlp2*⟩ {⟨*true code*⟩}{⟨*false code*⟩}
Execute ⟨*true code*⟩ if ⟨*tlp1*⟩ holds the same token list as ⟨*tlp2*⟩ and ⟨*false code*⟩ otherwise.

5.3 Token lists

\tlist_if_eq:nnTF
\tlist_if_eq:nnT
\tlist_if_eq:nnF
\tlist_if_eq:noTF
\tlist_if_eq:noT
\tlist_if_eq:noF

`\tlist_if_eq:nnTF` {⟨*tlist1*⟩} {⟨*tlist2*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
Execute ⟨*true code*⟩ if the two token lists ⟨*tlist1*⟩ and ⟨*tlist2*⟩ are identical.

\tlist_if_empty_p:n
\tlist_if_empty_p:o
\tlist_if_empty:nTF
\tlist_if_empty:nT
\tlist_if_empty:nF
\tlist_if_empty:oTF
\tlist_if_empty:oT
\tlist_if_empty:oF

`\tlist_if_empty:nTF` {⟨*tlist*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
Execute ⟨*true code*⟩ if ⟨*tlist*⟩ doesn't contain any tokens and ⟨*false code*⟩ otherwise.

\tlist_if_blank_p:n
\tlist_if_blank:nTF
\tlist_if_blank:nT
\tlist_if_blank:nF
\tlist_if_blank_p:o
\tlist_if_blank:oTF
\tlist_if_blank:oT
\tlist_if_blank:oF

\tlist_if_blank:nTF {⟨*tlist*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Execute ⟨*true code*⟩ if ⟨*tlist*⟩ is blank meaning that it is either empty or contains only blank spaces.

\tlist_to_lowercase:n
\tlist_to_uppercase:n

\tlist_to_lowercase:n {⟨*tlist*⟩}

\tlist_to_lowercase:n converts all tokens in ⟨*tlist*⟩ to their lower case representation. Similar for \tlist_to_uppercase:n.

TEXhackers note: These are the TEX primitives \lowercase and \uppercase renamed.

\tlist_to_str:n

\tlist_to_str:n {⟨*tlist*⟩}

This function turns its argument into a string where all characters have catcode 'other'.

TEXhackers note: This is the ε-TEX primitive \detokenize.

\tlist_map_function:nN
\tlp_map_function:NN
\tlp_map_function:cN

\tlist_map_function:nN {⟨*tlist*⟩} ⟨*function*⟩
\tlp_map_function:NN ⟨*tlp*⟩ ⟨*function*⟩

Runs through all elements in a tlist 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.

\tlist_map_inline:nn
\tlp_map_inline:Nn
\tlp_map_inline:cn

\tlist_map_inline:nn {⟨*tlist*⟩} {⟨*inline function*⟩}
\tlp_map_inline:Nn ⟨*tlp*⟩ {⟨*inline function*⟩}

Allows a syntax like \tlist_map_inline:nn {⟨*tlist*⟩} {\token_to_string:N ##1}. This renders it non-expandable though. Remember to double the #s for each level.

\tlist_map_variable:nNn
\tlp_map_variable:NNn
\tlp_map_variable:cNn

\tlist_map_variable:nNn {⟨*tlist*⟩} ⟨*temp*⟩ {⟨*action*⟩}
\tlp_map_variable:NNn ⟨*tlp*⟩ ⟨*temp*⟩ {⟨*action*⟩}

Assigns ⟨*temp*⟩ to each element on ⟨*tlist*⟩ and executes ⟨*action*⟩. As there is an assignment in this process it is not expandable.

TEXhackers note: This is the LATEX2 function `\@tfor` but with a more sane syntax. Also it works by tail recursion and so is faster as lists grow longer.

\tlist_map_break:w
\tlp_map_break:w

\tlist_map_break:w
For breaking out of a loop. You should take note of the `:w` as its usage must be precise!

\tlist_reverse:n

\tlist_reverse:n {⟨*token*$_1$⟩⟨*token*$_2$⟩...⟨*token*$_n$⟩}
Reverse the token list to result in ⟨*token*$_n$⟩...⟨*token*$_2$⟩⟨*token*$_1$⟩. Note that spaces in this token list are gobbled in the process.

5.3.1 Internal functions

\tlist_map_function_aux:Nn
\tlist_map_inline_aux:Nn
\tlist_map_variable_aux:Nnn

Internal helper functions for the ⟨*tlist*⟩ loops.

\tlist_if_blank_p_aux:w

5.4 Variables and constants

\C_job_name_tlp Constant that gets the 'job name' assigned when TEX starts.

TEXhackers note: This is the new name for the primitive `\jobname`. It is a constant that will be set by TEX and can not be overwritten by the package. Therefore the `C`

\c_empty_tlp Constant that is always empty.

TEXhackers note: This was named `\@empty` in LATEX2 and `\empty` in plain TEX.

`\c_relax_tlp` Constant holding the token that is assigned to a newly created control sequence by TEX.

`\l_tmpa_tlp`
`\l_tmpb_tlp`
`\g_tmpa_tlp`
`\g_tmpb_tlp` Scratch register for immediate use. They are not used by conditionals or predicate functions.

5.4.1 Internal functions

`\l_replace_tlp` Internal register used in the replace functions.

`\l_testa_tlp`
`\l_testb_tlp`
`\g_testa_tlp`
`\g_testb_tlp` Registers used for conditional processing if the engine doesn't support arbitrary string comparison.

`\tlp_put_left_aux:w` Used by `\tlp_put_left:Nn` and its variants.

`\tlp_to_str_aux:w` Function used to implement `\tlp_to_str:N`.

5.5 Search and replace

`\tlp_if_in:NnTF`
`\tlp_if_in:cnTF`
`\tlp_if_in:NnT`
`\tlp_if_in:cnT`
`\tlp_if_in:NnF`
`\tlp_if_in:cnF`
`\tlist_if_in:nnTF`
`\tlist_if_in:onTF` `\tlp_if_in:NnTF` ⟨*tlp*⟩ { ⟨*item*⟩ }{ ⟨*true code*⟩ }{ ⟨*false code*⟩ }
Function that tests if ⟨*item*⟩ is in ⟨*tlp*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed. Note that ⟨*item*⟩ cannot contain brace groups.

`\tlp_replace_in:Nnn`
`\tlp_replace_in:cnn`
`\tlp_greplace_in:Nnn`
`\tlp_greplace_in:cnn` `\tlp_replace_in:Nnn` ⟨*tlp*⟩ { ⟨*item1*⟩ }{ ⟨*item2*⟩ }

Replaces the leftmost occurrence of ⟨*item1*⟩ in ⟨*tlp*⟩ with ⟨*item2*⟩ if present, otherwise the ⟨*tlp*⟩ is left untouched.

```
\tlp_replace_all_in:Nnn
\tlp_replace_all_in:cnn
\tlp_greplace_all_in:Nnn
\tlp_greplace_all_in:cnn
```
`\tlp_replace_all_in:Nnn` ⟨*tlp*⟩ `{` ⟨*item1*⟩ `}{` ⟨*item2*⟩ `}`

Replaces *all* occurrences of ⟨*item1*⟩ in ⟨*tlp*⟩ with ⟨*item2*⟩.

```
\tlp_remove_in:Nn
\tlp_remove_in:cn
\tlp_gremove_in:Nn
\tlp_gremove_in:cn
```
`\tlp_remove_in:Nn` ⟨*tlp*⟩ `{` ⟨*item*⟩ `}`
Removes the leftmost occurrence of ⟨*item*⟩ from ⟨*tlp*⟩ if present.

```
\tlp_remove_all_in:Nn
\tlp_remove_all_in:cn
\tlp_gremove_all_in:Nn
\tlp_gremove_all_in:cn
```
`\tlp_remove_all_in:Nn` ⟨*tlp*⟩ `{` ⟨*item*⟩ `}`

Removes *all* occurrences of ⟨*item*⟩ from ⟨*tlp*⟩.

5.6 Heads or tails?

Here are some functions for grabbing either the head or tail of a list and perform some tests on it.

```
\tlist_head:n
\tlist_tail:n
\tlist_head_iii:n
\tlist_head_iii:f
\tlist_head:w
\tlist_tail:w
\tlist_head_iii:w
```
`\tlist_head:n` `{` ⟨*token1*⟩⟨*token2*⟩...⟨*token-n*⟩ `}`
`\tlist_tail:n` `{` ⟨*token1*⟩⟨*token2*⟩...⟨*token-n*⟩ `}`
These functions return either the head or the tail of a list, thus in the above example `\tlist_head:n` would return ⟨*token1*⟩ and `\tlist_tail:n` would return ⟨*token2*⟩...⟨*token-n*⟩. `\tlist_head_iii:n` returns the first three tokens. The `:w` versions require some care as they use a delimited argument internally.

TEXhackers note: These are the Lisp functions `car` and `cdr` but with LaTeX3 names.

\tlist_if_head_eq_meaning_p:nN
\tlist_if_head_eq_meaning:nNTF
\tlist_if_head_eq_meaning:nNTF
\tlist_if_head_eq_meaning:nNTF

\tlist_if_head_eq_meaning:nNTF { ⟨*token list*⟩ } ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

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:NN`.

\tlist_if_head_eq_charcode_p:nN
\tlist_if_head_eq_charcode:nNTF
\tlist_if_head_eq_charcode:nNTF
\tlist_if_head_eq_charcode:nNTF

\tlist_if_head_eq_charcode:nNTF { ⟨*token list*⟩ } ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

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 `\tlist_if_head_eq_charcode:fNTF` or similar).

\tlist_if_head_eq_catcode_p:nN
\tlist_if_head_eq_catcode:nNTF
\tlist_if_head_eq_catcode:nNTF
\tlist_if_head_eq_catcode:nNTF

\tlist_if_head_eq_catcode:nNTF { ⟨*token list*⟩ } ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

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.

6 LaTeX3 functions

All LaTeX3 functions contain a colon in their name. Characters following the colon are used to denote the number and the "type" of arguments that the function takes. An uppercase `N` is used to denote an argument that consists of a single token and a lowercase `n` is used when the argument can consist of several tokens surrounded by braces. In case of `n` arguments that consist of a single token the surrounding braces can be omitted in nearly all situations—functions that force the use of braces even for single token arguments are explicitly mentioned. For example, `\seq_gpush:Nn` is a function that takes two arguments, the first is a single token (the sequence) and the second may consist of several tokens surrounded by braces.

This concept of argument specification makes it easy to read the code and should be followed when defining new functions.

6.1 Expanding arguments of functions

Within code it is often necessary to expand or partially expand arguments before passing it on to some function. For example, if the token list pointer `\l_tmpa_tlp` contains the current file that should be pushed onto some stack, we can not write

```
\seq_gpush:Nn
   \g_file_name_stack
   \l_tmpa_tlp
```

since this would put the token `\l_tmpa_tlp` and not its contents on the stack. Instead a suitable number of `\exp_after:NN` would be necessary (together with extra braces) to change the order of execution, i.e.

```
\exp_after:NN
   \seq_gpush:Nn
\exp_after:NN
      \g_file_name_stack
\exp_after:NN
      {\l_tmpa_tlp}
```

The above example is probably the simplest case but is already shows how the code changes to something difficult to understand. Therefore LaTeX3 provides the programmer with a general scheme that keeps the code compact and easy to understand. To denote that some argument to a function needs special treatment one just uses different letters in the argument part of the function to mark the desired behavior. In the above example one would write

```
\seq_gpush:No
   \g_file_name_stack
   \l_tmpa_tlp
```

to achieve the desired effect. Here the `o` stands for expand this (the second) argument once before passing it to the function.

The following letters can be used to denote special treatment of arguments before passing it to the basic function:

o One time expanded token or token-list. In the latter case, effectively only the first token in the list gets expanded. Since the expansion might result in more than one token, the result is surrounded for further processing with braces.

x Fully expanded token or token-list. Like `o` but the argument is expanded using `\def:Npx` before it is passed on. This means that expansion takes place until only unexpandable tokens are left.

f Almost the same as the `x` type except here the token list is expanded fully until the first unexpandable token is found and the rest is left unchanged. Note that if this function finds a space at the beginning of the argument it will gobble it and not expand the next argument.

N,O,X Like `n`, `o`, `x` but the argument must be a single token without any braces around it.

c A character string or a token-list that ultimately expands to characters. This string (after expansion) is used to construct a command name that is eventually passed on.

C A character string or a token-list that ultimately expands to characters. From this string (after expansion) a command name is constructed and then this command name is expanded once (like `o`). The result of this is eventually passed on. In other words

```
\seq_gpush:NC
   \g_file_name_stack
   {l_tmpa_tlp}
```

Has the same effect as the example above.

Here are three new expansion types that may be useful but I'm not sure yet. Only time will tell... Proper documentation of these functions is postponed until later.

d This is pretty much like the `o` type except the token list get's expanded twice before being passed on. (`d` is for double)

E Sometimes you need to unpack a token list or something else but you don't want it to add the braces that the `o` type does. This is where you usually wind up with a lot of `\exp_after:NN`s and we would like to avoid that. This type works quite well with the other syntax but it won't work in certain circumstances: Since the generic expansion functions read their arguments when the expanded code is shuffled around, this type will have a problem if the last token you want to expand once is `\token_to_str:N` *and* you're in an argument expansion process involving arguments in braces such as the `n` and `o` type arguments. If you stick to functions invloving only `N` and `E` everything will work just fine. (`E` is for expanded, single token.)

e Same as above but the argument must be given in braces.

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

6.2 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_tlp
```

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.

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

Providing variants in this way in style files is uncritical as the `\def_new: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 `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_tlp 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_tlp` contains the text string `lur`. The straight forward approach is

```
\toks_set:No \l_tmpa_toks {\let:Nc \aaa {b \l_tmpa_tlp b}}
```

Unfortunately this only puts `\exp_args:NNc \let:NN \aaa {b \l_tmpa_tlp b}` into `\l_tmpa_toks` and not `\let: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 {\let:Nc \aaa {b \l_tmpa_tlp b}}
```

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

```
\def: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!

The available internal functions for argument expansion come in to flavours, some of them are faster then 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`, `o` or `f` in the last position) whenever possible.

6.3 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.

`\exp_args:Nc` `\exp_args:Nc` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
The first argument of ⟨*funct*⟩ (i.e., ⟨*arg1*⟩) 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 ⟨*funct*⟩ as the first argument. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged.

`\exp_args:NC` `\exp_args:Nc` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
The first argument of ⟨*funct*⟩ (i.e., ⟨*arg1*⟩) 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 which is then expanded once more. The result of this is then passed to ⟨*funct*⟩ as the first argument. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged.

`\exp_args:Nx` `\exp_args:Nx` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
The first argument of ⟨*funct*⟩ (i.e., ⟨*arg1*⟩) is fully expanded until only unexpandable

tokens remain, the result is surrounded by braces and passed to ⟨*funct*⟩. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged. As mentioned before, this type of function is relatively slow.

`\exp_args:Nf` `\exp_args:Nf` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
The first argument of ⟨*funct*⟩ (i.e., ⟨*arg1*⟩) undergoes full expansion until the first unexpandable token is encountered, the result is surrounded by braces and passed to ⟨*funct*⟩. ⟨*funct*⟩ may have more than one argument—all others are passed unchanged. Beware of its special behavior as explained above.

6.4 Manipulating two arguments

`\exp_args:NNx`
`\exp_args:Nnx`
`\exp_args:Ncx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx` `\exp_args:Nnx` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
The above functions all manipulate the first two arguments of ⟨*funct*⟩. They are all slow and non-expandable.

`\exp_args:NNo`
`\exp_args:NNf`
`\exp_args:Nno`
`\exp_args:NNc`
`\exp_args:Noo`
`\exp_args:NOo`
`\exp_args:NOc`
`\exp_args:Nco`
`\exp_args:Ncc`
`\exp_args:NNC` `\exp_args:NNo` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ...
These are the fast and expandable functions for the first two arguments.

6.5 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.

`\exp_args:Nnnx`
`\exp_args:Noox`
`\exp_args:Nnox`
`\exp_args:Ncnx` `\exp_args:Nnnx` ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ⟨*arg3*⟩ ...
All the above functions are non-expandable.

\exp_args:NnnN
\exp_args:Nnno
\exp_args:NNOo
\exp_args:NOOo
\exp_args:Nccc
\exp_args:NcNc
\exp_args:Nnnc
\exp_args:NcNo
\exp_args:Ncco
\exp_args:NNOo ⟨*funct*⟩ ⟨*arg1*⟩ ⟨*arg2*⟩ ⟨*arg3*⟩ ...
These are the fast and expandable functions for the first three arguments.

6.6 Internal functions and variables

\exp_after:NN \exp_after:NN ⟨*token1*⟩ ⟨*token2*⟩
This will expand ⟨*token2*⟩ once before processing ⟨*token1*⟩. This is similar to \exp_args:No except that no braces are put around the result of expanding ⟨*token2*⟩.

TEXhackers note: This is the primitive \expandafter which was renamed to fit into the naming conventions of LaTeX3.

\exp_not:N
\exp_not:c
\exp_not:n
\exp_not:N ⟨*token*⟩
\exp_not:n { ⟨*token list*⟩ }
This function will prohibit the expansion of ⟨*token*⟩ in situation where ⟨*token*⟩ would otherwise be replaced by it definition, e.g., inside an argument that is handled by the x convention.

TEXhackers note: \exp_not:N is the primitive \noexpand renamed and \exp_not:n is the ε-TeX primitive \unexpanded.

\exp_not:o \exp_not:o {⟨*token list*⟩}
Same as \exp_not:n except ⟨*token list*⟩ is expanded once and the result of this expansion is then prohibited from being expanded further.

\exp_not:E \exp_not:E ⟨*token*⟩
The name of this command is a lie. Perhaps it should be called "exp_perhaps_once". What it actually does is, it expands ⟨*token*⟩ and then issues an \exp_not:N to prohibit further expansion of the first token in the replacement text of ⟨*token*⟩. This means that if the replacement text of ⟨*token*⟩ consists of more than one token all further tokens are still subject to full expansion.

TEXhackers note: This command has no equivalent.

`\l_exp_tlp`

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.

7 Macro Counters

Instead of using counter registers for manipulation of integer values it is sometimes useful to keep such values in macros. For this LATEX3 offers the type "num".

One reason is the limited number of registers inside TEX. However, when using ε-TEX this is no longer an issue. It remains to be seen if there are other compelling reasons to keep this module.

It turns out there might be as with a ⟨*num*⟩ data type, the allocation module can do its bookkeeping without the aid of ⟨*int*⟩ registers.

7.1 Functions

`\num_new:N`
`\num_new:c` `\num_new:N` ⟨*num*⟩

Defines ⟨*num*⟩ to be a new variable of type `num` (initialized to zero). There is no way to define constant counters with these functions.

`\num_incr:N`
`\num_incr:c`
`\num_gincr:N`
`\num_gincr:c` `\num_incr:N` ⟨*num*⟩

Increments ⟨*num*⟩ by one. For global variables the global versions should be used.

`\num_decr:N`
`\num_decr:c`
`\num_gdecr:N`
`\num_gdecr:c` `\num_decr:N` ⟨*num*⟩

Decrements ⟨*num*⟩ by one. For global variables the global versions should be used.

`\num_zero:N`
`\num_zero:c`
`\num_gzero:N`
`\num_gzero:c` `\num_zero:N` ⟨*num*⟩

Resets ⟨*num*⟩ to zero. For global variables the global versions should be used.

\num_set:Nn
\num_set:cn
\num_gset:Nn
\num_gset:cn
\num_set:Nn ⟨num⟩ { ⟨integer⟩ }
These functions will set the ⟨num⟩ register to the ⟨integer⟩ value.

\num_gset_eq:NN
\num_gset_eq:cN
\num_gset_eq:Nc
\num_gset_eq:cc
\num_gset_eq:NN ⟨num1⟩ ⟨num2⟩
These functions will set the ⟨num1⟩ register equal to ⟨num2⟩.

\num_add:Nn
\num_gadd:Nn
\num_add:Nn ⟨num⟩ { ⟨integer⟩ }
These functions will add to the ⟨num⟩ register the value ⟨integer⟩. If the second argument is a ⟨num⟩ register too, the surrounding braces can be left out.

\num_use:N
\num_use:c
\num_use:N ⟨num⟩
This function returns the integer value kept in ⟨num⟩ in a way suitable for further processing.

TEXhackers note: Since these ⟨num⟩s are implemented as macros, the function `\num_use:N` is effectively a noop and mainly there for consistency with similar functions in other modules.

\num_eval:n
\num_eval:n { ⟨integer-expr⟩ }
Evaluates the integer expression allowing normal mathematical operators like `+-/*`.

\num_compare:nNnTF
\num_compare:cNcTF
\num_compare:nNnT
\num_compare:nNnF
\num_compare:nNnTF {⟨num expr⟩} ⟨rel⟩ {⟨num expr⟩} {⟨true⟩} {⟨false⟩}
These functions test two ⟨num⟩ expressions against each other. They are both evaluated by `\num_eval:n`.

\num_compare_p:nNn
\num_compare_p:nNn {⟨num expr⟩} ⟨rel⟩ {⟨num expr⟩}
A predicate version of the above functions.

7.2 Formatting a counter value

See the l3int module for ways of doing this.

7.3 Variable and constants

`\const_new:Nn` `\const_new:Nn  \c_⟨value⟩ { ⟨value⟩ }`
Defines a constant with ⟨*value*⟩. If the constant is negative or very large it requires an ⟨*int*⟩ register.

```
\c_minus_one
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_six
\c_seven
\c_nine
\c_ten
\c_eleven
\c_sixteen
\c_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_twenty_thousand
```
Set of constants denoting useful values.

TEXhackers note: Most of these constants have been available under LATEX2 under names like `\tw@`, `\thr@@` etc.

```
\l_tmpa_num
\l_tmpb_num
\l_tmpc_num
\g_tmpa_num
\g_tmpb_num
```
Scratch register for immediate use. They are not used by conditionals or predicate functions.

7.4 Primitive functions

`\num_eval:w` `\num_eval:w ⟨integer expression⟩ \scan_stop:`
Evaluates ⟨*integer expression*⟩. The evaluation stops when an unexpandable token of

catcode other than 12 is reached or `\scan_stop:` is read. The latter is gobbled by the scanner mechanism.

TEXhackers note: This is the ε-TEX primitive `\numexpr`.

`\if_num:w` `\if_num:w` ⟨*number1*⟩ ⟨*rel*⟩ ⟨*number2*⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`
Compare two numbers. It is recommended to use `\num_eval:n` to correctly evaluate and terminate these numbers. ⟨*rel*⟩ is one of `<`, `=` or `>` with catcode 12.

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

`\if_num_odd:w` `\if_num_odd:w` ⟨*number*⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`
Execute ⟨*true*⟩ if ⟨*number*⟩ is odd, ⟨*false*⟩ otherwise.

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

`\if_case:w` `\or:` `\if_case:w` ⟨*number*⟩ ⟨*case0*⟩ `\or:` ⟨*case1*⟩ `\or:` ... `\else:` ⟨*default*⟩ `\fi:`
Chooses case⟨*number*⟩. If you wish to use negative numbers as well, you can offset them with `\num_eval:n`.

TEXhackers note: These are the TEX primitives `\ifcase` and `\or`.

8 Sequences

LATEX3 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 ⟨*tlp*⟩ assume that the ⟨*tlp*⟩ 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.

8.1 Functions

`\seq_new:N`
`\seq_new:c` `\seq_new:N` ⟨*sequence*⟩
Defines ⟨*sequence*⟩ to be a variable of type sequences.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c` `\seq_clear:N` ⟨*sequence*⟩
These functions locally or globally clear ⟨*sequence*⟩.

`\seq_put_left:Nn`
`\seq_put_left:No`
`\seq_put_left:Nx`
`\seq_put_left:cn`
`\seq_put_right:Nn`
`\seq_put_right:No`
`\seq_put_right:Nx` `\seq_put_left:Nn` ⟨*sequence*⟩ ⟨*token list*⟩
Locally appends ⟨*token list*⟩ as a single item to the left or right of ⟨*sequence*⟩. ⟨*token list*⟩ might get expanded before appending.

`\seq_gput_left:Nn`
`\seq_gput_right:Nn`
`\seq_gput_right:Nc`
`\seq_gput_right:No`
`\seq_gput_right:cn`
`\seq_gput_right:co`
`\seq_gput_right:cc` `\seq_gput_left:Nn` ⟨*sequence*⟩ ⟨*token list*⟩
Globally appends ⟨*token list*⟩ as a single item to the left or right of ⟨*sequence*⟩.

`\seq_get:NN`
`\seq_get:cN` `\seq_get:NN` ⟨*sequence*⟩ ⟨*tlp*⟩
Functions that locally assign the left-most item of ⟨*sequence*⟩ to the token list pointer ⟨*tlp*⟩. Item is not removed from ⟨*sequence*⟩! If you need a global return value you need to code something like this:

```
\seq_get:NN ⟨sequence⟩ \l_tmpa_tlp
\tlp_gset_eq:NN ⟨global tlp⟩ \l_tmpa_tlp
```

But if this kind of construction is used often enough a separate function should be provided.

`\seq_set_eq:NN` `\seq_set_eq:NN ⟨seq1⟩ ⟨seq2⟩`
Function that locally makes ⟨*seq1*⟩ identical to ⟨*seq2*⟩.

`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc` `\seq_gset_eq:NN ⟨seq1⟩ ⟨seq2⟩`
Function that globally makes ⟨*seq1*⟩ identical to ⟨*seq2*⟩.

`\seq_gconcat:NNN`
`\seq_gconcat:ccc` `\seq_gconcat:NNN ⟨seq1⟩ ⟨seq2⟩ ⟨seq3⟩`
Function that conatenates ⟨*seq2*⟩ and ⟨*seq3*⟩ and globally assigns the result to ⟨*seq1*⟩.

`\seq_map_variable:NNn`
`\seq_map_variable:cNn` `\seq_map_variable:NNn ⟨sequence⟩ ⟨tlp⟩ { ⟨code using tlp⟩ }`

Every element in ⟨*sequence*⟩ is assigned to ⟨*tlp*⟩ and then ⟨*code using tlp*⟩ 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.

`\seq_map:NN` `\seq_map:NN ⟨sequences⟩ ⟨function⟩`
This function applies ⟨*function*⟩ (which must be a function with one argument) to every item of ⟨*sequence*⟩. ⟨*function*⟩ 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.

`\seq_map_inline:Nn`
`\seq_map_inline:cn` `\seq_map_inline:Nn ⟨sequence⟩ { ⟨inline function⟩ }`
Applies ⟨*inline function*⟩ (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 ⟨*sequence*⟩. ⟨*inline function*⟩ 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.

8.2 Predicates and conditionals

`\seq_if_empty_p:N` `\seq_if_empty_p:N` ⟨*sequence*⟩
This predicate returns 'true' if ⟨*sequence*⟩ is 'empty' i.e., doesn't contain any tokens.

`\seq_if_empty:NTF`
`\seq_if_empty:cTF`
`\seq_if_empty:NF`
`\seq_if_empty:cF`
`\seq_if_empty:NTF` ⟨*sequence*⟩ `{` ⟨*true code*⟩ `}{` ⟨*false code*⟩ `}`
Set of conditionals that test whether or not a particular ⟨*sequence*⟩ is empty and if so executes either ⟨*true code*⟩ or ⟨*false code*⟩.

`\seq_if_in:NnTF`
`\seq_if_in:cnTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`
`\seq_if_in:NnF`
`\seq_if_in:cnF`
`\seq_if_in:NnTF` ⟨*sequ*⟩ `{` ⟨*item*⟩ `}{` ⟨*true code*⟩ `}{` ⟨*false code*⟩ `}`
Function that tests if ⟨*item*⟩ is in ⟨*sequ*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed.

8.3 Internal functions

`\seq_if_empty_err:N` `\seq_if_empty_err:N` ⟨*sequence*⟩
Signals an LaTeX3 error if ⟨*sequence*⟩ is empty.

`\seq_pop_aux:nnNN` `\seq_pop_aux:nnNN` ⟨*assign1*⟩ ⟨*assign2*⟩ ⟨*sequence*⟩ ⟨*tlp*⟩
Function that assigns the left-most item of ⟨*sequence*⟩ to ⟨*tlp*⟩ using ⟨*assign1*⟩ and assigns the tail to ⟨*sequence*⟩ using ⟨*assign2*⟩. This function could be used to implement a global return function.

`\seq_get_aux:w`
`\seq_pop_aux:w`
`\seq_put_aux:Nnn`
`\seq_put_aux:w`
Functions used to implement put and get operations. They are not for meant for direct use.

`\seq_elt:w`
`\seq_elt_end:`
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.

9 Sequence Stacks

Special sequences in LaTeX3 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.)

9.1 Functions

`\seq_push:Nn`
`\seq_push:No`
`\seq_push:cn`
`\seq_gpush:Nn`
`\seq_gpush:No`
`\seq_gpush:cn` `\seq_push:Nn` ⟨*stack*⟩ `{` ⟨*token list*⟩ `}`
Locally or globally pushes ⟨*token list*⟩ as a single item onto the ⟨*stack*⟩. ⟨*token list*⟩ might get expanded before the operation.

`\seq_pop:NN`
`\seq_pop:cN`
`\seq_gpop:NN`
`\seq_gpop:cN` `\seq_pop:NN` ⟨*stack*⟩ ⟨*tlp*⟩
Functions that assign the top item of ⟨*stack*⟩ to the token list pointer ⟨*tlp*⟩ and removes it from ⟨*stack*⟩!

`\seq_top:NN`
`\seq_top:cN` `\seq_top:NN` ⟨*stack*⟩ ⟨*tlp*⟩
Functions that locally assign the top item of ⟨*stack*⟩ to the token list pointer ⟨*tlp*⟩. Item is not removed from ⟨*stack*⟩!

9.2 Predicates and conditionals

Use `seq` functions.

10 Allocating registers and the like

This module provides the basic mechanism for allocating TeX's registers. While designing this we have to take into account the following characteristics:

- `\box255` is reserved for use in the output routine, so it should not be allocated otherwise.

- TEX can load up 256 hyphenation patterns (registers `\tex_language:D` 0-255),
- TEX can load no more than 16 math families,
- TEX supports no more than 16 io-streams for reading (`\tex_read:D`) and 16 io-streams for writing (`\tex_write:D`),
- TEX supports no more than 256 inserts, Omega supports more.
- The other registers (`\tex_count:D`, `\tex_dimen:D`, `\tex_skip:D`, `\tex_muskip:D`, `\tex_box:D`, and `\tex_toks:D` range from 0 to 32768, but registers numbered above 255 are accessed somewhat less efficient.
- Registers could be allocated both globally and locally; the use of regsiters could also be globaly or locally. Here we provide support for globally allocated registers for both gloabl and local use and for locally allocated registers for local use only.

We also need to allow for some bookkeeping: we need to know which register was allocated last and which registers can not be allocated by the standard mechanisms.

10.1 Functions

`\alloc_setup_type:nnn` `\alloc_setup_type:nnn` ⟨*type*⟩ ⟨*g_start_num*⟩ ⟨*l_start_num*⟩

Sets up the storage needed for the administration of registers of type ⟨*type*⟩.
⟨*type*⟩ should be a token list in braces, it can be one of `int`, `dimen`, `skip`, `muskip`, `box`, `toks`, `ior`, `iow`, `pattern`, or `ins`.
⟨*g˙start˙num*⟩ is the number of the first not allocated global register, it will be incremented by 1 when the allocation is done. ⟨*l˙start˙num*⟩ is the number of the first not allocated local register, it will be decremented by 1 when the allocation is done.

`\alloc_reg:NnNN` `\alloc_reg:NnNN` ⟨*g-l*⟩ ⟨*type*⟩ ⟨*alloc_cmd*⟩ ⟨*cs*⟩
Performs the allocation of a register of type ⟨*type*⟩ to copntrol sequence ⟨*cs*⟩, using the command ⟨*alloc˙cmd*⟩. The `g` or `l` indicaties whether the allocation should be global or local. This macro is the basic building block for the definition of the `\`⟨*type*⟩`_new:N` commands

11 Low-level file i/o

TEX is capable of reading from and writing up to 16 individual streams. These i/o operations are accessable in LATEX3 with functions from the `\io..` modules. In most cases it will be sufficiant for the programmer to use the functions provided by the auxiliary file module, but here are the necessary functions for manipulating private streams.

Sometimes it is not know beforehand how much text is going to be written with a single call. As a result some internal TEX buffer may overflow. To avoid this kind of problem, LATEX3 maintains beside direct write operations like `\iow_expanded:Nn` also so called "long" writes where the output is broken into individual lines on every blank in the text to be written. The resulting files are difficult to read for humans but since they usually serve only as internal storage this poses no problem.

Beside the functions that immediately act (e.g., `\iow_expanded:Nn`, etc.) we also have deferred operations that are saved away until the next page is finished. This allow to expand the ¡tokens¿ at the right time to get correct page numbers etc.

11.1 Functions for output streams

`\iow_new:N`
`\iow_new:c` `\iow_new:N` ⟨*stream*⟩
Defines ⟨*stream*⟩ to be a new identifer denoting an output stream for use in subsequent functions.

TEXhackers note: `\iow_new:N` corresponds to the plain TEX `\newwrite` allocation routine.

`\iow_open:Nn`
`\iow_open:cn` `\iow_open:Nn` ⟨*stream*⟩ { ⟨*file name*⟩ }
Opens output stream ⟨*stream*⟩ to write to ⟨*file name*⟩. The output stream is immediately available for use. If the ⟨*stream*⟩ was already used as an output stream to some other file, this file gets closed first.[1] Also, all output streams still open at the end of the TEX run will be automatically closed.

`\iow_expanded:Nn`
`\iow_unexpanded:Nn` `\iow_expanded:Nn` ⟨*stream*⟩ { ⟨*tokens*⟩ }
This function immediately writes the expansion of ⟨*tokens*⟩ to the output stream ⟨*stream*⟩. If ⟨*stream*⟩ is not open output goes to the terminal. The variant `\iow_unexpanded:Nn` writes out ⟨*tokens*⟩ without any further expansion (verbatim).

`\iow_expanded_log:n`
`\iow_expanded_term:n`
`\iow_unexpanded_term:n` `\iow_expanded_log:n` { ⟨*tokens*⟩ }

[1] This is a precaution since on some OS it is possible to open the same file for output more than once which then results in some internal errors at the end of the run.

These functions write to the transcript or to the terminal respectively. So they are equivalent to `\iow_expanded:Nn` where ⟨*stream*⟩ is the transcript file (`\c_iow_log_stream`) or the terminal (`\c_io_term_stream`).

`\iow_long_expanded:Nx`
`\iow_long_unexpanded:Nn` `\iow_long_expanded:Nn` ⟨*stream*⟩ `{` ⟨*tokens*⟩ `}`

Like `\iow_expanded:Nn` but splits ⟨*tokens*⟩ at every blank into separate lines.

`\iow_unexpanded_if_avail:Nn`
`\iow_unexpanded_if_avail:cn` `\iow_unexpanded_if_avail:Nn` ⟨*stream*⟩ `{` ⟨*tokens*⟩ `}`

This special function first checks if the ⟨*stream*⟩ is open of writing. If not it does nothing otherwise it behaves like `\iow_unexpanded:Nn`.

`\iow_deferred_expanded:Nn`
`\iow_deferred_unexpanded:Nn` `\iow_deferred_expanded:Nn` ⟨*stream*⟩ `{` ⟨*tokens*⟩ `}`

These functions save away ⟨*tokens*⟩ until the next page is ready to be shipped out. Then, in case of `\iow_deferred_expanded:Nn` ⟨*tokens*⟩ get expanded and afterwards written to ⟨*stream*⟩. `\iow_deferred_expanded:Nn` also always needs `{}` around the second argument. The use of `\iow_deferred_unexpanded:Nn` is probably seldom necessary.

TEXhackers note: `\iow_deferred_expanded:Nn` was known as `\write`.

`\iow_newline:` `\iow_newline:`
Function that produces a new line when used within the ⟨*token list*⟩ that gets written some output stream in non-verbatim mode.

11.2 Functions for input streams

`\ior_new:N` `\ior_new:N` ⟨*stream*⟩
This function defines ⟨*stream*⟩ to be a new input stream constant.

TEXhackers note: This is the new name and new implementation for plain TEX's `\newread`.

`\ior_open:Nn` `\ior_open:Nn` ⟨*stream*⟩ `{` ⟨*file name*⟩ `}`
This function opens ⟨*stream*⟩ as an input stream for the external file ⟨*file name*⟩. If

⟨*file name*⟩ doesn't exist or is an empty file the stream is considered to be fully read, a condition which can be tested with `\ior_eof:NTF` etc. If ⟨*stream*⟩ was already used to read from some other file this file will be closed first. The input stream is ready for immediate use.

`\ior_close:N` `\ior_close:N` ⟨*stream*⟩
This function closes the read stream ⟨*stream*⟩.

TEXhackers note: This is a new name for `\closein` but it is considered bad practice to make use of this knowledge :-)

`\ior_eof:NTF`
`\ior_eof:NF` `\ior_eof:NTF` ⟨*stream*⟩ `{` ⟨*true code*⟩ `}{` ⟨*false code*⟩ `}`
Conditional that tests if some input stream is fully read. The condition is also true if the input stream is not open.

`\if_eof:w` `\if_eof:w` ⟨*stream*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

TEXhackers note: This is the primitive `\ifeof` but we allow only a ⟨*stream*⟩ and not a plain number after it.

`\ior_to:NN`
`\ior_gto:NN` `\ior_to:NN` ⟨*stream*⟩ ⟨*tlp*⟩
Functions that reads one or more lines (until an equal number of left and right braces are found) from the input stream ⟨*stream*⟩ and places the result locally or globally into ⟨*tlp*⟩. If ⟨*stream*⟩ is not open input is requested from the terminal.

11.3 Constants

`\c_iow_comment_char`
`\c_iow_lbrace_char`
`\c_iow_rbrace_char` Constants that can be used to represent comment character, left and right brace in token lists that should be written to a file.

`\c_io_term_stream` Input or output stream denoting the terminal. If used as an input stream the user is prompted with the name of the ⟨*tlp*⟩ (that is used in the call `\ior_to:NN` or `\ior_gto:NN`) followed by an equal sign. If you don't want an automatic prompt of this sort "misuse" `\c_iow_log_stream` as an input stream.

`\c_iow_log_stream` Output stream that writes only to the transcript file (e.g., the `.log` file on most systems). You may "misuse" this stream as an input stream. In this case it acts as a terminal stream without user prompting.

11.4 Internal functions

`\iow_long_expanded_aux:w` Function used to implement immediate writing where a new line is started at every blank.

`\tex_read:D`
`\tex_immediate:D`
`\tex_closeout:D`
`\tex_openin:D`
`\tex_openout:D`
These are the functions of the primitive interface to TEX.

TEXhackers note: The TEX primitives `\read`, `\immediate`, `\closeout`, `\openin`, and `\openout` are all renamed and should not be used by a programmer since the functionality is covered by the LATEX3 functions above.

12 Comma lists

LATEX3 implements a data type called 'clist (comma-lists)'. 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 comma-list might contain any tokens except for commas that are used to structure comma-lists (braces are need if commas are part of the value). It is also possible to map functions on such comma-lists so that they are executed for every item of the comma-list.

All functions that return items from a comma-list in some ⟨*tlp*⟩ assume that the ⟨*tlp*⟩ 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 comma-list can currently be done with either something like `\clist_put_left:Nn` or with a "stack" function like `\clist_push:Nn` which has the same effect. Maybe one should therefore remove the "left" functions totally.

12.1 Functions

\clist_new:N
\clist_new:c
\clist_new:N ⟨*comma-list*⟩
Defines ⟨*comma-list*⟩ to be a variable of type clist.

\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c
\clist_clear:N ⟨*comma-list*⟩
These functions locally or globally clear ⟨*comma-list*⟩.

\clist_put_left:Nn
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_right:Nn
\clist_put_right:No
\clist_put_right:Nx
\clist_put_left:Nn ⟨*comma-list*⟩ ⟨*token list*⟩
Locally appends ⟨*token list*⟩ as a single item to the left or right of ⟨*comma-list*⟩. ⟨*token list*⟩ might get expanded before appending.

\clist_gput_left:Nn
\clist_gput_right:Nn
\clist_gput_right:No
\clist_gput_right:cn
\clist_gput_right:co
\clist_gput_right:cc
\clist_gput_left:Nn ⟨*comma-list*⟩ ⟨*token list*⟩

Globally appends ⟨*token list*⟩ as a single item to the left or right of ⟨*comma-list*⟩.

\clist_get:NN
\clist_get:cN
\clist_get:NN ⟨*comma-list*⟩ ⟨*tlp*⟩
Functions that locally assign the left-most item of ⟨*comma-list*⟩ to the token list pointer ⟨*tlp*⟩. 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_tlp
\tlp_gset_eq:NN ⟨global tlp⟩ \l_tmpa_tlp
```

But if this kind of construction is used often enough a separate function should be provided.

\clist_set_eq:NN `\clist_set_eq:NN ⟨clist1⟩ ⟨clist2⟩`
Function that locally makes ⟨*clist1*⟩ identical to ⟨*clist2*⟩.

\clist_gset_eq:NN
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc `\clist_gset_eq:NN ⟨clist1⟩ ⟨clist2⟩`
Function that globally makes ⟨*clist1*⟩ identical to ⟨*clist2*⟩.

\clist_concat:NNN
\clist_gconcat:NNN
\clist_gconcat:NNc
\clist_gconcat:ccc `\clist_gconcat:NNN ⟨clist1⟩ ⟨clist2⟩ ⟨clist3⟩`
Function that concatenates ⟨*clist2*⟩ and ⟨*clist3*⟩ and globally assigns the result to ⟨*clist1*⟩.

\clist_remove_duplicates:N
\clist_gremove_duplicates:N `\clist_gremove_duplicates:N ⟨clist⟩`

Function that removes any duplicate entries in ⟨*clist*⟩.

\clist_use:N
\clist_use:c `\clist_use:N ⟨clist⟩`
Function that inserts the ⟨*clist*⟩ into the processing stream. Mainly useful if one knows what the ⟨*clist*⟩ contains, e.g., for displaying the content of template parameters.

12.2 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.

\clist_map_function:NN
\clist_map_function:cN
\clist_map_function:nN `\clist_map_function:NN ⟨comma-list⟩ ⟨function⟩`

This function applies ⟨*function*⟩ (which must be a function with one argument) to every item of ⟨*comma-list*⟩. ⟨*function*⟩ is not executed within a sub-group so that side effects can be achieved locally. The operation is expandable which means that it can be used within write operations etc.

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn
\clist_map_inline:Nn ⟨*comma-list*⟩ { ⟨*inline function*⟩ }

Applies ⟨*inline function*⟩ (which should be the direct coding for a function with one argument (i.e. use ##1 as the placeholder for this argument)) to every item of ⟨*comma-list*⟩. ⟨*inline function*⟩ 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. These functions can be nested.

\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn
\clist_map_variable:NNn ⟨*comma-list*⟩ ⟨*temp-var*⟩ { ⟨*action*⟩ }

Assigns ⟨*temp-var*⟩ to each element in ⟨*clist*⟩ and then executes ⟨*action*⟩ which should contain ⟨*temp-var*⟩. As the operation performs an assignment, it is not expandable.

TEXhackers note: These functions resemble the LaTeX 2_ε function \@for but does not borrow the somewhat strange syntax.

\clist_map_break:w
\clist_map_break:w
For breaking out of a loop. To be used inside TF type functions as in the example below.

```
\def_new:Npn \test_function:n #1 {
  \int_compare:nNnTF {#1}> 3 {\clist_map_break:w}{``#1''}
}
\clist_map_function:nN {1,2,3,4,5,6,7,8}\test_function:n
```

This would return ``1''``2''``3''.

12.3 Predicates and conditionals

\clist_if_empty_p:N
\clist_if_empty_p:N ⟨*comma-list*⟩
This predicate returns 'true' if ⟨*comma-list*⟩ is 'empty' i.e., doesn't contain any tokens.

\clist_if_empty:NTF
\clist_if_empty:cTF
\clist_if_empty:NF
\clist_if_empty:cF
\clist_if_empty:NTF ⟨*comma-list*⟩ { ⟨*true code*⟩ }{ ⟨*false code*⟩ }
Set of conditionals that test whether or not a particular ⟨*comma-list*⟩ is empty and if so executes either ⟨*true code*⟩ or ⟨*false code*⟩.

\clist_if_eq:NNTF ⟨*comma-list1*⟩ ⟨*comma-list2*⟩ { ⟨*true code*⟩ }{ ⟨*false code*⟩ }

Check if ⟨*comma-list1*⟩ and ⟨*comma-list2*⟩ are equal and execute either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

\clist_if_in:NnTF
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:coTF

\clist_if_in:NnTF ⟨*comma-list*⟩ { ⟨*item*⟩ }{ ⟨*true code*⟩ }{ ⟨*false code*⟩ }

Function that tests if ⟨*item*⟩ is in ⟨*comma-list*⟩. Depending on the result either ⟨*true code*⟩ or ⟨*false code*⟩ is executed.

12.4 Internal functions

\clist_if_empty_err:N \clist_if_empty_err:N ⟨*comma-list*⟩

Signals an LaTeX3 error if ⟨*comma-list*⟩ is empty.

\clist_pop_aux:nnNN \clist_pop_aux:nnNN ⟨*assign1*⟩ ⟨*assign2*⟩ ⟨*comma-list*⟩ ⟨*tlp*⟩

Function that assigns the left-most item of ⟨*comma-list*⟩ to ⟨*tlp*⟩ using ⟨*assign1*⟩ and assigns the tail to ⟨*comma-list*⟩ using ⟨*assign2*⟩. This function could be used to implement a global return function.

\clist_get_aux:w
\clist_pop_aux:w
\clist_pop_auxi:w
\clist_put_aux:NNnnNn

Functions used to implement put and get operations. They are not for meant for direct use.

\clist_map_function_aux:Nw
\clist_map_inline_aux:Nw
\clist_map_variable_aux:Nnw

Internal helper functions for the ⟨*clist*⟩ mapping functions.

\clist_concat_aux:NNNN
\clist_remove_duplicates_aux:NN
\clist_remove_duplicates_aux:n
\l_clist_remove_duplicates_clist

Functions that help concatenate ⟨*clist*⟩s and remove duplicate elements from a ⟨*clist*⟩.

12.5 Comma list Stacks

Special comma-lists in LaTeX3 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 `\clist_new:N` etc.)

`\clist_push:Nn`
`\clist_push:No`
`\clist_push:cn`
`\clist_gpush:Nn`
`\clist_gpush:No`
`\clist_gpush:cn`
`\clist_push:Nn` ⟨*stack*⟩ { ⟨*token list*⟩ }
Locally or globally pushes ⟨*token list*⟩ as a single item onto the ⟨*stack*⟩. ⟨*token list*⟩ might get expanded before the operation.

`\clist_pop:NN`
`\clist_pop:cN`
`\clist_gpop:NN`
`\clist_gpop:cN`
`\clist_pop:NN` ⟨*stack*⟩ ⟨*tlp*⟩
Functions that assign the top item of ⟨*stack*⟩ to the token list pointer ⟨*tlp*⟩ and removes it from ⟨*stack*⟩!

`\clist_top:NN`
`\clist_top:cN`
`\clist_top:NN` ⟨*stack*⟩ ⟨*tlp*⟩
Functions that locally assign the top item of ⟨*stack*⟩ to the token list pointer ⟨*tlp*⟩. Item is not removed from ⟨*stack*⟩!

13 Property lists

LaTeX3 implements a data structure which allows to store information associated with individual tokens.

13.1 Functions

`\prop_new:N`
`\prop_new:c`
`\prop_new:N` ⟨*plist*⟩
Defines ⟨*plist*⟩ to be a variable of type p-list.

`\prop_clear:N`
`\prop_gclear:N`
`\prop_clear:N` ⟨*plist*⟩
These functions locally or globally clear ⟨*plist*⟩.

\prop_put:NNn
\prop_put:ccn
\prop_gput:NNn
\prop_gput:NNo
\prop_gput:NOo
\prop_gput:Ncn
\prop_gput:OOo
\prop_gput:NOx
\prop_gput:cNn
\prop_gput:ccn
\prop_gput:cco
\prop_gput:ccx `\prop_put:NNn ⟨plist⟩ ⟨key⟩ {⟨token list⟩}`

Locally or globally associates ⟨*token list*⟩ with ⟨*key*⟩ in the p-list ⟨*plist*⟩. If ⟨*key*⟩ has already a meaning within ⟨*plist*⟩ this value is overwritten.

\prop_gput_if_new:NNn `\prop_gput_if_new:NNn ⟨plist⟩ ⟨key⟩ {⟨token list⟩}`

Globally associates ⟨*token list*⟩ with ⟨*key*⟩ in the p-list ⟨*plist*⟩ but only if ⟨*key*⟩ has so far no meaning within ⟨*plist*⟩. overwritten.

\prop_get:NNN
\prop_get:cNN
\prop_gget:NNN
\prop_gget:NcN
\prop_gget:cNN `\prop_get:NNN ⟨plist⟩ ⟨key⟩ ⟨tlp⟩`

If ⟨*info*⟩ is the information associated with ⟨*key*⟩ in the p-list ⟨*plist*⟩ then the token list pointer ⟨*tlp*⟩ gets ⟨*info*⟩ assigned. Otherwise its value is the special quark `\q_no_value`. The assignment is done either locally or globally.

\prop_set_eq:NN
\prop_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:cc `\prop_set_eq:NN ⟨plist 1⟩ ⟨plist 2⟩`

A fast assignment of ⟨*plist*⟩s.

\prop_get_gdel:NNN `\prop_get_gdel:NNN ⟨plist⟩ ⟨key⟩ ⟨tlp⟩`

Like `\prop_get:NNN` but additionally removes ⟨*key*⟩ (and its ⟨*info*⟩) from ⟨*plist*⟩.

\prop_del:NN
\prop_gdel:NN `\prop_del:NN ⟨plist⟩ ⟨key⟩`

Locally or globally deletes ⟨*key*⟩ and its ⟨*info*⟩ from ⟨*plist*⟩ if found. Otherwise does nothing.

\prop_map_function:NN
\prop_map_function:cN
\prop_map_function:cc
\prop_map_function:NN ⟨*plist*⟩ ⟨*function*⟩

Maps ⟨*function*⟩ which should be a function with two arguments (⟨*key*⟩ and ⟨*info*⟩) over every ⟨*key*⟩ ⟨*info*⟩ pair of ⟨*plist*⟩. Expandable.

\prop_map_inline:Nn
\prop_map_inline:cn
\prop_map_inline:Nn ⟨*plist*⟩ { ⟨*inline function*⟩ }
Just like \prop_map_function:NN but with the function of two arguments supplied as inline code. Within ⟨*inline function*⟩ refer to the arguments via ##1 (⟨*key*⟩) and ##2 (⟨*info*⟩). Fast but not expandable.

\prop_map_break:w
\prop_map_break:w
For breaking out of a loop. To be used inside TF type functions.

13.2 Predicates and conditionals

\prop_if_empty:NTF
\prop_if_empty:NTF ⟨*plist*⟩ {⟨*true code*⟩}{⟨*false code*⟩}
Set of conditionals that test whether or not a particular ⟨*plist*⟩ is empty.

\prop_if_eq:NNF
\prop_if_eq:ccF
\prop_if_eq:NNF ⟨*plist1*⟩ ⟨*plist2*⟩ {⟨*false code*⟩}
Execute ⟨*false code*⟩ if ⟨*plist1*⟩ doesn't hold the same token list as ⟨*plist2*⟩.

\prop_if_in:NNTF
\prop_if_in:NOTF
\prop_if_in:ccTF
\prop_if_in:NNTF ⟨*plist*⟩⟨*key*⟩{⟨*true code*⟩}{⟨*false code*⟩}
Tests if ⟨*key*⟩ is used in ⟨*plist*⟩ and then either executes ⟨*true code*⟩ or ⟨*false code*⟩.

13.3 Internal functions

\prop_put_aux:w
\prop_put_if_new_aux:w
Internal functions implementing the put operations.

\prop_get_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:NNn`
`\prop_map_inline_aux:Nn` Internal functions implementing the map operations.

`\l_prop_inline_level_num` Fake integer used in internal name for function used inside `\prop_map_inline:NN`.

`\prop_split_aux:NNn` `\prop_split_aux:NNn` ⟨*plist*⟩ ⟨*key*⟩ ⟨*cmd*⟩
Internal function that invokes ⟨*cmd*⟩ with 3 arguments: 1st is the beginning of ⟨*plist*⟩ before ⟨*key*⟩, 2nd is the value associated with ⟨*key*⟩, 3rd is the rest of ⟨*plist*⟩ after ⟨*key*⟩. If there is no key ⟨*key*⟩ in ⟨*plist*⟩, then the 2 arg is `\q_no_value` and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens ⟨*prop*⟩ `\q_no_value` at the end.

This function is used to implement various get operations.

14 Integers

LaTeX3 maintains two type of integer registers for internal use. One (associated with the name `num`) for low level uses in the allocation mechanism using macros only and `int`: the one described here.

The `int` type uses the built-in counter registers of TeX and is therefore relatively fast compared to the `num` type and should be preferred in all cases as there is little chance we should ever run out of registers when being based on at least ε-TeX.

14.1 Functions

`\int_new:N`
`\int_new:c`
`\int_new_l:N` `\int_new:N` ⟨*int*⟩
Globally defines ⟨*int*⟩ to be a new variable of type `int` although you can still choose if it should be a an `\l_` or `\g_` type. There is no way to define constant counters with these functions. The function `\int_new_l:N` defines ⟨*int*⟩ locally only.

TeXhackers note: `\int_new:N` is the equivalent to plain TeX's `\newcount`. However, the internal register allocation is done differently.

`\int_incr:N`
`\int_gincr:N`
`\int_gincr:c` `\int_incr:N` ⟨*int*⟩
Increments ⟨*int*⟩ by one. For global variables the global versions should be used.

\int_decr:N
\int_gdecr:N
\int_gdecr:c

`\int_decr:N` ⟨*int*⟩

Decrements ⟨*int*⟩ by one. For global variables the global versions should be used.

\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn

`\int_set:Nn` ⟨*int*⟩ `{` ⟨*integer expr*⟩ `}`

These functions will set the ⟨*int*⟩ register to the ⟨*integer expr*⟩ value. This value can contain simple calc-like expressions as provided by ε-TEX.

\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c

`\int_zero:N` ⟨*int*⟩

These functions sets the ⟨*int*⟩ register to zero either locally or globally.

\int_add:Nn
\int_add:cn
\int_gadd:Nn

`\int_add:Nn` ⟨*int*⟩ `{` ⟨*integer expr*⟩ `}`

These functions will add to the ⟨*int*⟩ register the value ⟨*integer expr*⟩. If the second argument is a ⟨*int*⟩ register too, the surrounding braces can be left out.

\int_sub:Nn
\int_gsub:Nn

`\int_gsub:Nn` ⟨*int*⟩ `{` ⟨*integer expr*⟩ `}`

These functions will subtract from the ⟨*int*⟩ register the value ⟨*integer expr*⟩. If the second argument is a ⟨*int*⟩ register too, the surrounding braces can be left out.

\int_use:N
\int_use:c

`\int_use:N` ⟨*int*⟩

This function returns the integer value kept in ⟨*int*⟩ in a way suitable for further processing.

TEXhackers note: The function `\int_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-explaining.

14.2 Formatting a counter value

`\int_to_arabic:n`
`\int_to_alph:n`
`\int_to_Alph:n`
`\int_to_roman:n`
`\int_to_Roman:n`
`\int_to_symbol:n`

`\int_to_alph:n { ⟨integer⟩ }`
`\int_to_alph:n ⟨int⟩`

If some ⟨*integer*⟩ or the the current value of a ⟨*int*⟩ should be displayed or typeset in a special ways (e.g., as uppercase roman numerals) these function can be used. We need braces if the argument is a simple ⟨*integer*⟩, they can be omitted in case of a ⟨*int*⟩. By default the letters produced by `\int_to_roman:n` and `\int_to_Roman:n` have catcode 11.

All functions are fully expandable and will therefore produce the correct output when used inside of deferred writes, etc. In case the number in an `alph` or `Alph` function is greater than the default base number (26) it follows a simple conversion rule so that 27 is turned into `aa`, 50 into `ax` and so on and so forth. These two functions can be modified quite easily to take a different base number and conversion rule so that other languages can be supported.

TEXhackers note: These are more or less the internal LATEX2 functions `\@arabic`, `\@alph`, `\Alph`, `\@roman`, `\@Roman`, and `\@fnsymbol` except that `\int_to_symbol:n` is also allowed outside math mode.

14.2.1 Internal functions

`\int_to_roman:w`

`\int_to_roman:w ⟨integer⟩ ⟨space⟩ or ⟨non-expandable token⟩`

Converts ⟨*integer*⟩ to it lowercase roman representation. Note that it produces a string of letters with catcode 12.

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

`\int_roman_lcuc_mapping:Nnn`
`\int_to_roman_lcuc:NN`

`\int_roman_lcuc_mapping:Nnn ⟨roman_char⟩ {⟨licr⟩} {⟨LICR⟩}`
`\int_to_roman_lcuc:NN ⟨roman_char⟩ ⟨char⟩`

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

\int_convert_number_with_rule:nnN
\int_alph_default_conversion_rule:n
\int_Alph_default_conversion_rule:n
\int_symbol_math_conversion_rule:n
\int_symbol_text_conversion_rule:n

\int_convert_number_with_rule:nnN {⟨*int1*⟩} {⟨*int2*⟩} ⟨*function*⟩
\int_alph_default_conversion_rule:n {⟨*int*⟩}

\int_convert_number_with_rule:nnN converts ⟨*int1*⟩ into letters, symbols, whatever as defined by ⟨*function*⟩. ⟨*int2*⟩ denotes the base number for the conversion.

14.3 Variable and constants

\c_max_int Constant that denote the maximum value which can be stored in an ⟨*int*⟩ register.

\l_tmpa_int
\l_tmpb_int
\l_tmpc_int
\g_tmpa_int
\g_tmpb_int

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

14.4 Testing and evaluating integer expressions

\int_eval:n
\int_div_truncate:nn
\int_div_round:nn
\int_mod:nn

\int_eval:n {⟨*int expr*⟩}
\int_div_truncate:n {⟨*int expr*⟩} {⟨*int expr*⟩}
\int_mod:nn {⟨*int expr*⟩} {⟨*int expr*⟩}

Evaluates the value of a integer expression so that \int_eval:n {3*5/4} puts 4 back into the input stream. Note that the results of divisions are rounded by the primitive operations. If you want the result of a division to be truncated use \int_div_truncate:nn. \int_div_round:nn is added for completeness. \int_mod:nn returns the remainder of a division. All of these functions are expandable.

TEXhackers note: \int_eval:n is the ε-TEXprimitive \numexpr turned into a function taking an argument.

\int_compare:nNnTF
\int_compare:nNnT
\int_compare:nNnF

\int_compare:nNnTF {⟨*int expr*⟩} ⟨*rel*⟩ {⟨*int expr*⟩} {⟨*true*⟩} {⟨*false*⟩}

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

```
\int_compare:nNnTF \l_temp_int < \c_zero {negative}{non-negative}
```

you can safely omit the braces.

TEXhackers note: This is the TEX primitive `\ifnum` turned into a function.

`\int_compare_p:nNn` `\int_compare_p:nNn {⟨int expr⟩} ⟨rel⟩ {⟨int expr⟩}`
A predicate version of the above mentioned functions.

`\int_if_odd:nTF`
`\int_if_odd_p:n` `\int_if_odd:nTF {⟨int expr⟩} {⟨true⟩} {⟨false⟩}`
These functions test if an integer expression is even or odd. We also define a predicate version of it.

TEXhackers note: This is the TEX primitive `\ifodd` turned into a function.

`\int_whiledo:nNnT`
`\int_whiledo:nNnF`
`\int_dowhile:nNnT`
`\int_dowhile:nNnF` `\int_whiledo:nNnT ⟨int expr⟩ ⟨rel⟩ ⟨int expr⟩ {⟨true⟩}`
`\int_whiledo:nNnT` tests the integer expressions and if true performs the body `T` until the test fails. `\int_dowhile:nNnT` is similar but executes the body first and then performs the check, thus ensuring that the body is executed at least once. The `F` versions are similar but continue the loop as long as the test is false. They could be omitted as it is just a matter of switching the arguments in the test.

14.5 Conversion

`\int_convert_from_base_ten:nn` `\int_convert_from_base_ten:nn {⟨number⟩}{⟨base⟩}`

Converts the base 10 number ⟨*number*⟩ into its equivalent representation written in base ⟨*base*⟩. Expandable.

`\int_convert_to_base_ten:nn` `\int_convert_to_base_ten:nn {⟨number⟩}{⟨base⟩}`

Converts the base ⟨*base*⟩ number ⟨*number*⟩ into its equivalent representation written in base 10. ⟨*number*⟩ can consist of digits and ascii letters. Expandable.

15 Length registers

LaTeX3 knows about two types of length registers for internal use: rubber lengths (`skip`s) and rigid lengths (`dim`s).

15.1 Skip registers

15.1.1 Functions

`\skip_new:N`
`\skip_new:c`
`\skip_new_l:N` `\skip_new:N` ⟨*skip*⟩
Defines ⟨*skip*⟩ to be a new variable of type `skip`.

TEXhackers note: `\skip_new:N` is the equivalent to plain TEX's `\newskip`. However, the internal register allocation is done differently.

`\skip_zero:N`
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c` `\skip_zero:N` ⟨*skip*⟩
Locally or globally reset ⟨*skip*⟩ to zero. For global variables the global versions should be used.

`\skip_set:Nn`
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn` `\skip_set:Nn` ⟨*skip*⟩ `{` ⟨*skip value*⟩ `}`
These functions will set the ⟨*skip*⟩ register to the ⟨*length*⟩ value.

`\skip_add:Nn`
`\skip_add:cn`
`\skip_gadd:Nn` `\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.

`\skip_sub:Nn`
`\skip_gsub:Nn` `\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.

`\skip_use:N`
`\skip_use:c` `\skip_use:N` ⟨*skip*⟩
This function returns the length value kept in ⟨*skip*⟩ in a way suitable for further processing.

TEXhackers note: The function `\skip_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.

`\skip_horizontal:N`
`\skip_horizontal:c`
`\skip_horizontal:n`
`\skip_vertical:N`
`\skip_vertical:c`
`\skip_vertical:n`
`\skip_horizontal:N` ⟨*skip*⟩
`\skip_horizontal:n { ⟨length⟩ }`
The `hor` functions insert ⟨*skip*⟩ or ⟨*length*⟩ with the TEX primitive `\hskip`. The `vertical` variants do the same with `\vskip`. The `n` versions evaluate ⟨*length*⟩ with `\skip_eval:n`.

`\skip_infinite_glue:nTF` `\skip_infinite_glue:nTF {⟨skip⟩} {⟨true⟩} {⟨false⟩}`

Checks if ⟨*skip*⟩ contains infinite stretch or shrink components and executes either ⟨*true*⟩ or ⟨*false*⟩. Also works on input like `3pt plus .5in`.

`\skip_split_finite_else_action:nnNN` `\skip_split_finite_else_action:nnNN {⟨skip⟩} {⟨actio`
⟨*dimen1*⟩ ⟨*dimen2*⟩

Checks if ⟨*skip*⟩ contains finite glue. If it does then it assigns ⟨*dimen1*⟩ the stretch component and ⟨*dimen2*⟩ the shrink component. If it contains infinite glue set ⟨*dimen1*⟩ and ⟨*dimen2*⟩ to zero and execute `#2` which is usually an error or warning message of some sort.

`\skip_eval:n` `\skip_eval:n {⟨skip expr⟩}`
Evaluates the value of ⟨*skip expr*⟩ so that `\skip_eval:n {5pt plus 3fil + 3pt minus 1fil}` puts `8.0pt plus 3.0fil minus 1.0fil` back into the input stream. Expandable.

TEXhackers note: This is the ε-TEX primitive `\glueexpr` turned into a function taking an argument.

15.1.2 Formatting a skip register value

15.1.3 Variable and constants

`\c_max_skip` Constant that denotes the maximum value which can be stored in a ⟨*skip*⟩ register.

`\c_zero_skip` Set of constants denoting useful values.

`\l_tmpa_skip`
`\l_tmpb_skip`
`\l_tmpc_skip`
`\g_tmpa_skip`
`\g_tmpb_skip` Scratch register for immediate use.

15.2 Dim registers

15.2.1 Functions

`\dim_new:N`
`\dim_new:c`
`\dim_new_l:N` `\dim_new:N` ⟨*dim*⟩
Defines ⟨*dim*⟩ to be a new variable of type `dim`.

TEXhackers note: `\dim_new:N` is the equivalent to plain TEX's `\newdimen`. However, the internal register allocation is done differently.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c` `\dim_zero:N` ⟨*dim*⟩
Locally or globally reset ⟨*dim*⟩ to zero. For global variables the global versions should be used.

`\dim_set:Nn`
`\dim_set:cn`
`\dim_gset:Nn`
`\dim_gset:cn` `\dim_set:Nn` ⟨*dim*⟩ `{` ⟨*dim value*⟩ `}`
These functions will set the ⟨*dim*⟩ register to the ⟨*dim value*⟩ value.

`\dim_add:Nn`
`\dim_add:cn`
`\dim_gadd:Nn`
`\dim_add:Nn` ⟨*dim*⟩ { ⟨*length*⟩ }
These functions will add to the ⟨*dim*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*dim*⟩ register too, the surrounding braces can be left out.

`\dim_sub:Nn`
`\dim_gsub:Nn`
`\dim_gsub:Nn` ⟨*dim*⟩ { ⟨*length*⟩ }
These functions will subtract from the ⟨*dim*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*dim*⟩ register too, the surrounding braces can be left out.

`\dim_use:N`
`\dim_use:c`
`\dim_use:N` ⟨*dim*⟩
This function returns the length value kept in ⟨*dim*⟩ in a way suitable for further processing.

TEXhackers 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_eval:n` `\dim_eval:n` {⟨*dim expr*⟩}
Evaluates the value of a dimension expression so that `\dim_eval:n {5pt+3pt}` puts `8pt` back into the input stream. Expandable.

TEXhackers note: This is the ε-TEX primitive `\dimexpr` turned into a function taking an argument.

`\if_dim:w` `\if_dim:w` ⟨*dimen1*⟩ ⟨*rel*⟩ ⟨*dimen2*⟩ ⟨*true*⟩ `\else:` ⟨*false*⟩ `\fi:`
Compare two dimensions. It is recommended to use `\dim_eval:n` to correctly evaluate and terminate these numbers. ⟨*rel*⟩ is one of `<`, `=` or `>` with catcode 12.

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

`\dim_compare:nNnTF`
`\dim_compare:nNnT`
`\dim_compare:nNnF`
`\dim_compare:nNnTF` {⟨*dim expr*⟩} ⟨*rel*⟩ {⟨*dim expr*⟩} {⟨*true*⟩} {⟨*false*⟩}
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.

TEXhackers note: This is the TEX primitive `\ifdim` turned into a function.

`\dim_compare_p:nNn` `\dim_compare_p:nNn` {⟨*dim expr*⟩} ⟨*rel*⟩ {⟨*dim expr*⟩}
Predicate version of the above functions.

`\dim_while:nNnT`
`\dim_while:nNnF`
`\dim_dowhile:nNnT`
`\dim_dowhile:nNnF` `\dim_while:nNnT` ⟨*dim expr*⟩ ⟨*rel*⟩ ⟨*dim expr*⟩ ⟨*true*⟩
`\dim_while:nNnT` tests the dimension expressions and if true performs the body `T` until the test fails. `\dim_dowhile:nNnT` is similar but executes the body first and then performs the check, thus ensuring that the body is executed at least once. The `F` versions are similar but continue the loop as long as the test is false.

15.2.2 Variable and constants

`\c_max_dim` Constant that denotes the maximum value which can be stored in a ⟨*dim*⟩ register.

`\c_zero_dim` Set of constants denoting useful values.

`\l_tmpa_dim`
`\l_tmpb_dim`
`\l_tmpc_dim`
`\l_tmpd_dim`
`\g_tmpa_dim`
`\g_tmpb_dim` Scratch register for immediate use.

15.3 Muskips

`\muskip_new:N`
`\muskip_new_l:N` `\muskip_new:N` ⟨*muskip*⟩
Defines ⟨*muskip*⟩ to be a new variable of type `muskip`.

TEXhackers note: `\muskip_new:N` is the equivalent to plain TEX's `\newmuskip`. However, the internal register allocation is done differently.

`\muskip_set:Nn`
`\muskip_gset:Nn` `\muskip_set:Nn` ⟨*muskip*⟩ `{` ⟨*muskip value*⟩ `}`
These functions will set the ⟨*muskip*⟩ register to the ⟨*length*⟩ value.

`\muskip_add:Nn`
`\muskip_gadd:Nn` `\muskip_add:Nn` ⟨*muskip*⟩ `{` ⟨*length*⟩ `}`
These functions will add to the ⟨*muskip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*muskip*⟩ register too, the surrounding braces can be left out.

`\muskip_sub:Nn`
`\muskip_gsub:Nn` `\muskip_gsub:Nn` ⟨*muskip*⟩ `{` ⟨*length*⟩ `}`
These functions will subtract from the ⟨*muskip*⟩ register the value ⟨*length*⟩. If the second argument is a ⟨*muskip*⟩ register too, the surrounding braces can be left out.

16 Token Registers

There is a second form beside token list pointers in which LATEX3 stores token lists, namely the internal TEX token registers. Functions dealing with these registers got the prefix `\toks_`. Unlike token list pointers we have an accessing function as one can see below.

The main difference between ⟨*toks*⟩ (token registers) and ⟨*tlp*⟩ (token list pointers) is their behavior regarding expansion. While ⟨*tlp*⟩'s expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denote by `x`) ⟨*toks*⟩'s expand always only up to one level, i.e., passing their contents without further expansion.

16.1 Functions

`\toks_new:N`
`\toks_new:c`
`\toks_new_l:N` `\toks_new:N` ⟨*toks*⟩
Defines ⟨*toks*⟩ to be a new token list register.

TEXhackers note: This is the LATEX3 allocation for what was called `\newtoks` in plain TEX.

\toks_set:Nn
\toks_set:No
\toks_set:Nd
\toks_set:Nf
\toks_set:Nx
\toks_set:cn
\toks_set:co
\toks_set:cf
\toks_set:cx
\toks_gset:Nn
\toks_gset:No
\toks_gset:Nx `\toks_set:Nn` ⟨*toks*⟩ {⟨*token list*⟩}

Defines ⟨*toks*⟩ to hold the token list ⟨*token list*⟩. Global variants of this command assign the value globally the other variants expand the ⟨*token list*⟩ up to a certain level before the assignment or interpret the ⟨*token list*⟩ as a character list and form a control sequence out of it.

TEXhackers note: `\toks_set:Nn` could have been specified in plain TEX by ⟨*toks*⟩ `=` `{` ⟨*token list*⟩ `}` but all other functions have no counterpart in plain TEX. Additionally the functions above will check for correct local and global assignments, something that isn't available in plain TEX.

\toks_gset_eq:NN `\toks_gset_eq:NN` ⟨*toks1*⟩ ⟨*toks2*⟩

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.

\toks_clear:N
\toks_gclear:N `\toks_clear:N` ⟨*toks*⟩

The ⟨*toks*⟩ is locally or globally cleared.

\toks_put_left:Nn
\toks_gput_left:Nn
\toks_put_right:Nn
\toks_put_right:No
\toks_put_right:Nx
\toks_gput_right:Nn
\toks_gput_right:No
\toks_gput_right:Nx `\toks_put_left:Nn` ⟨*toks*⟩ {⟨*token list*⟩}

These functions will append ⟨*token list*⟩ to the left or right of ⟨*toks*⟩. Assignment is done either locally or globally. If possible append to the right since this operation is faster.

\toks_use:N `\toks_use:N` ⟨*toks*⟩

Accesses the contents of ⟨*toks*⟩. Contrary to token list pointers ⟨*toks*⟩ can't be access simply by calling them directly.

TEXhackers note: Something like `\the` ⟨*toks*⟩.

`\toks_use_clear:N`
`\toks_use_gclear:N` `\toks_use_clear:N` ⟨*toks*⟩
Accesses the contents of ⟨*toks*⟩ 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.

16.2 Predicates and conditionals

`\toks_if_empty_p:N`
`\toks_if_empty:NTF`
`\toks_if_empty:NT`
`\toks_if_empty:NF`
`\toks_if_empty_p:c`
`\toks_if_empty:cTF`
`\toks_if_empty:cT`
`\toks_if_empty:cF` `\toks_if_empty:NTF` ⟨*toks*⟩ {⟨*true code*⟩}{⟨*false code*⟩}
Tests if ⟨*toks*⟩ is empty.

16.3 Variable and constants

`\c_empty_toks` Constant that is always empty.

`\l_tmpa_toks`
`\l_tmpb_toks`
`\g_tmpa_toks`
`\g_tmpb_toks` Scratch register for immediate use. They are not used by conditionals or predicate functions.

16.3.1 Internal functions

`\toks_put_left_aux:w` Used by `\toks_put_left:Nn` and its variants.

`\tex_toksdef:D` Primitive function for defining a ⟨*cs*⟩ to correspond to a token register should not be used by a programmer.

TEXhackers note: This function was named `\toksdef`.

17 Communicating with the user

Sometimes it is necesary to pass information back to the user about what is going on. The information can be just that, information, or it can be a warning that something might not happen to his expectation. It could also be that something has gone awry and that processing can't reliably continue without some help from the user. In such a case an error is signalled. When things are really bad, processing may have to stop as there is no way to enter additional commands that put things right again. In such a case we have a fatal error and the LaTeX run will be aborted.

17.1 Displaying the information

First of all we need a couple of fairly low level functions that deal with the job of passing the information to the user.

Real information is usually only written to the log file, while warnings are displayed on the screen as well.

`\err_info:nn`
`\err_warn:nn` `\err_info:nn { ⟨message⟩ } {⟨continuation⟩}`
The ⟨*message*⟩ will be written to the log file. When it contains the command `\err_newline:` a line break will occur and the new line will start with the ⟨*continuation*⟩. The function `\err_warn:nn` writes the message to the terminal as well. When an erroneous situation is encountered, a message is displayed and the user is given the opportunity to enter some additional code in an attempt to put things right. He may first ask for some help, in which case some extra text will be displayed to him.

`\err_interrupt:NNw` `\err_interrupt:NNw ⟨err id⟩ ⟨label⟩ ⟨more args⟩`
This function signals a user error by searching the error file denoted by ⟨*err id*⟩ for an error message associated with ⟨*label*⟩, i.e., specified by a corresponding `\err_interrupt_new:NNNnnn` command. Depending on the number of arguments specified as ⟨*argno*⟩ when the error message was defined, further arguments are read. Then the error message is displayed as explained in `\err_interrupt_new:NNNnnn`.

Finally, when something really serious occurs, LaTeX will tell the user about it and abort the run.

`\err_fatal:nn` `\err_fatal:nn { ⟨message⟩ } {⟨continuation⟩}`
Just displays the ⟨*message*⟩ and then aborts the LaTeX run.

`\err_newline:` `\err_newline:`
Is used to break an informational, warning or error message up into multiple lines. May be defined in such a way that the new line starts with a standard ⟨*continuation*⟩.

17.2 Storing the information

The informational and warning messages are usually short and can be stored as part of a macro; but error messages need to be more verbose. Therefor error messages are stored in external files which are read and searched for the correct error message at the time of the error. In this way it is possible to write extensive help texts without cluttering TEX's main memory.

17.2.1 Dealing with the error file

`\err_file_new:Nn` `\err_file_new:Nn` ⟨*err id*⟩ {⟨*err file name*⟩}
Opens a new error file to write errors to. ⟨*err id*⟩ is a unique identifier for the external ⟨*err file name*⟩. By convention ⟨*err id*⟩ is declared as a constant (i.e., starts with `\c_`) und ends with `_tlp`. If this command is issued while some other error file is open we get an internal error message.

`\err_file_close:N` `\err_file_close:N` ⟨*err id*⟩
Closes the currently open error file and checks that it matches ⟨*err id*⟩, i.e., that everything is alright in the code.

17.2.2 Declaring an error message in the error file

`\err_interrupt_new:NNNnnn`
`\err_interrupt_new:NNNnnn` ⟨*err id*⟩ ⟨*label*⟩ ⟨*argno*⟩
{ ⟨*short msg*⟩ }
{ ⟨*long msg*⟩ }
{ ⟨*recovery code*⟩ }

This function declares an new error message which can be addressed via `\err_interrupt:NNw`. The pair (⟨*err id*⟩, ⟨*label*⟩) has to be unique where ⟨*label*⟩ can be some otherwise arbitrary token (usually the function name in which the error routine is called. Actually, the pair (⟨*err id*⟩, expansion of ⟨*label*⟩) has to be unique since for reasons of speed, tests are carried out using `\if_meaning:NN`.

⟨*argno*⟩ specifies the number of extra arguments that will be supplied to the error routine when `\err_interrupt:NNw` is called. These arguments can be used within ⟨*short msg*⟩, ⟨*long msg*⟩, and/or ⟨*recovery code*⟩ to provide further information to the user. They are denoted with `#1`, `#2`, etc. within these arguments.

The ⟨*short msg*⟩ is displayed directly on the terminal if the error occurs, ⟨*long msg*⟩ is displayed when the user types `h` in response to the error prompt of TEX, and ⟨*recovery code*⟩ is executed afterwards. This means that ⟨*recovery code*⟩ is inserted after any deletions or insertions given by the user. All three arguments are expanded while they are written to the error file, therefore one has to prevent expansion of tokens with `\token_to_string:N` that should be expanded when the error is triggered.

17.3 Internal functions

`\err_display_aux:w` This function is constructed on the fly while reading the error file. It grabs following arguments from the code (if any) and then displays the error message and inserts the ⟨*recovery code*⟩.

`\err_interrupt_new_aux:w` Helper function used to write the error message info onto the error file.

`\err_msgline_aux:NNnnn` `\err_msgline_aux:NNnnn` ⟨*argno*⟩ ⟨*label*⟩ {⟨*short*⟩} {⟨*long msg*⟩}{⟨*recovery code*⟩}

Function written in front of every error message on the error file. It will be executed when the error file is read back in comparing ⟨*label*⟩ to `\l_err_label_token`. If they are the same, `\err_display_aux:w` will be defined and the reading process will stop.

`\err_message:x` `\err_message:x {`⟨*error message*⟩`}`
Function that directly triggers TEX's error handler. It should not be used directly.

TEXhackers note: This is the LATEX3 name for the `\errormessage` primitive.

17.4 Kernel specific functions

For a number of the functions described above specific variants are provided that are used in the kernel of LATEX3.

`\err_kernel_info:n` `\err_kernel_warn:n` `\err_kernel_fatal:n` `\err_kernel_info:n {`⟨*message*⟩`}`

`\err_kernel_interrupt:Nw` `\err_kernel_interrupt_new:NNnnn` Abbrivations for writing and accessing kernel error messages that go to the error file `\c_kernel_err_tlp`.

`\err_latex_bug:x` `\err_latex_bug:x {`⟨*error message*⟩`}`
Creates an internal error message. This is intended to be used in places that should not be reached in normal operation. Something is wrong with the code.

17.5 Variables and constants

`\iow_newline` Identifier denoting the character that triggers a line break in the output.

`\c_iow_err_stream` Output stream used to access the error files during their generation.

`\c_kernel_err_tlp` Identifier denoting the kernel error file. (Its contents is the name of the external file.)

`\g_err_curr_fname` Global variable containing the name of the currently open error file. Empty when no such file is open for writing.

`\tex_errorcontextlines:D` Variable determining the amount of macro expansion contents shown to the user when an error is triggered. LaTeX3 sets this to -1 since to the average user this contents is of no interest.

TeXhackers note: This is the LaTeX3 name for the TeX3 primitive `\errorcontextlines`.

`\g_err_help_toks` Token register that holds the message that will be shown if the user types `h` in response to an error message that was produced by `\err_message:x`.

TeXhackers note: This is the LaTeX3 name for the TeX primitive `\errhelp`.

`\l_err_label_token` Variable holding the ⟨*label*⟩ to look up in an error file.

18 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_`.

18.1 Generic functions

`\box_new:N`
`\box_new:c`
`\box_new_l:N` `\box_new:N` ⟨*box*⟩
Defines ⟨*box*⟩ to be a new variable of type `box`.

TEXhackers note: `\box_new:N` is the equivalent of plain TEX's `\newbox`. However, the internal register allocation is done differently.

`\if_hbox:N`
`\if_vbox:N`
`\if_box_empty:N` `\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.

TEXhackers note: These are the TEX primitives `\ifhbox`, `\ifvbox` and `\ifvoid`.

`\box_if_empty:NTF`
`\box_if_empty:cTF`
`\box_if_empty:NT`
`\box_if_empty:cT`
`\box_if_empty:NF`
`\box_if_empty:cF` `\box_if_empty:NTF` ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
Tests if ⟨*box*⟩ is empty (void) and executes `code` according to the test outcome.

TEXhackers note: `\box_if_empty:NTF` is the LATEX3 function name for `\ifvoid`.

`\box_set_eq:NN`
`\box_set_eq:cN`
`\box_set_eq:Nc`
`\box_set_eq:cc` `\box_set_eq:NN` ⟨*box1*⟩ ⟨*box2*⟩
Sets ⟨*box1*⟩ equal to ⟨*box2*⟩. Note that this eradicates the contents of ⟨*box2*⟩ afterwards.

`\box_gset_eq:NN`
`\box_gset_eq:cN`
`\box_gset_eq:Nc`
`\box_gset_eq:cc` `\box_gset_eq:NN` ⟨*box1*⟩ ⟨*box2*⟩
Globally sets ⟨*box1*⟩ equal to ⟨*box2*⟩.

\box_set_to_previous:N
\box_set_to_previous:c
\box_gset_to_previous:N
\box_gset_to_previous:c

\box_set_to_previous:N ⟨*box*⟩

Sets ⟨*box*⟩ equal to the previous box \R_previous_box and removes \R_previous_box from the current list (unless in outer vertical or math mode).

\box_move_right:nn
\box_move_left:nn
\box_move_up:nn
\box_move_down:nn

\box_move_left:nn {⟨*dimen*⟩} {⟨*box function*⟩}

Moves ⟨*box function*⟩ ⟨*dimen*⟩ in the direction specified. ⟨*box function*⟩ is either an operation on a box such as \box_use:N or a "raw" box specification like \vbox:n{xyz}.

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

\box_clear:N ⟨*box*⟩

Clears ⟨*box*⟩ by setting it to the constant \c_void_box. \box_gclear:N does it globally.

\box_use:N
\box_use:c
\box_use_clear:N
\box_use_clear:c

\box_use:N ⟨*box*⟩
\box_use_clear:N ⟨*box*⟩

\box_use:N puts a copy of ⟨*box*⟩ on the current list while \box_use_clear:N puts the box on the current list and then eradicates the contents of it.

TEXhackers note: \box_use:N and \box_use_clear:N are the TEX primitives \copy and \box with new (descriptive) names.

\box_ht:N
\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c

\box_ht:N ⟨*box*⟩

Returns the height, depth, and width of ⟨*box*⟩ for use in dimension settings.

TEXhackers note: These are the TEX primitives \ht, \dp and \wd.

\box_show:N
\box_show:c
\box_show:N ⟨*box*⟩
Writes the contents of ⟨*box*⟩ to the log file.

TEXhackers note: This is the TEX primitive \showbox.

\c_empty_box
\l_tmpa_box
\l_tmpb_box
\c_empty_box is the constantly empty box. The others are scratch boxes.

\R_previous_box
\R_previous_box is a read-only box register. You can set other boxes to this box, which will then be removed from the current list.

18.2 Horizontal mode

\hbox:n \hbox:n {⟨*contents*⟩}
Places a hbox of natural size.

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
\hbox_set:Nn ⟨*box*⟩ {⟨*contents*⟩}
Sets ⟨*box*⟩ to be a vertical mode box containing ⟨*contents*⟩. It has it's natural size. \hbox_gset:Nn does it globally.

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
\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.

\hbox_to_wd:nn \hbox_to_wd:nn {⟨*dimen*⟩} ⟨*contents*⟩
Places a ⟨*box*⟩ of width ⟨*dimen*⟩ containing ⟨*contents*⟩.

\hbox_set_inline_begin:N
\hbox_set_inline_end:
\hbox_gset_inline_begin:N
\hbox_gset_inline_end:

\hbox_set_inline_begin:N ⟨*box*⟩ ⟨*contents*⟩
\hbox_set_inline_end:

Sets ⟨*box*⟩ to contain ⟨*contents*⟩. This type is useful for use in environment definitions.

\hbox_unpack:N
\hbox_unpack_clear:N

\hbox_unpack:N ⟨*box*⟩

\hbox_unpack:N unpacks the contents of the ⟨*box*⟩ register and \hbox_unpack_clear:N also clears the ⟨*box*⟩ after unpacking it.

TEXhackers note: These are the TEX primitives \unhcopy and \unhbox.

18.3 Vertical mode

\vbox_set:Nn
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

\vbox_set:Nn ⟨*box*⟩ {⟨*contents*⟩}
Sets ⟨*box*⟩ to be a vertical mode box containing ⟨*contents*⟩. It has it's natural size. \vbox_gset:Nn does it globally.

\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn
\vbox_gset_to_ht:ccn

\vbox_set_to_ht:Nnn ⟨*box*⟩ {⟨*dimen*⟩} {⟨*contents*⟩}

Sets ⟨*box*⟩ to contain ⟨*contents*⟩ and have total height ⟨*dimen*⟩. \vbox_gset_to_ht:Nn does it globally.

\vbox_set_inline_begin:N
\vbox_set_inline_end:
\vbox_gset_inline_begin:N
\vbox_gset_inline_end:

\vbox_set_inline_begin:N ⟨*box*⟩ ⟨*contents*⟩
\vbox_set_inline_end:

Sets ⟨*box*⟩ to contain ⟨*contents*⟩. This type is useful for use in environment definitions.

\vbox_set_split_to_ht:NNn `\vbox_set_split_to_ht:NNn ⟨box1⟩ ⟨box2⟩ {⟨dimen⟩}`

Sets ⟨*box1*⟩ to contain the top ⟨*dimen*⟩ part of ⟨*box2*⟩.

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

\vbox:n `\vbox:n {⟨contents⟩}`
Places a vbox of natural size with baseline equal to the baseline of the last line in the box.

\vbox_to_ht:nn
\vbox_to_zero:n
`\vbox_to_ht:nn {⟨dimen⟩} ⟨contents⟩`
`\vbox_to_zero:n ⟨contents⟩`
Places a ⟨*box*⟩ of size ⟨*dimen*⟩ containing ⟨*contents*⟩.

\vbox_unpack:N
\vbox_unpack_clear:N
`\vbox_unpack:N ⟨box⟩`

`\vbox_unpack:N` unpacks the contents of the ⟨*box*⟩ register and `\vbox_unpack_clear:N` also clears the ⟨*box*⟩ after unpacking it.

TEXhackers note: These are the TEX primitives `\unvcopy` and `\unvbox`.

19 Control sequence functions extended ...

\cs_gen_sym:N
\cs_ggen_sym:N
`\cs_gen_sym:N ⟨tlp⟩`
These functions will generate a new control sequence name for use as a pointer, e.g. some tree structure like the LDB. The new unique name is returned locally in ⟨*tlp*⟩ for further use. The names are generated using the roman numeral representation of some special counters together with a prefix of `\l*` (local) or `\g*`(global).

\cs_record_name:N `\cs_record_name:N ⟨cs⟩`
Takes the ⟨*cs*⟩ and saves it in a special places for pre-compiling purposes on a file later on. All control sequences that are recorded with this function will be dumped by `\cs_dump:`. This function is internally automatically used to record all symbols generated by `\cs_gen_sym:N` and `\cs_ggen_sym:N`.

\cs_load_dump:n `\cs_load_dump:n { ⟨file name⟩ }`
Loads and executes the file ⟨*file name*⟩ if found. Then scans further ignoring everything

until finding `\cs_dump:` where normal execution continues. If ⟨*file name*⟩ is not found, the name is saved and normal execution of all following code is done until `\cs_dump:` is scanned. Then all symbols marked for dumping are dumped into ⟨*file name*⟩.

`\cs_dump:` Dumps the symbols recorded by `\cs_record_name:N` in the file given by the argument in `\cs_load_dump:n`. Dumping means that for every ⟨*cs*⟩ recorded by `\cs_record_name:N` a line

`\def:Npn` ⟨*cs*⟩ `{` ⟨*current meaning of cs*⟩ `}`

is written to this file. This means that when loading the file the definitions of all these ⟨*cs*⟩'s are directly available.

19.1 Internal variables

`\g_gen_sym_num`
`\g_ggen_sym_num` Holds the number of the last generated symbol by `\cs_gen_sym:N` or `\cs_ggen_sym:N`.

`\g_cs_dump_seq` Sequence in which the symbols to be dumped are stored.

`\c_cs_dump_stream` Output stream used for writing out the definitions of the recorded ⟨*tlp*⟩.

20 Quarks

A special type of constants in LaTeX3 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:NN`. 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_`.

20.1 Functions

\quark_new:N `\quark_new:N` ⟨*quark*⟩
Defines ⟨*quark*⟩ to be a new constant of type `quark`.

\quark_if_no_value_p:n
\quark_if_no_value:nTF
\quark_if_no_value:nF
\quark_if_no_value:nT
\quark_if_no_value_p:N
\quark_if_no_value:NTF
\quark_if_no_value:NT
\quark_if_no_value:NF

`\quark_if_no_value:nTF {`⟨*token list*⟩`}`
`  {`⟨*true code*⟩`}{`⟨*false code*⟩`}`
`\quark_if_no_value:NTF` ⟨*tlp*⟩
`  {`⟨*true code*⟩`}{`⟨*false 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 pointer use `\quark_if_no_value:NTF`, or `\quark_if_no_value:NF` or check the value directly with `\if_meaning:NN`. All those cases are faster then `\quark_if_no_value:nTF` so should be preferred.[2]

TEXhackers note: But be aware of the fact that `\if_meaning:NN` can result in an overflow of TEX'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.

\quark_if_nil_p:N
\quark_if_nil:NTF
\quark_if_nil:NT
\quark_if_nil:NF

`\quark_if_nil:NTF` ⟨*token*⟩
`  {`⟨*true code*⟩`}{`⟨*false 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.

\quark_if_nil_p:n
\quark_if_nil:nTF
\quark_if_nil:nT
\quark_if_nil:nF
\quark_if_nil_p:o
\quark_if_nil:oTF
\quark_if_nil:oT
\quark_if_nil:oF

`\quark_if_nil:nTF {`⟨*tokens*⟩`}`
`{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

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

[2] Clarify semantic of the "n" case ... i think it is not implement according to what we originally intended /FMi

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

20.2 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 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.

21 Control structures

21.1 Choosing modes

`\mode_vertical_p:`
`\mode_vertical:TF`
`\mode_vertical:T`
`\mode_vertical:F` `\mode_vertical:TF {⟨true code⟩} {⟨false code⟩}`
Determines if TEX is in vertical mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

`\mode_horizontal_p:`
`\mode_horizontal:TF`
`\mode_horizontal:T`
`\mode_horizontal:F` `\mode_horizontal:TF {⟨true code⟩} {⟨false code⟩}`
Determines if TEX is in horizontal mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

`\mode_inner_p:`
`\mode_inner:TF`
`\mode_inner:T`
`\mode_inner:F` `\mode_inner:TF {⟨true code⟩} {⟨false code⟩}`
Determines if TEX is in inner mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

\mode_math:TF
\mode_math:T
\mode_math:F
\mode_math:TF {⟨*true code*⟩} {⟨*false code*⟩}
Determines if TEX is in math mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

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

21.1.1 Alignment safe grouping and scanning

\scan_align_safe_stop: \scan_align_safe_stop:

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

\group_align_safe_begin:
\group_align_safe_end: \group_align_safe_begin: ⟨...⟩ \group_align_safe_end:

Encloses ⟨...⟩ inside a group but is safe inside an alignment cell. See the implementation of \peek_token_generic:NNTF for an application.

21.2 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.

\prg_replicate:nn \prg_replicate:nn { ⟨*number*⟩ } { ⟨*arg*⟩ }
Creates ⟨*number*⟩ copies of ⟨*arg*⟩. Expandable.

\prg_stepwise_function:nnnN \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. The \prg_stepwise_function:nnnN function is expandable.

\prg_stepwise_inline:nnnn {⟨*start*⟩} {⟨*step*⟩} {⟨*end*⟩} {⟨*action*⟩}

\prg_stepwise_inline:nnnn

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.

\prg_stepwise_variable:nnnn {⟨*start*⟩} {⟨*step*⟩} {⟨*end*⟩} ⟨*temp-var*⟩ {⟨*action*⟩}

\prg_stepwise_variable:nnnNn

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.

21.3 Conditionals and logical operations

LaTeX3 has two primary forms of conditional flow processing. The one type deals with the truth value of a test directly as in \cs_free:NTF where you test if a control sequence was undefined and then execute either the ⟨*true*⟩ or ⟨*false*⟩ part depending on the result and after exiting the underlying \if...\fi: structure. The second type has to do with predicate functions like \cs_free_p:N which return either \c_true or \c_false to be used in testing with \if:w.

This section describes a boolean data type which is closely connected to both parts 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:w test. Here TeX's original concept with \iffalse and \iftrue is a little difficult to handle so the easiest is simply to let a boolean either be \c_true or \c_false. This also means we get the logical operations And, Or, and Not which can then be used on both the boolean type and predicate functions. All functions by the name \prg_if_predicate_ are expandable and expect the input to also be fully expandable. More generic constructs do not contain predicate in their names.

21.3.1 The boolean data type

\bool_new:N ⟨*bool*⟩

\bool_new:N
\bool_new:c

Define a new boolean variable. The initial value is ⟨*false*⟩. A boolean is actually just either \c_true or \c_false.

\bool_set_true:N
\bool_set_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_true:N
\bool_gset_true:c
\bool_gset_false:N
\bool_gset_false:c

\bool_gset_false:N ⟨*bool*⟩

Set ⟨*bool*⟩ either true or false. We can also do this globally.

\bool_set_eq:NN
\bool_set_eq:Nc
\bool_set_eq:cN
\bool_set_eq:cc
\bool_gset_eq:NN
\bool_gset_eq:Nc
\bool_gset_eq:cN
\bool_gset_eq:cc

\bool_set_eq:NN ⟨*bool1*⟩ ⟨*bool2*⟩

Set ⟨*bool1*⟩ equal to the value of ⟨*bool2*⟩.

\bool_if:NTF
\bool_if:NT
\bool_if:NF
\bool_if_p:N

\bool_if:NTF ⟨*bool*⟩ {⟨*true*⟩} {⟨*false*⟩}
\bool_if_p:N ⟨*bool*⟩

Test the truth value of the boolean and execute the ⟨*true*⟩ or ⟨*false*⟩ code. `\bool_if_p:N` is a predicate function for use in `\if:w` tests.

\bool_whiledo:NT
\bool_whiledo:NF
\bool_dowhile:NT
\bool_dowhile:NF

\bool_whiledo:NT ⟨*bool*⟩ {⟨*true*⟩}
\bool_whiledo:NF ⟨*bool*⟩ {⟨*false*⟩}

The `T` versions execute the ⟨*true*⟩ code as long as the boolean is true and the `F` versions execute the ⟨*false*⟩ code as long as the boolean is false. The `whiledo` functions execute the body after testing the boolean and the `dowhile` functions executes the body first and then tests the boolean.

\l_tmpa_bool
\g_tmpa_bool

Reserved booleans.

21.3.2 Logical operations

Somewhat related to the subject of conditional flow processing is logical operators as these deal with ⟨*true*⟩ and ⟨*false*⟩ statements which is precisely what the predicate functions return.

\prg_if_predicate:nTF
\prg_if_predicate:nT
\prg_if_predicate:nF
\prg_if_predicate:nTF {⟨*predicate*⟩} {⟨*true*⟩} {⟨*false*⟩}

The first argument can either be a predicate function like `\cs_free_p:N \foo` or one of the logic tests below.

\prg_if_predicate_ors_p:n
\prg_if_predicate_or_p:nn
\prg_if_predicate_ands_p:n
\prg_if_predicate_and_p:nn
\prg_if_predicate_not_p:n
\prg_if_predicate_ors_p:n {⟨*predicate*⟩ ⟨*predicate*⟩ ...}
\prg_if_predicate_or_p:nn {⟨*predicate*⟩} {⟨*predicate*⟩}
\prg_if_predicate_ands_p:n {⟨*predicate*⟩ ⟨*predicate*⟩ ...}
\prg_if_predicate_and_p:nn {⟨*predicate*⟩} {⟨*predicate*⟩}
\prg_if_predicate_not_p:n {⟨*predicate*⟩}

`\prg_if_predicate_or_p:nn` compares the outcome of two predicate functions (or other logic tests) and if either turns out to be ⟨*true*⟩ returns ⟨*true*⟩ itself. `\prg_if_predicate_and_p:nn` works in a similar way but requires both outcomes to be ⟨*true*⟩ in order for it to return ⟨*true*⟩ itself. `\prg_if_predicate_not_p:n` reverses the truth value of a predicate test or a logic operation. Thus

```
\prg_if_predicate_not_p:n {\prg_if_predicate_not_p:n {\c_true}}
```

ultimately returns ⟨*true*⟩. The argument of the functions `\prg_if_predicate_ors_p:n` and `\prg_if_predicate_ands_p:n` take a list of predicate functions (and their arguments) and return either ⟨*true*⟩ or ⟨*false*⟩.

21.3.3 Generic loops

\prg_whiledo:nT
\prg_whiledo:nF
\prg_dowhile:nT
\prg_dowhile:nF
\prg_whiledo:nT {⟨*test*⟩} {⟨*true*⟩}
\prg_whiledo:nF {⟨*test*⟩} {⟨*false*⟩}

The `T` versions execute the ⟨*true*⟩ code as long as ⟨*test*⟩ is true and the `F` versions execute the ⟨*false*⟩ code as long as ⟨*test*⟩ is false. The `whiledo` functions execute the body after testing the boolean and the `dowhile` functions executes the body first and then tests the boolean. For the `T` versions, ⟨*test*⟩ should end with a function executing only the ⟨*true*⟩ code for some test such as `\tlp_if_eq:NNT`. Similarly the `F` types should end with `\tlp_if_eq:NNF`.

21.4 Sorting

\prg_quicksort:n
\prg_quicksort:n { {⟨*element 1*⟩} {⟨*element 2*⟩} ... {⟨*element n*⟩} }

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 elements are given as argument to the function \prg_quicksort_function:n which the programmer also controls.

\prg_quicksort_function:n \prg_quicksort_compare:nnTF	\prg_quicksort_function:n {⟨*element*⟩} \prg_quicksort_compare:nnTF {⟨*element 1*⟩} {⟨*element 2*⟩}

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

```
\def:NNn\prg_quicksort_function:n 1{{#1}}
\def:NNn\prg_quicksort_compare:nnTF 2{\num_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

```
\def:NNn\prg_quicksort_compare:nnTF 2{
  \num_compare:nNnTF{\tlist_compare:nn{#1}{#2}}>\c_zero }
```

22 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 TEX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such 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 by 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 py a single control sequence is a 'token list pointer' tlp. Functions for these two types are found in the l3tlp module.

22.1 Character tokens

Setting category codes of characters.

\char_set_catcode:nn
\char_set_catcode:w
\char_value_catcode:n
\char_value_catcode:w
\char_show_value_catcode:n
\char_show_value_catcode:w

\char_set_catcode:nn {⟨*char*⟩} {⟨*number*⟩}
\char_set_catcode:w ⟨*char*⟩ = ⟨*number*⟩
\char_value_catcode:n {⟨*char*⟩}
\char_show_value_catcode:n {⟨*char*⟩}

\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 prints the value on the terminal and in the log file. The :w should be avoided unless you have to fiddle with the catcode of { or }.

TEXhackers note: \char_set_catcode:w is the TEX primitive \catcode renamed.

\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

\char_set_lccode:nn {⟨*char*⟩} {⟨*number*⟩}
\char_set_lccode:w ⟨*char*⟩ = ⟨*number*⟩
\char_value_lccode:n {⟨*char*⟩}
\char_show_value_lccode:n {⟨*char*⟩}

Set the lower caser representation of ⟨*char*⟩ for when ⟨*char*⟩ is being converted in \tlist_to_lowercase:n. As above, the :w form is only for people who really, really know what they are doing.

TEXhackers note: \char_set_lccode:w is the TEX primitive \lccode renamed.

\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

\char_set_uccode:nn {⟨*char*⟩} {⟨*number*⟩}
\char_set_uccode:w ⟨*char*⟩ = ⟨*number*⟩
\char_value_uccode:n {⟨*char*⟩}
\char_show_value_uccode:n {⟨*char*⟩}

Set the uppercase representation of ⟨*char*⟩ for when ⟨*char*⟩ is being converted in \tlist_to_uppercase:n. As above, the :w form is only for people who really, really know what they are doing.

TEXhackers note: \char_set_uccode:w is the TEX primitive \uccode renamed.

\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	\char_set_sfcode:nn {⟨*char*⟩} {⟨*number*⟩} \char_set_sfcode:w ⟨*char*⟩ = ⟨*number*⟩ \char_value_sfcode:n {⟨*char*⟩} \char_show_value_sfcode:n {⟨*char*⟩}

Set the space factor for ⟨*char*⟩.

TEXhackers note: `\char_set_sfcode:w` is the TEX primitive `\sfcode` renamed.

22.2 Generic tokens

`\token_new:Nn` `\token_new:Nn ⟨token 1⟩ {⟨token 2⟩}`
Defines ⟨*token 1*⟩ to globally be a snapshot of ⟨*token 2*⟩. This will be an implicit representation of ⟨*token 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

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.

\token_if_group_begin_p:N \token_if_group_begin:NTF \token_if_group_begin:NT \token_if_group_begin:NF	\token_if_group_begin:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a begin group token.

\token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end:NT \token_if_group_end:NF	\token_if_group_end:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an end group token.

```
\token_if_math_shift_p:N
\token_if_math_shift:NTF
\token_if_math_shift:NT
\token_if_math_shift:NF
```
`\token_if_math_shift:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math shift token.

```
\token_if_aligment_tab_p:N
\token_if_aligment_tab:NTF
\token_if_aligment_tab:NT
\token_if_aligment_tab:NF
```
`\token_if_aligment_tab:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an aligment tab token.

```
\token_if_parameter_p:N
\token_if_parameter:NTF
\token_if_parameter:NT
\token_if_parameter:NF
```
`\token_if_parameter:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a parameter token.

```
\token_if_math_superscript_p:N
\token_if_math_superscript:NTF
\token_if_math_superscript:NT
\token_if_math_superscript:NF
```
`\token_if_math_superscript:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math superscript token.

```
\token_if_math_subscript_p:N
\token_if_math_subscript:NTF
\token_if_math_subscript:NT
\token_if_math_subscript:NF
```
`\token_if_math_subscript:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a math subscript token.

```
\token_if_space_p:N
\token_if_space:NTF
\token_if_space:NT
\token_if_space:NF
```
`\token_if_space:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}
Check if ⟨*token*⟩ is a space token.

\token_if_letter_p:N
\token_if_letter:NTF
\token_if_letter:NT
\token_if_letter:NF

\token_if_letter:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a letter token.

\token_if_other_char_p:N
\token_if_other_char:NTF
\token_if_other_char:NT
\token_if_other_char:NF

\token_if_other_char:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an other char token.

\token_if_active_char_p:N
\token_if_active_char:NTF
\token_if_active_char:NT
\token_if_active_char:NF

\token_if_active_char:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is an active char token.

\token_if_eq_meaning_p:NN
\token_if_eq_meaning:NNTF
\token_if_eq_meaning:NNT
\token_if_eq_meaning:NNF

\token_if_eq_meaning:NNTF ⟨*token1*⟩
⟨*token2*⟩{⟨*true*⟩}{⟨*false*⟩}

Check if the meaning of two tokens are identical.

\token_if_eq_catcode_p:NN
\token_if_eq_catcode:NNTF
\token_if_eq_catcode:NNT
\token_if_eq_catcode:NNF

\token_if_eq_catcode:NNTF ⟨*token1*⟩
⟨*token2*⟩{⟨*true*⟩}{⟨*false*⟩}

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

\token_if_eq_charcode_p:NN
\token_if_eq_catcode:NNTF
\token_if_eq_catcode:NNT
\token_if_eq_catcode:NNF

\token_if_eq_catcode:NNTF ⟨*token1*⟩
⟨*token2*⟩{⟨*true*⟩}{⟨*false*⟩}

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

\token_if_macro_p:N
\token_if_macro:NTF
\token_if_macro:NT
\token_if_macro:NF `\token_if_macro:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a macro.

\token_if_cs_p:N
\token_if_cs:NTF
\token_if_cs:NT
\token_if_cs:NF `\token_if_cs:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ 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.

\token_if_expandable_p:N
\token_if_expandable:NTF
\token_if_expandable:NT
\token_if_expandable:NF `\token_if_expandable:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is expandable or not. Note that ⟨*token*⟩ 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 ⟨*toks*⟩ register we're trying to to something with really a ⟨*toks*⟩ register at all?

\token_if_long_macro_p:N
\token_if_long_macro:NTF
\token_if_long_macro:NT
\token_if_long_macro:NF `\token_if_long_macro:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "long" macro.

\token_if_protected_macro_p:N
\token_if_protected_macro:NTF
\token_if_protected_macro:NT
\token_if_protected_macro:NF `\token_if_long_macro:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "protected" macro. This test does *not* return ⟨*true*⟩ if the macro is also "long", see below.

```
\token_if_protected_long_macro_p:N
\token_if_protected_long_macro:NTF
\token_if_protected_long_macro:NT
\token_if_protected_long_macro:NF
```
`\token_if_protected_long_macro:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a "protected long" macro.

```
\token_if_chardef_p:N
\token_if_chardef:NTF
\token_if_chardef:NT
\token_if_chardef:NF
```
`\token_if_chardef:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a chardef.

```
\token_if_mathchardef_p:N
\token_if_mathchardef:NTF
\token_if_mathchardef:NT
\token_if_mathchardef:NF
```
`\token_if_mathchardef:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a mathchardef.

```
\token_if_int_register_p:N
\token_if_int_register:NTF
\token_if_int_register:NT
\token_if_int_register:NF
```
`\token_if_int_register:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be an integer register.

```
\token_if_dim_register_p:N
\token_if_dim_register:NTF
\token_if_dim_register:NT
\token_if_dim_register:NF
```
`\token_if_dim_register:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a dimension register.

```
\token_if_skip_register_p:N
\token_if_skip_register:NTF
\token_if_skip_register:NT
\token_ifskip_register:NF
```
`\token_if_skip_register:NTF` ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a skip register.

\token_if_toks_register_p:N
\token_if_toks_register:NTF
\token_if_toks_register:NT
\token_if_toks_register:NF \token_if_toks_register:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is defined to be a toks register.

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N \token_get_arg_spec:N ⟨*token*⟩

If token is a macro with definition `\def_long: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 ⟨*token*⟩ isn't a macro, these functions return the `\scan_stop:` token.

22.2.1 Useless code: because we can!

\token_if_primitive_p:N
\token_if_primitive:NTF
\token_if_primitive:NT
\token_if_primitive:NF \token_if_primitive:NTF ⟨*token*⟩ {⟨*true*⟩} {⟨*false*⟩}

Check if ⟨*token*⟩ is a primitive. Probably not a very useful function.

22.3 Peeking ahead at the next token

\l_peek_token
\g_peek_token
\l_peek_search_token

Some useful variables. Initially they are set to `?`.

\peek_after:NN
\peek_gafter:NN \peek_after:NN ⟨*function*⟩⟨*token*⟩

Assign ⟨*token*⟩ to `\l_peek_token` and then run ⟨*function*⟩ which should perform some sort of test on this token. Leaves ⟨*token*⟩ in the input stream. `\peek_gafter:NN` does this globally to the token `\g_peek_token`.

TEXhackers note: This is the primitive `\futurelet` turned into a function.

```
\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF
```
`\peek_meaning:NTF` ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

`\peek_meaning:NTF` checks (by using `\if_meaning:NN`) if ⟨*token*⟩ equals the next token in the input stream and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly. `\peek_meaning_remove:NTF` does the same but additionally removes the token if found. The `ignore_spaces` versions skips blank spaces before making the decision.

```
\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF
```
`\peek_charcode:NTF` ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

Same as for the `\peek_meaning:NTF` functions above but these use `\if_charcode:w` to compare the tokens.

```
\peek_catcode:NTF
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF
```
`\peek_catcode:NTF` ⟨*token*⟩ {⟨*true*⟩}{⟨*false*⟩}

Same as for the `\peek_meaning:NTF` functions above but these use `\if_catcode:w` to compare the tokens.

```
\peek_token_generic:NNTF
\peek_token_remove_generic:NNTF
```
`\peek_token_generic:NNTF` ⟨*token*⟩⟨*function*⟩ {⟨*true*⟩}{⟨*false*⟩}

`\peek_token_generic:NNTF` looks ahead and checks if the next token in the input stream is equal to ⟨*token*⟩. It uses ⟨*function*⟩ to make that decision. `\peek_token_remove_generic:NNTF` does the same thing but additionally removes ⟨*token*⟩ from the input stream if it is found. This also works if ⟨*token*⟩ is either `\c_group_begin_token` or `\c_group_end_token`.

```
\peek_execute_branches_meaning:
\peek_execute_branches_charcode:
\peek_execute_branches_catcode:
```
`\peek_execute_branches_meaning:`

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 TEX is comparing tokens: meaning, character code, and category code.

22.3.1 Internal functions

`\l_peek_true_tlp`
`\l_peek_false_tlp`
These token list pointers are used internally when choosing either the true or false branches of a test.

`\peek_tmp:w`
Scratch function used to gobble tokens from the input stream.

`\l_peek_true_aux_tlp`
`\l_peek_true_remove_next_tlp`
These token list pointers 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.

23 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 pointer `\l_xref_curr_`⟨*type*⟩`_tlp` 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 TEX 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.