The LaTeX3 Sources

The LaTeX3 Project*

2008/08/05

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:

\⟨*module*⟩_⟨*description*⟩:⟨*arg-spec*⟩

where ⟨*module*⟩ is one of the (to be) chosen module names and ⟨*description*⟩ is a verbal description of the functionality. ⟨*arg-spec*⟩ 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 ⟨*description*⟩ is allowed to take further _ characters to separate words is necessary.

Currently there exists some functions which don't have a proper ⟨*module*⟩ name.

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

The ⟨*arg-spec*⟩ 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:

$\langle access\rangle\_\langle module\rangle\_\langle description\rangle\_\langle type\rangle$

⟨*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.

2.1 Using the modules

Most of the modules can be used on top of LaTeX and are loaded with the usual `\usepackage` or `\RequirePackage` instructions. As the packages use a coding syntax different from standard LaTeX it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`

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

Issues a catcode regime where spaces are ignored and colon and underscore are letters.

`l3names`
`\ProvidesExplPackage`
`\ProvidesExplClass`

`\RequirePackage{l3names}`
`\ProvidesExplPackage` {⟨*package*⟩}
{⟨*date*⟩} {⟨*version*⟩} {⟨*description*⟩}

The package `l3names` (this module) provides `\ProvidesExplPackage` which is a wrapper for `\ProvidesPackage` and sets up the LaTeX3 catcode settings for programming automatically. Similar for the relationship between `\ProvidesExplClass` and `\ProvidesClass`. Spaces are not ignored in the arguments of these commands.

`\GetIdInfo`
`\filename`
`\filenameext`
`\filedate`
`\fileversion`
`\filetimestamp`
`\fileauthor`

`\RequirePackage{l3names}`
`\GetIdInfo $Id:` ⟨*cvs or svn info field*⟩ `$ {`⟨*description*⟩`}`

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

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

```
\ProvidesExplPackage{\filename}{\filedate}{\fileversion}{\filedescription}
```

3 Starters

This is the base part of LaTeX3 defining things like `catcode`s and redefining the TeX primitives.

We start by setting up \catcodes that we need to define new commands. These are the ones for begin-group and end-group characters.[1]

```
⟨*initex⟩
\catcode`\{=1 % left brace is begin-group character
\catcode`\}=2 % right brace is end-group character
\catcode`\#=6 % hash mark is macro parameter character
\catcode`\^=7 %
\catcode`\^^I=10 % ascii tab is a blank space
⟨/initex⟩
```

Reason for \endlinechar=32 is that a line ending with a backslash will be interpreted as the token \␣ which seems most natural and since spaces are ignored it works as we intend elsewhere.

```
⟨*initex | package⟩
\catcode126=10\relax  % tilde is a space char.
\catcode32=9\relax    % space is ignored
\catcode9=9\relax     % tab also ignored
\endlinechar=32\relax % endline is space
\catcode95=11\relax   % underscore letter
\catcode58=11\relax   % colon letter
```

4 Setting up primitive names

Here is the function that renames TEX's primitives.

Normally the old name is left untouched, but the possibility of undefining the original names is made available by docstrip and package options. If nothing else, this gives a way of checking what 'old code' a package depends on...

If the package option 'removeoldnames' is used then some trick code is run after the end of this file, to skip past the code which has been inserted by LATEX 2_ε to manage the file name stack, this code would break if run once the TEX primitives have been undefined. (What a surprise!) **The option has been temporarily disabled.**

To get things started, give a new name for \let.

```
\let\tex_let:D\let
⟨/initex | package⟩
```

and now an internal function to possibly remove the old name.

```
⟨*initex⟩
\long\def\name_undefine:N#1{
    \tex_let:D#1\c_undefined}
⟨/initex⟩

⟨*package⟩
\DeclareOption{removeoldnames}{
  \long\def\name_undefine:N#1{
    \tex_let:D#1\c_undefined}}
```

[1] Well not needed while this file is running as a package on top of LATEX 2_ε, so omitted from the package code

```
\DeclareOption{keepoldnames}{
  \long\def\name_undefine:N#1{}}

\ExecuteOptions{keepoldnames}

\ProcessOptions
⟨/package⟩
```

The internal function to give the new name and possibly undefine the old name.

```
⟨*initex | package⟩
\long\def\name_primitive:NN#1#2{
  \tex_let:D #2 #1
  \name_undefine:N #1
      }
```

4.1 Assignments

In the current incarnation of this package, all TEX primitives are given a new name of the form `\tex_`*oldname*`:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
\name_primitive:NN \                    \tex_space:D
\name_primitive:NN \/                   \tex_italiccor:D
\name_primitive:NN \-                   \tex_hyphen:D
```

Now all the other primitives.

```
\name_primitive:NN \let                 \tex_let:D
\name_primitive:NN \def                 \tex_def:D
\name_primitive:NN \edef                \tex_edef:D
\name_primitive:NN \gdef                \tex_gdef:D
\name_primitive:NN \xdef                \tex_xdef:D
\name_primitive:NN \chardef             \tex_chardef:D
\name_primitive:NN \countdef            \tex_countdef:D
\name_primitive:NN \dimendef            \tex_dimendef:D
\name_primitive:NN \skipdef             \tex_skipdef:D
\name_primitive:NN \muskipdef           \tex_muskipdef:D
\name_primitive:NN \mathchardef         \tex_mathchardef:D
\name_primitive:NN \toksdef             \tex_toksdef:D
\name_primitive:NN \futurelet           \tex_futurelet:D
\name_primitive:NN \advance             \tex_advance:D
\name_primitive:NN \divide              \tex_divide:D
\name_primitive:NN \multiply            \tex_multiply:D
\name_primitive:NN \font                \tex_font:D
\name_primitive:NN \fam                 \tex_fam:D
\name_primitive:NN \global              \tex_global:D
\name_primitive:NN \long                \tex_long:D
\name_primitive:NN \outer               \tex_outer:D
\name_primitive:NN \setlanguage         \tex_setlanguage:D
\name_primitive:NN \globaldefs          \tex_globaldefs:D
\name_primitive:NN \afterassignment     \tex_afterassignment:D
\name_primitive:NN \aftergroup          \tex_aftergroup:D
```

```
\name_primitive:NN \expandafter          \tex_expandafter:D
\name_primitive:NN \noexpand             \tex_noexpand:D
\name_primitive:NN \begingroup           \tex_begingroup:D
\name_primitive:NN \endgroup             \tex_endgroup:D
\name_primitive:NN \halign               \tex_halign:D
\name_primitive:NN \valign               \tex_valign:D
\name_primitive:NN \cr                   \tex_cr:D
\name_primitive:NN \crcr                 \tex_crcr:D
\name_primitive:NN \noalign              \tex_noalign:D
\name_primitive:NN \omit                 \tex_omit:D
\name_primitive:NN \span                 \tex_span:D
\name_primitive:NN \tabskip              \tex_tabskip:D
\name_primitive:NN \everycr              \tex_everycr:D
\name_primitive:NN \if                   \tex_if:D
\name_primitive:NN \ifcase               \tex_ifcase:D
\name_primitive:NN \ifcat                \tex_ifcat:D
\name_primitive:NN \ifnum                \tex_ifnum:D
\name_primitive:NN \ifodd                \tex_ifodd:D
\name_primitive:NN \ifdim                \tex_ifdim:D
\name_primitive:NN \ifeof                \tex_ifeof:D
\name_primitive:NN \ifhbox               \tex_ifhbox:D
\name_primitive:NN \ifvbox               \tex_ifvbox:D
\name_primitive:NN \ifvoid               \tex_ifvoid:D
\name_primitive:NN \ifx                  \tex_ifx:D
\name_primitive:NN \iffalse              \tex_iffalse:D
\name_primitive:NN \iftrue               \tex_iftrue:D
\name_primitive:NN \ifhmode              \tex_ifhmode:D
\name_primitive:NN \ifmmode              \tex_ifmmode:D
\name_primitive:NN \ifvmode              \tex_ifvmode:D
\name_primitive:NN \ifinner              \tex_ifinner:D
\name_primitive:NN \else                 \tex_else:D
\name_primitive:NN \fi                   \tex_fi:D
\name_primitive:NN \or                   \tex_or:D
\name_primitive:NN \immediate            \tex_immediate:D
\name_primitive:NN \closeout             \tex_closeout:D
\name_primitive:NN \openin               \tex_openin:D
\name_primitive:NN \openout              \tex_openout:D
\name_primitive:NN \read                 \tex_read:D
\name_primitive:NN \write                \tex_write:D
\name_primitive:NN \closein              \tex_closein:D
\name_primitive:NN \newlinechar          \tex_newlinechar:D
\name_primitive:NN \input                \tex_input:D
\name_primitive:NN \endinput             \tex_endinput:D
\name_primitive:NN \inputlineno          \tex_inputlineno:D
\name_primitive:NN \errmessage           \tex_errmessage:D
\name_primitive:NN \message              \tex_message:D
\name_primitive:NN \show                 \tex_show:D
\name_primitive:NN \showthe              \tex_showthe:D
\name_primitive:NN \showbox              \tex_showbox:D
\name_primitive:NN \showlists            \tex_showlists:D
\name_primitive:NN \errhelp              \tex_errhelp:D
\name_primitive:NN \errorcontextlines    \tex_errorcontextlines:D
\name_primitive:NN \tracingcommands      \tex_tracingcommands:D
\name_primitive:NN \tracinglostchars     \tex_tracinglostchars:D
```

```
\name_primitive:NN \tracingmacros          \tex_tracingmacros:D
\name_primitive:NN \tracingonline          \tex_tracingonline:D
\name_primitive:NN \tracingoutput          \tex_tracingoutput:D
\name_primitive:NN \tracingpages           \tex_tracingpages:D
\name_primitive:NN \tracingparagraphs      \tex_tracingparagraphs:D
\name_primitive:NN \tracingrestores        \tex_tracingrestores:D
\name_primitive:NN \tracingstats           \tex_tracingstats:D
\name_primitive:NN \pausing                \tex_pausing:D
\name_primitive:NN \showboxbreadth         \tex_showboxbreadth:D
\name_primitive:NN \showboxdepth           \tex_showboxdepth:D
\name_primitive:NN \batchmode              \tex_batchmode:D
\name_primitive:NN \errorstopmode          \tex_errorstopmode:D
\name_primitive:NN \nonstopmode            \tex_nonstopmode:D
\name_primitive:NN \scrollmode             \tex_scrollmode:D
\name_primitive:NN \end                    \tex_end:D
\name_primitive:NN \csname                 \tex_csname:D
\name_primitive:NN \endcsname              \tex_endcsname:D
\name_primitive:NN \ignorespaces           \tex_ignorespaces:D
\name_primitive:NN \relax                  \tex_relax:D
\name_primitive:NN \the                    \tex_the:D
\name_primitive:NN \mag                    \tex_mag:D
\name_primitive:NN \language               \tex_language:D
\name_primitive:NN \mark                   \tex_mark:D
\name_primitive:NN \topmark                \tex_topmark:D
\name_primitive:NN \firstmark              \tex_firstmark:D
\name_primitive:NN \botmark                \tex_botmark:D
\name_primitive:NN \splitfirstmark         \tex_splitfirstmark:D
\name_primitive:NN \splitbotmark           \tex_splitbotmark:D
\name_primitive:NN \fontname               \tex_fontname:D
\name_primitive:NN \escapechar             \tex_escapechar:D
\name_primitive:NN \endlinechar            \tex_endlinechar:D
\name_primitive:NN \mathchoice             \tex_mathchoice:D
\name_primitive:NN \delimiter              \tex_delimiter:D
\name_primitive:NN \mathaccent             \tex_mathaccent:D
\name_primitive:NN \mathchar               \tex_mathchar:D
\name_primitive:NN \mskip                  \tex_mskip:D
\name_primitive:NN \radical                \tex_radical:D
\name_primitive:NN \vcenter                \tex_vcenter:D
\name_primitive:NN \mkern                  \tex_mkern:D
\name_primitive:NN \above                  \tex_above:D
\name_primitive:NN \abovewithdelims        \tex_abovewithdelims:D
\name_primitive:NN \atop                   \tex_atop:D
\name_primitive:NN \atopwithdelims         \tex_atopwithdelims:D
\name_primitive:NN \over                   \tex_over:D
\name_primitive:NN \overwithdelims         \tex_overwithdelims:D
\name_primitive:NN \displaystyle           \tex_displaystyle:D
\name_primitive:NN \textstyle              \tex_textstyle:D
\name_primitive:NN \scriptstyle            \tex_scriptstyle:D
\name_primitive:NN \scriptscriptstyle      \tex_scriptscriptstyle:D
\name_primitive:NN \nonscript              \tex_nonscript:D
\name_primitive:NN \eqno                   \tex_eqno:D
\name_primitive:NN \leqno                  \tex_leqno:D
\name_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
\name_primitive:NN \abovedisplayskip       \tex_abovedisplayskip:D
```

```
\name_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
\name_primitive:NN \belowdisplayskip      \tex_belowdisplayskip:D
\name_primitive:NN \displaywidowpenalty   \tex_displaywidowpenalty:D
\name_primitive:NN \displayindent         \tex_displayindent:D
\name_primitive:NN \displaywidth          \tex_displaywidth:D
\name_primitive:NN \everydisplay          \tex_everydisplay:D
\name_primitive:NN \predisplaysize        \tex_predisplaysize:D
\name_primitive:NN \predisplaypenalty     \tex_predisplaypenalty:D
\name_primitive:NN \postdisplaypenalty    \tex_postdisplaypenalty:D
\name_primitive:NN \mathbin               \tex_mathbin:D
\name_primitive:NN \mathclose             \tex_mathclose:D
\name_primitive:NN \mathinner             \tex_mathinner:D
\name_primitive:NN \mathop                \tex_mathop:D
\name_primitive:NN \displaylimits         \tex_displaylimits:D
\name_primitive:NN \limits                \tex_limits:D
\name_primitive:NN \nolimits              \tex_nolimits:D
\name_primitive:NN \mathopen              \tex_mathopen:D
\name_primitive:NN \mathord               \tex_mathord:D
\name_primitive:NN \mathpunct             \tex_mathpunct:D
\name_primitive:NN \mathrel               \tex_mathrel:D
\name_primitive:NN \overline              \tex_overline:D
\name_primitive:NN \underline             \tex_underline:D
\name_primitive:NN \left                  \tex_left:D
\name_primitive:NN \right                 \tex_right:D
\name_primitive:NN \binoppenalty          \tex_binoppenalty:D
\name_primitive:NN \relpenalty            \tex_relpenalty:D
\name_primitive:NN \delimitershortfall    \tex_delimitershortfall:D
\name_primitive:NN \delimiterfactor       \tex_delimiterfactor:D
\name_primitive:NN \nulldelimiterspace    \tex_nulldelimiterspace:D
\name_primitive:NN \everymath             \tex_everymath:D
\name_primitive:NN \mathsurround          \tex_mathsurround:D
\name_primitive:NN \medmuskip             \tex_medmuskip:D
\name_primitive:NN \thinmuskip            \tex_thinmuskip:D
\name_primitive:NN \thickmuskip           \tex_thickmuskip:D
\name_primitive:NN \scriptspace           \tex_scriptspace:D
\name_primitive:NN \noboundary            \tex_noboundary:D
\name_primitive:NN \accent                \tex_accent:D
\name_primitive:NN \char                  \tex_char:D
\name_primitive:NN \discretionary         \tex_discretionary:D
\name_primitive:NN \hfil                  \tex_hfil:D
\name_primitive:NN \hfilneg               \tex_hfilneg:D
\name_primitive:NN \hfill                 \tex_hfill:D
\name_primitive:NN \hskip                 \tex_hskip:D
\name_primitive:NN \hss                   \tex_hss:D
\name_primitive:NN \vfil                  \tex_vfil:D
\name_primitive:NN \vfilneg               \tex_vfilneg:D
\name_primitive:NN \vfill                 \tex_vfill:D
\name_primitive:NN \vskip                 \tex_vskip:D
\name_primitive:NN \vss                   \tex_vss:D
\name_primitive:NN \unskip                \tex_unskip:D
\name_primitive:NN \kern                  \tex_kern:D
\name_primitive:NN \unkern                \tex_unkern:D
\name_primitive:NN \hrule                 \tex_hrule:D
\name_primitive:NN \vrule                 \tex_vrule:D
```

```
\name_primitive:NN \leaders                \tex_leaders:D
\name_primitive:NN \cleaders               \tex_cleaders:D
\name_primitive:NN \xleaders               \tex_xleaders:D
\name_primitive:NN \lastkern               \tex_lastkern:D
\name_primitive:NN \lastskip               \tex_lastskip:D
\name_primitive:NN \indent                 \tex_indent:D
\name_primitive:NN \par                    \tex_par:D
\name_primitive:NN \noindent               \tex_noindent:D
\name_primitive:NN \vadjust                \tex_vadjust:D
\name_primitive:NN \baselineskip           \tex_baselineskip:D
\name_primitive:NN \lineskip               \tex_lineskip:D
\name_primitive:NN \lineskiplimit          \tex_lineskiplimit:D
\name_primitive:NN \clubpenalty            \tex_clubpenalty:D
\name_primitive:NN \widowpenalty           \tex_widowpenalty:D
\name_primitive:NN \exhyphenpenalty        \tex_exhyphenpenalty:D
\name_primitive:NN \hyphenpenalty          \tex_hyphenpenalty:D
\name_primitive:NN \linepenalty            \tex_linepenalty:D
\name_primitive:NN \doublehyphendemerits   \tex_doublehyphendemerits:D
\name_primitive:NN \finalhyphendemerits    \tex_finalhyphendemerits:D
\name_primitive:NN \adjdemerits            \tex_adjdemerits:D
\name_primitive:NN \hangafter              \tex_hangafter:D
\name_primitive:NN \hangindent             \tex_hangindent:D
\name_primitive:NN \parshape               \tex_parshape:D
\name_primitive:NN \hsize                  \tex_hsize:D
\name_primitive:NN \lefthyphenmin          \tex_lefthyphenmin:D
\name_primitive:NN \righthyphenmin         \tex_righthyphenmin:D
\name_primitive:NN \leftskip               \tex_leftskip:D
\name_primitive:NN \rightskip              \tex_rightskip:D
\name_primitive:NN \looseness              \tex_looseness:D
\name_primitive:NN \parskip                \tex_parskip:D
\name_primitive:NN \parindent              \tex_parindent:D
\name_primitive:NN \uchyph                 \tex_uchyph:D
\name_primitive:NN \emergencystretch       \tex_emergencystretch:D
\name_primitive:NN \pretolerance           \tex_pretolerance:D
\name_primitive:NN \tolerance              \tex_tolerance:D
\name_primitive:NN \spaceskip              \tex_spaceskip:D
\name_primitive:NN \xspaceskip             \tex_xspaceskip:D
\name_primitive:NN \parfillskip            \tex_parfillskip:D
\name_primitive:NN \everypar               \tex_everypar:D
\name_primitive:NN \prevgraf               \tex_prevgraf:D
\name_primitive:NN \spacefactor            \tex_spacefactor:D
\name_primitive:NN \shipout                \tex_shipout:D
\name_primitive:NN \vsize                  \tex_vsize:D
\name_primitive:NN \interlinepenalty       \tex_interlinepenalty:D
\name_primitive:NN \brokenpenalty          \tex_brokenpenalty:D
\name_primitive:NN \topskip                \tex_topskip:D
\name_primitive:NN \maxdeadcycles          \tex_maxdeadcycles:D
\name_primitive:NN \maxdepth               \tex_maxdepth:D
\name_primitive:NN \output                 \tex_output:D
\name_primitive:NN \deadcycles             \tex_deadcycles:D
\name_primitive:NN \pagedepth              \tex_pagedepth:D
\name_primitive:NN \pagestretch            \tex_pagestretch:D
\name_primitive:NN \pagefilstretch         \tex_pagefilstretch:D
\name_primitive:NN \pagefillstretch        \tex_pagefillstretch:D
```

```
\name_primitive:NN \pagefilllstretch      \tex_pagefilllstretch:D
\name_primitive:NN \pageshrink            \tex_pageshrink:D
\name_primitive:NN \pagegoal              \tex_pagegoal:D
\name_primitive:NN \pagetotal             \tex_pagetotal:D
\name_primitive:NN \outputpenalty         \tex_outputpenalty:D
\name_primitive:NN \hoffset               \tex_hoffset:D
\name_primitive:NN \voffset               \tex_voffset:D
\name_primitive:NN \insert                \tex_insert:D
\name_primitive:NN \holdinginserts        \tex_holdinginserts:D
\name_primitive:NN \floatingpenalty       \tex_floatingpenalty:D
\name_primitive:NN \insertpenalties       \tex_insertpenalties:D
\name_primitive:NN \lower                 \tex_lower:D
\name_primitive:NN \moveleft              \tex_moveleft:D
\name_primitive:NN \moveright             \tex_moveright:D
\name_primitive:NN \raise                 \tex_raise:D
\name_primitive:NN \copy                  \tex_copy:D
\name_primitive:NN \lastbox               \tex_lastbox:D
\name_primitive:NN \vsplit                \tex_vsplit:D
\name_primitive:NN \unhbox                \tex_unhbox:D
\name_primitive:NN \unhcopy               \tex_unhcopy:D
\name_primitive:NN \unvbox                \tex_unvbox:D
\name_primitive:NN \unvcopy               \tex_unvcopy:D
\name_primitive:NN \setbox                \tex_setbox:D
\name_primitive:NN \hbox                  \tex_hbox:D
\name_primitive:NN \vbox                  \tex_vbox:D
\name_primitive:NN \vtop                  \tex_vtop:D
\name_primitive:NN \prevdepth             \tex_prevdepth:D
\name_primitive:NN \badness               \tex_badness:D
\name_primitive:NN \hbadness              \tex_hbadness:D
\name_primitive:NN \vbadness              \tex_vbadness:D
\name_primitive:NN \hfuzz                 \tex_hfuzz:D
\name_primitive:NN \vfuzz                 \tex_vfuzz:D
\name_primitive:NN \overfullrule          \tex_overfullrule:D
\name_primitive:NN \boxmaxdepth           \tex_boxmaxdepth:D
\name_primitive:NN \splitmaxdepth         \tex_splitmaxdepth:D
\name_primitive:NN \splittopskip          \tex_splittopskip:D
\name_primitive:NN \everyhbox             \tex_everyhbox:D
\name_primitive:NN \everyvbox             \tex_everyvbox:D
\name_primitive:NN \nullfont              \tex_nullfont:D
\name_primitive:NN \textfont              \tex_textfont:D
\name_primitive:NN \scriptfont            \tex_scriptfont:D
\name_primitive:NN \scriptscriptfont      \tex_scriptscriptfont:D
\name_primitive:NN \fontdimen             \tex_fontdimen:D
\name_primitive:NN \hyphenchar            \tex_hyphenchar:D
\name_primitive:NN \skewchar              \tex_skewchar:D
\name_primitive:NN \defaulthyphenchar     \tex_defaulthyphenchar:D
\name_primitive:NN \defaultskewchar       \tex_defaultskewchar:D
\name_primitive:NN \number                \tex_number:D
\name_primitive:NN \romannumeral          \tex_romannumeral:D
\name_primitive:NN \string                \tex_string:D
\name_primitive:NN \lowercase             \tex_lowercase:D
\name_primitive:NN \uppercase             \tex_uppercase:D
\name_primitive:NN \meaning               \tex_meaning:D
\name_primitive:NN \penalty               \tex_penalty:D
```

```
\name_primitive:NN \unpenalty            \tex_unpenalty:D
\name_primitive:NN \lastpenalty          \tex_lastpenalty:D
\name_primitive:NN \special              \tex_special:D
\name_primitive:NN \dump                 \tex_dump:D
\name_primitive:NN \patterns             \tex_patterns:D
\name_primitive:NN \hyphenation          \tex_hyphenation:D
\name_primitive:NN \time                 \tex_time:D
\name_primitive:NN \day                  \tex_day:D
\name_primitive:NN \month                \tex_month:D
\name_primitive:NN \year                 \tex_year:D
\name_primitive:NN \jobname              \tex_jobname:D
\name_primitive:NN \everyjob             \tex_everyjob:D
\name_primitive:NN \count                \tex_count:D
\name_primitive:NN \dimen                \tex_dimen:D
\name_primitive:NN \skip                 \tex_skip:D
\name_primitive:NN \toks                 \tex_toks:D
\name_primitive:NN \muskip               \tex_muskip:D
\name_primitive:NN \box                  \tex_box:D
\name_primitive:NN \wd                   \tex_wd:D
\name_primitive:NN \ht                   \tex_ht:D
\name_primitive:NN \dp                   \tex_dp:D
\name_primitive:NN \catcode              \tex_catcode:D
\name_primitive:NN \delcode              \tex_delcode:D
\name_primitive:NN \sfcode               \tex_sfcode:D
\name_primitive:NN \lccode               \tex_lccode:D
\name_primitive:NN \uccode               \tex_uccode:D
\name_primitive:NN \mathcode             \tex_mathcode:D
```

Since LaTeX3 will require at least the ε-TeX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

```
\name_primitive:NN \ifdefined            \etex_ifdefined:D
\name_primitive:NN \ifcsname             \etex_ifcsname:D
\name_primitive:NN \unless               \etex_unless:D
\name_primitive:NN \eTeXversion          \etex_eTeXversion:D
\name_primitive:NN \eTeXrevision         \etex_eTeXrevision:D
\name_primitive:NN \marks                \etex_marks:D
\name_primitive:NN \topmarks             \etex_topmarks:D
\name_primitive:NN \firstmarks           \etex_firstmarks:D
\name_primitive:NN \botmarks             \etex_botmarks:D
\name_primitive:NN \splitfirstmarks      \etex_splitfirstmarks:D
\name_primitive:NN \splitbotmarks        \etex_splitbotmarks:D
\name_primitive:NN \unexpanded           \etex_unexpanded:D
\name_primitive:NN \detokenize           \etex_detokenize:D
\name_primitive:NN \scantokens           \etex_scantokens:D
\name_primitive:NN \showtokens           \etex_showtokens:D
\name_primitive:NN \readline             \etex_readline:D
\name_primitive:NN \tracingassigns       \etex_tracingassigns:D
\name_primitive:NN \tracingscantokens    \etex_tracingscantokens:D
\name_primitive:NN \tracingnesting       \etex_tracingnesting:D
\name_primitive:NN \tracingifs           \etex_tracingifs:D
\name_primitive:NN \currentiflevel       \etex_currentiflevel:D
\name_primitive:NN \currentifbranch      \etex_currentifbranch:D
\name_primitive:NN \currentiftype        \etex_currentiftype:D
\name_primitive:NN \tracinggroups        \etex_tracinggroups:D
```

```
\name_primitive:NN \currentgrouplevel     \etex_currentgrouplevel:D
\name_primitive:NN \currentgrouptype      \etex_currentgrouptype:D
\name_primitive:NN \showgroups            \etex_showgroups:D
\name_primitive:NN \showifs               \etex_showifs:D
\name_primitive:NN \interactionmode       \etex_interactionmode:D
\name_primitive:NN \lastnodetype          \etex_lastnodetype:D
\name_primitive:NN \iffontchar            \etex_iffontchar:D
\name_primitive:NN \fontcharht            \etex_fontcharht:D
\name_primitive:NN \fontchardp            \etex_fontchardp:D
\name_primitive:NN \fontcharwd            \etex_fontcharwd:D
\name_primitive:NN \fontcharic            \etex_fontcharic:D
\name_primitive:NN \parshapeindent        \etex_parshapeindent:D
\name_primitive:NN \parshapelength        \etex_parshapelength:D
\name_primitive:NN \parshapedimen         \etex_parshapedimen:D
\name_primitive:NN \numexpr               \etex_numexpr:D
\name_primitive:NN \dimexpr               \etex_dimexpr:D
\name_primitive:NN \glueexpr              \etex_glueexpr:D
\name_primitive:NN \muexpr                \etex_muexpr:D
\name_primitive:NN \gluestretch           \etex_gluestretch:D
\name_primitive:NN \glueshrink            \etex_glueshrink:D
\name_primitive:NN \gluestretchorder      \etex_gluestretchorder:D
\name_primitive:NN \glueshrinkorder       \etex_glueshrinkorder:D
\name_primitive:NN \gluetomu              \etex_gluetomu:D
\name_primitive:NN \mutoglue              \etex_mutoglue:D
\name_primitive:NN \lastlinefit           \etex_lastlinefit:D
\name_primitive:NN \interlinepenalties    \etex_interlinepenalties:D
\name_primitive:NN \clubpenalties         \etex_clubpenalties:D
\name_primitive:NN \widowpenalties        \etex_widowpenalties:D
\name_primitive:NN \displaywidowpenalties \etex_displaywidowpenalties:D
\name_primitive:NN \middle                \etex_middle:D
\name_primitive:NN \savinghyphcodes       \etex_savinghyphcodes:D
\name_primitive:NN \savingvdiscards       \etex_savingvdiscards:D
\name_primitive:NN \pagediscards          \etex_pagediscards:D
\name_primitive:NN \splitdiscards         \etex_splitdiscards:D
\name_primitive:NN \TeXXETstate           \etex_TeXXETstate:D
\name_primitive:NN \beginL                \etex_beginL:D
\name_primitive:NN \endL                  \etex_endL:D
\name_primitive:NN \beginR                \etex_beginR:D
\name_primitive:NN \endR                  \etex_endR:D
\name_primitive:NN \predisplaydirection   \etex_predisplaydirection:D
\name_primitive:NN \everyeof              \etex_everyeof:D
\name_primitive:NN \protected             \etex_protected:D
```

All major distributions[2] use pdfε-TEX as engine so we add these names as well. Since the pdfTEX team has been very good at prefixing most primitives with `pdf` (so far only five do not start with `pdf`) we do not give then a double `pdf` prefix. The list below covers pdfTEXv 1.30.4.

```
%% integer registers:
\name_primitive:NN \pdfoutput             \pdf_output:D
\name_primitive:NN \pdfminorversion       \pdf_minorversion:D
\name_primitive:NN \pdfcompresslevel      \pdf_compresslevel:D
\name_primitive:NN \pdfdecimaldigits      \pdf_decimaldigits:D
```

[2] At the time of writing MiKTEX does not but I have a gut feeling that will change.

```
\name_primitive:NN \pdfimageresolution    \pdf_imageresolution:D
\name_primitive:NN \pdfpkresolution       \pdf_pkresolution:D
\name_primitive:NN \pdftracingfonts       \pdf_tracingfonts:D
\name_primitive:NN \pdfuniqueresname      \pdf_uniqueresname:D
\name_primitive:NN \pdfadjustspacing      \pdf_adjustspacing:D
\name_primitive:NN \pdfprotrudechars      \pdf_protrudechars:D
\name_primitive:NN \efcode                \pdf_efcode:D
\name_primitive:NN \lpcode                \pdf_lpcode:D
\name_primitive:NN \rpcode                \pdf_rpcode:D
\name_primitive:NN \pdfforcepagebox       \pdf_forcepagebox:D
\name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
\name_primitive:NN \pdfinclusionerrorlevel\pdf_inclusionerrorlevel:D
\name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
\name_primitive:NN \pdfimagehicolor       \pdf_imagehicolor:D
\name_primitive:NN \pdfimageapplygamma    \pdf_imageapplygamma:D
\name_primitive:NN \pdfgamma              \pdf_gamma:D
\name_primitive:NN \pdfimagegamma         \pdf_imagegamma:D
%% dimen registers:
\name_primitive:NN \pdfhorigin            \pdf_horigin:D
\name_primitive:NN \pdfvorigin            \pdf_vorigin:D
\name_primitive:NN \pdfpagewidth          \pdf_pagewidth:D
\name_primitive:NN \pdfpageheight         \pdf_pageheight:D
\name_primitive:NN \pdflinkmargin         \pdf_linkmargin:D
\name_primitive:NN \pdfdestmargin         \pdf_destmargin:D
\name_primitive:NN \pdfthreadmargin       \pdf_threadmargin:D
%% token registers:
\name_primitive:NN \pdfpagesattr          \pdf_pagesattr:D
\name_primitive:NN \pdfpageattr           \pdf_pageattr:D
\name_primitive:NN \pdfpageresources      \pdf_pageresources:D
\name_primitive:NN \pdfpkmode             \pdf_pkmode:D
%% expandable commands:
\name_primitive:NN \pdftexrevision        \pdf_texrevision:D
\name_primitive:NN \pdftexbanner          \pdf_texbanner:D
\name_primitive:NN \pdfcreationdate       \pdf_creationdate:D
\name_primitive:NN \pdfpageref            \pdf_pageref:D
\name_primitive:NN \pdfxformname          \pdf_xformname:D
\name_primitive:NN \pdffontname           \pdf_fontname:D
\name_primitive:NN \pdffontobjnum         \pdf_fontobjnum:D
\name_primitive:NN \pdffontsize           \pdf_fontsize:D
\name_primitive:NN \pdfincludechars       \pdf_includechars:D
\name_primitive:NN \leftmarginkern        \pdf_leftmarginkern:D
\name_primitive:NN \rightmarginkern       \pdf_rightmarginkern:D
\name_primitive:NN \pdfescapestring       \pdf_escapestring:D
\name_primitive:NN \pdfescapename         \pdf_escapename:D
\name_primitive:NN \pdfescapehex          \pdf_escapehex:D
\name_primitive:NN \pdfunescapehex        \pdf_unescapehex:D
\name_primitive:NN \pdfstrcmp             \pdf_strcmp:D
\name_primitive:NN \pdfuniformdeviate     \pdf_uniformdeviate:D
\name_primitive:NN \pdfnormaldeviate      \pdf_normaldeviate:D
\name_primitive:NN \pdfmdfivesum          \pdf_mdfivesum:D
\name_primitive:NN \pdffilemoddate        \pdf_filemoddate:D
\name_primitive:NN \pdffilesize           \pdf_filesize:D
\name_primitive:NN \pdffiledump           \pdf_filedump:D
%% read-only integers:
```

```
\name_primitive:NN \pdftexversion         \pdf_texversion:D
\name_primitive:NN \pdflastobj            \pdf_lastobj:D
\name_primitive:NN \pdflastxform          \pdf_lastxform:D
\name_primitive:NN \pdflastximage         \pdf_lastximage:D
\name_primitive:NN \pdflastximagepages    \pdf_lastximagepages:D
\name_primitive:NN \pdflastannot          \pdf_lastannot:D
\name_primitive:NN \pdflastxpos           \pdf_lastxpos:D
\name_primitive:NN \pdflastypos           \pdf_lastypos:D
\name_primitive:NN \pdflastdemerits       \pdf_lastdemerits:D
\name_primitive:NN \pdfelapsedtime        \pdf_elapsedtime:D
\name_primitive:NN \pdfrandomseed         \pdf_randomseed:D
\name_primitive:NN \pdfshellescape        \pdf_shellescape:D
%% general commands:
\name_primitive:NN \pdfobj                \pdf_obj:D
\name_primitive:NN \pdfrefobj             \pdf_refobj:D
\name_primitive:NN \pdfxform              \pdf_xform:D
\name_primitive:NN \pdfrefxform           \pdf_refxform:D
\name_primitive:NN \pdfximage             \pdf_ximage:D
\name_primitive:NN \pdfrefximage          \pdf_refximage:D
\name_primitive:NN \pdfannot              \pdf_annot:D
\name_primitive:NN \pdfstartlink          \pdf_startlink:D
\name_primitive:NN \pdfendlink            \pdf_endlink:D
\name_primitive:NN \pdfoutline            \pdf_outline:D
\name_primitive:NN \pdfdest               \pdf_dest:D
\name_primitive:NN \pdfthread             \pdf_thread:D
\name_primitive:NN \pdfstartthread        \pdf_startthread:D
\name_primitive:NN \pdfendthread          \pdf_endthread:D
\name_primitive:NN \pdfsavepos            \pdf_savepos:D
\name_primitive:NN \pdfinfo               \pdf_info:D
\name_primitive:NN \pdfcatalog            \pdf_catalog:D
\name_primitive:NN \pdfnames              \pdf_names:D
\name_primitive:NN \pdfmapfile            \pdf_mapfile:D
\name_primitive:NN \pdfmapline            \pdf_mapline:D
\name_primitive:NN \pdffontattr           \pdf_fontattr:D
\name_primitive:NN \pdftrailer            \pdf_trailer:D
\name_primitive:NN \pdffontexpand         \pdf_fontexpand:D
%%\name_primitive:NN \vadjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
\name_primitive:NN \pdfliteral            \pdf_literal:D
%%\name_primitive:NN \special <pdfspecial spec>
\name_primitive:NN \pdfresettimer         \pdf_resettimer:D
\name_primitive:NN \pdfsetrandomseed      \pdf_setrandomseed:D
\name_primitive:NN \pdfnoligatures        \pdf_noligatures:D
```

What about Omega and Aleph? The status of both are unclear but let's start by adding a single primitive which we can use for testing if we have one of these engines.

```
\name_primitive:NN \textdir             \aleph_textdir:D
```

`\CodeStart` `\CodeStop`

Here we define functions that are used to turn on and off the special conventions used in the kernel of LaTeX3.

First of all, the space, tab and the return characters will all be ignored inside LaTeX3 code, the latter because endline is set to a space instead. When space characters are needed in LaTeX3 code the ~ character will be used for that purpose.

```
\tex_def:D\ExplSyntaxOn{
  \tex_def:D\ExplSyntaxStatus{00}
  \tex_catcode:D  126=10 \tex_relax:D % tilde is a space char.
  \tex_catcode:D   32=9  \tex_relax:D % space is ignored
  \tex_catcode:D    9=9  \tex_relax:D % tab also ignored
  \tex_endlinechar:D =32 \tex_relax:D % endline is space
  \tex_catcode:D   95=11 \tex_relax:D % underscore letter
  \tex_catcode:D   58=11 \tex_relax:D % colon letter
}

\tex_def:D\ExplSyntaxOff{
  \tex_def:D\ExplSyntaxStatus{01}
  \tex_catcode:D  126=13 \tex_relax:D
  \tex_catcode:D   32=10 \tex_relax:D
  \tex_catcode:D    9=10 \tex_relax:D
  \tex_endlinechar:D =13 \tex_relax:D
  \tex_catcode:D   95=8  \tex_relax:D
  \tex_catcode:D   58=12 \tex_relax:D
}
```

Temporary while names change.

```
\tex_let:D \CodeStart \ExplSyntaxOn
\tex_let:D \CodeStop \ExplSyntaxOff
```

`\NamesStart` `\NamesStop` Sometimes we need to be able to use names from the kernel of LaTeX3 without adhering it's conventions according to space characters. These macros provide the necessary settings.

```
\tex_def:D \NamesStart{
  \tex_catcode:D `\_=11\tex_relax:D
  \tex_catcode:D `\:=11\tex_relax:D
}
\tex_def:D \NamesStop{
  \tex_catcode:D `\_=8\tex_relax:D
  \tex_catcode:D `\:=12\tex_relax:D
}
```

`\GetIdInfo` `\GetIdInfoAuxi:w` `\GetIdInfoAuxii:w` `\GetIdInfoAuxCVS:w` `\GetIdInfoAuxSVN:w` Extract all information from a cvs or svn field. The formats are slightly different but at least the information is in the same positions so we check in the date format so see if it contains a `/` after the four-digit year. If it does it is cvs else svn and we extract information. To be on the safe side we ensure that spaces in the argument are seen.

```
\tex_def:D\GetIdInfo{
  \tex_begingroup:D
  \tex_catcode:D   32=10 \tex_relax:D % needed? Probably for now.
  \GetIdInfoAuxi:w
}
\tex_def:D\GetIdInfoAuxi:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
  \tex_endgroup:D
  \tex_def:D\filename{#2}
  \tex_def:D\fileversion{#4}
  \tex_def:D\filedescription{#9}
  \tex_def:D\fileauthor{#7}
  \GetIdInfoAuxii:w #5\tex_relax:D
```

```
  #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
}
\tex_def:D\GetIdInfoAuxii:w #1#2#3#4#5#6\tex_relax:D{
  \tex_ifx:D#5/
    \tex_expandafter:D\GetIdInfoAuxCVS:w
  \tex_else:D
    \tex_expandafter:D\GetIdInfoAuxSVN:w
  \tex_fi:D
}
\tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
                             #2\tex_relax:D#3\tex_relax:D{
  \tex_def:D\filedate{#2}
  \tex_def:D\filenameext{#1}
  \tex_def:D\filetimestamp{#3}
```

When creating the format we want the information in the log straight away.

```
⟨initex⟩\tex_immediate:D\tex_write:D-1
⟨initex⟩  {\filename;~ v\fileversion,~\filedate;~\filedescription}
}
\tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2-#3-#4
                             \tex_relax:D#5Z\tex_relax:D{
  \tex_def:D\filenameext{#1}
  \tex_def:D\filedate{#2/#3/#4}
  \tex_def:D\filetimestamp{#5}
⟨−package⟩\tex_immediate:D\tex_write:D-1
⟨−package⟩  {\filename;~ v\fileversion,~\filedate;~\filedescription}
}
⟨/initex | package⟩
```

Finally some corrections in the case we are running over LaTeX 2_ε.

We want to set things up so that experimental packages and regular packages can coexist with the former using the LaTeX3 programming catcode settings. Since it cannot be the task of the end user to know how a package is constructed under the hood we make it so that the experimental packages have to identify themselves. As an example it can be done as

```
\RequirePackage{l3names}
\ProvidesExplPackage{agent}{2007/08/28}{007}{bonding module}
```

or by using the `\file`⟨*field*⟩ informations from `\GetIdInfo` as the packages in this distribution do like this:

```
\RequirePackage{l3names}
\GetIdInfo$Id: l3names.dtx 621 2007-09-01 20:14:19Z morten $
         {L3 Experimental Box module}
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
```

`\ProvidesExplPackage`
`\ProvidesExplClass`

First up is the identification. Rather trivial

```
⟨*package⟩
\tex_def:D \ProvidesExplPackage#1#2#3#4{
  \ProvidesPackage{#1}[#2~v#3~#4]
  \ExplSyntaxOn
}
\tex_def:D \ProvidesExplClass#1#2#3#4{
  \ProvidesClass{#1}[#2~v#3~#4]
  \ExplSyntaxOn
}
```

\org@onefilewithoptions
\@onefilewithoptions
\@popfilename

The idea behind the code is to record whether or not the LaTeX3 syntax is on or off when about to load a file with class or package extension. This status stored in the parameter `\ExplSyntaxStatus` and set by `\ExplSyntaxOn` and `\ExplSyntaxOff` to `00` and `01` respectively is puched onto the stack `\ExplSyntaxStack`. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again.

`\@popfilename` is appended with a preamble check. If the catcode of `@` is being reset it is a fair assumption that we are back in the usual preamble and so we switch off our syntax as well.

```
\tex_let:D \org@onefilewithoptions\@onefilewithoptions
\tex_def:D \@onefilewithoptions#1[#2][#3]#4{
  \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus\ExplSyntaxStack }
  \ExplSyntaxOff
  \org@onefilewithoptions{#1}[{#2}][{#3}]{#4}
  \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\tex_relax:D
}
\g@addto@macro\@popfilename{%
  \tex_ifnum:D\tex_the:D\tex_catcode:D`\@=12\tex_relax:D
    \ExplSyntaxOff
  \tex_fi:D
}
```

\ExplSyntaxPopStack
\ExplSyntaxStack

Popping the stack is simple: Take the first two tokens which are either the sequence `00` or `01` and use them in an if test. The stack is initially empty.

```
\tex_def:D\ExplSyntaxPopStack#1#2#3\tex_relax:D{
  \tex_def:D\ExplSyntaxStack{#3}
  \tex_if:D#1#2
    \ExplSyntaxOn
  \tex_else:D
    \ExplSyntaxOff
  \tex_fi:D
}
\tex_def:D\ExplSyntaxStack{}
```

A few of the 'primitives' assigned above have already been stolen by LaTeX, so assign them by hand to the saved real primitive.

```
\tex_let:D\tex_input:D        \@@input
\tex_let:D\tex_underline:D    \@@underline
\tex_let:D\tex_end:D          \@@end
\tex_let:D\tex_everymath:D    \frozen@everymath
```

```
\tex_let:D\tex_everydisplay:D \frozen@everydisplay
\tex_let:D\tex_italiccor:D    \@@italiccorr
\tex_let:D\tex_hyphen:D       \@@hyph
```

TEX has a nasty habit of inserting a command with the name `\par` so we had better make sure that that command at least has a definition.

```
\tex_let:D\par          \tex_par:D

\tex_ifx:D\name_undefine:N\@gobble

  \AtEndOfPackage{\ExplSyntaxOff}
  \tex_def:D\name_pop_stack:w{}

\tex_else:D
```

But if traditional TEX code is disabled, do this...

As mentioned above, The LATEX 2_ε package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the TEX primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```
\tex_def:D\ProvidesPackage{
  \tex_begingroup:D
  \ExplSyntaxOff
  \package_provides:w}

\tex_def:D\package_provides:w#1#2[#3]{
  \tex_endgroup:D
  \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
  \tex_expandafter:D\tex_xdef:D
    \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}
```

In this case the catcode preserving stack is not maintained and `\CodeStart` conventions stay in force once on. You'll need to turn then off explicitly with `\CodeStop` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```
\tex_def:D\name_pop_stack:w#1\relax{%
  \ExplSyntaxOff
  \tex_expandafter:D\@p@pfilename\@currnamestack\@nil
  \tex_let:D\default@ds\@unknownoptionerror
  \tex_global:D\tex_let:D\ds@\@empty
  \tex_global:D\tex_let:D\@declaredoptions\@empty}

\tex_def:D\@p@pfilename#1#2#3#4\@nil{%
  \tex_gdef:D\@currname{#1}%
  \tex_gdef:D\@currext{#2}%
  \tex_catcode:D`\@#3%
  \tex_gdef:D\@currnamestack{#4}}
```

```
\tex_def:D\NeedsTeXFormat#1{}
\tex_def:D\RequirePackage#1{
  \tex_expandafter:D\tex_ifx:D
    \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
      \ExplSyntaxOn
      \tex_input:D#1.sty\tex_relax:D
  \tex_fi:D}
\tex_fi:D
```

The `\futurelet` just forces the special end of file marker to vanish, so the argument of `\name_pop_stack:w` does not cause an end-of-file error. (Normally I use `\expandafter` for this trick, but here the next token is in fact `\let` and that may be undefined.)

```
\tex_futurelet:D\name_tmp:\name_pop_stack:w
⟨/package⟩
```

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

5 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.

5.1 Predicates and conditionals

5.1.1 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 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_catcode:w ⟨*token1*⟩ ⟨*token2*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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

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

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

\if_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.

5.1.2 Non-primitive conditionals

\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.

5.2 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_orelse: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:
\use_arg_i_after_orelse:nw {⟨arg⟩} \or:/\else: ⟨balanced text⟩ \fi:

Executes ⟨*arg*⟩ after executing closing out `\fi:`. `\use_arg_i_after_orelse:nw` can be used anywhere where `\use_arg_i_after_else:nw` or `\use_arg_i_after_or:nw` are used.

5.3 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.

5.4 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.

5.4.1 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.

5.4.2 Undefining functions

\cs_gundefine:N `\cs_gundefine:N` ⟨*cs*⟩

Undefines the control sequence.

5.4.3 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`.

5.5 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.

5.6 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.

5.7 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.

5.8 Engine specific definitions

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

5.9 The Implementation

We start by ensuring that the required packages are loaded. We need l3names to get things going but we actually need it very early on, so it is loaded at the very top of this file. Also, most of the code below won't run until l3expan has been loaded.

5.9.1 Renaming some TEX primitives (again)

\let:NwN Having given all the tex primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.[3]

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨*initex | package⟩
\tex_let:D \let:NwN            \tex_let:D
```

\if_true:
\if_false:
\else:
\fi:
\reverse_if:N
\if:w
\if_charcode:w
\if_catcode:w
Then some conditionals.

```
\let:NwN   \if_true:           \tex_iftrue:D
\let:NwN   \if_false:          \tex_iffalse:D
\let:NwN   \else:              \tex_else:D
\let:NwN   \fi:                \tex_fi:D
\let:NwN   \reverse_if:N       \etex_unless:D
\let:NwN   \if:w               \tex_if:D
\let:NwN   \if_charcode:w     \tex_if:D
\let:NwN   \if_catcode:w      \tex_ifcat:D
```

\if_meaning:NN
\if_token_eq:NN
\if_cs_meaning_eq:NN
Some different names for \ifx.[4]

```
\let:NwN   \if_meaning:NN      \tex_ifx:D
\let:NwN   \if_token_eq:NN     \tex_ifx:D
\let:NwN   \if_cs_meaning_eq:NN\tex_ifx:D
```

\if_mode_math:
\if_mode_horizontal:
\if_mode_vertical:
\if_mode_inner:
TEX lets us detect some if its modes.

```
\let:NwN   \if_mode_math:      \tex_ifmmode:D
\let:NwN   \if_mode_horizontal:\tex_ifhmode:D
\let:NwN   \if_mode_vertical:  \tex_ifvmode:D
\let:NwN   \if_mode_inner:     \tex_ifinner:D
```

[3] This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the "tex...D" name in the cases where no good alternative exists.

[4] MH: Clean up at some point

\if_cs_exist:N
\if_cs_exist:w

```
\let:NwN   \if_cs_exist:N      \etex_ifdefined:D
\let:NwN   \if_cs_exist:w      \etex_ifcsname:D
```

\exp_after:NN
\exp_not:N
\exp_not:n

The three \exp_ functions are used in the l3expan module where they are described.

```
\let:NwN   \exp_after:NN       \tex_expandafter:D
\let:NwN   \exp_not:N          \tex_noexpand:D
\let:NwN   \exp_not:n          \etex_unexpanded:D
```

\io_put_deferred:Nx
\token_to_meaning:N
\token_to_string:N
\cs:w
\cs_end:
\cs_meaning:N
\cs_meaning:c
\cs_show:N
\cs_show:c

```
\let:NwN   \io_put_deferred:Nx \tex_write:D
\let:NwN   \token_to_meaning:N \tex_meaning:D
\let:NwN   \token_to_string:N  \tex_string:D
\let:NwN   \cs:w               \tex_csname:D
\let:NwN   \cs_end:            \tex_endcsname:D
\let:NwN   \cs_meaning:N       \tex_meaning:D
\tex_def:D \cs_meaning:c #1{\exp_after:NN\cs_meaning:N\cs:w #1\cs_end:}
\let:NwN   \cs_show:N          \tex_show:D
\tex_def:D \cs_show:c #1{\exp_after:NN\cs_show:N\cs:w #1\cs_end:}
```

\scan_stop:
\group_begin:
\group_end:

The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the l3prg module.

```
\let:NwN   \scan_stop:         \tex_relax:D
\let:NwN   \group_begin:       \tex_begingroup:D
\let:NwN   \group_end:         \tex_endgroup:D
```

\group_execute_after:N

```
\let:NwN \group_execute_after:N \tex_aftergroup:D
```

\the_internal:D
\pref_global:D
\pref_long:D
\pref_protected:D

These following names are temporary and should be removed as soon as possible (April 1998).

```
\let:NwN   \the_internal:D     \tex_the:D
\let:NwN   \pref_global:D      \tex_global:D
```

\pref_long:D has been documented for years but didn't exist... Added it and the robustness prefix.

```
\let:NwN   \pref_long:D        \tex_long:D
\let:NwN   \pref_protected:D   \etex_protected:D
```

5.9.2 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

\def:Npn
\def:Npx
\def_long:Npn
\def_long:Npx
\def_protected:Npn
\def_protected:Npx
\def_protected_long:Npn
\def_protected_long:Npx

All assignment functions in LaTeX3 should be naturally robust; after all, the TeX primitives for assignments are and it can be a cause of problems if others aren't.

```
\let:NwN   \def:Npn            \tex_def:D
\let:NwN   \def:Npx            \tex_edef:D
\pref_protected:D \def:Npn \def_long:Npn {\pref_long:D \def:Npn}
\pref_protected:D \def:Npn \def_long:Npx {\pref_long:D \def:Npx}
\pref_protected:D \def:Npn \def_protected:Npn {\pref_protected:D \def:Npn}
\pref_protected:D \def:Npn \def_protected:Npx {\pref_protected:D \def:Npx}
\def_protected:Npn \def_protected_long:Npn {
  \pref_protected:D \pref_long:D \def:Npn
}
\def_protected:Npn \def_protected_long:Npx {
  \pref_protected:D \pref_long:D \def:Npx
}
```

\gdef:Npn
\gdef:Npx
\gdef_long:Npn
\gdef_long:Npx
\gdef_protected:Npn
\gdef_protected:Npx
\gdef_protected_long:Npn
\gdef_protected_long:Npx

Global versions of the above functions.

```
\let:NwN   \gdef:Npn           \tex_gdef:D
\let:NwN   \gdef:Npx           \tex_xdef:D
\def_protected:Npn \gdef_long:Npn {\pref_long:D \gdef:Npn}
\def_protected:Npn \gdef_long:Npx {\pref_long:D \gdef:Npx}
\def_protected:Npn \gdef_protected:Npn {\pref_protected:D \gdef:Npn}
\def_protected:Npn \gdef_protected:Npx {\pref_protected:D \gdef:Npx}
\def_protected:Npn \gdef_protected_long:Npn {
  \pref_protected:D \pref_long:D \gdef:Npn
}
\def_protected:Npn \gdef_protected_long:Npx {
  \pref_protected:D \pref_long:D \gdef:Npx
}
```

5.9.3 Predicate implementation

I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either `TT` or `TF` to be tested using `\if:w` but this was later changed to `00` and `01`, so they can be used in logical operations (see the l3prg module). We need this from the get-go.

\c_true
\c_false

Here are the canonical boolean values.

```
\def:Npn \c_true  {00}
\def:Npn \c_false {01}
```

5.9.4 Defining and checking (new) functions

\c_minus_one
\c_sixteen

We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal but the rest are defined in the l3num module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. Otherwise the l3int module is required but it can't be used until the allocation has been set up properly! The actual allocation mechanism is in l3alloc and as TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```
⟨*!initex⟩
\let:NwN \c_minus_one\m@ne
⟨/!initex⟩
⟨*!package⟩
\tex_countdef:D \c_minus_one = 10 \scan_stop:
\c_minus_one = -1 \scan_stop:
⟨/!package⟩
\tex_chardef:D \c_sixteen = 16\scan_stop:
```

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The definitions here are only temporary, they will be redefined later on.

`\io_put_log:x` `\io_put_term:x` We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal.

```
\def:Npn \io_put_log:x{
      \tex_immediate:D\io_put_deferred:Nx \c_minus_one }
\def:Npn \io_put_term:x{
      \tex_immediate:D\io_put_deferred:Nx \c_sixteen }
```

`\err_latex_bug:x` This will show internal errors.

```
\def:Npn\err_latex_bug:x#1{
   \io_put_term:x{This~is~a~LaTeX~bug!~Check~coding!}\tex_errmessage:D{#1}}
```

`\cs_record_meaning:N` This macro will be used later on for tracing purposes. But we need some more modules to define it, so we just give some dummy definition here.

```
⟨*trace⟩
\def:Npn \cs_record_meaning:N#1{}
⟨/trace⟩
```

We need these two to make `\chk_new_cs:N` bulletproof.

```
\def_long:Npn \use_none:n  #1{}
\def_long:Npn \use_arg_i:n #1{#1}
```

`\chk_new_cs:N` This command is called by `\def_new:Npn` and `\let_new:NN` etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if ⟨*csname*⟩ is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```
\def:Npn \chk_new_cs:N #1{
  \if:w \cs_if_free_p:N #1
    \exp_after:NN \use_none:n
  \else:
```

```
    \exp_after:NN \use_arg_i:n
  \fi:
  {
    \err_latex_bug:x {Command~name~`\token_to_string:N #1'~
                      already~defined!~
                      Current~meaning:~\token_to_meaning:N #1
                     }
  }
⟨*trace⟩
  \cs_record_meaning:N#1
%     \io_put_term:x{Defining~\token_to_string:N #1~on~%}
  \io_put_log:x{Defining~\token_to_string:N #1~on~
                line~\tex_the:D \tex_inputlineno:D}
⟨/trace⟩
}
```

On 2005/11/20 Morten said: I think names for testing if a certain condition is true or false should always contain `if` to avoid confusion. This is what we do for lots of other types of test functions. For now I have defined both names for the functions checking names of control sequences.

`\cs_if_exist_p:N` Expands into `\c_true` if the control sequence given as its argument *is* in use.

```
\def:Npn \cs_if_exist_p:N #1{
  \if:w \cs_if_free_p:N #1
    \c_false
  \else:
    \c_true \fi:}
```

`\chk_if_exist_cs:N` `\chk_if_exist_cs:c` This function issues a warning message when the control sequence in its argument does not exist.

```
\def:Npn \chk_if_exist_cs:N #1 {
  \if:w \cs_if_exist_p:N #1
  \else:
    \err_latex_bug:x{Command~ `\token_to_string:N #1'~
                     not~ yet~ defined!}
  \fi:}
\def:Npn \chk_if_exist_cs:c #1 {
  \exp_after:NN \chk_if_exist_cs:N \cs:w #1\cs_end: }
```

`\cs_if_free_p:N` Expands into `\c_true` if the control sequence given as its argument is not yet in use. Note that we make sure to expand into `\c_false` if the control sequence is textually `\c_undefined` or `\scan_stop:`, so that we don't end up (re)defining them.

```
\def:Npn \cs_if_free_p:N #1{
  \if_cs_exist:N #1
    \if_meaning:NN#1\scan_stop:
      \if:w\cs_if_eq_name_p:NN #1\scan_stop:
        \c_false \else: \c_true \fi:
    \else:
       \c_false
    \fi:
```

```
  \else:
    \if:w \cs_if_eq_name_p:NN #1\c_undefined
      \c_false \else: \c_true \fi:
  \fi:
}
\let:NwN \cs_free_p:N \cs_if_free_p:N
```

`\str_if_eq_p:nn` `\str_if_eq_p_aux:w`

Takes 2 lists of characters as arguments and expands into `\c_true` if they are equal, and `\c_false` otherwise. Note that in the current implementation spaces in these strings are ignored.[5]

```
\def:Npn \str_if_eq_p:nn #1#2{
  \str_if_eq_p_aux:w #1\scan_stop:\\#2\scan_stop:\\
}
\def:Npn \str_if_eq_p_aux:w #1#2\\#3#4\\{
  \if_meaning:NN#1#3
    \if_meaning:NN#1\scan_stop:\c_true \else:
    \if_meaning:NN#3\scan_stop:\c_false \else:
    \str_if_eq_p_aux:w #2\\#4\\\fi:\fi:
  \else:\c_false \fi:}
```

`\cs_if_eq_name_p:NN`

An application of the above function, already streamlined for speed, so I put it in here. It takes two control sequences as arguments and expands into true iff they have the same name. We make it long in case one of them is `\par`!

```
\def_long:Npn \cs_if_eq_name_p:NN #1#2{
  \exp_after:NN\exp_after:NN
  \exp_after:NN\str_if_eq_p_aux:w
  \exp_after:NN\token_to_string:N
  \exp_after:NN#1
  \exp_after:NN\scan_stop:
  \exp_after:NN\\
  \token_to_string:N#2\scan_stop:\\}
```

`\str_if_eq_var_p:nf` `\str_if_eq_var_start:nnN` `\str_if_eq_var_stop:w`

A variant of `\str_if_eq_p:nn` which has the advantage of obeying spaces in at least the second argument. See l3quark for an application. From the hand of David Kastrup with slight modifications to make it fit with the remainder of the expl3 language.

The macro builds a string of `\if:w \fi:` pairs from the first argument. The idea is to turn the comparison of `ab` and `cde` into

```
\tex_number:D
  \if:w \scan_stop: \if:w b\if:w a cde\scan_stop: '\fi: \fi: \fi:
 13
```

The `'` is important here. If all tests are true, the `'` is read as part of the number in which case the returned number is `13` in octal notation so `\tex_number:D` returns `11`. If one test returns false the `'` is never seen and then we get just `13`. We wrap the whole process in an external `\if:w` in order to make it return either `\c_true` or `\c_false` since some parts of l3prg expect a predicate to return one of these two tokens.

[5] This is a function which could use `\tlist_compare:xx`.

```
\def:Npn \str_if_eq_var_p:nf#1{
  \if:w \tex_number:D\str_if_eq_var_start:nnN{}{}#1\scan_stop:
}
\def:Npn\str_if_eq_var_start:nnN#1#2#3{
  \if:w#3\scan_stop:\exp_after:NN\str_if_eq_var_stop:w\fi:
  \str_if_eq_var_start:nnN{\if:w#3#1}{#2\fi:}
}
\def:Npn\str_if_eq_var_stop:w\str_if_eq_var_start:nnN#1#2#3{
  #1#3\scan_stop:'#213~\c_true\else:\c_false\fi:
}
```

5.9.5 More new definitions

\def_new:Npn
\def_new:Npx
\def_long_new:Npn
\def_long_new:Npx
\def_protected_new:Npn
\def_protected_new:Npx
\def_protected_long_new:Npn
\def_protected_long_new:Npx

These are like \def:Npn and \let:NN, but they first check that the argument command is not already in use. You may use \pref_global:D, \pref_long:D, \pref_protected:D, and \tex_outer:D as prefixes.

```
\def_protected:Npn \def_new:Npn #1{\chk_new_cs:N #1
                        \def:Npn #1}
\def_protected:Npn \def_new:Npx #1{\chk_new_cs:N #1
                        \def:Npx #1}
\def_protected:Npn \def_long_new:Npn #1{\chk_new_cs:N #1
                                  \def_long:Npn #1}
\def_protected:Npn \def_long_new:Npx #1{\chk_new_cs:N #1
                                  \def_long:Npx #1}
\def_protected:Npn \def_protected_new:Npn #1{\chk_new_cs:N #1
                                  \def_protected:Npn #1}
\def_protected:Npn \def_protected_new:Npx #1{\chk_new_cs:N #1
                                  \def_protected:Npx #1}
\def_protected:Npn \def_protected_long_new:Npn #1{\chk_new_cs:N #1
                                  \def_protected_long:Npn #1}
\def_protected:Npn \def_protected_long_new:Npx #1{\chk_new_cs:N #1
                                  \def_protected_long:Npx #1}
```

\gdef_new:Npn
\gdef_new:Npx
\gdef_long_new:Npn
\gdef_long_new:Npx
\gdef_protected_new:Npn
\gdef_protected_new:Npx
\gdef_protected_long_new:Npn
\gdef_protected_long_new:Npx

Global versions of the above functions.

```
\def_protected_new:Npn \gdef_new:Npn #1{\chk_new_cs:N #1
                          \gdef:Npn #1}
\def_protected_new:Npn \gdef_new:Npx #1{\chk_new_cs:N #1
                          \gdef:Npx #1}
\def_protected_new:Npn \gdef_long_new:Npn #1{\chk_new_cs:N #1
                                  \gdef_long:Npn #1}
\def_protected_new:Npn \gdef_long_new:Npx #1{\chk_new_cs:N #1
                                  \gdef_long:Npx #1}
\def_protected_new:Npn \gdef_protected_new:Npn #1{\chk_new_cs:N #1
                                  \gdef_protected:Npn #1}
\def_protected_new:Npn \gdef_protected_new:Npx #1{\chk_new_cs:N #1
                                  \gdef_protected:Npx #1}
\def_protected_new:Npn \gdef_protected_long_new:Npn #1{\chk_new_cs:N #1
                                  \gdef_protected_long:Npn #1}
\def_protected_new:Npn \gdef_protected_long_new:Npx #1{\chk_new_cs:N #1
                                  \gdef_protected_long:Npx #1}
```

\def:cpn
\def:cpx
\gdef:cpn
\gdef:cpx
\def_new:cpn
\def_new:cpx
\gdef_new:cpn
\gdef_new:cpx

Like \def:Npn and \def_new:Npn, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module.). Global versions are also provided.

\def:cpn⟨*string*⟩⟨*rep-text*⟩ will turn ⟨*string*⟩ into a csname and then assign ⟨*rep-text*⟩ to it by using \def:Npn. This means that there might be a parameter string between the two arguments.

```
\def_new:Npn \def:cpn #1{\exp_after:NN \def:Npn \cs:w #1\cs_end:}
\def_new:Npn \def:cpx #1{\exp_after:NN \def:Npx \cs:w #1\cs_end:}
\def_new:Npn \gdef:cpn #1{\exp_after:NN \gdef:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef:cpx #1{\exp_after:NN \gdef:Npx \cs:w #1\cs_end:}
\def_new:Npn \def_new:cpn #1{\exp_after:NN \def_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_new:cpx #1{\exp_after:NN \def_new:Npx \cs:w #1\cs_end:}
\def_new:Npn \gdef_new:cpn #1{\exp_after:NN \gdef_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_new:cpx #1{\exp_after:NN \gdef_new:Npx \cs:w #1\cs_end:}
```

\def_long:cpn
\def_long:cpx
\gdef_long:cpn
\gdef_long:cpx
\def_long_new:cpn
\def_long_new:cpx
\gdef_long_new:cpn
\gdef_long_new:cpx

Variants of the \def_long:Npn versions which make a csname out of the first arguments. We may also do this globally.

```
\def_new:Npn \def_long:cpn #1{\exp_after:NN \def_long:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_long:cpx #1{
  \exp_after:NN\def_long:Npx\cs:w #1\cs_end:}
\def_new:Npn \gdef_long:cpn #1{
  \exp_after:NN \gdef_long:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_long:cpx #1{
  \exp_after:NN\gdef_long:Npx\cs:w #1\cs_end:}
\def_new:Npn \def_long_new:cpn #1{
  \exp_after:NN \def_long_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_long_new:cpx #1{
  \exp_after:NN \def_long_new:Npx \cs:w #1\cs_end:}
\def_new:Npn \gdef_long_new:cpn #1{
  \exp_after:NN \gdef_long_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_long_new:cpx #1{
  \exp_after:NN \gdef_long_new:Npx \cs:w #1\cs_end:}
```

\def_protected:cpn
\def_protected:cpx
\gdef_protected:cpn
\gdef_protected:cpx
\def_protected_new:cpn
\def_protected_new:cpx
\gdef_protected_new:cpn
\gdef_protected_new:cpx

Variants of the \def_protected:Npn versions which make a csname out of the first arguments. We may also do this globally.

```
\def_new:Npn \def_protected:cpn #1{
  \exp_after:NN \def_protected:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_protected:cpx #1{
  \exp_after:NN\def_protected:Npx\cs:w #1\cs_end:}
\def_new:Npn \gdef_protected:cpn #1{
  \exp_after:NN \gdef_protected:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_protected:cpx #1{
  \exp_after:NN\gdef_protected:Npx\cs:w #1\cs_end:}
\def_new:Npn \def_protected_new:cpn #1{
  \exp_after:NN \def_protected_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_protected_new:cpx #1{
  \exp_after:NN \def_protected_new:Npx \cs:w #1\cs_end:}
\def_new:Npn \gdef_protected_new:cpn #1{
  \exp_after:NN \gdef_protected_new:Npn \cs:w #1\cs_end:}
```

```
\def_new:Npn \gdef_protected_new:cpx #1{
  \exp_after:NN \gdef_protected_new:Npx \cs:w #1\cs_end:}
```

\def_protected_long:cpn
\def_protected_long:cpx
\gdef_protected_long:cpn
\gdef_protected_long:cpx
\def_protected_long_new:cpn
\def_protected_long_new:cpx
\gdef_protected_long_new:cpn
\gdef_protected_long_new:cpx

Variants of the \def_protected_long:Npn versions which make a csname out of the first arguments. We may also do this globally.

```
\def_new:Npn \def_protected_long:cpn #1{
  \exp_after:NN \def_protected_long:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_protected_long:cpx #1{
  \exp_after:NN\def_protected_long:Npx\cs:w #1\cs_end:}
\def_new:Npn \gdef_protected_long:cpn #1{
  \exp_after:NN \gdef_protected_long:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_protected_long:cpx #1{
  \exp_after:NN\gdef_protected_long:Npx\cs:w #1\cs_end:}
\def_new:Npn \def_protected_long_new:cpn #1{
  \exp_after:NN \def_protected_long_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \def_protected_long_new:cpx #1{
  \exp_after:NN \def_protected_long_new:Npx \cs:w #1\cs_end:}
\def_new:Npn \gdef_protected_long_new:cpn #1{
  \exp_after:NN \gdef_protected_long_new:Npn \cs:w #1\cs_end:}
\def_new:Npn \gdef_protected_long_new:cpx #1{
  \exp_after:NN \gdef_protected_long_new:Npx \cs:w #1\cs_end:}
```

\def_aux_0:NNn
\def_aux_1:NNn
\def_aux_2:NNn
\def_aux_3:NNn
\def_aux_4:NNn
\def_aux_5:NNn
\def_aux_6:NNn
\def_aux_7:NNn
\def_aux_8:NNn
\def_aux_9:NNn

Defining a function with n arguments. First some helper functions.

```
\def_new:cpn {def_aux_0:NNn} #1#2 {#1 #2 }
\def_new:cpn {def_aux_1:NNn} #1#2 {#1 #2 ##1 }
\def_new:cpn {def_aux_2:NNn} #1#2 {#1 #2 ##1##2 }
\def_new:cpn {def_aux_3:NNn} #1#2 {#1 #2 ##1##2##3 }
\def_new:cpn {def_aux_4:NNn} #1#2 {#1 #2 ##1##2##3##4 }
\def_new:cpn {def_aux_5:NNn} #1#2 {#1 #2 ##1##2##3##4##5 }
\def_new:cpn {def_aux_6:NNn} #1#2 {#1 #2 ##1##2##3##4##5##6 }
\def_new:cpn {def_aux_7:NNn} #1#2 {#1 #2 ##1##2##3##4##5##6##7 }
\def_new:cpn {def_aux_8:NNn} #1#2 {#1 #2 ##1##2##3##4##5##6##7##8 }
\def_new:cpn {def_aux_9:NNn} #1#2 {#1 #2 ##1##2##3##4##5##6##7##8##9 }
```

\def_aux:NNnn
\def_aux:Ncnn
\def_arg_number_error_msg:Nn

Then the function itself which checks for the existance of such a helper function. If it doesn't exist, return an error. Otherwise call it to define #2 with the correct number of arguments.

```
\def_protected_long_new:Npn \def_aux:NNnn #1#2#3#4 {
  \cs_if_really_exist:cTF {def_aux_\tex_the:D\etex_numexpr:D #3 :NNn}
  {
    \cs_use:c {def_aux_\tex_the:D\etex_numexpr:D #3 :NNn} #1 #2 {#4}
  }
  { \def_arg_number_error_msg:Nn #2{#3} }
}
\def_new:Npn \def_aux:Ncnn #1#2{
  \exp_after:NN \def_aux:NNnn \exp_after:NN #1 \cs:w #2\cs_end:}
```

The error message.

```
\def_new:Npn \def_arg_number_error_msg:Nn #1#2 {
  \err_latex_bug:x{
```

```
    You're~ trying~ to~ define~ the~ command~ `\token_to_string:N #1'~
    with~ \use_arg_i:n{\tex_the:D\etex_numexpr:D #2\scan_stop:} ~
    arguments~ but~ I~ only~ allow~ 0-9~ arguments.~ I~ can~ probably~
    not~ help~ you~ here
  }
}
```

`\def_aux_use_0_parameter:` `\def_aux_use_1_parameter:` `\def_aux_use_2_parameter:` `\def_aux_use_3_parameter:` `\def_aux_use_4_parameter:` `\def_aux_use_5_parameter:` `\def_aux_use_6_parameter:` `\def_aux_use_7_parameter:` `\def_aux_use_8_parameter:` `\def_aux_use_9_parameter:`

Something similar to `\def_aux_9:NNn` but for using the parameters.

```
\def:cpn{def_aux_use_0_parameter:}{}
\def:cpn{def_aux_use_1_parameter:}{{##1}}
\def:cpn{def_aux_use_2_parameter:}{{##1}{##2}}
\def:cpn{def_aux_use_3_parameter:}{{##1}{##2}{##3}}
\def:cpn{def_aux_use_4_parameter:}{{##1}{##2}{##3}{##4}}
\def:cpn{def_aux_use_5_parameter:}{{##1}{##2}{##3}{##4}{##5}}
\def:cpn{def_aux_use_6_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}}
\def:cpn{def_aux_use_7_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}{##7}}
\def:cpn{def_aux_use_8_parameter:}{
  {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}}
\def:cpn{def_aux_use_9_parameter:}{
  {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}{##9}}
```

`\def:NNn` `\def:NNx` `\def:cNn` `\def:cNx` `\gdef:NNn` `\gdef:NNx` `\gdef:cNn` `\gdef:cNx` `\def_new:NNn` `\def_new:NNx` `\def_new:cNn` `\def_new:cNx` `\gdef_new:NNn` `\gdef_new:NNx` `\gdef_new:cNn` `\gdef_new:cNx`

Defining macros without delimited arguments is now relatively easy. First local and global versions of the usual `\def:Npn` operation.

```
\def_new:Npn \def:NNn { \def_aux:NNnn \def:Npn }
\def_new:Npn \def:NNx { \def_aux:NNnn \def:Npx }
\def_new:Npn \def:cNn { \def_aux:Ncnn \def:Npn }
\def_new:Npn \def:cNx { \def_aux:Ncnn \def:Npx }
\def_new:Npn \gdef:NNn { \def_aux:NNnn \gdef:Npn }
\def_new:Npn \gdef:NNx { \def_aux:NNnn \gdef:Npx }
\def_new:Npn \gdef:cNn { \def_aux:Ncnn \gdef:Npn }
\def_new:Npn \gdef:cNx { \def_aux:Ncnn \gdef:Npx }
\def_new:Npn \def_new:NNn { \def_aux:NNnn \def_new:Npn }
\def_new:Npn \def_new:NNx { \def_aux:NNnn \def_new:Npx }
\def_new:Npn \def_new:cNn { \def_aux:Ncnn \def_new:Npn }
\def_new:Npn \def_new:cNx { \def_aux:Ncnn \def_new:Npx }
\def_new:Npn \gdef_new:NNn { \def_aux:NNnn \gdef_new:Npn }
\def_new:Npn \gdef_new:NNx { \def_aux:NNnn \gdef_new:Npx }
\def_new:Npn \gdef_new:cNn { \def_aux:Ncnn \gdef_new:Npn }
\def_new:Npn \gdef_new:cNx { \def_aux:Ncnn \gdef_new:Npx }
```

`\def_long:NNn` `\def_long:NNx` `\def_long:cNn` `\def_long:cNx` `\gdef_long:NNn` `\gdef_long:NNx` `\gdef_long:cNn` `\gdef_long:cNx` `\def_long_new:NNn` `\def_long_new:NNx` `\def_long_new:cNn` `\def_long_new:cNx` `\gdef_long_new:NNn` `\gdef_long_new:NNx` `\gdef_long_new:cNn` `\gdef_long_new:cNx`

Long versions of the above.

```
\def_new:Npn \def_long:NNn { \def_aux:NNnn \def_long:Npn }
\def_new:Npn \def_long:NNx { \def_aux:NNnn \def_long:Npx }
\def_new:Npn \def_long:cNn { \def_aux:Ncnn \def_long:Npn }
\def_new:Npn \def_long:cNx { \def_aux:Ncnn \def_long:Npx }
\def_new:Npn \gdef_long:NNn { \def_aux:NNnn \gdef_long:Npn }
\def_new:Npn \gdef_long:NNx { \def_aux:NNnn \gdef_long:Npx }
\def_new:Npn \gdef_long:cNn { \def_aux:Ncnn \gdef_long:Npn }
\def_new:Npn \gdef_long:cNx { \def_aux:Ncnn \gdef_long:Npx }
\def_new:Npn \def_long_new:NNn { \def_aux:NNnn \def_long_new:Npn }
\def_new:Npn \def_long_new:NNx { \def_aux:NNnn \def_long_new:Npx }
```

```
\def_new:Npn \def_long_new:cNn { \def_aux:Ncnn \def_long_new:Npn }
\def_new:Npn \def_long_new:cNx { \def_aux:Ncnn \def_long_new:Npx }
\def_new:Npn \gdef_long_new:NNn { \def_aux:NNnn \gdef_long_new:Npn }
\def_new:Npn \gdef_long_new:NNx { \def_aux:NNnn \gdef_long_new:Npx }
\def_new:Npn \gdef_long_new:cNn { \def_aux:Ncnn \gdef_long_new:Npn }
\def_new:Npn \gdef_long_new:cNx { \def_aux:Ncnn \gdef_long_new:Npx }
```

\def_protected:NNn
\def_protected:NNx
\def_protected:cNn
\def_protected:cNx
\gdef_protected:NNn
\gdef_protected:NNx
\gdef_protected:cNn
\gdef_protected:cNx
\def_protected_new:NNn
\def_protected_new:NNx
\def_protected_new:cNn
\def_protected_new:cNx
\gdef_protected_new:NNn
\gdef_protected_new:NNx
\gdef_protected_new:cNn
\gdef_protected_new:cNx

Protected versions of the above.

```
\def_new:Npn \def_protected:NNn { \def_aux:NNnn \def_protected:Npn }
\def_new:Npn \def_protected:NNx { \def_aux:NNnn \def_protected:Npx }
\def_new:Npn \def_protected:cNn { \def_aux:Ncnn \def_protected:Npn }
\def_new:Npn \def_protected:cNx { \def_aux:Ncnn \def_protected:Npx }
\def_new:Npn \gdef_protected:NNn { \def_aux:NNnn \gdef_protected:Npn }
\def_new:Npn \gdef_protected:NNx { \def_aux:NNnn \gdef_protected:Npx }
\def_new:Npn \gdef_protected:cNn { \def_aux:Ncnn \gdef_protected:Npn }
\def_new:Npn \gdef_protected:cNx { \def_aux:Ncnn \gdef_protected:Npx }
\def_new:Npn \def_protected_new:NNn { \def_aux:NNnn \def_protected_new:Npn }
\def_new:Npn \def_protected_new:NNx { \def_aux:NNnn \def_protected_new:Npx }
\def_new:Npn \def_protected_new:cNn { \def_aux:Ncnn \def_protected_new:Npn }
\def_new:Npn \def_protected_new:cNx { \def_aux:Ncnn \def_protected_new:Npx }
\def_new:Npn \gdef_protected_new:NNn { \def_aux:NNnn \gdef_protected_new:Npn }
\def_new:Npn \gdef_protected_new:NNx { \def_aux:NNnn \gdef_protected_new:Npx }
\def_new:Npn \gdef_protected_new:cNn { \def_aux:Ncnn \gdef_protected_new:Npn }
\def_new:Npn \gdef_protected_new:cNx { \def_aux:Ncnn \gdef_protected_new:Npx }
```

\def_protected_long:NNn
\def_protected_long:NNx
\def_protected_long:cNn
\def_protected_long:cNx
\gdef_protected_long:NNn
\gdef_protected_long:NNx
\gdef_protected_long:cNn
\gdef_protected_long:cNx
\def_protected_long_new:NNn
\def_protected_long_new:NNx
\def_protected_long_new:cNn
\def_protected_long_new:cNx
\gdef_protected_long_new:NNn
\gdef_protected_long_new:NNx
\gdef_protected_long_new:cNn
\gdef_protected_long_new:cNx

And finally both long and protected.

```
\def_new:Npn \def_protected_long:NNn { \def_aux:NNnn \def_protected_long:Npn }
\def_new:Npn \def_protected_long:NNx { \def_aux:NNnn \def_protected_long:Npx }
\def_new:Npn \def_protected_long:cNn { \def_aux:Ncnn \def_protected_long:Npn }
\def_new:Npn \def_protected_long:cNx { \def_aux:Ncnn \def_protected_long:Npx }
\def_new:Npn \gdef_protected_long:NNn { \def_aux:NNnn \gdef_protected_long:Npn }
\def_new:Npn \gdef_protected_long:NNx { \def_aux:NNnn \gdef_protected_long:Npx }
\def_new:Npn \gdef_protected_long:cNn { \def_aux:Ncnn \gdef_protected_long:Npn }
\def_new:Npn \gdef_protected_long:cNx { \def_aux:Ncnn \gdef_protected_long:Npx }
\def_new:Npn \def_protected_long_new:NNn {
  \def_aux:NNnn \def_protected_long_new:Npn }
\def_new:Npn \def_protected_long_new:NNx {
  \def_aux:NNnn \def_protected_long_new:Npx }
\def_new:Npn \def_protected_long_new:cNn {
  \def_aux:Ncnn \def_protected_long_new:Npn }
\def_new:Npn \def_protected_long_new:cNx {
  \def_aux:Ncnn \def_protected_long_new:Npx }
\def_new:Npn \gdef_protected_long_new:NNn {
  \def_aux:NNnn \gdef_protected_long_new:Npn }
\def_new:Npn \gdef_protected_long_new:NNx {
  \def_aux:NNnn \gdef_protected_long_new:Npx }
\def_new:Npn \gdef_protected_long_new:cNn {
  \def_aux:Ncnn \gdef_protected_long_new:Npn }
\def_new:Npn \gdef_protected_long_new:cNx {
  \def_aux:Ncnn \gdef_protected_long_new:Npx }
```

\let:NN
\let:cN
\let:Nc
\let:cc

These macros allow us to copy the definition of a control sequence to another control sequence.

```
\def_protected_long_new:Npn \let:NN #1{
```

\let_new:NN
\let_new:cN
\let_new:Nc
\let_new:cc

The = sign allows us to define funny char tokens like .= itself or ␣ with this function. For the definition of \c_space_chartok{~} to work we need the ~ after the =

```
                                \let:NwN #1=~}
\def_new:Npn\let:cN #1 {\exp_after:NN\let:NN\cs:w#1\cs_end:}
\def_new:Npn\let:Nc{\exp_args:NNc\let:NN}
\def_new:Npn\let:cc{\exp_args:Ncc\let:NN}
\def_new:Npn \let_new:NN #1{\chk_new_cs:N #1
                                \let:NN #1}
\def_new:Npn \let_new:cN {\exp_args:Nc \let_new:NN}
\def_new:Npn \let_new:Nc {\exp_args:NNc \let_new:NN}
\def_new:Npn \let_new:cc {\exp_args:Ncc \let_new:NN}
```

\glet:NN
\glet:cN
\glet:Nc
\glet:cc

These are global versions of some of the previously defined functions.

\glet_new:NN
\glet_new:cN
\glet_new:Nc
\glet_new:cc

```
\def_protected_new:Npn \glet:NN {\pref_global:D \let:NN}
\def_protected_new:Npn \glet:Nc {\exp_args:NNc \glet:NN}
\def_protected_new:Npn \glet:cN {\exp_args:Nc \glet:NN}
\def_new:Npn \glet:cc {\exp_args:Ncc \glet:NN}
\def_new:Npn \glet_new:NN #1{\chk_new_cs:N #1
                                \tex_global:D\let:NN #1}
\def_new:Npn \glet_new:cN {\exp_args:Nc \glet_new:NN}
\def_new:Npn \glet_new:Nc {\exp_args:NNc \glet_new:NN}
\def_new:Npn \glet_new:cc {\exp_args:Ncc \glet_new:NN}
```

\def:No
\gdef:No

\def:No expands its second argument one time before making the definition.

```
\def_new:Npn \def:No{\exp_args:NNo\def:Npn}
\def_new:Npn \gdef:No  {\exp_args:NNo\gdef:Npn}
```

\def_test_function_aux:Nnnn
\def_test_function_aux:Nnnx
\def_test_function:npn
\def_test_function:npx
\def_long_test_function:npn
\def_long_test_function:npx
\def_test_function_new:npn
\def_test_function_new:npx
\def_long_test_function_new:npn
\def_long_test_function_new:npx

We will often be defining several almost identical TF, T and F type functions so it makes sense for us to define a small function that will do this for us so that we are less likely to introduce typos (it does tend to happen). By doing it in two steps as below we can still retain a simple interface where you write the TEX parameters as usual. Just don't do it when you're already within a conditional!

I think the ways of exiting conditionals below are as fast as they get. Using \reverse_if:N instead of \else: didn't give any difference I could measure.

```
\def_long_new:Npn \def_test_function_aux:Nnnn #1#2#3#4{
  #1 {#2TF} #3 {#4
    \exp_after:NN\use_arg_i:nn\else:\exp_after:NN\use_arg_ii:nn\fi:}
  #1 {#2FT} #3 {#4
    \exp_after:NN\use_arg_ii:nn\else:\exp_after:NN\use_arg_i:nn\fi:}
  #1 {#2T} #3 {#4
    \else:\exp_after:NN\use_none:nn\fi:\use_arg_i:n}
  #1 {#2F} #3 {#4
    \exp_after:NN\use_none:nn\fi:\use_arg_i:n}}
\def_long_new:Npn \def_test_function_aux:Nnnx #1#2#3#4{
```

```
  #1 {#2TF} #3 {#4
    \exp_not:n{\exp_after:NN\use_arg_i:nn\else:\exp_after:NN\use_arg_ii:nn\fi:}}
  #1 {#2FT} #3 {#4
    \exp_not:n{\exp_after:NN\use_arg_ii:nn\else:\exp_after:NN\use_arg_i:nn\fi:}}
  #1 {#2T} #3 {#4
    \exp_not:n{\else:\exp_after:NN\use_none:nn\fi:\use_arg_i:n}}
  #1 {#2F} #3 {#4
    \exp_not:n{\exp_after:NN\use_none:nn\fi:\use_arg_i:n}}}
\def_long_new:Npn \def_test_function:npn #1#2#{
  \def_test_function_aux:Nnnn \def:cpn {#1}{#2}
}
\def_long_new:Npn \def_test_function:npx #1#2#{
  \def_test_function_aux:Nnnx \def:cpx {#1}{#2}
}
\def_long_new:Npn \def_long_test_function:npn #1#2#{
  \def_test_function_aux:Nnnn \def_long:cpn {#1}{#2}
}
\def_long_new:Npn \def_long_test_function:npx #1#2#{
  \def_test_function_aux:Nnnx \def_long:cpx {#1}{#2}
}
\def_long_new:Npn \def_test_function_new:npn #1#2#{
  \def_test_function_aux:Nnnn \def_new:cpn {#1}{#2}
}
\def_long_new:Npn \def_long_test_function_new:npn #1#2#{
  \def_test_function_aux:Nnnn \def_long_new:cpn {#1}{#2}
}
\def_long_new:Npn \def_test_function_new:npx #1#2#{
  \def_test_function_aux:Nnnx \def_new:cpx {#1}{#2}
}
\def_long_new:Npn \def_long_test_function_new:npx #1#2#{
  \def_test_function_aux:Nnnx \def_long_new:cpx {#1}{#2}
}
```

5.9.6 Further checking

`\cs_if_free:NTF` `\cs_if_free:NT` `\cs_if_free:NF`

The old `\@ifundefined` of LaTeX 2.09 is re-implemented in the function `\cs_free:cTF`, again in a way that `\else:` and `\fi:` are removed. In this implementation this is absolutely necessary because functions inside the conditional parts expect to read further input from outside the conditional. Actually, the first part of the code below is more general, since it checks ⟨*csnames*⟩ directly and therefore allows both `\scan_stop:` and `\c_undefined`.

```
\def_long_test_function_new:npn {cs_if_free:N}#1{\if:w\cs_if_free_p:N #1}
\let:NN \cs_free:NTF \cs_if_free:NTF
\let:NN \cs_free:NT \cs_if_free:NT
\let:NN \cs_free:NF \cs_if_free:NF
```

`\cs_if_free:cTF` `\cs_if_free:cT` `\cs_if_free:cF`

We have to implement the `c` variants 'by hand' because a different test is necessary and I don't want the overhead for the test with `\if:w`. What a mistake Don made by making this a feature of `\cs:w`. If I'm not totally mistaken this feature alone has cost him more than 600$ for bug-checks.

```
\def_long_test_function_new:npn {cs_if_free:c}#1{
  \exp_after:NN \if_meaning:NN \cs:w#1\cs_end: \scan_stop:}
\let:NN \cs_free:cTF \cs_if_free:cTF
\let:NN \cs_free:cT \cs_if_free:cT
\let:NN \cs_free:cF \cs_if_free:cF
```

\cs_if_really_free:cTF
\cs_if_really_free:cT
\cs_if_really_free:cF

These versions are for special control sequences that can only be formed through `\cs:w ... \cs_end:`. They do not turn the control sequence formed into `\scan_stop:`.

```
\def_long_test_function_new:npn {cs_if_really_free:c}#1{
  \reverse_if:N\if_cs_exist:w #1\cs_end:}
\let:NN \cs_really_free:cTF \cs_if_really_free:cTF
\let:NN \cs_really_free:cT \cs_if_really_free:cT
\let:NN \cs_really_free:cF \cs_if_really_free:cF
```

\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_really_exist:cTF
\cs_if_really_exist:cT
\cs_if_really_exist:cF

Now the same functions but with reverse logic: test if the control sequence exists.

```
\def_long_test_function_new:npn {cs_if_exist:N}#1{\if:w\cs_if_exist_p:N #1}
\def_long_test_function_new:npn {cs_if_exist:c}#1{
  \exp_after:NN\reverse_if:N
  \exp_after:NN \if_meaning:NN \cs:w#1\cs_end: \scan_stop:}
\def_long_test_function_new:npn {cs_if_really_exist:c}#1{
  \if_cs_exist:w #1\cs_end:}
```

5.9.7 Freeing memory

\cs_gundefine:N

The following function is used to free the main memory from the definition of some function that isn't in use any longer.

```
\def_new:Npn \cs_gundefine:N #1{\glet:NN #1\c_undefined}
```

5.9.8 Engine specific definitions

\engine_if_aleph:TF

In some cases it will be useful to know which engine we're running.

```
\def_test_function_new:npn {engine_if_aleph:}{\if_cs_exist:N \aleph_textdir:D}
```

5.9.9 Selecting tokens

\use_arg_i:n

This macro grabs its argument and returns it back to the input (with outer braces removed).

```
%\def_long_new:Npn \use_arg_i:n #1{#1}% moved earlier
```

\use:c
\cs_use:c
\use:cc

This macro grabs its argument and returns a csname from it.

```
\def_new:Npn \use:c #1{\cs:w #1\cs_end:}
\def_new:Npn \cs_use:c #1 { \cs:w#1\cs_end: }
```

THE NAME IS COMPLETELY WRONG!!!!! Morten says: Perhaps this is really `\exp_args:cc` instead?

```
\def_new:Npn \use:cc #1#2
  {\cs:w #1\exp_after:NN\cs_end:\cs:w #2\cs_end:}
```

`\use_arg_i:nn` `\use_arg_ii:nn` These macros are needed to provide functions with true and false cases, as introduced by Michael some time ago. By using `\exp_after:NN \use_arg_i:nn \else:` constructions it is possible to write code where the true or false case is able to access the following tokens from the input stream, which is not possible if the `\c_true` syntax is used.

```
\def_long_new:Npn \use_arg_i:nn #1#2{#1}
\def_long_new:Npn \use_arg_ii:nn #1#2{#2}
```

`\use_arg_i:nnn` `\use_arg_ii:nnn` `\use_arg_iii:nnn` `\use_arg_i:nnnn` `\use_arg_ii:nnnn` `\use_arg_iii:nnnn` `\use_arg_iv:nnnn` We also need something for picking up arguments from a longer list.

```
\def_long_new:NNn \use_arg_i:nnn    3{#1}
\def_long_new:NNn \use_arg_ii:nnn   3{#2}
\def_long_new:NNn \use_arg_iii:nnn  3{#3}
\def_long_new:NNn \use_arg_i:nnnn   4{#1}
\def_long_new:NNn \use_arg_ii:nnnn  4{#2}
\def_long_new:NNn \use_arg_iii:nnnn 4{#3}
\def_long_new:NNn \use_arg_iv:nnnn  4{#4}
```

`\use_arg_i_ii:nn` And a function for grabbing two arguments and returning them again.

```
\def_long_new:NNn\use_arg_i_ii:nn 2{#1#2}
```

`\use_none_delimit_by_q_nil:w` `\use_none_delimit_by_q_stop:w` Functions that gobble everything until they see either `\q_nil` or `\q_stop` resp.

```
\def_long_new:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
\def_long_new:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
```

`use_arg_i_delimit_by_q_nil:nw` `se_arg_i_delimit_by_q_stop:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\def_long_new:Npn \use_arg_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
\def_long_new:Npn \use_arg_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
```

`\use_arg_i_after_fi:nw` `\use_arg_i_after_else:nw` `\use_arg_i_after_or:nw` `\use_arg_i_after_or:nw` Returns the first argument after ending the conditional.

```
\def_long_new:Npn \use_arg_i_after_fi:nw #1\fi:{\fi: #1}
\def_long_new:Npn \use_arg_i_after_else:nw #1\else:#2\fi:{\fi: #1}
\def_long_new:Npn \use_arg_i_after_or:nw #1\or: #2\fi: {\fi:#1}
\def_long_new:Npn \use_arg_i_after_orelse:nw #1 #2#3\fi: {\fi:#1}
```

5.9.10 Gobbling tokens from input

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

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although defining `\use_none:nnn` and above as separate calls of `\use_none:n` and `\use_none:nn` is slightly faster, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```
%\def_long_new:NNn \use_none:n 1{}% moved earlier
\def_long_new:NNn \use_none:nn 2{}
\def_long_new:NNn \use_none:nnn 3{}
\def_long_new:NNn \use_none:nnnn 4{}
\def_long_new:NNn \use_none:nnnnn 5{}
\def_long_new:NNn \use_none:nnnnnn 6{}
\def_long_new:NNn \use_none:nnnnnnn 7{}
\def_long_new:NNn \use_none:nnnnnnnn 8{}
\def_long_new:NNn \use_none:nnnnnnnnn 9{}
```

5.9.11 Scratch functions

\gtmp:w

This function is for global scratch definitions that are used immediately afterwards. It should be used when we need a function that operates on input, i.e. has arguments. If we want to save only some tokens for later use, token-list scratch variables should be used.

```
\def_new:Npn \gtmp:w {}
```

\tmp:w

This is a local version of the previous function.

```
\def_new:Npn \tmp:w {}
```

\use_noop:

I don't think this function belongs here, but one place is as good as any other. I want to use this function when I want to express 'no operation'.

```
\def_new:Npn \use_noop: {}
```

5.9.12 Strings and input stream token lists

\cs_to_str:N

This converts a control sequence into the character string of its name, removing the leading escape character.

```
\def_new:Npn \cs_to_str:N {\exp_after:NN\use_none:n \token_to_string:N}
```

\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

Check if two control sequences are identical.

```
\def_test_function_new:npn {cs_if_eq:NN} #1#2{\if_meaning:NN #1#2}
\def_new:Npn \cs_if_eq:cNTF {\exp_args:Nc \cs_if_eq:NNTF}
\def_new:Npn \cs_if_eq:cNT {\exp_args:Nc \cs_if_eq:NNT}
\def_new:Npn \cs_if_eq:cNF {\exp_args:Nc \cs_if_eq:NNF}
\def_new:Npn \cs_if_eq:NcTF {\exp_args:NNc \cs_if_eq:NNTF}
\def_new:Npn \cs_if_eq:NcT {\exp_args:NNc \cs_if_eq:NNT}
\def_new:Npn \cs_if_eq:NcF {\exp_args:NNc \cs_if_eq:NNF}
\def_new:Npn \cs_if_eq:ccTF {\exp_args:Ncc \cs_if_eq:NNTF}
\def_new:Npn \cs_if_eq:ccT {\exp_args:Ncc \cs_if_eq:NNT}
\def_new:Npn \cs_if_eq:ccF {\exp_args:Ncc \cs_if_eq:NNF}
```

Finally some code that is needed as we do not distribute the file module at the moment (so we simply define the needed function via an existing LaTeX command) and some other stuff which was set up elsewhere, in undistributed modules.

```
\def_new:Npn\file_not_found:nTF #1#2#3{\IfFileExists{#1}{#3}{#2}}
```

5.9.13 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

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

   ```
   \cs_free:cTF{abc}{...}{...}
   ```

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

2. Functions that return a special type of boolean value which can be tested by the function `\if:w`. All functions of this type have names that end with `_p` in the description part. For example

   ```
   \cs_free_p:N
   ```

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

   ```
   \if:w \cs_free_p:N \l_foo_bar ... \else: ... \fi:
   ```

3. Actually there is a third one, namely the original concept used in plain TeX. This belongs to the second form but needs further thoughts.

Important to note is that conditionals with ⟨*true code*⟩ and/or ⟨*false code*⟩ are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream while the predicate implementations always have an `\else:` or `\fi:` interfering. This can be important in scanner implementations.

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

6 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.

6.1 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.

6.2 Constants

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

6.3 Internal functions

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

6.4 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3basics}
⟨package⟩\RequirePackage{l3int,l3prg}
⟨∗initex | package⟩
```

The `\chk` module contains those functions that are used primarily during the development of LaTeX3 for checking that things are not mixed up too badly. All these functions are of type 'e' since they issue error messages if certain conditions are violated.

6.4.1 Checking variable assignments

`\chk_local:N` `\chk_local_aux:w`

This function checks that its argument is a proper local variable, i.e. its name starts with `\l` or `\L`. It is not allowed that the name starts with `\g` or `\G`, which means that we do not allow to update global variables locally. But see `\pref_global_chk:` below for the encoding of functions that might accept global variables in certain situations. Not checked is the case that the argument isn't a variable at all, i.e. it doesn't have a `_` as second letter. Maybe we should add this for safety during the implementation since it will find certain errors involving wrong expansion in earlier stage.

```
\def_new:Npn \chk_local:N #1{
  \exp_after:NN \chk_local_aux:w \token_to_string:N#1\q_stop}

\def_new:Npn \chk_local_aux:w #1#2#3\q_stop{
   \if_num:w\tex_uccode:D‘#2=‘G\scan_stop:
       \err_latex_bug:x{Local~mismatch:~local~function~called~with~
                     global~variable:^^J\text_put_four_sp: #1#2#3~
                     on~line~\tex_the:D\tex_inputlineno:D}
   \else:
     \if_num:w\tex_uccode:D‘#2=‘L\scan_stop:
     \else:
       \err_latex_bug:x{Variable~mismatch:~function~not~called~with~
                     proper~variable:^^J\text_put_four_sp: #1#2#3~
                     on~line~\tex_the:D\tex_inputlineno:D}\fi:
   \fi:}
```

We set the `\l_iow_new_line_code` at this point, just in case we run into errors.

```
\tex_newlinechar:D=‘\^^J
```

`\chk_global:N` `\chk_global_aux:w`

`\chk_global:N` is similar to `\chk_local:N` but looks only for global variables.

```
\def_new:Npn \chk_global:N #1{\exp_after:NN
                              \chk_global_aux:w \token_to_string:N#1\q_stop}
\def_new:Npn \chk_global_aux:w #1#2#3\q_stop{
   \if_num:w\tex_uccode:D‘#2=‘L\scan_stop:
       \err_latex_bug:x{Global~mismatch:~global~function~called~with~
                    local~variable:~#1#2#3~
                    on~line~\tex_the:D\tex_inputlineno:D}
```

```
  \else:
    \if_num:w\tex_uccode:D‘#2=‘G\scan_stop:
    \else:
      \err_latex_bug:x{Variable~mismatch:~function~not~called~with~
                    proper~variable:~#1#2#3~
                    on~line~\tex_the:D\tex_inputlineno:D}\fi:\fi:}
```

`\pref_global_chk:`
`\chk_local_or_pref_global:N`

To allow implementations where we precede some function with `\pref_global:D` without loosing the possibility to check for the correct type of a variable, the following helper functions can be used: `\chk_local_or_pref_global:N` ⟨*variable*⟩ is the variable check which is usually `\let:NN` to `\chk_local:N`, i.e. it will check for local variables. 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 issues a `\pref_global:D`. After being used, `\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 of course, that `\foo_bar:n` is defined in a way that a `\pref_global:D` does work. Such a scheme has to be used carefully, but its advantage is that the checking version has the same structure as a streamlined version, we only have to change `\pref_global_chk:` into `\pref_global:D` and omit the `\chk_local_or_pref_global:N` function in the body.

```
\def_new:Npn \pref_global_chk: {
    \gdef:Npn \chk_local_or_pref_global:N ##1{
          \chk_global:N ##1
          \glet:NN \chk_local_or_pref_global:N \chk_local:N}
    \pref_global:D}
\let_new:NN \chk_local_or_pref_global:N \chk_local:N
```

`\chk_var_or_const:N`
`\chk_var_or_const_aux:w`

`\chk_var_or_const:N` is used in situations where we want to check that we have a variable (or a constant) but do not care whether or not it is global (for example, in allocation routines).

```
\def_new:Npn \chk_var_or_const:N #1{\exp_after:NN
     \chk_var_or_const_aux:w \token_to_string:N#1\q_stop }
\def_new:Npn \chk_var_or_const_aux:w #1#2#3\q_stop {
    \if_num:w\tex_uccode:D‘#2=‘L\scan_stop:
    \else:
      \if_num:w\tex_uccode:D‘#2=‘G\scan_stop:
      \else:
        \if_num:w\tex_uccode:D‘#2=‘C\scan_stop:
        \else:
```

We also allow the beast to be a quark, i.e. to start with `\q`.

```
          \if_charcode:w#2q\scan_stop:
          \else:
```

We might also want to allow that it is a user definable variable which means that its name consists of letters only. We could check this by testing that there is no _, but this is not implemented so far.

```
          \err_latex_bug:x{Variable~mismatch:~function~not~called~with~
                      proper~variable:^^J\text_put_four_sp: #1#2#3~
                      on~line~\tex_the:D\tex_inputlineno:D}\fi:\fi:\fi:
    \fi:}
```

6.4.2 Doing some tracing

\tracingall
\absolutelytracingall
\donotcheck

During \tracingall we don't want to see all this code coming from the checking functions since something more substantial is probably wrong. Therefore this definition of it turns the functions of as far as possible.

```
\def_new:Npn\donotcheck{
  \let:NN \chk_global:N \use_none:n
  \let:NN \chk_local:N \use_none:n
  \let:NN \chk_local_or_pref_global:N \use_none:n
  \let:NN \pref_global_chk: \pref_global:D
  \let:NN \chk_new_cs:N \use_none:n
  \let:NN \chk_exist_cs:N \use_none:n
  \let:NN \chk_var_or_const:N \use_none:n
  \let:NN \cs_record_name:N \use_none:n
  \let:NN \cs_record_name:c \use_none:n
  \let:NN \cs_record_meaning:N \use_none:n
  \let:NN \register_record_name:N \use_none:n
}
\def_new:Npn\absolutelytracingall{
```

We do the settings by hand to avoid uninteresting lines in the log file as much as possible.

```
  \pref_global:D\g_trace_commands_status\c_two
  \pref_global:D\g_trace_statistics_status\c_two
  \pref_global:D\g_trace_pages_status\c_one
  \pref_global:D\g_trace_output_status\c_one
  \pref_global:D\g_trace_chars_status\c_one
  \pref_global:D\g_trace_macros_status\c_two
  \pref_global:D\g_trace_paragraphs_status\c_one
  \pref_global:D\g_trace_restores_status\c_one
  \pref_global:D\g_trace_box_breadth_int\c_ten_thousand
  \pref_global:D\g_trace_box_depth_int\c_ten_thousand
  \pref_global:D\g_trace_online_status\c_one
  \tex_errorstopmode:D}
%
% Use LaTeX2e definition for now.
%\def_new:Npn\tracingall{
%  \donotcheck
%  \absolutelytracingall
%}
```

\tracingoff

This macro turns off all tracing.

```
\def_new:Npn\tracingoff{
```

First we turn off `\g_trace_online_status` so that we get minimal rubbish on the terminal. Of course, in the log file all assignments are shown.

```
  \pref_global:D\g_trace_online_status\c_zero
  \pref_global:D\g_trace_commands_status\c_zero
  \pref_global:D\g_trace_statistics_status\c_zero
  \pref_global:D\g_trace_pages_status\c_zero
  \pref_global:D\g_trace_output_status\c_zero
  \pref_global:D\g_trace_chars_status\c_zero
  \pref_global:D\g_trace_macros_status\c_zero
  \pref_global:D\g_trace_paragraphs_status\c_zero
  \pref_global:D\g_trace_restores_status\c_zero
  \pref_global:D\g_trace_box_breadth_int\c_zero
  \pref_global:D\g_trace_box_depth_int\c_zero
}
```

6.4.3 Tracing modules

`\traceon` `\traceoff` Turning the tracing of modules on or off. (primitive version).

```
⟨*trace⟩
\def_new:Npn\traceon#1{
  \clist_map_inline:nn {#1}{
    \cs_free:cF{g_trace_ ##1 _status}
    {\int_gincr:c{g_trace_ ##1 _status}}
  }
}
\def_new:Npn\traceoff#1{
  \clist_map_inline:nn {#1}{
    \cs_free:cF{g_trace_ ##1 _status}
    {\int_gdecr:c{g_trace_ ##1 _status}}
  }
}
⟨/trace⟩
⟨−trace⟩\let_new:NN\traceon\use_none:n
⟨−trace⟩\let_new:NN\traceoff\use_none:n
⟨/initex | package⟩
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

7 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.

7.1 Functions

\tlp_new:N
\tlp_new:c
\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.

The form `\tlp_new:N` initializes the token list pointer with an empty value.

\tlp_use:N
\tlp_use:c
\tlp_use:N ⟨*tlp*⟩

Function that inserts the ⟨*tlp*⟩ into the processing stream. Instead of `\tlp_use:N` simply placing the ⟨*tlp*⟩ into the input stream is also supported.`\tlp_use:c` will complain is the ⟨*tlp*⟩ hasn't been declared previously!

\tlp_set:Nn
\tlp_set:Nc
\tlp_set:No
\tlp_set:Nd
\tlp_set:Nf
\tlp_set:Nx
\tlp_gset:Nn
\tlp_gset:Nc
\tlp_gset:No
\tlp_gset:Nd
\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_put_left:Nx
\tlp_gput_left:Nn
\tlp_gput_left:No
\tlp_gput_left:Nx
\tlp_put_right:Nn
\tlp_put_right:No
\tlp_put_right:Nx
\tlp_put_right:cc
\tlp_gput_right:Nn
\tlp_gput_right:No
\tlp_gput_right:Nx
\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.

A word of warning is appropriate here: Token list pointers are implemented as macros and as such currently inherit some of the peculiarities of how TEX handles `#`s in the argument of macros. In particular, the following actions are legal

```
\tlp_set:Nn \l_tmpa_tlp{##1}
\tlp_put_right:Nn \l_tmpa_tlp{##2}
\tlp_set:No \l_tmpb_tlp{\l_tmpa_tlp ##3}
```

`x` type expansions where macros being expanded contain `#`s do not work and will not work until there is an `\expanded` primitive in the engine. If you want them to work you must double `#`s another level.

\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'.

7.2 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.

7.3 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.

\tlist_elt_count:n \tlist_elt_count:o \tlp_elt_count:N	\tlist_elt_count:n {⟨*token list*⟩} \tlp_elt_count:N ⟨*tlp*⟩

Returns the number of elements in the token list. Brace groups encountered count as one element. Note that spaces in this token list are gobbled in the process.

7.3.1 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`

7.4 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.

7.4.1 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_to_str_aux:w` Function used to implement `\tlp_to_str:N`.

7.5 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*⟩.

7.6 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.

7.7 The Implementation

We start by ensuring that the required packages are loaded.

```
1279 ⟨package⟩\ProvidesExplPackage
```

```
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package&!check⟩\RequirePackage{l3basics}
⟨package & check⟩\RequirePackage{l3chk}

⟨*initex | package⟩
```

A token list pointer is a control sequence that holds tokens. The interface is similar to that for token registers, but beware that the behavior vis á vis `\def:Npx` etc. ... is different. (You see this comes from Denys' implementation.)

`\tlp_new:N` `\tlp_new:c` `\tlp_new:Nn` `\tlp_new:cn` `\tlp_new:Nx`

We provide one allocation function (which checks that the name is not used) and two clear functions that locally or globally clear the token list. The allocation function has two arguments to specify an initial value. This is the only way to give values to constants.

```
\def_long_new:Npn \tlp_new:Nn #1#2{
  \chk_new_cs:N #1
```

If checking we don't allow constants to be defined.

```
⟨*check⟩
  \chk_var_or_const:N #1
⟨/check⟩
```

Otherwise any variable type is allowed.

```
  \gdef:Npn #1{#2}
}
\def_new:Npn \tlp_new:cn {\exp_args:Nc \tlp_new:Nn }
\def_long_new:Npn \tlp_new:Nx #1#2{
  \chk_new_cs:N #1
⟨check⟩ \chk_var_or_const:N #1
  \gdef:Npx #1{#2}
}
\def_new:Npn \tlp_new:N #1{\tlp_new:Nn #1{}}
\def_new:Npn \tlp_new:c #1{\tlp_new:cn {#1}{}}
```

`\tlp_use:N` `\tlp_use:c`

Perhaps this should just be enabled when checking?

```
\def_new:Npn \tlp_use:N #1 {
  \if_meaning:NN #1 \scan_stop:
```

If ⟨*tlp*⟩ equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cd_end:` that was created by mistake somewhere.

```
    \err_latex_bug:x {Token~list~pointer~ `\token_to_string:N #1'~
                      has~ an~ erroneous~ structure!}
  \else:
    \exp_after:NN #1
  \fi:
}
\def_new:Npn \tlp_use:c {\exp_args:Nc \tlp_use:N}
```

\tlp_set:Nn
\tlp_set:No
\tlp_set:Nd
\tlp_set:Nf
\tlp_set:Nx
\tlp_set:cn
\tlp_set:co
\tlp_set:cx

To set token lists to a specific value to type of functions are available: `\tlp_set_eq:NN` takes two token-lists as its arguments assign the first the contents of the second; `\tlp_set:Nn` has as its second argument a 'real' list of tokens. One can view `\tlp_set_eq:NN` as a special form of `\tlp_set:No`. Both functions have global counterparts.

During development we check if the token list that is being assigned to exists. If not, a warning will be issued.

\tlp_gset:Nn
\tlp_gset:No
\tlp_gset:Nd
\tlp_gset:Nx
\tlp_gset:cn
\tlp_gset:cx

```
1308 ⟨*check⟩
1309 \def_long_new:Npn \tlp_set:Nn #1#2{
1310   \chk_exist_cs:N #1  \def:Npn #1{#2}
```

We use `\chk_local_or_pref_global:N` after the assignment to allow constructs with `\pref_global_chk:`. But one should note that this is less efficient then using the real global variant since they are built-in.

```
1311   \chk_local_or_pref_global:N #1
1312 }
1313 \def_long_new:Npn \tlp_set:Nx #1#2{
1314   \chk_exist_cs:N #1  \def:Npx #1{#2}  \chk_local:N #1
1315 }
```

The the global versions.

```
1316 \def_long_new:Npn \tlp_gset:Nn #1#2{
1317   \chk_exist_cs:N #1  \gdef:Npn #1{#2}  \chk_global:N #1
1318 }
1319 \def_long_new:Npn \tlp_gset:Nx #1#2{
1320   \chk_exist_cs:N #1  \gdef:Npx #1{#2}  \chk_global:N #1
1321 }
1322 ⟨/check⟩
```

For some functions like `\tlp_set:Nn` we need to define the 'non-check' version with arguments since we want to allow constructions like `\tlp_set:Nn\l_tmpa_tlp\foo` and so we can't use the primitive TEX command.

```
1323 ⟨*!check⟩
1324 \def_long_new:Npn\tlp_set:Nn#1#2{\def:Npn#1{#2}}
1325 \def_long_new:Npn\tlp_set:Nx#1#2{\def:Npx#1{#2}}
1326 \def_long_new:Npn\tlp_gset:Nn#1#2{\gdef:Npn#1{#2}}
1327 \def_long_new:Npn\tlp_gset:Nx#1#2{\gdef:Npx#1{#2}}
1328 ⟨/!check⟩
```

The remaining functions can just be defined with help from the expansion module.

```
1329 \def_new:Npn \tlp_set:No {\exp_args:NNo \tlp_set:Nn}
1330 \def_new:Npn \tlp_set:Nd {\exp_args:NNd \tlp_set:Nn}
1331 \def_new:Npn \tlp_set:Nf {\exp_args:NNf \tlp_set:Nn}
1332 \def_new:Npn \tlp_set:cn {\exp_args:Nc  \tlp_set:Nn}
1333 \def_new:Npn \tlp_set:co {\exp_args:Nco \tlp_set:Nn}
1334 \def_new:Npn \tlp_set:cx {\exp_args:Nc  \tlp_set:Nx}
1335 \def_new:Npn \tlp_gset:No {\exp_args:NNo \tlp_gset:Nn}
1336 \def_new:Npn \tlp_gset:Nd {\exp_args:NNd \tlp_gset:Nn}
1337 \def_new:Npn \tlp_gset:cn {\exp_args:Nc \tlp_gset:Nn}
1338 \def_new:Npn \tlp_gset:cx {\exp_args:Nc \tlp_gset:Nx}
```

\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

For setting token list pointers equal to each other. First checking:

```
⟨*check⟩
\def_new:Npn \tlp_set_eq:NN #1#2{
  \chk_exist_cs:N #1  \let:NN #1#2
  \chk_local_or_pref_global:N #1  \chk_var_or_const:N #2
}
\def_new:Npn \tlp_gset_eq:NN #1#2{
  \chk_exist_cs:N #1  \glet:NN #1#2
  \chk_global:N #1  \chk_var_or_const:N #2
}
⟨/check⟩
```

Non-checking versions are easy.

```
⟨*!check⟩
\let_new:NN \tlp_set_eq:NN \let:NN
\let_new:NN \tlp_gset_eq:NN \glet:NN
⟨/!check⟩
```

The rest again with the expansion module.

```
\def_new:Npn \tlp_set_eq:Nc {\exp_args:NNc \tlp_set_eq:NN}
\def_new:Npn \tlp_set_eq:cN {\exp_args:Nc  \tlp_set_eq:NN}
\def_new:Npn \tlp_set_eq:cc {\exp_args:Ncc \tlp_set_eq:NN}
\def_new:Npn \tlp_gset_eq:Nc {\exp_args:NNc \tlp_gset_eq:NN}
\def_new:Npn \tlp_gset_eq:cN {\exp_args:Nc  \tlp_gset_eq:NN}
\def_new:Npn \tlp_gset_eq:cc {\exp_args:Ncc \tlp_gset_eq:NN}
```

\tlp_clear:N
\tlp_clear:c
\tlp_gclear:N
\tlp_gclear:c

Clearing a token list pointer.

```
\def_new:Npn \tlp_clear:N #1{\tlp_set_eq:NN #1\c_empty_tlp}
\def_new:Npn \tlp_clear:c {\exp_args:Nc \tlp_clear:N}
\def_new:Npn \tlp_gclear:N #1{\tlp_gset_eq:NN #1\c_empty_tlp}
\def_new:Npn \tlp_gclear:c {\exp_args:Nc \tlp_gclear:N}
```

\tlp_clear_new:N
\tlp_clear_new:c

These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.

```
⟨*check⟩
\def_new:Npn \tlp_clear_new:N #1{
  \chk_var_or_const:N #1
  \if:w \cs_exist_p:N #1
    \tlp_clear:N #1
  \else:
    \tlp_new:Nn #1{}
  \fi:
}
⟨/check⟩
⟨−check⟩\let_new:NN \tlp_clear_new:N \tlp_clear:N
\def_new:Npn \tlp_clear_new:c {\exp_args:Nc \tlp_clear_new:N}
```

\tlp_gclear_new:N
\tlp_gclear_new:c

These are the global versions of the above.

```
⟨*check⟩
\def_new:Npn \tlp_gclear_new:N #1{
  \chk_var_or_const:N #1
  \if:w \cs_exist_p:N #1
    \tlp_gclear:N #1
  \else:
    \tlp_new:Nn #1{}
  \fi:}
⟨/check⟩
⟨−check⟩\let_new:NN \tlp_gclear_new:N \tlp_gclear:N
\def_new:Npn \tlp_gclear_new:c {\exp_args:Nc \tlp_gclear_new:N}
```

\tlp_put_left:Nn
\tlp_put_left:No
\tlp_put_left:Nx
\tlp_gput_left:Nn
\tlp_gput_left:No
\tlp_gput_left:Nx

We can add tokens to the left (either globally or locally). It is not quite as easy as we would like because we have to ensure the assignments

```
\tlp_set:Nn \l_tmpa_tlp{##1abc##2def}
\tlp_set:Nn \l_tmpb_tlp{##1abc}
\tlp_put_right:Nn \l_tmpb_tlp {##2def}
```

cause \l_tmpa_tlp and \l_tmpb_tlp to be identical. The old code did not succeed in doing this (it gave an error) and so we use a different technique where the item(s) to be added are first stored in a temporary pointer and then added using an x type expansion combined with the appropriate level of non-expansion. Putting the tokens directly into one assignment does not work unless we want full expansion. Note (according to the warning earlier) TEX does not allow us to treat #s the same in all cases. Tough.

```
\def_long_new:Npn \tlp_put_left:Nn #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_set:Nx #1{\exp_not:o{\l_exp_tlp}\exp_not:o{#1}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_put_left:No #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_set:Nx #1{\exp_not:d{\l_exp_tlp}\exp_not:o{#1}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_put_left:Nx #1#2{
  \tlp_set:Nx #1{#2\exp_not:o{#1}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_gput_left:Nn #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_gset:Nx #1{\exp_not:o{\l_exp_tlp}\exp_not:o{#1}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_gput_left:No #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_gset:Nx #1{\exp_not:d{\l_exp_tlp}\exp_not:o{#1}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_gput_left:Nx #1#2{
  \tlp_gset:Nx #1{#2\exp_not:o{#1}}
```

```
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_put_left:cn{\exp_args:Nc\tlp_put_left:Nn}
\def_long_new:Npn \tlp_put_left:co{\exp_args:Nc\tlp_put_left:No}
\def_long_new:Npn \tlp_put_left:cx{\exp_args:Nc\tlp_put_left:Nx}
\def_long_new:Npn \tlp_gput_left:cn{\exp_args:Nc\tlp_gput_left:Nn}
\def_long_new:Npn \tlp_gput_left:co{\exp_args:Nc\tlp_gput_left:No}
\def_long_new:Npn \tlp_gput_left:cx{\exp_args:Nc\tlp_gput_left:Nx}
```

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

These are variants of the functions above, but for adding tokens to the right.

```
\def_long_new:Npn \tlp_put_right:Nn #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_set:Nx #1{\exp_not:o{#1}\exp_not:o{\l_exp_tlp}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_gput_right:Nn #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_gset:Nx #1{\exp_not:o{#1}\exp_not:o{\l_exp_tlp}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_put_right:No #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_set:Nx #1{\exp_not:o{#1}\exp_not:d{\l_exp_tlp}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long_new:Npn \tlp_gput_right:No #1#2{
  \tlp_set:Nn \l_exp_tlp{#2}
  \tlp_gset:Nx #1{\exp_not:o{#1}\exp_not:d{\l_exp_tlp}}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long:Npn \tlp_put_right:Nx #1#2{
  \tlp_set:Nx #1{\exp_not:o{#1}#2}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_long:Npn \tlp_gput_right:Nx #1#2{
  \tlp_gset:Nx #1{\exp_not:o{#1}#2}
⟨check⟩ \chk_local_or_pref_global:N #1
}
\def_new:Npn \tlp_gput_right:cn {\exp_args:Nc \tlp_gput_right:Nn}
\def_new:Npn \tlp_gput_right:co {\exp_args:Nc \tlp_gput_right:No}
\def_new:Npn \tlp_put_right:cc {\exp_args:Ncc \tlp_put_right:Nn}
```

\tlp_gset:Nc
\tlp_set:Nc

These two functions are included because they are necessary in Denys' implementations. The :Nc convention (see the expansion module) is very unusual at first sight, but it works nicely over all modules, so we would like to keep it.

Construct a control sequence on the fly from #2 and save it in #1.

```
\def_new:Npn \tlp_gset:Nc {
⟨*check⟩
  \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
  \tlp_set:Nc}
```

\pref_global_chk: will turn the variable check in \tlp_set:No into a global check.

```
\def_new:Npn \tlp_set:Nc #1#2{\tlp_set:No #1{\cs:w#2\cs_end:}}
```

We also provide a few conditionals, both in expandable form (with \c_true) and in 'brace-form', the latter are denoted by TF at the end, as explained elsewhere.

\tlp_if_empty_p:N
\tlp_if_empty_p:c

Returns \c_true iff the token list in the argument is empty.

```
\def_new:Npn \tlp_if_empty_p:N #1{
  \if_meaning:NN#1\c_empty_tlp \c_true \else: \c_false \fi:}
\def_new:Npn \tlp_if_empty_p:c {\exp_args:Nc\tlp_if_empty_p:N}
```

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

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```
\def_test_function_new:npn {tlp_if_empty:N} #1{
  \if_meaning:NN#1\c_empty_tlp}
\def_new:Npn \tlp_if_empty:cTF {\exp_args:Nc \tlp_if_empty:NTF}
\def_new:Npn \tlp_if_empty:cT {\exp_args:Nc \tlp_if_empty:NT}
\def_new:Npn \tlp_if_empty:cF {\exp_args:Nc \tlp_if_empty:NF}
```

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

Returns \c_true iff the two token list pointers are equal.

```
\def_new:Npn \tlp_if_eq_p:NN #1#2{
  \if_meaning:NN#1#2 \c_true \else: \c_false \fi:}
\def_new:Npn \tlp_if_eq_p:Nc {\exp_args:NNc\tlp_if_empty_p:NN}
\def_new:Npn \tlp_if_eq_p:cN {\exp_args:Nc\tlp_if_empty_p:NN}
\def_new:Npn \tlp_if_eq_p:cc {\exp_args:Ncc\tlp_if_empty_p:NN}
```

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

These function tests whether the token list pointers that are in its first two arguments are equal.

```
\def_test_function_new:npn {tlp_if_eq:NN} #1#2{\if_meaning:NN#1#2}
\def_new:Npn \tlp_if_eq:cNTF{\exp_args:Nc \tlp_if_eq:NNTF}
\def_new:Npn \tlp_if_eq:cNT {\exp_args:Nc \tlp_if_eq:NNT}
\def_new:Npn \tlp_if_eq:cNF {\exp_args:Nc \tlp_if_eq:NNF}
\def_new:Npn \tlp_if_eq:NcTF{\exp_args:NNc \tlp_if_eq:NNTF}
\def_new:Npn \tlp_if_eq:NcT {\exp_args:NNc \tlp_if_eq:NNT}
\def_new:Npn \tlp_if_eq:NcF {\exp_args:NNc \tlp_if_eq:NNF}
\def_new:Npn \tlp_if_eq:ccTF{\exp_args:Ncc \tlp_if_eq:NNTF}
\def_new:Npn \tlp_if_eq:ccT {\exp_args:Ncc \tlp_if_eq:NNT}
\def_new:Npn \tlp_if_eq:ccF {\exp_args:Ncc \tlp_if_eq:NNF}
```

\c_empty_tlp
\c_relax_tlp

Two constants which are often used.

```
\tlp_new:Nn \c_empty_tlp {}
\tlp_new:Nn \c_relax_tlp {\scan_stop:}
```

\g_tmpa_tlp
\g_tmpb_tlp

Global temporary token list pointers. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
\tlp_new:Nn \g_tmpa_tlp{}
\tlp_new:Nn \g_tmpb_tlp{}
```

\l_testa_tlp
\l_testb_tlp
\g_testa_tlp
\g_testb_tlp

Global and local temporaries. These are the ones for test routines. This means that one can safely use other temporaries when calling test routines.

```
\tlp_new:Nn \l_testa_tlp {}
\tlp_new:Nn \l_testb_tlp {}
\tlp_new:Nn \g_testa_tlp {}
\tlp_new:Nn \g_testb_tlp {}
```

\l_tmpa_tlp
\l_tmpb_tlp

These are local temporary token list pointers.

```
\tlp_new:Nn \l_tmpa_tlp{}
\tlp_new:Nn \l_tmpb_tlp{}
```

\tlp_to_str:N
\tlp_to_str:c
\tlp_to_str_aux:w

These functions return the replacement text of a token list as a string list with all characters catcoded to 'other'.

```
\def_new:Npn \tlp_to_str:N {\exp_after:NN\tlp_to_str_aux:w
  \token_to_meaning:N}
\def_new:Npn \tlp_to_str_aux:w #1>{}
\def_new:Npn\tlp_to_str:c{\exp_args:Nc\tlp_to_str:N}
```

\tlist_if_empty_p:n

It would be tempting to just use `\if_meaning:NN\q_nil#1\q_nil` as a test since this works really well. However it fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true:` a `\else:` b `\fi:` because then `\if_true:` is used by `\if_meaning:NN`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the end starts executing... A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept `\q_nil` as the first token.

```
\def_long_new:Npn \tlist_if_empty_p:n #1{
  \exp_after:NN\if_meaning:NN\exp_after:NN\q_nil\tlist_to_str:n{#1}\q_nil
    \c_true
  \else:
    \c_false
  \fi:
}
```

\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

```
\def_new:Npn \tlist_if_empty_p:o {\exp_args:No\tlist_if_empty_p:n}
\def_long_test_function_new:npn{tlist_if_empty:n}#1{
  \if:w\tlist_if_empty_p:n{#1}}
\def_long_test_function_new:npn{tlist_if_empty:o}#1{
  \if:w\tlist_if_empty_p:o{#1}}
```

\tlist_if_blank_p:n
\tlist_if_blank_p_aux:w

This is based on the answers in "Around the Bend No 2" but is safer as the tests listed there all have one small flaw: If the input in the test is two tokens with the same meaning as the internal delimiter, they will fail since one of them is mistaken for the actual delimiter. In our version below we make sure to pass the input through `\tlist_to_str:n` which ensures that all the tokens are converted to catcode 12. However we use an `a` with

catcode 11 as delimiter so we can *never* get into the same problem as the solutions in "Around the Bend No 2".

```
\def_long_new:Npn \tlist_if_blank_p:n #1{
  \exp_after:NN\tlist_if_blank_p_aux:w\tlist_to_str:n{#1}aa..\q_nil
}
\def_new:Npn \tlist_if_blank_p_aux:w #1#2a#3#4\q_nil{
  \if_meaning:NN #3#4\c_true\else:\c_false\fi:}
```

`\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`

Variations on the original function above.

```
\def_long_test_function_new:npn{tlist_if_blank:n}#1{
  \if:w\tlist_if_blank_p:n{#1}}
\def:Npn \tlist_if_blank_p:o{\exp_args:No\tlist_if_blank_p:n}
\def_long_test_function_new:npn{tlist_if_blank:o}#1{
  \if:w\tlist_if_blank_p:o{#1}}
```

`\tlist_to_lowercase:n` `\tlist_to_uppercase:n`

Just some names for a few primitives.

```
\let_new:NN \tlist_to_lowercase:n \tex_lowercase:D
\let_new:NN \tlist_to_uppercase:n \tex_uppercase:D
```

`\tlist_to_str:n`

Another name for a primitive.

```
\let_new:NN \tlist_to_str:n \etex_detokenize:D
```

`\tlist_map_function:nN` `\tlp_map_function:NN` `\tlp_map_function:cN` `\tlist_map_function_aux:NN`

Expandable loop macro for `tlist`s. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

```
\def_long_new:Npn \tlist_map_function:nN #1#2{
  \tlist_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop
}
\def_new:Npn \tlp_map_function:NN #1#2{
  \exp_after:NN \tlist_map_function_aux:Nn
  \exp_after:NN #2 #1 \q_recursion_tail \q_recursion_stop
}
\def_long_new:Npn \tlist_map_function_aux:Nn #1#2{
  \quark_if_recursion_tail_stop:n{#2}
  #1{#2} \tlist_map_function_aux:Nn  #1
}
\def_new:Npn\tlp_map_function:cN{\exp_args:Nc\tlp_map_function:NN}
```

`\tlist_map_inline:nn` `\tlp_map_inline:Nn` `\tlp_map_inline:cn` `\tlist_map_inline_aux:n` `\g_tlp_inline_level_num`

The inline functions are straight forward by now. We use a little a trick with the fake counter `\g_tlp_inline_level_num` to make them nestable.[6] We can also make use of `\tlist_map_function:Nn` from before.

```
\def_long_new:Npn \tlist_map_inline:nn #1#2{
  \num_gincr:N \g_tlp_inline_level_num
  \gdef_long:cpn {tlist_map_inline_ \num_use:N \g_tlp_inline_level_num :n}
  ##1{#2}
```

[6] This should be a proper integer, but I don't want to mess with the dependencies right now...

```
  \exp_args:Nc \tlist_map_function_aux:Nn
  {tlist_map_inline_ \num_use:N \g_tlp_inline_level_num :n}
  #1 \q_recursion_tail\q_recursion_stop
  \num_gdecr:N \g_tlp_inline_level_num
}
\def_long_new:Npn \tlp_map_inline:Nn #1#2{
  \num_gincr:N \g_tlp_inline_level_num
  \gdef_long:cpn {tlist_map_inline_ \num_use:N \g_tlp_inline_level_num :n}
  ##1{#2}
  \exp_args:NcE \tlist_map_function_aux:Nn
  {tlist_map_inline_ \num_use:N \g_tlp_inline_level_num :n}
  #1 \q_recursion_tail\q_recursion_stop
  \num_gdecr:N \g_tlp_inline_level_num
}
\def_new:Npn\tlp_map_inline:cN{\exp_args:Nc\tlp_map_inline:NN}
\tlp_new:Nn \g_tlp_inline_level_num{0}
```

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

`\tlist_map:nNn` ⟨*tlist*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\def_long_new:Npn \tlist_map_variable:nNn #1#2#3{
  \tlist_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
}
\def_new:Npn \tlp_map_variable:NNn {\exp_args:No \tlist_map_variable:nNn}
\def_new:Npn \tlp_map_variable:cNn {\exp_args:Nc \tlp_map_variable:NNn}
```

`\tlist_map_variable_aux:NnN` The general loop. Assign the temp variable `#1` to the current item `#3` and then check if that's the stop marker. If it is, break the loop. If not, execute the action `#2` and continue.

```
\def_long_new:Npn \tlist_map_variable_aux:Nnn #1#2#3{
  \tlp_set:Nn #1{#3}
  \quark_if_recursion_tail_stop:N #1
  #2 \tlist_map_variable_aux:Nnn #1{#2}
}
```

`\tlist_map_break:w` `\tlp_map_break:w` The break statement.

```
\let_new:NN \tlist_map_break:w \use_none_delimit_by_q_recursion_stop:w
\let_new:NN \tlp_map_break:w \tlist_map_break:w
```

`\tlist_elt_count:n` `\tlist_elt_count:o` `\tlp_elt_count:n` Count number of elements within a token list or token list pointer. Brace groups within the list are read as a single element. `\num_elt_count:n` grabs the element and replaces it by `+1`. The `0` to ensure it works on an empty list.

```
\def_long_new:Npn \tlist_elt_count:n #1{
  \num_value:w \num_eval:w  0
  \tlist_map_function:nN {#1}\num_elt_count:n
  \num_eval_end:
}
\def_new:Npn \tlist_elt_count:o {\exp_args:No\tlist_elt_count:n}
\def_new:Npn \tlp_elt_count:N #1{
  \num_value:w \num_eval:w  0
  \tlp_map_function:NN #1 \num_elt_count:n
  \num_eval_end:
}
```

\tlist_if_eq:nnTF
\tlist_if_eq:nnT
\tlist_if_eq:nnF

Test if two token lists are identical. pdfTEX contains a most interesting primitive for expandable string comparison so we make use of it if available. Presumable it will be in the final version.

Firstly we give it an appropriate name. Note that this primitive actually performs an `x` type expansion but it is still expandable! Hence we must program these functions backwards to add `\exp_not:n`. We provide the combinations for the types `n`, `o` and `x`.

```
\let_new:NN \tlist_compare:xx \pdfstrcmp
\def_long_new:NNn \tlist_compare:nn 2{
  \tlist_compare:xx{\exp_not:n{#1}}{\exp_not:n{#2}}
}
\def_long_new:NNn \tlist_compare:nx 1{
  \tlist_compare:xx{\exp_not:n{#1}}
}
\def_long_new:NNn \tlist_compare:xn 2{
  \tlist_compare:xx{#1}{\exp_not:n{#2}}
}
\def_long_new:NNn \tlist_compare:no 2{
  \tlist_compare:xx{\exp_not:n{#1}}{\exp_not:n\exp_after:NN{#2}}
}
\def_long_new:NNn \tlist_compare:on 2{
  \tlist_compare:xx{\exp_not:n\exp_after:NN{#1}}{\exp_not:n{#2}}
}
\def_long_new:NNn \tlist_compare:oo 2{
  \tlist_compare:xx{\exp_not:n\exp_after:NN{#1}}{\exp_not:n\exp_after:NN{#2}}
}
\def_long_new:NNn \tlist_compare:xo 2{
  \tlist_compare:xx{#1}{\exp_not:n\exp_after:NN{#2}}
}
\def_long_new:NNn \tlist_compare:ox 2{
  \tlist_compare:xx{\exp_not:n\exp_after:NN{#1}}{\exp_not:n{#2}}
}
```

Since we have a lot of basically identical functions to define we define one to define the rest. Unfortunately we aren't quite set up to use the new `\tlist_map_inline:nn` function yet.

```
\def:Npn \tmp:w #1{
  \def_long_new:cNx {tlist_if_eq_p:#1} 2{
    \exp_not:N \if_num:w
    \exp_after:NN \exp_not:N \cs:w tlist_compare:#1\cs_end:{##1}{##2}
    \exp_not:n{ =\c_zero \c_true \else: \c_false \fi: }
  }
  \def_long_test_function_new:npx{tlist_if_eq:#1}##1##2{
    \exp_not:N \if_num:w
    \exp_after:NN \exp_not:N \cs:w tlist_compare:#1\cs_end:{##1}{##2}
    \exp_not:n{ =\c_zero }
  }
}
\tmp:w{xx}  \tmp:w{nn}  \tmp:w{oo}  \tmp:w{xn}  \tmp:w{nx}
\tmp:w{on}  \tmp:w{no}  \tmp:w{xo}  \tmp:w{ox}
```

However all of this only makes sense if we actually have that primitive. Therefore we disable it again if it is not there and define `\tlist_if_eq:nn` the old fashioned (and

unexpandable) way.

```
\cs_if_really_free:cT{pdf_strcmp:D}{
  \def_long_test_function:npn{tlist_if_eq:nn}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:n{#1}}
    \tlp_set:Nx \l_testb_tlp {\exp_not:n{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:no}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:n{#1}}
    \tlp_set:Nx \l_testb_tlp {\exp_not:o{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:nx}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:n{#1}}
    \tlp_set:Nx \l_testb_tlp {#2}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:on}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:o{#1}}
    \tlp_set:Nx \l_testb_tlp {\exp_not:n{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:oo}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:o{#1}}
    \tlp_set:Nx \l_testb_tlp {\exp_not:o{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:ox}#1#2{
    \tlp_set:Nx \l_testa_tlp {\exp_not:o{#1}}
    \tlp_set:Nx \l_testb_tlp {#2}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:xn}#1#2{
    \tlp_set:Nx \l_testa_tlp {#1}
    \tlp_set:Nx \l_testb_tlp {\exp_not:n{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:xo}#1#2{
    \tlp_set:Nx \l_testa_tlp {#1}
    \tlp_set:Nx \l_testb_tlp {\exp_not:o{#2}}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
  \def_long_test_function:npn{tlist_if_eq:xx}#1#2{
    \tlp_set:Nx \l_testa_tlp {#1}
    \tlp_set:Nx \l_testb_tlp {#2}
    \if_meaning:NN\l_testa_tlp \l_testb_tlp
  }
}
```

7.7.1 Checking for and replacing tokens

\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

See the replace functions for further comments. In this part we don't care too much about brace stripping since we are not interested in passing on the tokens which are split

off in the process.

```
\def_long:Npn \tlp_if_in:NnTF #1#2{
  \def_long:Npn\tmp:w ##1 #2 ##2\q_stop{
    \quark_if_no_value:nFT{##2}
  }
  \exp_after:NN \tmp:w #1 #2 \q_no_value \q_stop
}
\def_new:Npn \tlp_if_in:cnTF {\exp_args:Nc\tlp_if_in:NnTF}
\def_long:Npn \tlp_if_in:NnT #1#2{
  \def_long:Npn\tmp:w ##1 #2 ##2\q_stop{
    \quark_if_no_value:nF{##2}
  }
  \exp_after:NN \tmp:w #1 #2 \q_no_value \q_stop
}
\def_new:Npn \tlp_if_in:cnT {\exp_args:Nc\tlp_if_in:NnT}
\def_long:Npn \tlp_if_in:NnF #1#2{
  \def_long:Npn\tmp:w ##1 #2 ##2\q_stop{
    \quark_if_no_value:nT{##2}
  }
  \exp_after:NN \tmp:w #1 #2 \q_no_value \q_stop
}
\def_new:Npn \tlp_if_in:cnF {\exp_args:Nc\tlp_if_in:NnF}
\def_long_new:Npn \tlist_if_in:nnTF #1#2{
  \def_long:Npn\tmp:w ##1 #2 ##2\q_stop{
    \quark_if_no_value:nFT{##2}
  }
  \tmp:w #1 #2 \q_no_value \q_stop
}
\def_new:Npn \tlist_if_in:onTF {\exp_args:No\tlist_if_in:nnTF}
```

`\l_tlp_replace_tlp` A temp variable for the replace operations.

```
\tlp_new:Nn\l_tlp_replace_tlp{}
```

`\tlp_replace_in:Nnn` `\tlp_replace_in:cnn` `\tlp_greplace_in:Nnn` `\tlp_greplace_in:cnn` `\tlp_replace_in_aux:NNnn` Replacing the first item in a token list pointer goes like this: Define a temporary function with delimited arguments containing the search term and take a closer look at what is left. We append the expansion of the tlp with the search term plus the quark `\q_no_value`. If the search term isn't present this last one is found and the following token is the quark, so we test for that. If the search term is present we will have to split off the `#3\q_no_value` we had, so we define yet another function with delimited arguments to do this. The advantage here is that now we have a special end sequence so there is no problem if the search term appears more than once. Only problem left is to prevent brace stripping in both ends, so we prepend the expansion of the tlp with `\q_mark` later to be gobbled and also prepend the remainder of the first split operation with `\q_mark` also to be gobbled again later on.

```
\def_long_new:NNn \tlp_replace_in_aux:NNnn 4{
  \def_long:Npn \tmp:w ##1#3##2\q_stop{
    \quark_if_no_value:nF{##2}
    {
```

At this point `##1` starts with a `\q_mark` so remove it.

```
      \tlp_set:Nx\l_tlp_replace_tlp{\exp_not:o{\use_none:n##1#4}}
      \def_long:Npn \tmp:w ####1#3\q_no_value{
        \tlp_put_right:Nx \l_tlp_replace_tlp {\exp_not:o{\use_none:n ####1}}
      }
      \tmp:w \q_mark ##2
    }
```

Now all that is done is setting the token list pointer equal to the expansion of the token register.

```
    #1#2\l_tlp_replace_tlp
  }
```

Here is where we start the process. Note that the tlp might start with a space token so we use this little trick with `\use_arg_i:n` to prevent it from being removed.

```
  \use_arg_i:n{\exp_after:NN \tmp:w\exp_after:NN\q_mark}
  #2#3 \q_no_value\q_stop
}
```

Now the various versions doing the replacement either globally or locally.

```
\def_new:Npn \tlp_replace_in:Nnn {\tlp_replace_in_aux:NNnn \tlp_set_eq:NN}
\def_new:Npn \tlp_replace_in:cnn{\exp_args:Nc\tlp_replace_in:Nnn}
\def_new:Npn \tlp_greplace_in:Nnn {\tlp_replace_in_aux:NNnn \tlp_gset_eq:NN}
\def_new:Npn \tlp_greplace_in:cnn{\exp_args:Nc\tlp_greplace_in:Nnn}
```

`\tlp_replace_all_in:Nnn`
`\tlp_greplace_all_in:cnn`
`\tlp_replace_all_in:Nnn`
`\tlp_greplace_all_in:cnn`
`\tlp_replace_all_in_aux:NNnn`

The version for replacing *all* occurrences of the search term is fairly easy since we just have to keep doing the replacement on the split-off part until all are replaced. Otherwise it is pretty much the same as above.

```
\def_long:NNn \tlp_replace_all_in_aux:NNnn 4{
  \tlp_clear:N \l_tlp_replace_tlp
  \def_long:Npn \tmp:w ##1#3##2\q_stop{
    \quark_if_no_value:nTF{##2}
    {
      \tlp_put_right:Nx \l_tlp_replace_tlp {\exp_not:o{\use_none:n##1}}
    }
    {
      \tlp_put_right:Nx \l_tlp_replace_tlp {\exp_not:o{\use_none:n##1 #4}}
      \tmp:w \q_mark##2 \q_stop
    }
  }
  \use_arg_i:n{\exp_after:NN \tmp:w\exp_after:NN\q_mark}
  #2#3 \q_no_value\q_stop
  #1#2\l_tlp_replace_tlp
}
```

Now the various forms.

```
\def_new:Npn \tlp_replace_all_in:Nnn {
  \tlp_replace_all_in_aux:NNnn \tlp_set_eq:NN}
\def_new:Npn \tlp_replace_all_in:cnn{\exp_args:Nc\tlp_replace_all_in:Nnn}
\def_new:Npn \tlp_greplace_all_in:Nnn {
  \tlp_replace_all_in_aux:NNnn \tlp_gset_eq:NN}
\def_new:Npn \tlp_greplace_all_in:cnn{\exp_args:Nc\tlp_greplace_all_in:Nnn}
```

\tlp_remove_in:Nn
\tlp_remove_in:cn
\tlp_gremove_in:Nn
\tlp_gremove_in:cn

Next comes a series of removal functions. I have just implemented them as subcases of the replace functions for now (I'm lazy).

```
\def_long_new:NNn \tlp_remove_in:Nn  2{\tlp_replace_in:Nnn #1{#2}{}}
\def_long_new:NNn \tlp_gremove_in:Nn 2{\tlp_greplace_in:Nnn #1{#2}{}}
\def_new:Npn \tlp_remove_in:cn{\exp_args:Nc\tlp_remove_in:Nn}
\def_new:Npn \tlp_gremove_in:cn{\exp_args:Nc\tlp_gremove_in:Nn}
```

\tlp_remove_all_in:Nn
\tlp_remove_all_in:Nn
\tlp_gremove_all_in:Nn
\tlp_gremove_all_in:Nn

Same old, same old.

```
\def_long_new:Npn \tlp_remove_all_in:Nn #1#2{
  \tlp_replace_all_in:Nnn #1{#2}{}
}
\def_long_new:Npn \tlp_gremove_all_in:Nn #1#2{
  \tlp_greplace_all_in:Nnn #1{#2}{}
}
\def_new:Npn \tlp_remove_all_in:cn{\exp_args:Nc\tlp_remove_all_in:Nn}
\def_new:Npn \tlp_gremove_all_in:cn{\exp_args:Nc\tlp_gremove_all_in:Nn}
```

7.7.2 Heads or tails?

\tlist_head:n
\tlist_head_i:n
\tlist_tail:n
\tlist_tail:f
\tlist_head_iii:n
\tlist_head_iii:f
\tlist_head:w
\tlist_tail:w
\tlist_head_iii:w

These functions pick up either the head or the tail of a list. `\tlist_head_iii:n` returns the first three items on a list.

```
\def_long_new:Npn \tlist_head:n #1{\tlist_head:w #1\q_nil}
\let_new:NN \tlist_head_i:n \tlist_head:n
\def_long_new:Npn \tlist_tail:n #1{\tlist_tail:w #1\q_nil}
\def_new:Npn \tlist_tail:f {\exp_args:Nf \tlist_tail:n}
\def_long_new:Npn \tlist_head_iii:n #1{\tlist_head_iii:w #1\q_nil}
\def_new:Npn \tlist_head_iii:f {\exp_args:Nf \tlist_head_iii:n}
\let_new:NN \tlist_head:w \use_arg_i_delimit_by_q_nil:nw
\def_long_new:Npn \tlist_tail:w #1#2\q_nil{#2}
\def_long_new:Npn \tlist_head_iii:w #1#2#3#4\q_nil{#1#2#3}
```

tlist_if_head_eq_meaning_p:nN
tlist_if_head_eq_meaning:nNTF
\tlist_if_head_eq_meaning:nNT
\tlist_if_head_eq_meaning:nNF
st_if_head_eq_charcode_p:nNTF
list_if_head_eq_charcode:nNTF
tlist_if_head_eq_charcode:nNT
tlist_if_head_eq_charcode:nNF
ist_if_head_eq_catcode_p:nNTF
tlist_if_head_eq_catcode:nNTF
\tlist_if_head_eq_catcode:nNT
\tlist_if_head_eq_catcode:nNF

When we want to check if the first token of a list equals something specific it is usually either to see if it is a control sequence or a character. Hence we make two different functions as the internal test is different. `\tlist_if_head_meaning_eq:nNTF` uses `\if_meaning:NN` and will consider the tokens $\texttt{b}_{11}$ and $\texttt{b}_{12}$ different. `\tlist_if_head_char_eq:nNTF` on the other hand only compares character codes so would regard $\texttt{b}_{11}$ and $\texttt{b}_{12}$ as equal but would also regard two primitives as equal.

```
\def_long_new:Npn \tlist_if_head_eq_meaning_p:nN #1#2{
  \exp_after:NN \if_meaning:NN \tlist_head:w #1\q_nil#2
    \c_true
  \else:
    \c_false
  \fi:
}
\def_long_test_function_new:npn {tlist_if_head_eq_meaning:nN}#1#2{
  \if:w \tlist_if_head_eq_meaning_p:nN{#1}#2}
```

For the charcode and catcode versions we insert `\exp_not:N` in front of both tokens. If you need them to expand fully as TEX does itself with these you can use an `f` type expansion.

```
\def_long_new:Npn \tlist_if_head_eq_charcode_p:nN #1#2{
  \exp_after:NN\if_charcode:w \exp_after:NN\exp_not:N
    \tlist_head:w #1\q_nil\exp_not:N#2
   \c_true
 \else:
   \c_false
 \fi:
}
\def_long_test_function_new:npn {tlist_if_head_eq_charcode:nN}#1#2{
 \if:w\tlist_if_head_eq_charcode_p:nN{#1}#2}
```

Actually the default is already an `f` type expansion.

```
\def_long_new:Npn \tlist_if_head_eq_charcode_p:fN #1#2{
  \exp_after:NN\if_charcode:w \tlist_head:w #1\q_nil\exp_not:N#2
   \c_true
 \else:
   \c_false
 \fi:
}
\def_long_test_function_new:npn {tlist_if_head_eq_charcode:fN}#1#2{
 \if:w\tlist_if_head_eq_charcode_p:fN{#1}#2}
\def_long_new:Npn \tlist_if_head_eq_catcode_p:nN #1#2{
  \exp_after:NN\if_charcode:w \exp_after:NN\exp_not:N
    \tlist_head:w #1\q_nil\exp_not:N#2
   \c_true
 \else:
   \c_false
 \fi:
}
\def_long_test_function_new:npn {tlist_if_head_eq_catcode:nN}#1#2{
 \if:w\tlist_if_head_eq_catcode_p:nN{#1}#2}
```

`\tlist_reverse:n` `\tlist_reverse_aux:nN` Reversal of a token list is done by taking one token at a time and putting it in front of the ones before it.

```
\def_long_new:Npn \tlist_reverse:n #1{
  \tlist_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
}
\def_long_new:Npn \tlist_reverse_aux:nN #1#2{
 \quark_if_recursion_tail_stop_do:nn {#2}{ #1 }
 \tlist_reverse_aux:nN {#2#1}
}
```

As this package relies heavily on a lot of the expansion tricks used in l3expan we make sure to load it automatically at the end when used as a package. Probably not needed but I'm just such a nice guy...

```
⟨package⟩\RequirePackage{l3expan}
⟨package⟩\RequirePackage{l3num}\par
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

8 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.

8.1 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.) It is often useful in conjunction with a forced expansion.

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.

8.2 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.

When pdfTeX 1.50 arrives, it will contain a primitive for performing the equivalent of an `x` expansion after only one expansion and most importantly: as an expandable operation.

`\exp_arg:x` `\exp_arg:x { ⟨arg⟩ }`
⟨*arg*⟩ is expanded fully using an `x` expansion.

8.3 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.

8.4 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.

8.5 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.

8.6 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:d`
`\exp_not:f`
`\exp_not:o {`⟨*token list*⟩`}`
Same as `\exp_not:n` except ⟨*token list*⟩ is expanded once for the `o` type and twice for the `d` type 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.

`\exp_stop_f:` ⟨*f expansion*⟩ ... `\exp_stop_f:`
This function stops an `f` type expansion. An example use is one such as

```
\tlp_set:Nf \l_tmpa_tlp {
  \if_case:w \l_tmpa_int
  \or:   \use_arg_i_after_orelse:nw{\exp_stop_f: \textbullet}
  \or:   \use_arg_i_after_orelse:nw{\exp_stop_f: \textendash}
  \else: \use_arg_i_after_orelse:nw{\exp_stop_f: else-item}
  \fi:
}
```

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

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

`\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.

8.7 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3tlp}
⟨*initex | package⟩
```

8.7.1 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version

of x is going to have to use one of the LaTeX3 names for \def:Npx at some point, and so is never going to be expandable.[7])

In a later section some common cases are coded by a more direct method, typically using calls to \exp_after:NN.

\l_exp_tlp We need a scratch token list pointer.

```
\tlp_new:Nn\l_exp_tlp{}
```

This code uses internal functions with names that start with \:: to perform the expansions. All macros are long as this turned out to be desirable since the tokens undergoing expansion may be arbitrary user input.

\exp_arg_next:nnn This is basically the same function as \Dexp_arg_next:nnn.

```
\def_long_new:Npn\exp_arg_next:nnn#1#2#3{
  #2\::{#3#1}
}
```

\::n This function is used to skip an argument that doesn't need to be expanded.

```
\def_long_new:Npn\::n#1\::#2#3{
  #1\::{#2{#3}}
}
```

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
\def_long_new:Npn\::N#1\::#2#3{
  #1\::{#2#3}
}
```

\::c This function is used to skip an argument that is turned into as control sequence without expansion.

```
\def_long_new:Npn\::c#1\::#2#3{
  \exp_after:NN\exp_arg_next:nnn\cs:w #3\cs_end:{#1}{#2}
}
```

\::o This function is used to expand an argument once.

```
\def_long_new:Npn\::o#1\::#2#3{
  \exp_after:NN\exp_arg_next:nnn\exp_after:NN{\exp_after:NN{#3}}{#1}{#2}
}
```

[7] However, some primitives have certain characteristics that means that their arguments undergo an x type expansion but the primitive is in fact still expandable. We shall make it very clear when such a function is expandable.

`\::f` This function is used to expand a token list until the first unexpandable token is found.
`\exp_stop_f:` The underlying `\int_to_roman:w -‘0` expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once TEX had fully expanded `\let:Nc \aaa {b \l_tmpa_tlp b}` into `\let:NwN \aaa = \blurb` which then turned out to contain the non-expandable token `\let:NwN`. Since the expansion of `\int_to_roman:w -‘0` is ⟨*null*⟩, we wind up with a fully expanded list, only TEX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
\def_long_new:Npn\::f#1\:::#2#3{
  \exp_after:NN\exp_arg_next:nnn
  \exp_after:NN{\exp_after:NN{\int_to_roman:w -‘0 #3}}
  {#1}{#2}
}
\def_new:Npn \exp_stop_f: {~}
```

`\::x` This function is used to expand an argument fully. If the pdfTEX primitive `\expanded` is present, we use it.

```
\let_new:NN \exp_arg:x \expanded % Move eventually.
\cs_if_free:NTF\exp_arg:x{
  \def_long_new:Npn\::x#1\:::#2#3{
    % \tlp_set:Nx\l_exp_tlp{{{#3}}}
    \def:Npx \l_exp_tlp{{{#3}}}
    \exp_after:NN\exp_arg_next:nnn\l_exp_tlp{#1}{#2}}
}
{
  \def_long_new:Npn\::x#1\:::#2#3{
    \exp_after:NN\exp_arg_next:nnn
    \exp_after:NN{\exp_arg:x{{#3}}}{#1}{#2}
  }
}
```

`\:::` Just another name for the identity function.

```
\def_long_new:Npn\:::#1{#1}
```

`\::C` This function creates a control sequence out of `#3` and expands that once before passing it on to `\exp_arg_next:nnn`.

```
\def_long_new:Npn\::C#1\:::#2#3{
  \exp_after:NN\exp_C_aux:nnn\cs:w #3\cs_end:{#1}{#2}}
```

`\exp_C_aux:nnn` A helper function for `\::C` wich expands its argument before passing it on to `\exp_arg_next:nnn`.

```
\def_long_new:Npn\exp_C_aux:nnn #1{
  \exp_after:NN
```

```
  \exp_arg_next:nnn
  \exp_after:NN
     {
  \exp_after:NN
      {#1}
     }
}
```

Here are some that might not stay but let's see.

`\::E` This function is used to expand an argument once and return it *without* braces. Use this only when you feel pretty comfortable about your input! Actually this is pretty much just generic wrapper for `\exp_after:NN`.

```
\def_long_new:Npn\::E#1\:::#2#3{
  \exp_after:NN\exp_arg_next:nnn \exp_after:NN{#3}{#1}{#2}
}
```

`\::e` Same as `\::E` really but conceptually they are different.

```
\def_long_new:Npn\::e#1\:::#2#3{
  \exp_after:NN\exp_arg_next:nnn \exp_after:NN{#3}{#1}{#2}
}
```

`\::d` This function is used to expand an argument twice. Mostly useful for `toks` type things.

```
\def_long_new:Npn\::d#1\:::#2#3{
  \exp_after:NN\exp_after:NN\exp_after:NN\exp_arg_next:nnn
  \exp_after:NN\exp_after:NN\exp_after:NN{
  \exp_after:NN\exp_after:NN\exp_after:NN{#3}}{#1}{#2}
}
```

We do most of them by hand here. This also means that we get a name for `\exp_after:NN` that fits with the rest of the code.

```
\let:NN \exp_args:NE \exp_after:NN
\def:Npn \exp_args:NNE #1{\exp_args:NE#1\exp_args:NE}
\def:Npn \exp_args:NNNE #1#2{\exp_args:NE#1\exp_args:NE#2\exp_args:NE}
\def:Npn \exp_args:NEE #1{\exp_args:NE\exp_args:NE\exp_args:NE#1\exp_args:NE}
\def:Npn \exp_args:NcE #1#2{\exp_after:NN #1\cs:w #2\exp_after:NN\cs_end:}
\def:Npn \exp_args:Nd {\::d\:::}
\def:Npn \exp_args:NNd {\::N\::d\:::}
```

`\exp_args:NC` `\exp_args:Ncx` `\exp_args:Ncco` `\exp_args:Nccx` `\exp_args:Ncnx` `\exp_args:NcNc` `\exp_args:Nf` `\exp_args:NNf` `\exp_args:Nfo` `\exp_args:Nnf` `\exp_args:NNno` `\exp_args:NnnN` `\exp_args:Nnno` `\exp_args:Nnnx` `\exp_args:Nno` `\exp_args:Nnox` `\exp_args:NNx` `\exp_args:Nnx` `\exp_args:Noo` Here are the actual function definitions, using the helper functions above.

```
%\def:Npn \exp_args:NNNo {\::N\::N\::o\:::}
%\def:Npn \exp_args:NNOo {\::N\::O\::o\:::}
%\def:Npn \exp_args:NNc {\::N\::c\:::}
%\def:Npn \exp_args:NNo {\::N\::o\::\:::}
%\def:Npn \exp_args:NOOo {\::O\::O\::o\:::}
%\def:Npn \exp_args:NOc {\::O\::c\:::}
%\def:Npn \exp_args:NOo {\::O\::o\:::}
%\def:Npn \exp_args:Nc {\::c\:::}
```

```
%\def:Npn \exp_args:Ncc {\::c\::c\:::}
%\def:Npn \exp_args:Nccc {\::c\::c\::c\:::}
%\def:Npn \exp_args:Nco {\::c\::o\:::}
%\def:Npn \exp_args:No {\::o\:::}

\def:Npn \exp_args:NC {\::C\:::}
\def:Npn \exp_args:NNC {\::N\::C\:::}
\def:Npn \exp_args:NNf {\::N\::f\:::}
\def:Npn \exp_args:NNno {\::N\::n\::o\:::}
\def:Npn \exp_args:NNnx {\::N\::n\::x\:::}  % new
\def:Npn \exp_args:NNoo {\::N\::o\::o\:::}  % new
\def:Npn \exp_args:NNox {\::N\::o\::x\:::}  % new
\def:Npn \exp_args:NNx {\::N\::x\:::}
\def:Npn \exp_args:NcNc {\::c\::N\::c\:::}
\def:Npn \exp_args:NcNo {\::c\::N\::o\:::}
\def:Npn \exp_args:Ncco {\::c\::c\::o\:::}
\def:Npn \exp_args:Ncco {\::c\::c\::o\:::}
\def:Npn \exp_args:Nccx {\::c\::c\::x\:::}
\def:Npn \exp_args:Ncnx {\::c\::n\::x\:::}
\def:Npn \exp_args:Ncx {\::c\::x\:::}
\def:Npn \exp_args:Nf {\::f\:::}
\def:Npn \exp_args:Nfo{\::f\::o\:::}
\def:Npn \exp_args:Nnf {\::n\::f\:::}
\def:Npn \exp_args:NnnN {\::n\::n\::N\:::}   %% Strange one this one...
\def:Npn \exp_args:Nnnc {\::n\::n\::c\:::}
\def:Npn \exp_args:Nnno {\::n\::n\::o\:::}
\def:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
\def:Npn \exp_args:Nno {\::n\::o\:::}
\def:Npn \exp_args:Nnox {\::n\::o\::x\:::}
\def:Npn \exp_args:Nnx {\::n\::x\:::}
\def:Npn \exp_args:Noo {\::o\::o\:::}
\def:Npn \exp_args:Noox {\::o\::o\::x\:::}
\def:Npn \exp_args:Nox {\::o\::x\:::}
\def:Npn \exp_args:Nx {\::x\:::}
\def:Npn \exp_args:Nxo {\::x\::o\:::}
\def:Npn \exp_args:Nxx {\::x\::x\:::}
```

8.7.2 Preventing expansion

`\exp_not:o`
`\exp_not:d`
`\exp_not:f`

```
\def_long_new:Npn\exp_not:o#1{\exp_not:n\exp_after:NN{#1}}
\def_long_new:Npn\exp_not:d#1{
  \exp_not:n\exp_after:NN\exp_after:NN\exp_after:NN{#1}
}
\def_long_new:Npn\exp_not:f#1{
  \exp_not:n\exp_after:NN{\int_to_roman:w -`0 #1}
}
```

`\exp_not:E` Two helper functions, which we can probably live without it.
`\exp_not:c`

```
\def_new:Npn\exp_not:E{\exp_after:NN\exp_not:N}
\def_long_new:Npn\exp_not:c#1{\exp_after:NN\exp_not:N\cs:w#1\cs_end:}
```

8.7.3 Single token expansion

Expansion for arguments that are single tokens is done with the functions below. I first thought of using a different module name but then I saw that this wouldn't do since I could then never determine for, say, `\seq_put:no` whether this means single, or general expansion. Therefore I decided to use uppercase 'O' for single expansion.

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\pref_global:D` for example. This together with the fact that the above concept is much slower in general means that we should convert whenever possible and perhaps remove all remaining occurences by hand-encoding in the end.

`\exp_args:No` `\exp_args:NOo` `\exp_args:NNo` `\exp_args:NNO` `\exp_args:NOOo` `\exp_args:NNOo` `\exp_args:NNNo`

This looks somewhat horrible but it runs well with the other syntax. It is important to see that these functions really need single tokens as arguments whenever capital letters are used.

```
\def_long_new:Npn \exp_args:No #1#2{\exp_after:NN#1\exp_after:NN{#2}}
\def_long_new:Npn \exp_args:NOo #1#2#3{\exp_after:NN\exp_args:No \exp_after:NN#1
  \exp_after:NN#2\exp_after:NN{#3}}
\def_long_new:Npn \exp_args:NOOo #1#2#3#4{\exp_after:NN\exp_args:NOo
  \exp_after:NN#1\exp_after:NN#2\exp_after:NN#3\exp_after:NN{#4}}
\def_long_new:Npn \exp_args:NNo #1#2#3{\exp_after:NN#1\exp_after:NN#2
  \exp_after:NN{#3}}
\def_long_new:Npn \exp_args:NNO #1#2#3 {\exp_after:NN#1
  \exp_after:NN#2 #3}
\def_long_new:Npn \exp_args:NNOo #1#2#3#4{\exp_after:NN\exp_args:NNo
  \exp_after:NN#1\exp_after:NN#2\exp_after:NN#3\exp_after:NN{#4}}
\def_long_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:NN#1\exp_after:NN#2
  \exp_after:NN#3\exp_after:NN{#4}}
```

`\exp_args:Nc` `\exp_args:NNc` `\exp_args:NOc` `\exp_args:Ncc` `\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```
\def_long_new:Npn \exp_args:Nc #1#2{\exp_after:NN#1\cs:w#2\cs_end:}
\def_long_new:Npn \exp_args:NNc #1#2#3{\exp_after:NN#1\exp_after:NN#2
    \cs:w#3\cs_end:}
\def_long_new:Npn \exp_args:NOc#1#2#3{\exp_after:NN\exp_args:No\exp_after:NN
    #1\exp_after:NN#2\cs:w#3\cs_end:}
\def_long_new:Npn \exp_args:Ncc #1#2#3{\exp_after:NN#1
    \cs:w#2\exp_after:NN\cs_end:\cs:w#3\cs_end:}
\def_long_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:NN#1
    \cs:w#2\exp_after:NN\cs_end:\cs:w#3\exp_after:NN
      \cs_end:\cs:w #4\cs_end:}
```

`\exp_args:Nco`

If we force that the third argument always has braces, we could implement this function with less tokens and only two arguments.

```
\def_long_new:Npn \exp_args:Nco #1#2#3{\exp_after:NN#1\cs:w#2\exp_after:NN
     \cs_end:\exp_after:NN{#3}}
```

`\exp_def_form:nnn`

This command is a recent addition which was actually added when we wrote the article for TUGboat (while most of the other code goes way back to 1993).

```
\def:Npn\exp_def_form:nnn#1#2#3{
   \exp_after:NN
   \def:Npn
     \cs:w
        #1:#3
       \exp_after:NN
     \cs_end:
     \exp_after:NN
       {
        \cs:w
           exp_args:N#3
          \exp_after:NN
        \cs_end:
        \cs:w
           #1:#2
        \cs_end:
       }
```

We also have to test if `exp_args:N#3` is already defined and if not define it via the `\::` commands using the chars in `#3`

```
    \cs_if_free:cT
          {exp_args:N#3}
          {\def:cpx {exp_args:N#3}
                    {\exp_args_form_x:w #3 :}
          }
}
```

`\exp_args_form_x:w` This command grabs char by char outputting `\::#1` (not expanded further) until we see a `:`. That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```
\def_new:Npn\exp_args_form_x:w #1 {
  \exp_not:c{::#1}
  \if_meaning:NN #1 :
  \else:
    \exp_after:NN\exp_args_form_x:w
  \fi:}
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

9 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.

9.1 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_add:cn
\num_gadd:Nn
\num_gadd:cn
\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.

\num_max_of:nn
\num_min_of:nn
\num_max_of:nn {⟨num expr⟩} {⟨num expr⟩}
Return the largest or smallest of two ⟨num⟩ expressions.

\num_abs:n
\num_abs:n {⟨num expr⟩}
Return the numerical value of a ⟨num⟩ expression.

\num_elt_count:n
\num_elt_count:n {⟨balanced text⟩}
Discards ⟨balanced text⟩ and puts a +1 in the input stream. Used to count elements in a token list.

9.2 Formatting a counter value

See the l3int module for ways of doing this.

9.3 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.

9.4 Primitive functions

\num_value:w `\num_value:w ⟨integer⟩`
`\num_value:w ⟨tokens⟩  ⟨optional space⟩`
Expands ⟨*tokens*⟩ until an ⟨*integer*⟩ is formed. One space may be gobbled in the process. Preferably use with `\num_eval:n`.

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

`\num_eval:w` `\num_eval:w` ⟨*integer expression*⟩ `\num_eval_end:`
Evaluates ⟨*integer expression*⟩. The evaluation stops when an unexpandable token of catcode other than 12 is reached or `\num_eval_end:` 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`.

9.5 The Implementation

We start by ensuring that the required packages are loaded.

```
1983 ⟨package⟩\ProvidesExplPackage
1984 ⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
1985 ⟨package&!check⟩\RequirePackage{l3expan}\par
1986 ⟨package & check⟩\RequirePackage{l3chk}\par
1987 ⟨*initex | package⟩
```

`\num_value:w` `\num_eval:w` `\num_eval_end:` `\if_num:w` `\if_num_odd:w` `\if_case:w` `\or:`

Here are the remaining primitives for number comparisons and expressions.

```
1988 \let_new:NN \num_value:w        \tex_number:D
1989 \let_new:NN \num_eval:w         \etex_numexpr:D
1990 \let_new:NN \num_eval_end:      \scan_stop:
1991 \let_new:NN \if_num:w           \tex_ifnum:D
1992 \let_new:NN \if_num_odd:w       \tex_ifodd:D
1993 \let_new:NN \if_case:w          \tex_ifcase:D
1994 \let_new:NN \or:                \tex_or:D
```

Functions that support LaTeX's user accessible counters should be added here, too. But first the internal counters.

`\num_incr:N` `\num_decr:N` `\num_gincr:N` `\num_gdecr:N` Incrementing and decrementing of integer registers is done with the following functions.

```
\def:Npn \num_incr:N #1{\num_add:Nn#1 1}
\def:Npn \num_decr:N #1{\num_add:Nn#1 \c_minus_one}
\def:Npn \num_gincr:N #1{\num_gadd:Nn#1 1}
\def:Npn \num_gdecr:N #1{\num_gadd:Nn#1 \c_minus_one}
```

`\num_incr:c` `\num_decr:c` `\num_gincr:c` `\num_gdecr:c` We also need ...

```
\def_new:Npn \num_incr:c {\exp_args:Nc \num_incr:N}
\def_new:Npn \num_decr:c {\exp_args:Nc \num_decr:N}
\def_new:Npn \num_gincr:c {\exp_args:Nc \num_gincr:N}
\def_new:Npn \num_gdecr:c {\exp_args:Nc \num_gdecr:N}
```

`\num_zero:N` `\num_zero:c` `\num_gzero:N` `\num_gzero:c` We also need ...

```
\def_new:Npn \num_zero:N #1 {\num_set:Nn #1 0}
\def_new:Npn \num_gzero:N #1 {\num_gset:Nn #1 0}
\def_new:Npn \num_zero:c {\exp_args:Nc \num_zero:N}
\def_new:Npn \num_gzero:c {\exp_args:Nc \num_gzero:N}
```

`\num_new:N` `\num_new:c` Allocate a new ⟨*num*⟩ variable and initialize it with zero.

```
\def_new:Npn \num_new:N #1{\tlp_new:Nn #1{0}}
\def_new:Npn \num_new:c {\exp_args:Nc \num_new:N}
```

`\num_eval:n` This function enables us to do all the operations without the aid of an ⟨*int*⟩ register.

```
\def_new:Npn \num_eval:n #1{\num_eval:w #1\num_eval_end:}
```

`\num_set:Nn` `\num_set:cn` `\num_gset:Nn` `\num_gset:cn` Assigning values to ⟨*num*⟩ registers.

```
\def_new:Npn \num_set:Nn #1#2{
  \tlp_set:No #1{ \tex_number:D \num_eval:n {#2} }
}
\def_new:Npn \num_gset:Nn {\pref_global:D \num_set:Nn}
\def_new:Npn \num_set:cn {\exp_args:Nc \num_set:Nn }
\def_new:Npn \num_gset:cn {\exp_args:Nc \num_gset:Nn }
```

`\num_set_eq:NN` `\num_set_eq:cN` `\num_set_eq:Nc` `\num_set_eq:cc` Setting ⟨*num*⟩ registers equal to each other.

```
\let_new:NN \num_set_eq:NN \tlp_set_eq:NN
\def_new:Npn \num_set_eq:cN {\exp_args:Nc \num_set_eq:NN}
\def_new:Npn \num_set_eq:Nc {\exp_args:NNc \num_set_eq:NN}
\def_new:Npn \num_set_eq:cc {\exp_args:Ncc \num_set_eq:NN}
```

`\num_gset_eq:NN` `\num_gset_eq:cN` `\num_gset_eq:Nc` `\num_gset_eq:cc` Setting ⟨*num*⟩ registers equal to each other.

```
\let_new:NN \num_gset_eq:NN \tlp_gset_eq:NN
\def_new:Npn \num_gset_eq:cN {\exp_args:Nc \num_gset_eq:NN}
\def_new:Npn \num_gset_eq:Nc {\exp_args:NNc \num_gset_eq:NN}
\def_new:Npn \num_gset_eq:cc {\exp_args:Ncc \num_gset_eq:NN}
```

\num_add:Nn
\num_add:cn
\num_gadd:Nn
\num_gadd:cn

Adding is easily done as the second argument goes through \num_eval:n.

```
\def_new:Npn \num_add:Nn #1#2 {\num_set:Nn #1{#1+#2}}
\def_new:Npn \num_add:cn {\exp_args:Nc\num_add:Nn}
\def_new:Npn \num_gadd:Nn {\pref_global:D \num_add:Nn}
\def_new:Npn \num_gadd:cn {\exp_args:Nc\num_gadd:Nn}
```

\num_use:N
\num_use:c

Here is how num macros are accessed:

```
\let_new:NN\num_use:N \use_arg_i:n
\let_new:NN\num_use:c \cs_use:c
```

\num_compare:nNnTF
\num_compare:nNnT
\num_compare:nNnF

Simple comparison tests.

```
\def_test_function_new:npn {num_compare:nNn}#1#2#3{
  \if_num:w \num_eval:n {#1}#2\num_eval:n {#3}
}
\def_new:Npn \num_compare:cNcTF { \exp_args:NcNc\num_compare:nNnTF }
```

\num_compare_p:nNn

A predicate function.

```
\def_new:Npn \num_compare_p:nNn #1#2#3{
  \if_num:w \num_eval:n {#1}#2\num_eval:n {#3}
    \c_true
  \else:
    \c_false
  \fi:
}
```

\num_max_of:nn
\num_min_of:nn
\num_abs:n

Functions for min, max, and absolute value.

```
\def_new:Npn \num_abs:n#1{
  \if_num:w \num_eval:n{#1}<\c_zero \exp_after:NN -\fi: #1
}
\def_new:Npn \num_max_of:nn#1#2{\num_compare:nNnTF {#1}>{#2}{#1}{#2}}
\def_new:Npn \num_min_of:nn#1#2{\num_compare:nNnTF {#1}<{#2}{#1}{#2}}
```

\num_elt_count:n
\num_elt_count_prop:Nn

Helper function for counting elements in a list.

```
\def_long_new:Npn \num_elt_count:n #1 { + 1 }
\def_long_new:Npn \num_elt_count_prop:Nn #1#2 { + 1 }
```

\l_tmpa_num
\l_tmpb_num
\l_tmpc_num
\g_tmpa_num
\g_tmpb_num

We provide an number local and two global ⟨*num*⟩s, maybe we need more or less.

```
\num_new:N \l_tmpa_num
\num_new:N \l_tmpb_num
\num_new:N \l_tmpc_num
\num_new:N \g_tmpa_num
\num_new:N \g_tmpb_num
```

9.5.1 Defining constants

As stated, most constants can be defined as `\tex_chardef:D` or `\tex_mathchardef:D` but that's engine dependent. Omega/Aleph allows `\tex_chardef:D`s up to 65535 which is also the maximum number of registers of all types.

```
\const_new:Nn
\c_max_register_num
\const_new_aux:Nw
```

```
%    \begin{macrocode}
\engine_if_aleph:TF
{
  \let_new:NN \const_new_aux:Nw \tex_chardef:D
  \const_new_aux:Nw \c_max_register_num = 65535 \scan_stop:
}
{
  \let_new:NN \const_new_aux:Nw \tex_mathchardef:D
  \const_new_aux:Nw \c_max_register_num = 32767 \scan_stop:
}
\def_new:Npn \const_new:Nn #1#2 {
  \num_compare:nNnTF  {#2} > \c_minus_one
  {
    \num_compare:nNnTF {#2} > \c_max_register_num
    {\int_new:N #1 \int_set:Nn #1{#2}}
    {\chk_new_cs:N #1 \const_new_aux:Nw #1 = #2 \scan_stop: }
  }
  {\int_new:N #1 \int_set:Nn #1{#2}}
}
```

```
\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_thirty_two
\c_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three
\c_ten_thousand_four
\c_twenty_thousand
```

And the usual constants, others are still missing. Please, make every constant a real constant at least for the moment. We can easily convert things in the end when we have found what constants are used in critical places and what not.

```
%% \tex_countdef:D \c_minus_one = 10 \scan_stop:
%% \c_minus_one = -1 \scan_stop:        %% in l3basics
%% \tex_chardef:D \c_sixteen    = 16\scan_stop: %% in l3basics
\const_new:Nn \c_zero   {0}
\const_new:Nn \c_one    {1}
\const_new:Nn \c_two    {2}
\const_new:Nn \c_three  {3}
\const_new:Nn \c_four   {4}
\const_new:Nn \c_six    {6}
\const_new:Nn \c_seven  {7}
\const_new:Nn \c_nine   {9}
\const_new:Nn \c_ten    {10}
\const_new:Nn \c_eleven {11}
\const_new:Nn \c_thirty_two {32}
```

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```
\const_new:Nn \c_hundred_one          {101}
\const_new:Nn \c_twohundred_fifty_five {255}
\const_new:Nn \c_twohundred_fifty_six {256}
\const_new:Nn \c_thousand             {1000}
```

```
\const_new:Nn \c_ten_thousand          {10000}
\const_new:Nn \c_ten_thousand_one      {10001}
\const_new:Nn \c_ten_thousand_two      {10002}
\const_new:Nn \c_ten_thousand_three    {10003}
\const_new:Nn \c_ten_thousand_four     {10004}
\const_new:Nn \c_twenty_thousand       {20000}
```

⟨/initex | package⟩

10 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.

10.1 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.

10.2 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.

10.3 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.

11 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.)

11.1 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*⟩!

11.2 Predicates and conditionals

Use `seq` functions.

11.3 Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package&!check⟩\RequirePackage{l3quark}
⟨package&!check⟩\RequirePackage{l3tlp}
⟨package & check⟩\RequirePackage{l3chk}
⟨package⟩\RequirePackage{l3expan}
```

A sequence is a control sequence whose top-level expansion is of the form '`\seq_elt:w` ⟨*text*$_1$⟩ `\seq_elt_end:` ... `\seq_elt:w` ⟨*text*$_n$⟩ ...'. We use explicit delimiters instead of braces around ⟨*text*⟩ to allow efficient searching for an item in the sequence.

`\seq_elt:w` `\seq_elt_end:` We allocate the delimiters and make them errors if executed.

```
⟨*initex | package⟩
\let_new:NN \seq_elt:w \ERROR
\let_new:NN \seq_elt_end: \ERROR
```

`\seq_new:N` `\seq_new:c` Sequences are implemented using token lists.

```
\def_new:Npn \seq_new:N #1{\tlp_new:Nn #1{}}
\def_new:Npn \seq_new:c {\exp_args:Nc \seq_new:N}
```

`\seq_clear:N` `\seq_clear:c` `\seq_gclear:N` `\seq_gclear:c` Clearing a sequence is the same as clearing a token list.

```
\let_new:NN \seq_clear:N \tlp_clear:N
\let_new:NN \seq_clear:c \tlp_clear:c
\let_new:NN \seq_gclear:N \tlp_gclear:N
\let_new:NN \seq_gclear:c \tlp_gclear:c
```

`\seq_clear_new:N` `\seq_clear_new:c` `\seq_gclear_new:N` `\seq_gclear_new:c` Clearing a sequence is the same as clearing a token list.

```
\let_new:NN \seq_clear_new:N \tlp_clear_new:N
\let_new:NN \seq_clear_new:c \tlp_clear_new:c
\let_new:NN \seq_gclear_new:N \tlp_gclear_new:N
\let_new:NN \seq_gclear_new:c \tlp_gclear_new:c
```

`\seq_if_empty_p:N` A predicate which evaluates to `\c_true` iff the sequence is empty.

```
\let_new:NN \seq_if_empty_p:N \tlp_if_empty_p:N
```

`\seq_if_empty:NTF` `\seq_if_empty:cTF` `\seq_if_empty:NF` `\seq_if_empty:cF` `\seq_if_empty:NTF`⟨*seq*⟩⟨*true case*⟩⟨*false case*⟩ will check whether the ⟨*seq*⟩ is empty and then select one of the other arguments. `seq_if_empty:cTF` turns its first argument into a control sequence to get the name of the sequence.

```
\let_new:NN \seq_if_empty:NTF \tlp_if_empty:NTF
\def_new:Npn \seq_if_empty:cTF {\exp_args:Nc\seq_if_empty:NTF}
```

A variant of this, is only to do something if the sequence is *not* empty.

```
\let_new:NN \seq_if_empty:NF \tlp_if_empty:NF
\def_new:Npn \seq_if_empty:cF {\exp_args:Nc\seq_if_empty:NF}
```

\seq_if_empty_err:N Signals an error if the sequence is empty.

```
\def_new:Npn \seq_if_empty_err:N #1{\if_meaning:NN#1\c_empty_tlp
```

As I said before, I don't think we need to provide checks for this kind of error, since it is a severe internal macro package error that can not be produced by the user directly. Can it? So the next line of code should be probably removed.

```
  \tlp_clear:N \l_testa_tlp % catch prefixes
  \err_latex_bug:x{Empty~sequence~`\token_to_string:N#1'}\fi:}
```

\seq_get:NN
\seq_get:cN
\seq_get:NN ⟨*sequence*⟩⟨*cmd*⟩ defines ⟨*cmd*⟩ to be the left˙most element of ⟨*sequence*⟩.

```
\def_new:Npn \seq_get:NN #1{
  \seq_if_empty_err:N #1
  \exp_after:NN\seq_get_aux:w #1\q_stop}
\def_new:Npn \seq_get_aux:w \seq_elt:w #1\seq_elt_end:
                #2\q_stop #3{\tlp_set:Nn #3{#1}}
\def_new:Npn \seq_get:cN {\exp_args:Nc \seq_get:NN}
```

\seq_pop_aux:nnNN
\seq_pop_aux:w
\seq_pop_aux:nnNN ⟨*def*$_1$⟩ ⟨*def*$_2$⟩ ⟨*sequence*⟩ ⟨*cmd*⟩ assigns the left˙most element of ⟨*sequence*⟩ to ⟨*cmd*⟩ using ⟨*def*$_2$⟩, and assigns the tail of ⟨*sequence*⟩ to ⟨*sequence*⟩ using ⟨*def*$_1$⟩.

```
\def_new:Npn \seq_pop_aux:nnNN #1#2#3{
  \seq_if_empty_err:N #3
  \exp_after:NN\seq_pop_aux:w #3\q_stop #1#2#3}
\def_new:Npn \seq_pop_aux:w \seq_elt:w #1\seq_elt_end:
                #2\q_stop #3#4#5#6{#3#5{#2}#4#6{#1}}
```

\seq_put_aux:Nnn \seq_put_aux:Nnn ⟨*sequence*⟩ ⟨*left*⟩ ⟨*right*⟩ adds the elements specified by ⟨*left*⟩ to the left of ⟨*sequence*⟩, and those specified by ⟨*right*⟩ to the right.

```
\def_new:Npn \seq_put_aux:Nnn #1{
  \exp_after:NN\seq_put_aux:w #1\q_stop #1}
\def_new:Npn \seq_put_aux:w #1\q_stop #2#3#4{\tlp_set:Nn #2{#3#1#4}}
```

\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
Here are the usual operations for adding to the left and right.

```
\def_new:Npn \seq_put_left:Nn #1#2{
```

We can't put in a \use_noop: instead of {} since this argument is passed literally (and we would end up with many \use_noop:s inside the sequences.

```
        \seq_put_aux:Nnn #1{\seq_elt:w #2\seq_elt_end:}{}}
\def_new:Npn \seq_put_left:cn {\exp_args:Nc\seq_put_left:Nn}
\def_new:Npn \seq_put_left:No {\exp_args:NNo\seq_put_left:Nn}
\def_new:Npn \seq_put_left:Nx {\exp_args:Nnx\seq_put_left:Nn}
\def_new:Npn \seq_put_right:Nn #1#2{
        \seq_put_aux:Nnn #1{}{\seq_elt:w #2\seq_elt_end:}}
\def_new:Npn \seq_put_right:No {\exp_args:NNo\seq_put_right:Nn}
\def_new:Npn \seq_put_right:Nx {\exp_args:NNx\seq_put_right:Nn}
```

\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

An here the global variants.

```
\def_new:Npn \seq_gput_left:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \seq_put_left:Nn}
\def_new:Npn \seq_gput_right:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \seq_put_right:Nn}
\def_new:Npn \seq_gput_right:No {\exp_args:NNo \seq_gput_right:Nn}
\def_new:Npn \seq_gput_right:Nc {\exp_args:NNc \seq_gput_right:Nn}
\def_new:Npn \seq_gput_right:cn {\exp_args:Nc  \seq_gput_right:Nn}
\def_new:Npn \seq_gput_right:co {\exp_args:Nco \seq_gput_right:Nn}
\def_new:Npn \seq_gput_right:cc {\exp_args:Ncc \seq_gput_right:Nn}
```

\seq_map_variable:NNn
\seq_map_variable:cNn
\seq_map_variable_aux:nw
\seq_map_break:w

Nothing spectacular here. The shuffling of the arguments in `\seq_map_variable:NNn` below could also be done with `\exp_args:NNnE`.

```
\def_new:Npn \seq_map_variable_aux:Nnw #1#2\seq_elt:w#3\seq_elt_end:{
  \tlp_set:Nn #1{#3}
  \quark_if_nil:NT #1 \seq_map_break:w
  #2
  \seq_map_variable_aux:Nnw #1{#2}
}
\def_new:Npn \seq_map_variable:NNn #1#2#3{
  \tlp_set:Nx #2 {\exp_not:n{\seq_map_variable_aux:Nnw #2{#3}}}
  \exp_after:NN #2 #1 \seq_elt:w \q_nil\seq_elt_end: \q_stop
}
\def_new:Npn \seq_map_variable:cNn{\exp_args:Nc\seq_map_variable:Nn}
\let_new:NN \seq_map_break:w \use_none_delimit_by_q_stop:w
```

\seq_map:NN

`\seq_map:NN` ⟨*sequence*⟩ ⟨*cmd*⟩ applies ⟨*cmd*⟩ to each element of ⟨*sequence*⟩, from left to right. Since we don't have braces, this implementation is not very efficient. It might be better to say that ⟨*cmd*⟩ must be a function with one argument that is delimited by `\seq_elt_end:`.

```
\def_new:Npn \seq_map:NN #1#2{
  \def:Npn \seq_elt:w ##1\seq_elt_end: {#2{##1}}#1
  \let:NN \seq_elt:w \ERROR
}
```

\seq_map_inline:Nn
\seq_map_inline:cn

When no braces are used, this version of mapping seems more natural.

```
\def_new:Npn \seq_map_inline:Nn #1#2{
  \def:Npn \seq_elt:w ##1\seq_elt_end: {#2}#1
  \let:NN \seq_elt:w \ERROR
}
\def_new:Npn \seq_map_inline:cn{\exp_args:Nc\seq_map_inline:Nn}
```

\seq_set_eq:NN
\seq_set_eq:Nc

We can set one seq equal to another.

```
\let_new:NN  \seq_set_eq:NN \let:NN
\def_new:Npn \seq_set_eq:Nc {\exp_args:NNc \seq_set_eq:NN}
```

\seq_gset_eq:NN
\seq_gset_eq:cN
\seq_gset_eq:Nc
\seq_gset_eq:cc

An of course globally which seems to be needed far more often.[8]

```
\let_new:NN \seq_gset_eq:NN \glet:NN
\def_new:Npn \seq_gset_eq:cN {\exp_args:Nc \seq_gset_eq:NN}
\def_new:Npn \seq_gset_eq:Nc {\exp_args:NNc \seq_gset_eq:NN}
\def_new:Npn \seq_gset_eq:cc {\exp_args:Ncc \seq_gset_eq:NN}
```

\seq_gconcat:NNN
\seq_gconcat:ccc

\seq_gconcat:NNN ⟨*seq 1*⟩ ⟨*seq 2*⟩ ⟨*seq 3*⟩ will globally assign ⟨*seq 1*⟩ the concatenation of ⟨*seq 2*⟩ and ⟨*seq 3*⟩.

```
\def_new:Npn \seq_gconcat:NNN #1#2#3{
  \tlp_gset:Nx #1 {\exp_not:o{#2}\exp_not:o{#3}}
}
\def_new:Npn \seq_gconcat:ccc{\exp_args:Nccc\seq_gconcat:NNN}
```

\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 ⟨*seq*⟩⟨*item*⟩ ⟨*true case*⟩ ⟨*false case*⟩ will check whether ⟨*item*⟩ is in ⟨*seq*⟩ and then either execute the ⟨*true case*⟩ or the ⟨*false case*⟩. ⟨*true case*⟩ and ⟨*false case*⟩ may contain incomplete \if_charcode:w statements.

```
\def_new:Npn \seq_if_in:NnTF #1#2{
  \def:Npn\tmp:w
      ##1\seq_elt:w #2\seq_elt_end: ##2##3\q_stop{
```

Note that ##2 contains exactly one token which we can compare with \q_no_value.

```
        \if_meaning:NN\q_no_value##2
          \exp_after:NN\use_arg_ii:nn
        \else:
          \exp_after:NN\use_arg_i:nn
        \fi:
      }
  \exp_after:NN
  \tmp:w #1\seq_elt:w
  #2\seq_elt_end: \q_no_value \q_stop}
\def_new:Npn \seq_if_in:coTF {\exp_args:Nco \seq_if_in:NnTF}
\def_new:Npn \seq_if_in:cnTF {\exp_args:Nc \seq_if_in:NnTF}
\def_new:Npn \seq_if_in:cxTF {\exp_args:Ncx \seq_if_in:NnTF}

\def_new:Npn \seq_if_in:NnF #1#2 { \seq_if_in:NnTF #1{#2}\use_noop: }
\def_new:Npn \seq_if_in:cnF {\exp_args:Nc \seq_if_in:NnF}
```

11.3.1 Stack operations

We build stacks from sequences, but here we put the specific functions together.

[8] To save a bit of space these functions could be made identical to those from the tlp or clist module.

\seq_push:Nn
\seq_push:No
\seq_push:cn
\seq_pop:NN
\seq_pop:cN

Since sequences can be used as stacks, we ought to have both 'push' and 'pop'. In most cases they are nothing more then new names for old functions.

```
\let_new:NN \seq_push:Nn \seq_put_left:Nn
\let_new:NN \seq_push:No \seq_put_left:No
\let_new:NN \seq_push:cn \seq_put_left:cn
\def_new:Npn \seq_pop:NN {\seq_pop_aux:nnNN \tlp_set:Nn \tlp_set:Nn}
\def_new:Npn \seq_pop:cN {\exp_args:Nc \seq_pop:NN}
```

\seq_gpush:Nn
\seq_gpush:No
\seq_gpush:cn
\seq_gpush:NC
\seq_gpop:NN
\seq_gpop:cN

I don't agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of \seq_gpop:NN the value is nevertheless returned locally.

```
\let_new:NN \seq_gpush:Nn \seq_gput_left:Nn
\def_new:Npn \seq_gpush:No {\exp_args:NNo \seq_gpush:Nn}
\def_new:Npn \seq_gpush:cn {\exp_args:Nc \seq_gpush:Nn}
\def_new:Npn \seq_gpush:NC {\exp_args:NNC \seq_gpush:Nn}
\def_new:Npn \seq_gpop:NN {\seq_pop_aux:nnNN \tlp_gset:Nn \tlp_set:Nn}
\def_new:Npn \seq_gpop:cN {\exp_args:Nc \seq_gpop:NN}
```

\seq_top:NN
\seq_top:cN

Looking at the top element of the stack without removing it is done with this operation.

```
\let_new:NN \seq_top:NN \seq_get:NN
\let_new:NN \seq_top:cN \seq_get:cN
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩
```

12 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.

12.1 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

12.2 The Implementation

We start by ensuring that the required packages are loaded when this file is loaded as a package on top of LaTeX 2ε.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3expan}
⟨package⟩\RequirePackage{l3num}
⟨package⟩\RequirePackage{l3seq}\par

⟨*initex | package⟩
```

`\alloc_setup_type:nnn` For each type of register we need to 'counters' that hold the last allocated global or local register. We also need a sequence to store the 'excpetions'.

```
\def_new:Npn \alloc_setup_type:nnn #1 #2 #3{
  \num_new:c  {g_ #1 _allocation_num}
  \num_new:c  {l_ #1 _allocation_num}
  \seq_new:c  {g_ #1 _allocation_seq}
  \num_set:cn {g_ #1 _allocation_num}{#2}
  \num_set:cn {l_ #1 _allocation_num}{#3}
}
```

`\alloc_next_g:n` `\alloc_next_l:n` These routines find the next free register. For globally allocated registers we first increment the counter that keeps track of them.

```
\def_new:Npn \alloc_next_g:n #1 {
  \num_gincr:c {g_ #1 _allocation_num}
```

Then we need to check whether we have run out of registers.

```
  \num_compare:cNcTF  {g_ #1 _allocation_num} = {l_ #1 _allocation_num}
    {\io_put_term:x{We~ ran~ out~ of~ registers~ of~ type~ g_#1!}}
    {
```

We also need to check wether the value of the counter already occurs in the list of already allocated registers.

```
      \seq_if_in:cxTF {g_ #1 _allocation_seq}
                      {\num_use:c{g_ #1 _allocation_num}}
        {\io_put_term:x{\num_use:c{g_ #1 _allocation_num}~Already~ allocated!}
```

If it does, we find the next value.

```
          \alloc_next_g:n {#1} }
        {\use_noop:}
        }
```

By now the `.._allocation_num` counter will contain the number of the register we will assign a control seuence for.

```
  }
```

For the locally allocated registers we have a similar function.

```
\def_new:Npn \alloc_next_l:n #1 {
  \num_gdecr:c {l_ #1 _allocation_num}
  \num_compare:cNcTF  {g_ #1 _allocation_num} = {l_ #1 _allocation_num}
    {\io_put_term:x{We~ ran~ out~ of~ registers~ of~ type~ l_#1!}}
    {
      \seq_if_in:cxTF {g_ #1 _allocation_seq}
                      {\num_use:c{l_ #1 _allocation_num}}
        {\io_put_term:x{\num_use:c{l_ #1 _allocation_num}~Already~ allocated!}
          \alloc_next_l:n {#1} }
        {\io_put_term:x{\num_use:c{l_ #1 _allocation_num}~Free!}}
      }
  }
```

`\alloc_reg:NnNN` This internal macro performs the actual allocation. It's first argument is either 'g' for a globally allocated register or 'l' for a locally allocated register. The second argument is the type of register to allocate, the third argument is the command to use and the fourth argument is the control sequence that is to be defined to point to the register.

```
\def_new:Npn \alloc_reg:NnNN #1 #2 #3 #4{
```

It first checks that the control sequence that is to denote the register does not already exist.

```
  \chk_new_cs:N #4
```

Next, it decides whether a prefix is needed for the allocation command;

```
  \if:w#1g
    \exp_after:NN \pref_global:D
  \fi:
```

And finally the actual allocation takes place.

```
  #3 #4 \num_use:c{#1_ #2 _allocation_num}
  %%\cs_record_meaning:N#1
```

All that's left to do is write a message in the log file.

```
  \io_put_log:x{
    \token_to_string:N#4=#2~register~\num_use:c{#1_ #2 _allocation_num}}
```

Finally, it calls `\alloc_next_<g/l>` to find the next free register number.

```
  \cs_use:c{alloc_next_#1:n} {#2}
 }
```

⟨∗showmemory⟩
\showMemUsage
⟨/showmemory⟩
⟨/initex | package⟩

13 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.

13.1 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.[9] 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*⟩ `}`

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.

[9] 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.

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.

13.2 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.

13.3 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.

13.4 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.

13.5 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package & check⟩\RequirePackage{l3chk}\par
⟨package⟩\RequirePackage{l3toks}\par
⟨*initex | package⟩
```

This section is primarily concerned with input and output streams. The naming conventions for i/o streams is `ior` (for read) and `iow` (for write) as module names. e.g. `\c_ior_test_stream` is an input stream variable called 'test'.

13.5.1 Output streams

`\iow_new:N` `\iow_new:c` Allocation of new output streams is done by these functions. As we currently do not distribute a new allocation module we nick the `\newwrite` function.

```
⟨*initex⟩
\alloc_setup_type:nnn {iow} \c_zero \c_sixteen
\def_new:Npn \iow_new:N #1 {\alloc_reg:NnNN g {iow} \tex_chardef:D #1}
⟨/initex⟩
⟨package⟩\let:NN \iow_new:N \newwrite
\def_new:Npn \iow_new:c {\exp_args:Nc \iow_new:N}
```

`\iow_open:Nn` `\iow_open:cn` To open streams for reading or writing the following two functions are provided. The streams are opened immediately.

From some bad experiences on the mainframe, I learned that it is better to force the close before opening a dataset for writing. We have to check whether this is also necessary in case of `\tex_openin:D`.

```
\def_new:Npn \iow_open:Nn #1#2{\iow_close:N #1
     \tex_immediate:D\tex_openout:D#1#2\scan_stop:}
\def_new:Npn \iow_open:cn {\exp_args:Nc \iow_open:Nn}
```

`\iow_close:N` Since we close output streams prior to opening, a separate closing operation is probably not necessary. But here it is, just in case.... Actually you will need this if you intend to write and then read in the same pass from some stream.

```
\def_new:Npn \iow_close:N {\tex_immediate:D\tex_closeout:D}
```

`\c_io_term_stream` `\c_iow_log_stream` Here we allocate two output streams for writing to the transcript file only (`\c_iow_log_stream`) and to both the terminal and transcript file (`\c_io_term_stream`). The latter can also be used to read from therefore it is called `..io_...`

```
\let_new:NN \c_io_term_stream \c_sixteen
\let_new:NN \c_iow_log_stream \c_minus_one
```

Immediate writing

`\iow_expanded:Nn` An abbreviation for an often used operation, which immediately writes its second argument to the output stream.

```
\def_new:Npn \iow_expanded:Nn {\tex_immediate:D\iow_deferred_expanded:Nn}
```

`\iow_unexpanded:Nn` This routine writes the second argument verbatim onto the output stream. If this stream isn't open, the output goes to the terminal. If the first argument is no output stream at all, we get an internal error.

```
\def_new:Npn \iow_unexpanded:Nn #1#2{
   \iow_expanded:Nn #1{\exp_not:n{#2}}}
```

\iow_expanded_log:n
\iow_expanded_term:n

Now we redefine two functions for which we needed a definition very early on. They both write their second argument fully expanded to the output stream.

```
\def:Npn \iow_expanded_log:n {\iow_expanded:Nn \c_iow_log_stream}
\def:Npn \iow_expanded_term:n{\iow_expanded:Nn \c_io_term_stream}
```

The second one isn't exactly equivalent to the old `\typeout` since we need to control expansion in the function we provide for the user.

\iow_unexpanded_term:n

This function writes its argument verbatim to the the terminal.

```
\def_new:Npn \iow_unexpanded_term:n {\iow_unexpanded:Nn \c_io_term_stream}
```

\iow_unexpanded_if_avail:Nn
\iow_unexpanded_if_avail:cn

`\iow_unexpanded_if_avail:Nn` ⟨*stream*⟩ ⟨*code*⟩. This routine writes its second argument unexpanded to the stream given by the first argument, provided that this stream was opened for writing. Note, that `#` characters get doubled within ⟨*code*⟩.

```
\def_new:Npn \iow_unexpanded_if_avail:Nn #1{
```

In this routine we have to check whether or not the output stream that was requested is defined at all. So we check if the name is still free.

```
  \cs_free:NTF #1\use_none:n {\iow_unexpanded:Nn #1}}
```

Note: the next function could be streamlined for speed if we use the faster `\cs_free:cTF`. (space viz time).

```
\def_new:Npn \iow_unexpanded_if_avail:cn {
        \exp_args:Nc \iow_unexpanded_if_avail:Nn }
```

\iow_long_expanded:Nn
\iow_long_unexpanded:Nn
\iow_long_expanded_aux:w

Another type of writing onto an output stream is used for potentially long token sequences. We break the output lines at every blank in the second argument. This avoids the problem of buffer overflow when reading back, or badly broken lines on systems with limited file records. The only thing we have to take care of, is the danger of two blanks in succession since these get converted into a `\par` when we read the stuff back. But this can happen only if things like two spaces find their way into the second argument. Usually, multiple spaces are removed by TEX's scanner.

```
\def_new:Npn \iow_long_expanded_aux:w #1#2#3{
   \group_begin:\tex_newlinechar:D`\ #1#2{#3}\group_end:}
\def_new:Npn \iow_long_expanded:Nn {\iow_long_expanded_aux:w
                                    \iow_expanded:Nn}
\def_new:Npn \iow_long_unexpanded:Nn {\iow_long_expanded_aux:w
                                      \iow_unexpanded:Nn}
```

Deferred writing With ε-TEX available deferred writing is easy. The comments below are old.

Deferred writing to output streams is a bit more complicated because there seems to be no nice hack for writing unexpanded. The only relatively sure bet is to use `\token_to_meaning:N` expansion of some token list. That's the way the following functions are implemented.

Another possibility would be to reserve a certain number of scratch token registers that could be used to hold the tokens until after the next \tex_shipout:D. But such an approach would probably fail because of the limited number of available token registers that would need to be reserved for this special application.

\iow_deferred_expanded:Nn First the easy part, this is the primitive.

```
\let:NN \iow_deferred_expanded:Nn \tex_write:D
```

\iow_deferred_unexpanded:Nn Now the harder part:

```
\def_new:Npn \iow_deferred_unexpanded:Nn #1#2{
  \iow_deferred_expanded:Nn{\exp_not:n{#2}}
}
%% Old implementation:
%\def_new:Npn \iow_deferred_unexpanded:Nn #1#2{
%  \tlp_set:Nn \l_tmpa_tlp {#2}
%  \tlp_set:Nx \l_tmpb_tlp
%              {\iow_deferred_expanded:Nn #1{\tlp_to_str:N \l_tmpa_tlp}}
%  \l_tmpb_tlp}
```

Long forms of these functions are not possible since the deferred writing will restore the value of \tex_newlinechar:D before it will have a chance to act. But on the other hand it is nevertheless possible to make all deferred writes long by setting the \tex_newlinechar:D inside the output routine just before the \tex_shipout:D. The only disadvantage of this method is the fact that messages to the terminal during this time will also then break at spaces. But we should consider this.

Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream.

```
\def_new:Npn \iow_newline: {^^J}
```

\c_iow_comment_char
\c_iow_lbrace_char
\c_iow_rbrace_char We also need to be able to write braces and the comment character. We achieve this by defining global constants to expand into a version of these characters with \tex_catcode:D = 12.

```
\tlp_new:Nx \c_iow_comment_char {\cs_to_str:N\%}
```

To avoid another allocation function which is probably only necessary here we use the \def:Npx command directly.

```
\tlp_new:Nx \c_iow_lbrace_char{\cs_to_str:N\{}
\tlp_new:Nx \c_iow_rbrace_char{\cs_to_str:N\}}
```

13.5.2 Input streams

`\ior_new:N` Allocation of new input streams is done by this function. As we currently do not distribute a new allocation module we nick the `\newwread` function.

```
⟨*initex⟩
\alloc_setup_type:nnn {ior} \c_zero \c_sixteen
\def_new:Npn \ior_new:N #1 {\alloc_reg:NnNN g {ior} \tex_chardef:D #1}
⟨/initex⟩
⟨package⟩\let:NN \ior_new:N \newread
```

`\ior_open:Nn` `\ior_close:Nn` Processing of input-streams (via `\tex_openin:D` and `closein`) is always 'immediate' as far as TEX is concerned. An extra `\tex_immediate:D` is silently ignored.

```
\let:NN \ior_close:N \tex_closein:D
\def_new:Npn \ior_open:Nn #1#2{\ior_close:N #1\scan_stop:
                               \tex_openin:D#1#2\scan_stop:}
```

`\ior_eof:NTF` `\ior_eof:NTF` ⟨*stream*⟩ ⟨*true case*⟩ ⟨*false case*⟩. To test if some particular input stream is exhausted the following conditional is provided:

```
\def_new:Npn \ior_eof:NTF #1{\if_eof:w#1
     \exp_after:NN\use_arg_i:nn \else:
     \exp_after:NN\use_arg_ii:nn \fi:}
```

`\ior_eof:NF` `\if_eof:w` `\ior_eof:NF` ⟨*stream*⟩ ⟨*false case*⟩. Do something if if there is still something to read from this file:

```
\let:NN \if_eof:w \tex_ifeof:D
\def_new:Npn \ior_eof:NF #1{\if_eof:w#1
       \exp_after:NN \use_none:nn \fi: \use_arg_i:n}
```

`\ior_to:NN` `\ior_gto:NN` And here we read from files.

```
⟨*check⟩
\def_new:Npn \ior_to:NN #1#2{\tex_read:D#1to#2
   \chk_local_or_pref_global:N #2}
⟨/check⟩
⟨−check⟩ \def_new:Npn \ior_to:NN #1{\tex_read:D#1to}
\def_new:Npn \ior_gto:NN {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \ior_to:NN}
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

14 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.

14.1 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.

14.2 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 2ε 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''.

14.3 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
`\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.

14.4 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*⟩.

14.5 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*⟩!

Use `clist` functions.

14.6 The Implementation

We start by ensuring that the required packages are loaded.

```
2357 ⟨package⟩\ProvidesExplPackage
2358 ⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
2359 ⟨*package⟩
```

```
\NeedsTeXFormat{LaTeX2e}
⟨!check⟩\RequirePackage{l3prg,l3quark}
⟨check⟩\RequirePackage{l3chk}
⟨/package⟩
⟨*initex | package⟩
```

`\clist_new:N` `\clist_new:c` Comma-Lists are implemented using token lists.

```
\def_new:Npn \clist_new:N #1{\tlp_new:Nn #1{}}
\def_new:Npn \clist_new:c {\exp_args:Nc \clist_new:N}
```

`\clist_clear:N` `\clist_clear:c` `\clist_gclear:N` `\clist_gclear:c` Clearing a comma-list is the same as clearing a token list.

```
\let_new:NN \clist_clear:N \tlp_clear:N
\let_new:NN \clist_clear:c \tlp_clear:c
\let_new:NN \clist_gclear:N \tlp_gclear:N
\let_new:NN \clist_gclear:c \tlp_gclear:c
```

`\clist_set_eq:NN` We can set one ⟨*clist*⟩ equal to another.

```
\let_new:NN \clist_set_eq:NN \let:NN
```

`\clist_set_eq:NN` `\clist_set_eq:cN` `\clist_set_eq:Nc` `\clist_set_eq:cc` An of course globally which seems to be needed far more often.

```
\let_new:NN \clist_gset_eq:NN \glet:NN
\def_new:Npn \clist_gset_eq:cN {\exp_args:Nc \clist_gset_eq:NN}
\def_new:Npn \clist_gset_eq:Nc {\exp_args:NNc \clist_gset_eq:NN}
\def_new:Npn \clist_gset_eq:cc {\exp_args:Ncc \clist_gset_eq:NN}
```

`\clist_if_empty_p:N` A predicate which evaluates to `\c_true` iff the comma-list is empty.

```
\let_new:NN \clist_if_empty_p:N \tlp_if_empty_p:N
```

`\clist_if_empty:NTF` `\clist_if_empty:NTF`⟨*clist*⟩⟨*true case*⟩⟨*false case*⟩ will check whether the ⟨*clist*⟩ is empty `\clist_if_empty:NT` and then select one of the other arguments. `\clist_if_empty:cTF` turns its first argu- `\clist_if_empty:NF` ment into a control comma-list to get the name of the comma-list. `\clist_if_empty:cTF` `\clist_if_empty:cT` `\clist_if_empty:cF`

```
\def_test_function_new:npn {clist_if_empty:N}#1{\if_meaning:NN#1\c_empty_tlp}
\def_new:Npn \clist_if_empty:cTF {\exp_args:Nc\clist_if_empty:NTF}
\def_new:Npn \clist_if_empty:cT {\exp_args:Nc\clist_if_empty:NT}
\def_new:Npn \clist_if_empty:cF {\exp_args:Nc\clist_if_empty:NF}
```

`\clist_if_empty_err:N` Signals an error if the comma-list is empty.

```
\def_new:Npn \clist_if_empty_err:N #1{
  \if_meaning:NN#1\c_empty_tlp
    \tlp_clear:N \l_testa_tlp % catch prefixes
    \err_latex_bug:x{Empty~comma-list~`\token_to_string:N#1'}
  \fi:}
```

`\clist_if_eq:NNTF` As comma lists are token list pointers internally this is just an alias.

```
\let_new:NN \clist_if_eq:NNTF \tlp_if_eq:NNTF
```

\clist_get:NN
\clist_get:cN

\clist_get:NN ⟨*comma-list*⟩⟨*cmd*⟩ defines ⟨*cmd*⟩ to be the left-most element of ⟨*comma-list*⟩.

```
2387 \def_new:Npn \clist_get:NN #1{
2388   \clist_if_empty_err:N #1
2389   \exp_after:NN\clist_get_aux:w #1,\q_stop}

2390 \def_new:Npn \clist_get_aux:w  #1,#2\q_stop #3{\tlp_set:Nn #3{#1}}

2391 \def_new:Npn \clist_get:cN {\exp_args:Nc \clist_get:NN}
```

\clist_pop_aux:nnNN
\clist_pop_aux:w
\clist_pop_auxi:w

\clist_pop_aux:nnNN ⟨*def*$_1$⟩ ⟨*def*$_2$⟩ ⟨*comma-list*⟩ ⟨*cmd*⟩ assigns the left-most element of ⟨*comma-list*⟩ to ⟨*cmd*⟩ using ⟨*def*$_2$⟩, and assigns the tail of ⟨*comma-list*⟩ to ⟨*comma-list*⟩ using ⟨*def*$_1$⟩.

```
2392 \def_new:Npn \clist_pop_aux:nnNN #1#2#3{
2393   \clist_if_empty_err:N #3
2394   \exp_after:NN\clist_pop_aux:w #3,\q_nil\q_stop #1#2#3}

2395 \def_new:Npn \clist_pop_aux:w  #1,#2\q_stop #3#4#5#6{
2396    #4#6{#1}
2397    #3#5{#2}
```

If there was only one element in the original clist, it now contains only `\q_nil`.

```
2398    \quark_if_nil:NTF #5
2399    { #3#5{} }
2400    { \clist_pop_auxi:w #2 #3#5 }
2401 }

2402 \def_new:Npn\clist_pop_auxi:w #1,\q_nil #2#3  {#2#3{#1}}
```

\clist_put_aux:NNnnNn

The generic put function.

```
2403 \def_new:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6{
```

When adding we have to distinguish between an empty ⟨*clist*⟩ and one that contains at least one item (otherwise we accumulate commas).

```
2404   \clist_if_empty:NTF#5 {#1 #5{#6}}
```

MH says: Perhaps we should make sure that empty arguments don't get on the stack as that is probably a mistake. That's what I've implemented here. Since `\tlist_if_empty:nF` is expandable prefixes are still allowed.

```
2405   { \tlist_if_empty:nF {#6}{ #2 #5{#3#6#4} } }
2406 }
```

\clist_put_left:Nn
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn

The operations for adding to the left.

```
2407 \def_new:Npn \clist_put_left:Nn {
2408   \clist_put_aux:NNnnNn \tlp_set:Nn \tlp_put_left:Nn {} ,
2409 }
2410 \def_new:Npn \clist_put_left:cn {\exp_args:Nc \clist_put_left:Nn}
2411 \def_new:Npn \clist_put_left:No {\exp_args:NNo\clist_put_left:Nn}
2412 \def_new:Npn \clist_put_left:Nx {\exp_args:Nnx\clist_put_left:Nn}
```

\clist_gput_left:Nn Global versions.

```
\def_new:Npn \clist_gput_left:Nn {
  \clist_put_aux:NNnnNn \tlp_gset:Nn \tlp_gput_left:Nn {} ,
}
```

\clist_put_right:Nn
\clist_put_right:cn
\clist_put_right:No
\clist_put_right:Nx
Adding something to the right side is almost the same.

```
\def_new:Npn \clist_put_right:Nn {
  \clist_put_aux:NNnnNn \tlp_set:Nn \tlp_put_right:Nn , {}
}
\def_new:Npn \clist_put_right:cn {\exp_args:Nc \clist_put_right:Nn}
\def_new:Npn \clist_put_right:No {\exp_args:Nno\clist_put_right:Nn}
\def_new:Npn \clist_put_right:Nx {\exp_args:Nnx\clist_put_right:Nn}
```

\clist_gput_right:Nn
\clist_gput_right:No
\clist_gput_right:cn
\clist_gput_right:co
\clist_gput_right:cc
\clist_gput_right:NC
An here the global variants.

```
\def_new:Npn \clist_gput_right:Nn {
  \clist_put_aux:NNnnNn \tlp_gset:Nn \tlp_gput_right:Nn , {}
}
\def_new:Npn \clist_gput_right:No {\exp_args:NNo \clist_gput_right:Nn}
\def_new:Npn \clist_gput_right:cn {\exp_args:Nc  \clist_gput_right:Nn}
\def_new:Npn \clist_gput_right:co {\exp_args:Nco \clist_gput_right:Nn}
\def_new:Npn \clist_gput_right:cc {\exp_args:Ncc \clist_gput_right:Nn}

\def_new:Npn \clist_gput_right:NC {\exp_args:NNC \clist_gput_right:Nn}
```

\clist_map_function:nN
\clist_map_function:cN
\clist_map_function:NN
\clist_map_function:NN ⟨*comma-list*⟩ ⟨*cmd*⟩ applies ⟨*cmd*⟩ to each element of ⟨*comma-list*⟩, from left to right.

```
\def_new:Npn \clist_map_function:NN #1#2{
  \clist_if_empty:NF #1
  {
    \exp_after:NN \clist_map_function_aux:Nw
    \exp_after:NN #2 #1 , \q_recursion_tail , \q_recursion_stop
  }
}
\def_new:Npn \clist_map_function:cN{\exp_args:Nc\clist_map_function:NN}
\def_new:Npn \clist_map_function:nN #1#2{
  \tlist_if_blank:nF {#1}
  { \clist_map_function_aux:Nw #2 #1 , \q_recursion_tail , \q_recursion_stop }
}
```

\clist_map_function_aux:Nw The general loop. Tests if we hit the first stop marker and exits if we did. If we didn't, place the function #1 in front of the element #2, which is surrounded by braces.

```
\def_long_new:Npn \clist_map_function_aux:Nw #1#2,{
  \quark_if_recursion_tail_stop:n{#2}
  #1{#2}
  \clist_map_function_aux:Nw #1
}
```

\clist_map_break:w The break statement is easy. Same as in other modules, gobble everything up to the special recursion stop marker.

```
\let_new:NN \clist_map_break:w \use_none_delimit_by_q_recursion_stop:w
```

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

The inline type is faster but not expandable. In order to make it nestable, we use a counter to keep track of the nesting level so that all of the functions called have distict names. A simpler approach would of course be to use grouping and thus the save stack but then you lose the ability to do things locally.

A funny little thing occured in one document: The command setting up the first call of \clist_map_inline:Nn was used in a tabular cell and the inline code used \\ so the loop broke as soon as this happened. Lesson to be learned from this: If you wish to have group like structure but not using the groupings of TEX, then do every operation globally.

```
\int_new:N \g_clist_inline_level_int
\def_long_new:Npn \clist_map_inline:Nn #1#2{
  \clist_if_empty:NF #1
  {
    \int_gincr:N \g_clist_inline_level_int
    \gdef_long:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    ##1{#2}
```

Recall that the E in \exp_args:NcE means 'single token expanded once and no braces added'. It is a lot more efficient to carry over the special function rather than constructing the same csname over and over again, so we just do it once. We reuse \clist_map_function_aux:Nw for the actual loop.

```
    \exp_args:NcE \clist_map_function_aux:Nw
    {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    #1 , \q_recursion_tail , \q_recursion_stop
    \int_gdecr:N \g_clist_inline_level_int
  }
}
\def_new:Npn \clist_map_inline:cn{\exp_args:Nc\clist_map_inline:Nn}
\def_long_new:Npn \clist_map_inline:nn #1#2{
  \tlist_if_empty:nF {#1}
  {
    \int_gincr:N \g_clist_inline_level_int
    \gdef_long:cpn {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    ##1{#2}
    \exp_args:Nc \clist_map_function_aux:Nw
    {clist_map_inline_ \int_use:N \g_clist_inline_level_int :n}
    #1 , \q_recursion_tail , \q_recursion_stop
    \int_gdecr:N \g_clist_inline_level_int
  }
}
```

\clist_map_variable:nNn
\clist_map_variable:NNn
\clist_map_variable:cNn

\clist_map:NNn ⟨*comma-list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\def_new:Npn \clist_map_variable:nNn #1#2#3{
  \tlist_if_empty:nF{#1}
```

```
  {
    \clist_map_variable_aux:Nnw #2 {#3} #1
    , \q_recursion_tail , \q_recursion_stop
  }
}
\def_new:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
\def_new:Npn \clist_map_variable:cNn {\exp_args:Nc \clist_map_variable:NNn}
```

`\clist_map_variable_aux:Nnw` The general loop. Assign the temp variable `#1` to the current item `#3` and then check if that's the stop marker. If it is, break the loop. If not, execute the action `#2` and continue.

```
\def_new:Npn \clist_map_variable_aux:Nnw #1#2#3,{
  \def:Npn #1{#3}
  \quark_if_recursion_tail_stop:N #1
  #2 \clist_map_variable_aux:Nnw #1{#2}
}
```

`\clist_concat_aux:NNNN` `\clist_concat:NNN` `\clist_gconcat:NNN` `\clist_gconcat:NNc` `\clist_gconcat:ccc` `\clist_gconcat:NNN` ⟨*clist 1*⟩ ⟨*clist 2*⟩ ⟨*clist 3*⟩ will globally assign ⟨*clist 1*⟩ the concatenation of ⟨*clist 2*⟩ and ⟨*clist 3*⟩.

```
\def_new:Npn \clist_concat_aux:NNNN #1#2#3#4{
  \toks_set:No \l_tmpa_toks {#3}
  \toks_set:No \l_tmpb_toks {#4}
  #1 #2 {
    \toks_use:N \l_tmpa_toks
```

Again the situation is a bit more complicated because of the use of commas between items, so if either list is empty we have to avoid adding a comma.

```
    \toks_if_empty:NF \l_tmpa_toks {\toks_if_empty:NF \l_tmpb_toks ,}
    \toks_use:N \l_tmpb_toks
  }
}
\def_new:Npn \clist_concat:NNN {\clist_concat_aux:NNNN \tlp_set:Nx}
\def_new:Npn \clist_gconcat:NNN {\clist_concat_aux:NNNN \tlp_gset:Nx}
```

And the usual versions.

```
\def_new:Npn \clist_gconcat:NNc{\exp_args:Nnnc\clist_gconcat:NNN}
\def_new:Npn \clist_gconcat:ccc{\exp_args:Nccc\clist_gconcat:NNN}
```

`list_remove_duplicates_aux:NN` `clist_remove_duplicates_aux:n` `\clist_remove_duplicates:N` `\clist_gremove_duplicates:N` Removing duplicate entries in a ⟨*clist*⟩ is fairly straight forward. We use a temporary variable and then go through the list from left to right. For each element check if the element is already present in the list.

```
\def:Npn \clist_remove_duplicates_aux:NN #1#2 {
  \clist_clear:N \l_clist_remove_duplicates_clist
  \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
  #1 #2 \l_clist_remove_duplicates_clist
}
\def:Npn \clist_remove_duplicates_aux:n #1 {
  \clist_if_in:NnTF \l_clist_remove_duplicates_clist {#1} {}
  {\clist_put_right:Nn \l_clist_remove_duplicates_clist {#1}}
}
```

The high level functions are just for telling if it should be a local or global setting.

```
\def_new:Npn \clist_remove_duplicates:N {
  \clist_remove_duplicates_aux:NN \clist_set_eq:NN
}
\def_new:Npn \clist_gremove_duplicates:N {
  \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
}
```

clist_remove_duplicates_clist

```
\clist_new:N \l_clist_remove_duplicates_clist
```

\clist_use:N
\clist_use:c

Using a ⟨*clist*⟩ is just executing it but...

```
\def_new:Npn \clist_use:N #1 {
  \if_meaning:NN #1 \scan_stop:
```

... if ⟨*clist*⟩ equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cs_end:` that was created by mistake somewhere.

```
    \err_latex_bug:x {Comma~list~ `\token_to_string:N #1'~
                      has~ an~ erroneous~ structure!}
  \else:
    \exp_after:NN #1
  \fi:
}
\def_new:Npn \clist_use:c {\exp_args:Nc \clist_use:N}
```

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

`\clist_if_in:NnTF` ⟨*clist*⟩⟨*item*⟩ ⟨*true case*⟩ ⟨*false case*⟩ will check whether ⟨*item*⟩ is in ⟨*clist*⟩ and then either execute the ⟨*true case*⟩ or the ⟨*false case*⟩. ⟨*true case*⟩ and ⟨*false case*⟩ may contain incomplete `\if_charcode:w` statements.

```
\def_new:Npn \clist_if_in:NnTF #1#2{
  \def:Npn \tmp:w ##1 ,#2, ##2##3\q_stop{
    \if_meaning:NN\q_no_value##2
      \exp_after:NN\use_arg_ii:nn
    \else:
      \exp_after:NN\use_arg_i:nn
    \fi:
  }
  \exp_after:NN \tmp:w
  \exp_after:NN , #1, #2, \q_no_value \q_stop
}
\def_new:Npn \clist_if_in:NoTF {\exp_args:NNo \clist_if_in:NnTF}
\def_new:Npn \clist_if_in:coTF {\exp_args:Nco \clist_if_in:NnTF}
\def_new:Npn \clist_if_in:cnTF {\exp_args:Nc \clist_if_in:NnTF}
```

14.6.1 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

\clist_push:Nn
\clist_push:No
\clist_push:cn
\clist_pop:NN
\clist_pop:cN

Since comma-lists can be used as stacks, we ought to have both 'push' and 'pop'. In most cases they are nothing more then new names for old functions.

```
\let_new:NN \clist_push:Nn \clist_put_left:Nn
\let_new:NN \clist_push:No \clist_put_left:No
\let_new:NN \clist_push:cn \clist_put_left:cn
\def_new:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tlp_set:Nn \tlp_set:Nn}
\def_new:Npn \clist_pop:cN {\exp_args:Nc \clist_pop:NN}
```

\clist_gpush:Nn
\clist_gpush:No
\clist_gpush:cn
\clist_gpop:NN
\clist_gpop:cN

I don't agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of \clist_gpop:NN the value is nevertheless returned locally.

```
\let_new:NN \clist_gpush:Nn \clist_gput_left:Nn
\def_new:Npn \clist_gpush:No {\exp_args:NNo \clist_gpush:Nn}
\def_new:Npn \clist_gpush:cn {\exp_args:Nc \clist_gpush:Nn}
\def_new:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tlp_gset:Nn \tlp_set:Nn}
\def_new:Npn \clist_gpop:cN {\exp_args:Nc \clist_gpop:NN}
```

\clist_top:NN
\clist_top:cN

Looking at the top element of the stack without removing it is done with this operation.

```
\let_new:NN \clist_top:NN \clist_get:NN
\let_new:NN \clist_top:cN \clist_get:cN
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩
```

15 Property lists

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

15.1 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:Nc
\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*⟩). Nestable.

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

15.2 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*⟩.

15.3 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.

`\g_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.

15.4 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3toks}\par
⟨package⟩\RequirePackage{l3quark}\par
⟨∗initex | package⟩
```

A property list is a token register whose contents is of the form '⟨*q˙prop*⟩key$_1$⟨*q˙prop*⟩⟨*val*$_1$⟩ ... ⟨*q˙prop*⟩key$_n$⟨*q˙prop*⟩⟨*valn*⟩'. The properties have to be single token, the values might be arbitrary token lists they get surrounded by braces.

\q_prop

```
\quark_new:N\q_prop
```

To get values from property-lists, token lists should be passed to the appropriate functions.

\prop_new:N
\prop_new:c

Property lists are implemented as token lists.

```
\def_new:Npn \prop_new:N #1{\toks_new:N #1}
\def_new:Npn \prop_new:c {\exp_args:Nc \prop_new:N}
```

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

The same goes for clearing a property list, either locally or globally.

```
\let_new:NN \prop_clear:N \toks_clear:N
\def_new:Npn \prop_clear:c {\exp_args:Nc\prop_clear:N}
\let_new:NN \prop_gclear:N \toks_gclear:N
\def_new:Npn \prop_gclear:c {\exp_args:Nc\prop_gclear:N}
```

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

```
\def_long_new:Npn \prop_split_aux:Nnn #1#2#3{
  \def:Npn \tmp:w ##1\q_prop#2\q_prop##2##3\q_stop {#3{##1}{##2}{##3}}
  \exp_after:NN\tmp:w \toks_use:N#1\q_prop#2\q_prop\q_no_value \q_stop
}
```

\prop_get:NnN
\prop_get:cnN
\prop_get_aux:w

\prop_get:NNN ⟨*plist*⟩⟨*prop*⟩⟨*tlp*⟩ defines ⟨*tlp*⟩ to be the value associated with ⟨*prop*⟩ in ⟨*plist*⟩, \q_no_value if not found.

```
\def_long_new:NNn \prop_get:NnN 2{
  \prop_split_aux:Nnn #1{#2}\prop_get_aux:w}
\def_long_new:NNn \prop_get_aux:w 4{\tlp_set:Nx#4{\exp_not:n{#2}}}
\def_new:Npn \prop_get:cnN { \exp_args:Nc \prop_get:NnN }
```

\prop_gget:NnN
\prop_gget:NcN
\prop_gget:cnN
\prop_gget_aux:w

The global version of the previous function.

```
\def_long_new:NNn \prop_gget:NnN 2{
  \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
\def_new:Npn \prop_gget:NcN {\exp_args:NNc \prop_gget:NnN}
\def_new:Npn \prop_gget:cnN {\exp_args:Nc \prop_gget:NnN}
\def_long_new:NNn \prop_gget_aux:w 4{\tlp_gset:Nx#4{\exp_not:n{#2}}}
```

\prop_get_gdel:NNN
\prop_get_del_aux:w

\prop_get_gdel:NNN is the same as \prop_get:NNN but the ⟨*property˙key*⟩ and its value are afterwards globally removed from ⟨*property˙list*⟩. One probably also needs the local variants or only the local one, or... We decide this later.

```
\def_long_new:NNn \prop_get_gdel:NnN 3{
  \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1}{#2}}}
\def_long_new:NNn \prop_get_del_aux:w 6{
  \tlp_set:Nx #1{\exp_not:n{#5}}
  \quark_if_no_value:NF #1 {
    \def:Npn \tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
    \tmp:w #6}
}
```

\prop_put:Nnn
\prop_put:ccn
\prop_gput:Nnn
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:Nox
\prop_gput:Noo
\prop_gput:Ncn
\prop_gput:Ooo
\prop_gput:cNn
\prop_gput:ccn
\prop_gput:cco
\prop_gput:ccx
\prop_put_aux:w

\prop_put:Nnn ⟨*plist*⟩⟨*prop*⟩⟨*val*⟩ adds/changes the value associated with ⟨*prop*⟩ in ⟨*plist*⟩ to ⟨*val*⟩.

```
\def_long_new:NNn \prop_put:Nnn 2{
  \prop_split_aux:Nnn #1{#2}{
    \prop_clear:N #1
    \prop_put_aux:w {\toks_put_right:Nn #1}{#2}}
}
\def_new:Npn \prop_put:ccn {\exp_args:Ncc \prop_put:Nnn }

\def_long_new:NNn \prop_gput:Nnn 2{
  \prop_split_aux:Nnn #1{#2}{
    \prop_gclear:N #1
    \prop_put_aux:w {\toks_gput_right:Nn #1}{#2}}
}

\def_long_new:NNn \prop_put_aux:w 6{
  #1{\q_prop#2\q_prop{#6}#3}
  \tlist_if_empty:nF{#5}
  {
    \def:Npn \tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
    \tmp:w #5
  }
}
\def_new:Npn \prop_gput:Nno {\exp_args:NNno \prop_gput:Nnn}
```

```
\def_new:Npn \prop_gput:Nnx {\exp_args:NNnx \prop_gput:Nnn}
\def_new:Npn \prop_gput:Nox {\exp_args:NNox \prop_gput:Nnn}
\def_new:Npn \prop_gput:Noo {\exp_args:NNoo \prop_gput:Nnn}
\def_new:Npn \prop_gput:Ncn {\exp_args:NNc  \prop_gput:Nnn}
\def_new:Npn \prop_gput:Ooo {\exp_args:NOoo \prop_gput:Nnn}
\def_new:Npn \prop_gput:cnn {\exp_args:Nc   \prop_gput:Nnn}
\def_new:Npn \prop_gput:ccn {\exp_args:Ncc  \prop_gput:Nnn}
\def_new:Npn \prop_gput:cco {\exp_args:Ncco \prop_gput:Nnn}
\def_new:Npn \prop_gput:ccx {\exp_args:Nccx \prop_gput:Nnn}
```

`\prop_del:Nn` `\prop_gdel:Nn` `\prop_del_aux:w`

`\prop_del:NN` ⟨*plist*⟩⟨*prop*⟩ deletes the entry for ⟨*prop*⟩ in ⟨*plist*⟩, if any.

```
\def_long_new:NNn \prop_del:Nn 2{
  \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}}
\def_long_new:NNn \prop_gdel:Nn 2{
  \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}}
\def_long_new:NNn \prop_del_aux:w 5{
   \def:Npn \tmp:w {#4}
   \quark_if_no_value:NF \tmp:w
     {\def:Npn \tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{#3##1}}
      \tmp:w #5}}
```

`\prop_if_in:NnTF` `\prop_if_in:NoTF` `\prop_if_in:ccTF` `\prop_if_in_aux:w`

`\prop_if_in:NNTF` ⟨*property˙list*⟩ ⟨*property˙key*⟩ ⟨*true˙case*⟩ ⟨*false˙case*⟩ will check whether or not ⟨*property˙key*⟩ is on the ⟨*property˙list*⟩ and then select either the true or false case.

```
\def_new:NNn \prop_if_in:NnTF 2{
  \prop_split_aux:Nnn #1{#2}\prop_if_in_aux:w}
\def_new:NNn \prop_if_in_aux:w 3{\quark_if_no_value:nFT {#2}}

\def_new:Npn \prop_if_in:NoTF {\exp_args:NNo \prop_if_in:NnTF}
\def_new:Npn \prop_if_in:ccTF {\exp_args:Ncc \prop_if_in:NnTF}
```

`\prop_gput_if_new:Nnn`

`\prop_gput_if_new:NNn` ⟨*property˙list*⟩ ⟨*property˙key*⟩ ⟨*property˙value*⟩ is equivalent to

```
\prop_if_in:NNTF ⟨property⟩⟨property˙key⟩
     {}%
     {\prop_gput:Nnn
          ⟨property˙list⟩
          ⟨property˙key⟩
          ⟨property˙value⟩}
```

Here we go (listening to Porgy & Bess in a recording with Ella F. and Louis A. which makes writing macros sometimes difficult; I find myself humming instead of working):

```
\def_long_new:NNn \prop_gput_if_new:Nnn 2{
  \prop_split_aux:Nnn #1{#2}{\prop_put_if_new_aux:w #1{#2}}}
\def_long_new:NNn \prop_put_if_new_aux:w 6{
  \tlist_if_empty:nT {#5}{#1{\q_prop#2\q_prop{#6}#3}}}
```

`\prop_set_eq:NN` `\prop_set_eq:Nc` `\prop_set_eq:cN` `\prop_set_eq:cc` `\prop_gset_eq:NN` `\prop_gset_eq:Nc` `\prop_gset_eq:cN` `\prop_gset_eq:cc`

This makes two ⟨*plist*⟩s have the same contents.

```
\let_new:NN \prop_set_eq:NN  \toks_set_eq:NN
```

```
\let_new:NN \prop_set_eq:Nc  \toks_set_eq:Nc
\let_new:NN \prop_set_eq:cN  \toks_set_eq:cN
\let_new:NN \prop_set_eq:cc  \toks_set_eq:cc
\let_new:NN \prop_gset_eq:NN \toks_gset_eq:NN
\let_new:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
\let_new:NN \prop_gset_eq:cN \toks_gset_eq:cN
\let_new:NN \prop_gset_eq:cc \toks_gset_eq:cc
```

`\prop_if_empty_p:N` `\prop_if_empty_p:c` `\prop_if_empty:NTF` `\prop_if_empty:NT` `\prop_if_empty:NF` `\prop_if_empty:cTF` `\prop_if_empty:cT` `\prop_if_empty:cF`

This conditional takes a ⟨*plist*⟩ as its argument and evaluates either the true or the false case, depending on whether or not ⟨*plist*⟩ contains any properties.

```
\let_new:NN \prop_if_empty_p:N \toks_if_empty_p:N
\let_new:NN \prop_if_empty_p:c \toks_if_empty_p:c
\let_new:NN \prop_if_empty:NTF \toks_if_empty:NTF
\let_new:NN \prop_if_empty:NT  \toks_if_empty:NT
\let_new:NN \prop_if_empty:NF  \toks_if_empty:NF
\let_new:NN \prop_if_empty:cTF \toks_if_empty:cTF
\let_new:NN \prop_if_empty:cT  \toks_if_empty:cTF
\let_new:NN \prop_if_empty:cF  \toks_if_empty:cF
```

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

This function test whether the property lists that are in its first two arguments are equal; if they are not `#3` is executed.

```
\def_new:NNn \prop_if_eq:NNTF 2 {
  \tlist_if_eq:xxTF{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:NNn \prop_if_eq:NNT 2 {
  \tlist_if_eq:xxT{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:NNn \prop_if_eq:NNF 2 {
  \tlist_if_eq:xxF{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:Npn \prop_if_eq:NcTF {\exp_args:NNc \prop_if_eq:NNTF}
\def_new:Npn \prop_if_eq:NcT {\exp_args:NNc \prop_if_eq:NNT}
\def_new:Npn \prop_if_eq:NcF {\exp_args:NNc \prop_if_eq:NNF}
\def_new:Npn \prop_if_eq:cNTF {\exp_args:Nc \prop_if_eq:NNTF}
\def_new:Npn \prop_if_eq:cNT {\exp_args:Nc \prop_if_eq:NNT}
\def_new:Npn \prop_if_eq:cNF {\exp_args:Nc \prop_if_eq:NNF}
\def_new:Npn \prop_if_eq:ccTF {\exp_args:Ncc \prop_if_eq:NNTF}
\def_new:Npn \prop_if_eq:ccT {\exp_args:Ncc \prop_if_eq:NNT}
\def_new:Npn \prop_if_eq:ccF {\exp_args:Ncc \prop_if_eq:NNF}
```

`\prop_map_function:NN` `\prop_map_function:cN` `\prop_map_function:Nc` `\prop_map_function:cc` `\prop_map_function_aux:w`

Maps a function on every entry in the property list. The function must take 2 arguments: a key and a value.

```
\def_new:Npn \prop_map_function:NN #1#2{
  \exp_after:NN \prop_map_function_aux:w
  \exp_after:NN #2 \toks_use:N #1 \q_prop{}\q_prop \q_no_value \q_stop
}
\def_new:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \if:w \tlist_if_empty_p:n{#2}
    \exp_after:NN \prop_map_break:w
  \fi:
```

```
  #1{#2}{#3}
  \prop_map_function_aux:w #1
}
% problem with the above impl, is that an empty key stops the mapping but all
% other functions in the module allow the use of empty keys (as one value)

\def:Npn \prop_map_function:NN #1#2{
  \exp_after:NN \prop_map_function_aux:w
  \exp_after:NN #2 \toks_use:N #1 \q_prop \q_no_value \q_prop \q_no_value
}
\def:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \quark_if_no_value:nF{#2}
    {
     #1{#2}{#3}
     \prop_map_function_aux:w #1
    }
}
% problem with the above impl is that \quark_if_no_value:nF is fairly slow and
%  if \quark_if_no_value:NF is used instead we have to do an assignment thus
%  making the mapping not expandable (is that important?)

\def:Npn \prop_map_function:NN #1#2{
    \exp_after:NN \prop_map_function_aux:w
    \exp_after:NN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
}
\def:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \if_meaning:NN \q_nil #2
    \exp_after:NN \prop_map_break:w
  \fi:
  #1{#2}{#3}
  \prop_map_function_aux:w #1
}

% (potential) problem with the above impl is that it will returns true is #2
% contains more than just \q_nil thus executing whatever follows. Claim: this
% can't happen :-) so we should be ok

\def_new:Npn \prop_map_function:cN {\exp_args:Nc \prop_map_function:NN }
\def_new:Npn \prop_map_function:Nc {\exp_args:NNc \prop_map_function:NN }
\def_new:Npn \prop_map_function:cc {\exp_args:Ncc \prop_map_function:NN}
```

\prop_map_inline:Nn
\prop_map_inline:cn
\g_prop_inline_level_num

The inline functions are straight forward. It takes longer to test if the list is empty than to run it on an empty list so we don't waste time doing that.

```
\num_new:N \g_prop_inline_level_num
\def_new:Npn \prop_map_inline:Nn #1#2 {
  \num_gincr:N \g_prop_inline_level_num
  \gdef_long:cpn {prop_map_inline_ \num_use:N \g_prop_inline_level_num :n}
    ##1##2{#2}
  \prop_map_function:Nc #1
    {prop_map_inline_ \num_use:N \g_prop_inline_level_num :n}
  \num_gdecr:N \g_prop_inline_level_num
}
\def_new:Npn \prop_map_inline:cN { \exp_args:Nc \prop_map_inline:NN }
```

`\prop_map_break:w` The break statement.

```
\let_new:NN \prop_map_break:w \use_none_delimit_by_q_stop:w
```

Finally a bunch of compatibility commands with the old syntax: they will vanish soon!

```
\def:Npn \prop_put:NNn {\typeout{Warning:~name~
                    changed~ to~ \string\prop_put:Nnn}\prop_put:Nnn}
\def:Npn \prop_gput:NNn  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gput:Nnn }\prop_gput:Nnn }
\def:Npn \prop_gput:NNo  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gput:Nno }\prop_gput:Nno }
\def:Npn \prop_gput:cNn  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gput:cnn }\prop_gput:cnn }
\def:Npn \prop_gput_if_new:NNn  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gput_if_new:Nnn }\prop_gput_if_new:Nnn }
\def:Npn \prop_get:NNN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_get:NnN }\prop_get:NnN }
\def:Npn \prop_get:cNN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_get:cnN }\prop_get:cnN }
\def:Npn \prop_gget:NNN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gget:NnN }\prop_gget:NnN }
\def:Npn \prop_gget:cNN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gget:cnN }\prop_gget:cnN }
\def:Npn \prop_get_gdel:NNN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_get_gdel:NnN }\prop_get_gdel:NnN }
\def:Npn \prop_del:NN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_del:Nn }\prop_del:Nn }
\def:Npn \prop_gdel:NN  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_gdel:Nn }\prop_gdel:Nn }
\def:Npn \prop_if_in:NNTF  {\typeout{Warning:~name~
                    changed~ to~ \string\prop_if_in:NnTF }\prop_if_in:NnTF }
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
%\showMemUsage
⟨/showmemory⟩

⟨*unused⟩
\file_input_stop:

⟨/unused⟩
```

16 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.

16.1 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_incr:c`
`\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_decr:c`
`\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_gadd:cn` `\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_sub:cn
\int_gsub:Nn
\int_gsub:cn

\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.

16.2 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.

16.2.1 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.

16.3 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.

16.4 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_max_of:nn`
`\int_min_of:nn`
`\int_max_of:nn` {⟨*int expr*⟩} {⟨*int expr*⟩}

Return the largest or smallest of two integer expressions.

`\int_abs:n` `\int_abs:n` {⟨*int expr*⟩}

Return the numerical value of an integer expression.

`\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.

16.5 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.

16.6 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package&!check⟩\RequirePackage{l3num}
⟨package & check⟩\RequirePackage{l3chk}
⟨∗initex | package⟩
```

\int_to_roman:w
\int_to_number:w
\int_advance:w

A new name for the primitives.

```
\let_new:NN \int_to_roman:w \tex_romannumeral:D
\let_new:NN \int_to_number:w \tex_number:D
\let_new:NN \int_advance:w \tex_advance:D
```

Functions that support LaTeX's user accessible counters should be added here, too. But first the internal counters.

\int_incr:N
\int_decr:N
\int_gincr:N
\int_gdecr:N
\int_incr:c
\int_decr:c
\int_gincr:c
\int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

```
\def_new:Npn \int_incr:N #1{\int_advance:w#1\c_one
⟨∗check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
⟨∗check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \int_gincr:N {
```

We make sure that a local variable is not updated globally by changing the internal test (i.e. \chk_local_or_pref_global:N) before making the assignment. This is done by \pref_global_chk: which also issues the necessary \pref_global:D. This is

not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \int_incr:N}
\def_new:Npn \int_gdecr:N {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \int_decr:N}
```

With the `\int_add:Nn` functions we can shorten the above code. If this makes it too slow ...

```
\def:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
\def:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
\def:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
\def:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}
\def:Npn \int_incr:c {\exp_args:Nc\int_incr:N}
\def:Npn \int_decr:c {\exp_args:Nc\int_decr:N}
\def:Npn \int_gincr:c {\exp_args:Nc\int_gincr:N}
\def:Npn \int_gdecr:c {\exp_args:Nc\int_gdecr:N}
```

`\int_new:N` `\int_new_l:N` `\int_new:c` Allocation of a new internal counter is already done above. Here we define the next likely variant.

```
⟨*initex⟩
\alloc_setup_type:nnn {int} \c_eleven \c_max_register_num
\def_new:Npn \int_new:N   #1 {\alloc_reg:NnNN g {int} \tex_countdef:D#1}
\def_new:Npn \int_new_l:N #1 {\alloc_reg:NnNN l {int} \tex_countdef:D#1}
⟨/initex⟩
⟨package⟩\let:NN \int_new:N \newcount% allocation better nick the LaTeX one...
\def_new:Npn \int_new:c {\exp_args:Nc \int_new:N}
```

`\int_set:Nn` `\int_set:cn` `\int_gset:Nn` `\int_gset:cn` Setting counters is again something that I would like to make uniform at the moment to get a better overview.

```
\def_new:Npn \int_set:Nn #1#2{#1 \int_eval:w #2\int_eval_end:
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \int_gset:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \int_set:Nn }
\def_new:Npn \int_set:cn {\exp_args:Nc \int_set:Nn }
\def_new:Npn \int_gset:cn {\exp_args:Nc \int_gset:Nn }
```

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

Functions that reset an ⟨*int*⟩ register to zero.

```
\def_new:Npn \int_zero:N  #1 {#1=\c_zero}
\def_new:Npn \int_zero:c  #1 {\exp_args:Nc \int_zero:N}
\def_new:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
\def_new:Npn \int_gzero:c {\exp_args:Nc \int_gzero:N}
```

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Adding and substracting to and from a counter . . . We should think of using these functions

```
\def_new:Npn \int_add:Nn #1#2{
```

We need to say by in case the first argument is a register accessed by its number, e.g., \count23. Not that it should ever happen but. . .

```
    \int_advance:w #1 by \int_eval:w #2\int_eval_end:
⟨*check⟩
    \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn\int_add:cn{\exp_args:Nc\int_add:Nn}
\def_new:Npn \int_sub:Nn #1#2{
    \int_advance:w #1-\int_eval:w #2\int_eval_end:
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \int_gadd:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \int_add:Nn }
\def_new:Npn \int_gsub:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \int_sub:Nn }
\def_new:Npn \int_gadd:cn{\exp_args:Nc\int_gadd:Nn}
\def_new:Npn \int_sub:cn{\exp_args:Nc\int_sub:Nn}
\def_new:Npn \int_gsub:cn{\exp_args:Nc\int_gsub:Nn}
```

\int_use:N
\int_use:c

Here is how counters are accessed:

```
\let_new:NN \int_use:N \tex_the:D
\def_new:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}
```

\int_to_arabic:n

Nothing exciting here.

```
\def_new:Npn \int_to_arabic:n #1{\int_to_number:w \int_eval:n{#1}}
```

\int_roman_lcuc_mapping:Nnn

Using TEX's built-in feature for producing roman numerals has some surprising features. One is the the characters resulting from \int_to_roman:w have category code 12 so they

may fail in certain comparison tests. Therefore we use a mapping from the character TEX produces to the character we actually want which will give us letters with category code 11.

```
\def_new:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
  \def:cpn {int_to_lc_roman_#1:}{#2}
  \def:cpn {int_to_uc_roman_#1:}{#3}
}
```

Here are the default mappings. I haven't found any examples of say Turkish doing the mapping `i \i I` but at least there is a possibility for it if needed. Note: I have now asked a Turkish person and he tells me they do the `i I` mapping.

```
\int_roman_lcuc_mapping:Nnn i i I
\int_roman_lcuc_mapping:Nnn v v V
\int_roman_lcuc_mapping:Nnn x x X
\int_roman_lcuc_mapping:Nnn l l L
\int_roman_lcuc_mapping:Nnn c c C
\int_roman_lcuc_mapping:Nnn d d D
\int_roman_lcuc_mapping:Nnn m m M
```

For the delimiter we cheat and let it gobble its arguments instead.

```
\int_roman_lcuc_mapping:Nnn Q \use_none:nn \use_none:nn
```

`\int_to_roman:n` `\int_to_Roman:n` `\int_to_roman_lcuc:NN`

The commands for producing the lower and upper case roman numerals run a loop on one character at a time and also carries some information for upper or lower case with it. We put it through `\int_eval:n` first which is safer and more flexible.

```
\def_new:Npn \int_to_roman:n #1 {
  \exp_after:NN \int_to_roman_lcuc:NN \exp_after:NN l
    \int_to_roman:w \int_eval:n {#1} Q
}
\def_new:Npn \int_to_Roman:n #1 {
  \exp_after:NN \int_to_roman_lcuc:NN \exp_after:NN u
    \int_to_roman:w \int_eval:n {#1} Q
}
\def_new:Npn \int_to_roman_lcuc:NN #1#2{
  \cs_use:c {int_to_#1c_roman_#2:}
  \int_to_roman_lcuc:NN #1
}
```

`_convert_number_with_rule:nnN`

This is our major workhorse for conversions. `#1` is the number we want converted, `#2` is the base number, and `#3` is the function converting the number. This function expects to receive a non-negative integer and as such is ideal for something using `\if_case:w` internally.

The basic example is this: We want to convert the number 50 (`#1`) into an alphabetic equivalent `ax`. For the English language our list contains 26 elements so this is our argument `#2` while the function `#3` just turns `1` into `a`, `2` into `b`, etc. Hence our goal is to turn 50 into the sequence `#3{1}#1{24}` so what we do is to first divide 50 by 26 and truncating the result returning 1. Then before we execute this we call the function again but this time on the result of the remainder of the division. This goes on until the

remainder is less than or equal to the base number where we just call the function `#3` directly on the number.

We do a little pre-expansion of the arguments below as they otherwise have a tendency to grow quite large.

```
\def:Npn \int_convert_number_with_rule:nnN #1#2#3{
  \int_compare:nNnTF {#1}>{#2}
  {
    \exp_args:No \int_convert_number_with_rule:nnN
      { \int_use:N\int_div_truncate:nn {#1-1}{#2} }{#2}
      #3
```

Note that we have to nudge our modulus function so it won't return 0 as that wouldn't work with `\if_case:w` when that expects a positive number to produce a letter.

```
    \exp_args:No #3 { \int_use:N\int_eval:n{1+\int_mod:nn {#1-1}{#2}} }
  }
  { \exp_args:No #3{ \int_use:N\int_eval:n{#1} } }
}
```

As can be seen it is even simpler to convert to number systems that contain 0, since then we don't have to add or subtract 1 here and there.

`lph_default_conversion_rule:n` `lph_default_conversion_rule:n` Now we just set up a default conversion rule. Ideally every language should have one such rule, as say in Danish there are 29 letters in the alphabet.

```
\def_new:Npn \int_alph_default_conversion_rule:n #1{
  \if_case:w #1
    \or: a\or: b\or: c\or: d\or: e\or: f
    \or: g\or: h\or: i\or: j\or: k\or: l
    \or: m\or: n\or: o\or: p\or: q\or: r
    \or: s\or: t\or: u\or: v\or: w\or: x
    \or: y\or: z
  \fi:
}
\def_new:Npn \int_Alph_default_conversion_rule:n #1{
  \if_case:w #1
    \or: A\or: B\or: C\or: D\or: E\or: F
    \or: G\or: H\or: I\or: J\or: K\or: L
    \or: M\or: N\or: O\or: P\or: Q\or: R
    \or: S\or: T\or: U\or: V\or: W\or: X
    \or: Y\or: Z
  \fi:
}
```

`\int_to_alph:n` `\int_to_Alph:n` The actual functions are just instances of the generic function. The second argument of `\int_convert_number_with_rule:nnN` should of course match the number of `\or:`s in the conversion rule.

```
\def_new:Npn \int_to_alph:n #1{
  \int_convert_number_with_rule:nnN {#1}{26}
    \int_alph_default_conversion_rule:n
}
```

```
\def_new:Npn \int_to_Alph:n #1{
  \int_convert_number_with_rule:nnN {#1}{26}
    \int_Alph_default_conversion_rule:n
}
```

`\int_to_symbol:n` Turning a number into a symbol is also easy enough.

```
\def_new:Npn \int_to_symbol:n #1{
  \mode_if_math:TF
  {
    \int_convert_number_with_rule:nnN {#1}{9}
      \int_symbol_math_conversion_rule:n
  }
  {
    \int_convert_number_with_rule:nnN {#1}{9}
      \int_symbol_text_conversion_rule:n
  }
}
```

`symbol_math_conversion_rule:n` `symbol_text_conversion_rule:n` Nothing spectacular here.

```
\def_new:Npn \int_symbol_math_conversion_rule:n #1 {
  \if_case:w #1
    \or: *
    \or: \dagger
    \or: \ddagger
    \or: \mathsection
    \or: \mathparagraph
    \or: \|
    \or: **
    \or: \dagger\dagger
    \or: \ddagger\ddagger
  \fi:
}
\def_new:Npn \int_symbol_text_conversion_rule:n #1 {
  \if_case:w #1
    \or: \textasteriskcentered
    \or: \textdagger
    \or: \textdaggerdbl
    \or: \textsection
    \or: \textparagraph
    \or: \textbardbl
    \or: \textasteriskcentered\textasteriskcentered
    \or: \textdagger\textdagger
    \or: \textdaggerdbl\textdaggerdbl
  \fi:
}
```

`\l_tmpa_int` `\l_tmpb_int` `\l_tmpc_int` `\g_tmpa_int` `\g_tmpb_int` `\l_loop_int` We provide four local and two global scratch counters, maybe we need more or less.

```
\int_new:N \l_tmpa_int
\int_new:N \l_tmpb_int
\int_new:N \l_tmpc_int
\int_new:N \g_tmpa_int
```

```
\int_new:N \g_tmpb_int
\int_new:N \l_loop_int  % a variable for use in loops (whilenum etc)
```

`\int_eval:n` `\int_eval:w` `\int_eval_end:` Evaluating a calc expression using normal operators. Many of these are exactly the same as the ones in the num module so we just use them.

```
\let_new:NN \int_eval:n    \num_eval:n
\let_new:NN \int_eval:w    \num_eval:w
\let_new:NN \int_eval_end: \num_eval_end:
```

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
\const_new:Nn \c_max_int {2147483647}
```

`\int_pre_eval_one_arg:Nn` `\int_pre_eval_two_args:Nnn` These might be handy when handing down values to other functions. All they do is evaluate the number in advance.

```
\def:Npn \int_pre_eval_one_arg:Nnn #1#2{\exp_args:No#1{\int_eval:w#2}}
\def:Npn \int_pre_eval_two_args:Nnn #1#2#3{
  \exp_args:Noo#1{\int_use:N\int_eval:w#2}{\int_use:N\int_eval:w#3}
}
```

`\int_div_truncate:nn` `\int_div_round:nn` `\int_mod:nn` `\int_div_truncate_raw:nn` `\int_div_round_raw:nn` `\int_mod_raw:nn` As `\num_eval:w` rounds the result of a division we also provide a version that truncates the result.

```
\def_new:Npn \int_div_truncate:nn {
  \int_pre_eval_two_args:Nnn\int_div_truncate_raw:nn
}
```

Initial version didn't work correctly with eTEX's implementation.

```
%\def_new:Npn \int_div_truncate_raw:nn #1#2 {
%  \int_eval:n{ (2*#1 - #2) / (2* #2) }
%}
```

New version by Heiko:

```
\def_new:Npn \int_div_truncate_raw:nn #1#2 {
  \int_eval:w
    \if_num:w \int_eval:w#1 = \c_zero
      0
    \else:
      (#1
      \if_num:w  \int_eval:w #1 < \c_zero
        \if_num:w \int_eval:w#2 < \c_zero
          -( #2 +
        \else:
          +( #2 -
        \fi:
      \else:
        \if_num:w \int_eval:w #2 < \c_zero
          +( #2 +
        \else:
          -( #2 -
```

```
        \fi:
      \fi:
      1)/2)
    \fi:
    /(#2)
  \int_eval_end:
}
```

For the sake of completeness:

```
\def_new:Npn \int_div_round:nn {
  \int_pre_eval_two_args:Nnn\int_div_round_raw:nn
}
\def_new:Npn \int_div_round_raw:nn #1#2 {\int_eval:n{#1/#2}}
```

Finally there's the modulus operation.

```
\def_new:Npn \int_mod:nn {\int_pre_eval_two_args:Nnn\int_mod_raw:nn}
\def_new:Npn \int_mod_raw:nn #1#2 {
  \int_eval:n{ #1 - \int_div_truncate_raw:nn {#1}{#2} * #2 }
}
```

`\int_compare:nNnTF` `\int_compare:nNnT` `\int_compare:nNnF` Simple comparison tests.

```
\let_new:NN \int_compare:nNnTF \num_compare:nNnTF
\let_new:NN \int_compare:nNnT \num_compare:nNnT
\let_new:NN \int_compare:nNnF \num_compare:nNnF
```

`\int_max_of:nn` `\int_min_of:nn` `\int_abs:n` Simple comparison tests.

```
\let_new:NN \int_max_of:nn \num_max_of:nn
\let_new:NN \int_min_of:nn \num_min_of:nn
\let_new:NN \int_abs:nn \num_abs:nn
```

`\int_compare_p:nNn` A predicate function.

```
\let_new:NN \int_compare_p:nNn \num_compare_p:nNn
```

`\int_if_odd_p:n` `\int_if_odd:nTF` `\int_if_odd:nT` `\int_if_odd:nF` A predicate function.

```
\def_new:Npn \int_if_odd_p:n #1 {
  \if_num_odd:w \int_eval:n{#1}
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {int_if_odd:n}#1{\if_num_odd:w \int_eval:n{#1}}
```

`\int_whiledo:nNnT` `\int_whiledo:nNnF` `\int_dowhile:nNnT` `\int_dowhile:nNnF` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `dowhile` does it the other way round. The have to be defined as "long" since the `T` argument might contain `\par` tokens.

```
\def_long_new:Npn \int_whiledo:nNnT #1#2#3#4{
```

```
  \int_compare:nNnT {#1}#2{#3}{#4 \int_whiledo:nNnT {#1}#2{#3}{#4}}
}
\def_long_new:Npn \int_whiledo:nNnF #1#2#3#4{
  \int_compare:nNnF {#1}#2{#3}{#4 \int_whiledo:nNnF {#1}#2{#3}{#4}}
}
\def_long_new:Npn \int_dowhile:nNnT #1#2#3#4{
  #4 \int_compare:nNnT {#1}#2{#3}{\int_dowhile:nNnT {#1}#2{#3}{#4}}
}
\def_long_new:Npn \int_dowhile:nNnF #1#2#3#4{
  #4 \int_compare:nNnF {#1}#2{#3}{\int_dowhile:nNnF {#1}#2{#3}{#4}}
}
```

16.6.1 Scanning and conversion

Conversion between different numbering schemes requires meticulous work. A number can be preceeded by any number of + and/or -. We define a generic function which will return the sign and/or the remainder.

\int_get_sign_and_digits:n
\int_get_sign:n
\int_get_digits:n
_get_sign_and_digits_aux:nNNN
_get_sign_and_digits_aux:oNNN

A number may be preceeded by any number of +s and −s. Start out by assuming we have a positive number.

```
\def_new:Npn \int_get_sign_and_digits:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true \c_true \c_true
}
\def_new:Npn \int_get_sign:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true \c_true \c_false
}
\def_new:Npn \int_get_digits:n #1{
  \int_get_sign_and_digits_aux:nNNN {#1} \c_true \c_false \c_true
}
```

Now check the first character in the string. Only a - can change if a number is positive or negative, hence we reverse the boolean governing this. Then gobble the - and start over.

```
\def_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4{
  \tlist_if_head_eq_charcode:fNTF {#1} -
  {
    \bool_if:NTF #2
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_false #3#4 }
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} \c_true  #3#4 }
  }
```

The other cases are much simpler since we either just have to gobble the + or exit immediately and insert the correct sign.

```
  {
    \tlist_if_head_eq_charcode:fNTF {#1} +
    { \int_get_sign_and_digits_aux:oNNN {\use_none:n #1} #2#3#4}
    {
```

The boolean #3 is for printing the sign while #4 is for printing the digits.

```
      \bool_double_if:NNnnnn #3#4
      { \bool_if:NF #2 - #1 }
      { \bool_if:NF #2 -    }
      { #1 }  {  }
    }
  }
}
\def_new:Npn \int_get_sign_and_digits_aux:oNNN{
  \exp_args:No\int_get_sign_and_digits_aux:nNNN
}
```

\int_convert_from_base_ten:nn
\convert_from_base_ten_aux:nnn
\convert_from_base_ten_aux:non
\convert_from_base_ten_aux:fon

#1 is the base 10 number to be converted to base #2. We split off the sign first, print if if there and then convert only the number. Since this is supposedly a base 10 number we can let TEX do the reading of + and -.

```
\def:Npn \int_convert_from_base_ten:nn#1#2{
  \num_compare:nNnTF {#1}<\c_zero
  {
    - \int_convert_from_base_ten_aux:non {}
    { \int_use:N \int_eval:n {-#1} }
  }
  {
    \int_convert_from_base_ten_aux:non {}
    { \int_use:N \int_eval:n {#1} }
  }
  {#2}
}
```

The algorithm runs like this:

1. If the number ⟨*num*⟩ is greater than ⟨*base*⟩, calculate modulus of ⟨*num*⟩ and ⟨*base*⟩ and carry that over for next round. The remainder is calculated as a truncated division of ⟨*num*⟩ and ⟨*base*⟩. Start over with these new values.
2. If ⟨*num*⟩ is less than or equal to ⟨*base*⟩ convert it to the correct symbol, print the previously calculated digits and exit.

#1 is the carried over result, #2 the remainder and #3 the base number.

```
\def_new:Npn \int_convert_from_base_ten_aux:nnn#1#2#3{
  \num_compare:nNnTF {#2}<{#3}
  { \int_convert_number_to_letter:n{#2} #1 }
  {
    \int_convert_from_base_ten_aux:fon
    {
      \int_convert_number_to_letter:n {\int_use:N\int_mod_raw:nn {#2}{#3}}
      #1
    }
    {\int_use:N \int_div_truncate_raw:nn{#2}{#3}}
    {#3}
  }
}
\def:Npn \int_convert_from_base_ten_aux:non{
  \exp_args:Nno\int_convert_from_base_ten_aux:nnn
```

```
}
\def:Npn \int_convert_from_base_ten_aux:fon{
  \exp_args:Nfo\int_convert_from_base_ten_aux:nnn
}
```

`nt_convert_number_to_letter:n` Turning a number for a different base into a letter or digit.

```
\def:Npn \int_convert_number_to_letter:n #1{ \if_case:w \int_eval:w
  #1-10\scan_stop: \exp_after:NN A \or: \exp_after:NN B \or:
  \exp_after:NN C \or: \exp_after:NN D \or: \exp_after:NN E \or:
  \exp_after:NN F \or: \exp_after:NN G \or: \exp_after:NN H \or:
  \exp_after:NN I \or: \exp_after:NN J \or: \exp_after:NN K \or:
  \exp_after:NN L \or: \exp_after:NN M \or: \exp_after:NN N \or:
  \exp_after:NN O \or: \exp_after:NN P \or: \exp_after:NN Q \or:
  \exp_after:NN R \or: \exp_after:NN S \or: \exp_after:NN T \or:
  \exp_after:NN U \or: \exp_after:NN V \or: \exp_after:NN W \or:
  \exp_after:NN X \or: \exp_after:NN Y \or: \exp_after:NN Z \else:
  \use_arg_i_after_fi:nw{ #1 }\fi: }
```

`\int_convert_to_base_ten:nn` `#1` is the number, `#2` is its base. First we get the sign, then use only the digits/letters from it and pass that onto a new function.

```
\def:Npn \int_convert_to_base_ten:nn #1#2 {
  \int_use:N\int_eval:n{
    \int_get_sign:n{#1}
    \exp_args:Nf\int_convert_to_base_ten_aux:nn {\int_get_digits:n{#1}}{#2}
  }
}
```

This is an intermediate function to get things started.

```
\def_new:Npn \int_convert_to_base_ten_aux:nn #1#2{
  \int_convert_to_base_ten_auxi:nnN {0}{#2} #1 \q_nil
}
```

Here we check each letter/digit and calculate the next number. `#1` is the previously calculated result (to be multiplied by the base), `#2` is the base and `#3` is the next letter/digit to be added.

```
\def_new:Npn \int_convert_to_base_ten_auxi:nnN#1#2#3{
  \quark_if_nil:NTF #3
  {#1}
  {\exp_args:No\int_convert_to_base_ten_auxi:nnN
    {\int_use:N \int_eval:n{ #1*#2+\int_convert_letter_to_number:N #3} }
    {#2}
  }
}
```

This is for turning a letter or digit into a number. This function also takes care of handling lowercase and uppercase letters. Hence `a` is turned into `11` and so is `A`.

```
\def:Npn \int_convert_letter_to_number:N #1{
  \int_compare:nNnTF{`#1}<{58}{#1}
  {
```

```
    \int_eval:n{ `#1 -
      \if:w\int_compare_p:nNn{`#1}<{91}
      55
      \else:
      87
      \fi:
    }
  }
}
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

17 Length registers

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

17.1 Skip registers

17.1.1 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_gadd:cn`
`\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*⟩`} {`⟨*action*⟩
⟨*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.

17.1.2 Formatting a skip register value

17.1.3 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.

17.2 Dim registers

17.2.1 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_gadd:cn

`\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.

17.2.2 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.

17.3 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.

17.4 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package&!check⟩\RequirePackage{l3int}
⟨package&!check⟩\RequirePackage{l3prg}
⟨package & check⟩\RequirePackage{l3chk}
⟨*initex | package⟩
```

17.4.1 Skip registers

\skip_new:N
\skip_new:c
\skip_new_l:N

Allocation of a new internal registers.

```
⟨*initex⟩
\alloc_setup_type:nnn {skip} \c_zero \c_max_register_num
\def_new:Npn\skip_new:N   #1 {\alloc_reg:NnNN g {skip} \tex_skipdef:D #1 }
\def_new:Npn\skip_new_l:N #1 {\alloc_reg:NnNN l {skip} \tex_skipdef:D #1 }
⟨/initex⟩
⟨package⟩\let:NN \skip_new:N \newskip
\def_new:Npn \skip_new:c {\exp_args:Nc \skip_new:N}
```

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Setting skips is again something that I would like to make uniform at the moment to get a better overview.

```
\def_new:Npn \skip_set:Nn #1#2{#1\skip_eval:n{#2}
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \skip_gset:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \skip_set:Nn }
\def_new:Npn \skip_set:cn {\exp_args:Nc \skip_set:Nn }
\def_new:Npn \skip_gset:cn {\exp_args:Nc \skip_gset:Nn }
```

\skip_zero:N
\skip_gzero:N
\skip_zero:c
\skip_gzero:c

Reset the register to zero.

```
\def_new:Npn \skip_zero:N #1{#1\c_zero_skip \scan_stop:
⟨*check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \skip_gzero:N {
```

We make sure that a local variable is not updated globally by changing the internal test (i.e. \chk_local_or_pref_global:N) before making the assignment. This is done by \pref_global_chk: which also issues the necessary \pref_global:D. This is not very efficient, but this code will be only included for debugging purposes. Using \pref_global:D in front of the local function is better in the production versions.

```
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \skip_zero:N}
\def_new:Npn \skip_zero:c  {\exp_args:Nc \skip_zero:N}
\def_new:Npn \skip_gzero:c {\exp_args:Nc \skip_gzero:N}
```

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_sub:Nn
\skip_gsub:Nn

Adding and subtracting to and from ¡skip¿s

```
\def_new:Npn \skip_add:Nn #1#2{
```

We need to say by in case the first argment is a register accessed by its number, e.g., \skip23.

```
    \tex_advance:D#1 by \skip_eval:n{#2}
⟨*check⟩
    \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn\skip_add:cn{\exp_args:Nc\skip_add:Nn}
\def_new:Npn \skip_sub:Nn #1#2{
    \tex_advance:D#1-\skip_eval:n{#2}
⟨*check⟩
\chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \skip_gadd:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \skip_add:Nn }
\def_new:Npn \skip_gsub:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \skip_sub:Nn }
```

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Inserting skips.

```
\let_new:NN \skip_horizontal:N \tex_hskip:D
\def_new:Npn \skip_horizontal:c {\exp_args:Nc\skip_horizontal:N}
\def_new:Npn \skip_horizontal:n #1{\skip_horizontal:N \skip_eval:n{#1}}
\let_new:NN \skip_vertical:N \tex_vskip:D
\def_new:Npn \skip_vertital:c {\exp_args:Nc\skip_vertical:N}
\def_new:Npn \skip_vertical:n #1{\skip_vertical:N \skip_eval:n{#1}}
```

\skip_use:N
\skip_use:c

Here is how skip registers are accessed:

```
\let_new:NN \skip_use:N \tex_the:D
\def_new:Npn \skip_use:c #1{\exp_args:Nc\skip_use:N}
```

\skip_eval:n

Evaluating a calc expression.

```
\def_new:Npn \skip_eval:n #1 {\etex_glueexpr:D #1 \scan_stop:}
```

\l_tmpa_skip
\l_tmpb_skip
\l_tmpc_skip
\g_tmpa_skip
\g_tmpb_skip

We provide three local and two global scratch registers, maybe we need more or less.

```
%%\chk_new_cs:N \l_tmpa_skip
%%\tex_skipdef:D\l_tmpa_skip 255  %currently taken up by \skip@
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \l_tmpc_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip
```

\c_zero_skip
\c_max_skip

```
⟨*!package⟩
\skip_new:N \c_zero_skip
\skip_set:Nn \c_zero_skip {0pt}
\skip_new:N \c_max_skip
\skip_set:Nn \c_max_skip {16383.99999pt}
⟨/!package⟩
⟨*!initex⟩
\let:NN \c_zero_skip \z@
\let:NN \c_max_skip  \maxdimen
⟨/!initex⟩
```

\skip_infinite_glue:nTF

With ε-TEX we all of a sudden get access to a lot information we should otherwise consider ourselves lucky to get. One is the stretch and shrink components of a skip register and the order or those components. `\skip_infinite_glue:nTF` tests it directly by looking at the stretch and shrink order. If either of the predicate functions return ⟨*true*⟩ `\prg_logic_or_p:nn` will return ⟨*true*⟩ and the logic test will take the true branch.

```
\def_new:Npn \skip_infinite_glue:nTF #1{
  \predicate:nTF {
      \int_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
      \int_compare_p:nNn {\etex_glueshrinkorder:D #1 }  > \c_zero
  }
}
```

\skip_split_finite_else_action:nnNN This macro is useful when performing error checking in certain circumstances. If the ⟨*skip*⟩ register holds finite glue it sets #3 and #4 to the stretch and shrink component resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are global.

```
\def_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
  \skip_infinite_glue:nTF {#1}
  {
    #3 = \c_zero_skip
    #4 = \c_zero_skip
    #2
  }
  {
    #3 = \etex_gluestretch:D #1 \scan_stop:
    #4 = \etex_glueshrink:D  #1 \scan_stop:
  }
}
```

17.4.2 Dimen registers

\dim_new:N
\dim_new:c
\dim_new_l:N
Allocating ⟨*dim*⟩ registers...

```
⟨*initex⟩
\alloc_setup_type:nnn {dimen} \c_zero \c_max_register_num
\def_new:Npn \dim_new:N   #1 {\alloc_reg:NnNN g {dimen} \tex_dimendef:D #1 }
\def_new:Npn \dim_new_l:N #1 {\alloc_reg:NnNN l {dimen} \tex_dimendef:D #1 }
⟨/initex⟩
⟨package⟩\let:NN \dim_new:N \newdimen
\def_new:Npn \dim_new:c {\exp_args:Nc \dim_new:N}
```

\dim_set:Nn
\dim_gset:Nn
\dim_set:cn
\dim_set:Nc
\dim_gset:cn
\dim_gset:Nc
\dim_gset:cc
We add \dim_eval:n in order to allow simple arithmetic and a space just for those using \dimen1 or alike. See OR!

```
\def_new:Npn \dim_set:Nn #1#2{#1~ \dim_eval:n{#2}}
\def_new:Npn \dim_gset:Nn {\pref_global:D \dim_set:Nn }
\def_new:Npn \dim_set:cn {\exp_args:Nc \dim_set:Nn }
\def_new:Npn \dim_set:Nc {\exp_args:NNc \dim_set:Nn }
\def_new:Npn \dim_gset:cn {\exp_args:Nc \dim_gset:Nn }
\def_new:Npn \dim_gset:Nc {\exp_args:NNc \dim_gset:Nn }
\def_new:Npn \dim_gset:cc {\exp_args:Ncc \dim_gset:Nn }
```

\dim_zero:N
\dim_gzero:N
\dim_zero:c
\dim_gzero:N
Resetting.

```
\def_new:Npn \dim_zero:N #1{#1\c_zero_skip}
\def_new:Npn \dim_gzero:N {\pref_global:D \dim_zero:N}
\def_new:Npn \dim_zero:c  {\exp_args:Nc \dim_zero:N}
\def_new:Npn \dim_gzero:c {\exp_args:Nc \dim_gzero:N}
```

\dim_add:Nn
\dim_add:cn
\dim_add:Nc
\dim_gadd:Nn
\dim_gadd:cn
Addition.

```
\def_new:Npn \dim_add:Nn #1#2{
```

We need to say by in case the first argment is a register accessed by its number, e.g., \dimen23.

```
    \tex_advance:D#1 by \dim_eval:n{#2}\scan_stop:
}
\def_new:Npn\dim_add:cn{\exp_args:Nc\dim_add:Nn}
\def_new:Npn\dim_add:Nc{\exp_args:NNc\dim_add:Nn}
\def_new:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
\def_new:Npn\dim_gadd:cn{\exp_args:Nc\dim_gadd:Nn}
```

`\dim_sub:Nn` `\dim_sub:cn` `\dim_sub:Nc` `\dim_gsub:Nn` `\dim_gsub:cn` Subtracting.

```
\def_new:Npn \dim_sub:Nn #1#2{\tex_advance:D#1-#2\scan_stop:}
\def_new:Npn\dim_sub:cn{\exp_args:Nc\dim_sub:Nn}
\def_new:Npn\dim_sub:Nc{\exp_args:NNc\dim_sub:Nn}
\def_new:Npn \dim_gsub:Nn {\pref_global:D \dim_sub:Nn }
\def_new:Npn\dim_gsub:cn{\exp_args:Nc\dim_gsub:Nn}
```

`\dim_use:N` `\dim_use:c` Accessing a ⟨*dim*⟩.

```
\let_new:NN \dim_use:N \tex_the:D
\def_new:Npn \dim_use:c {\exp_args:Nc\dim_use:N}
```

`\l_tmpa_dim` `\l_tmpb_dim` `\l_tmpc_dim` `\l_tmpd_dim` `\g_tmpa_dim` `\g_tmpb_dim` Some scratch registers.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \l_tmpc_dim
\dim_new:N \l_tmpd_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

`\c_zero_dim` `\c_max_dim` Just aliases.

```
\let_new:NN \c_zero_dim \c_zero_skip
\let_new:NN \c_max_dim \c_max_skip
```

`\dim_eval:n` Evaluating a calc expression.

```
\def_new:Npn \dim_eval:n #1 {\etex_dimexpr:D #1 \scan_stop:}
```

`\if_dim:w` The comparison primitive.

```
\let_new:NN \if_dim:w \tex_ifdim:D
```

`\dim_compare:nNnTF` `\dim_compare:nNnT` `\dim_compare:nNnF` Check the expression and choose branch.

```
\def_new:Npn \dim_compare:nNnTF #1#2#3{
  \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
    \exp_after:NN \use_arg_i:nn
  \else:
    \exp_after:NN \use_arg_ii:nn
  \fi:
```

```
}
\def_new:Npn \dim_compare:nNnT #1#2#3{
  \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
    \exp_after:NN \use_arg_ii:nn
  \fi:
  \use_none:n
}
\def_new:Npn \dim_compare:nNnF #1#2#3{
  \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
    \exp_after:NN \use_none:n
  \else:
    \exp_after:NN \use_arg_i:n
  \fi:
}
```

`\dim_compare_p:nNn` A predicate function.

```
\def_new:Npn \dim_compare_p:nNn #1#2#3{
  \if_dim:w \dim_eval:n {#1} #2 \dim_eval:n {#3}
    \c_true
  \else:
    \c_false
  \fi:
}
```

`\dim_while:nNnT` `\dim_while:nNnF` `\dim_dowhile:nNnT` `\dim_dowhile:nNnF` `while` and `do-while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\def_new:Npn \dim_while:nNnT #1#2#3#4{
  \dim_compare:nNnT {#1}#2{#3}{#4 \dim_while:nNnT {#1}#2{#3}{#4}}
}
\def_new:Npn \dim_while:nNnF #1#2#3#4{
  \dim_compare:nNnF {#1}#2{#3}{#4 \dim_while:nNnF {#1}#2{#3}{#4}}
}
\def_new:Npn \dim_dowhile:nNnT #1#2#3#4{
  #4 \dim_compare:nNnT {#1}#2{#3}{\dim_dowhile:nNnT {#1}#2{#3}{#4}}
}
\def_new:Npn \dim_dowhile:nNnF #1#2#3#4{
  #4 \dim_compare:nNnF {#1}#2{#3}{\dim_dowhile:nNnF {#1}#2{#3}{#4}}
}
```

17.4.3 Muskips

`\muskip_new:N` `\muskip_new_l:N` And then we add muskips.

```
⟨*initex⟩
\alloc_setup_type:nnn {muskip} \c_zero \c_max_register_num
\def_new:Npn \muskip_new:N   #1{\alloc_reg:NnNN g {muskip} \tex_muskipdef:D #1}
\def_new:Npn \muskip_new_l:N #1{\alloc_reg:NnNN l {muskip} \tex_muskipdef:D #1}
⟨/initex⟩
⟨package⟩\let_new:NN \muskip_new:N \newmuskip     % nicked from LaTeX
```

\muskip_set:Nn
\muskip_gset:Nn
\muskip_add:Nn
\muskip_gadd:Nn
\muskip_sub:Nn
\muskip_gsub:Nn

Simple functions for muskips.

```
\def_new:Npn \muskip_set:Nn#1#2{#1\etex_muexpr:D#2\scan_stop:}
\def_new:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
\def_new:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
\def_new:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
\def_new:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
\def_new:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}
```

```
⟨/initex | package⟩
```

18 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.

18.1 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.

18.2 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.

18.3 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.

18.3.1 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.

18.4 The Implementation

We start by ensuring that the required packages are loaded. We check for l3expan since this a basic package that is essential for use of any higher-level package.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package & check⟩\RequirePackage{l3chk}\par
⟨package⟩\RequirePackage{l3expan}\par
⟨*initex | package⟩
```

\toks_new:N
\toks_new_l:N
\toks_new:c

Allocates a new token register. This function is already defined above.

```
⟨*initex⟩
\alloc_setup_type:nnn {toks} \c_zero \c_max_register_num
\def_new:Npn \toks_new:N   #1{\alloc_reg:NnNN g {toks} \tex_toksdef:D #1}
\def_new:Npn \toks_new_l:N #1{\alloc_reg:NnNN l {toks} \tex_toksdef:D #1}
⟨/initex⟩
⟨package⟩\let:NN \toks_new:N \newtoks   % nick from LaTeX for the moment
\def_new:Npn \toks_new:c {\exp_args:Nc\toks_new:N}
```

\toks_clear:N
\toks_gclear:N

These functions clear a token register, either locally or globally.

```
\def_new:Npn \toks_clear:N #1{#1\c_empty_toks
⟨*check⟩
   \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \toks_gclear:N {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \toks_clear:N}
```

\toks_use:N
\toks_use:c

This function just returns the contents of a token register.

```
\let_new:NN \toks_use:N \the_internal:D
\def_new:Npn \toks_use:c {\exp_args:Nc\toks_use:N}
```

\toks_use_clear:N
\toks_use_gclear:N

These functions clear a token register (locally or globally) after returning the contents.

They make sure that clearing the register does not interfere with following tokens. In other words, the contents of the register might operate on what follows in the input stream. A direct implementation will save one `\exp_after:NN` but for the sake of checking we do it this way now.

```
\def_new:Npn \toks_use_clear:N#1{
   \exp_after:NN
     \toks_clear:N
   \exp_after:NN
        #1
     \toks_use:N#1}
\def_new:Npn \toks_use_gclear:N{
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \toks_use_clear:N}
```

\toks_put_left:Nn
\toks_put_left:No
\toks_gput_left:Nn
\toks_gput_left:Nx
\toks_put_left_aux:w

`\toks_put_left:Nn` ⟨*toks*⟩⟨*stuff*⟩ adds the tokens of *stuff* on the 'left-side' of the token register ⟨*toks*⟩. `\toks_put_left:No` does the same, but expands the tokens once. We need to look out for brace stripping so we add a token, which is then later removed.

```
\def_new:Npn \toks_put_left:Nn #1{
```

```
  \exp_after:NN\toks_put_left_aux:w\exp_after:NN\q_mark
                          \toks_use:N #1\q_stop #1}
\def_new:Npn \toks_put_left:No {\exp_args:NNo \toks_put_left:Nn}
\def_new:Npn \toks_gput_left:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \toks_put_left:Nn}
\def_new:Npn \toks_gput_left:Nx {\exp_args:NNx \toks_gput_left:Nn}
```

A helper function for `\toks_put_left:Nn`. Its arguments are subsequently the tokens of ⟨*stuff*⟩, the token register ⟨*toks*⟩ and the current contents of ⟨*toks*⟩. We make sure to remove the token we inserted earlier.

```
\def_long_new:Npn \toks_put_left_aux:w #1\q_stop #2#3{
  #2\exp_after:NN{\use_arg_i:nn{#3}#1}
⟨*check⟩
     \chk_local_or_pref_global:N #2
⟨/check⟩
}
```

`\toks_put_right:Nn` `\toks_gput_right:Nn` `\toks_put_right:No` `\toks_put_right:Nd` `\toks_put_right:Nf` `\toks_put_right:Nx` `\toks_gput_right:No` `\toks_gput_right:Nx`

These macros add a list of tokens to the right of a token register.

```
\def_long_new:Npn \toks_put_right:Nn #1#2{#1\exp_after:NN{\toks_use:N #1#2}
⟨*check⟩
        \chk_local_or_pref_global:N #1
⟨/check⟩
}
\def_new:Npn \toks_gput_right:Nn {
⟨*check⟩
   \pref_global_chk:
⟨/check⟩
⟨−check⟩ \pref_global:D
   \toks_put_right:Nn}
```

`\toks_gput_right:Nx` expands its (second) argument.

```
⟨check⟩\def_new:Npn \toks_put_right:No {\exp_args:NNo \toks_put_right:Nn }
⟨−check⟩\def_long_new:Npn\toks_put_right:No#1#2{#1\exp_after:NN\exp_after:NN
⟨−check⟩\exp_after:NN{\exp_after:NN\toks_use:N\exp_after:NN #1#2}}
⟨check⟩\def_new:Npn \toks_put_right:Nd {\exp_args:NNd \toks_put_right:Nn }
⟨−check⟩\def_long_new:Npn\toks_put_right:Nd#1#2{
⟨−check⟩  \exp_after:NN\toks_put_right:No\exp_after:NN#1\exp_after:NN{#2}}
```

We implement `\toks_put_right:Nf` by hand because I think I might use it in the l3keyval module in which case it is going to be used a lot.

```
⟨check⟩\def_new:Npn \toks_put_right:Nf {\exp_args:NNf \toks_put_right:Nn }
⟨−check⟩\def_long_new:Npn \toks_put_right:Nf #1#2{
⟨−check⟩  #1\exp_after:NN\exp_after:NN\exp_after:NN{
⟨−check⟩    \exp_after:NN\toks_use:N\exp_after:NN #1\int_to_roman:w -‘0#2}}
\def_new:Npn \toks_put_right:Nx {\exp_args:NNx \toks_put_right:Nn }
\def_new:Npn \toks_gput_right:No {\exp_args:NNo\toks_gput_right:Nn}
\def_new:Npn \toks_gput_right:Nx {\exp_args:NNx\toks_gput_right:Nn}
```

\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_set:Nn`⟨*toks*⟩⟨*stuff*⟩ stores ⟨*stuff*⟩ without expansion in ⟨*toks*⟩. `\toks_set:No` and `\toks_set:Nx` expand ⟨*stuff*⟩ once and fully.

```
⟨*check⟩
\def_new:Npn \toks_set:Nn #1{\chk_local:N #1#1}
⟨/check⟩
```

If we don't check if ⟨*toks*⟩ is a local register then the `\toks_set:Nn` function has nothing to do.

```
⟨−check⟩ \let_new:NN \toks_set:Nn\use_noop:
⟨−check⟩ \def_long_new:Npn \toks_set:No#1#2{#1\exp_after:NN{#2}}
⟨−check⟩ \def_long_new:Npn \toks_set:Nd#1#2{
⟨−check⟩ #1\exp_after:NN\exp_after:NN\exp_after:NN{#2}}
⟨check⟩ \def_new:Npn \toks_set:No {\exp_args:NNo \toks_set:Nn}
⟨check⟩ \def_new:Npn \toks_set:Nd {\exp_args:NNd \toks_set:Nn}
\def_new:Npn \toks_set:Nx {\exp_args:NNx \toks_set:Nn}
```

We implement `\toks_set:Nf` by hand when not checking because this is going to be used *extensively* in keyval processing!

```
⟨check⟩\def_new:Npn \toks_set:Nf {\exp_args:NNf \toks_set:Nn}
⟨−check⟩\def_long_new:Npn\toks_set:Nf #1#2{
⟨−check⟩  #1\exp_after:NN{\int_to_roman:w -‘0#2}}
\def_new:Npn \toks_set:cf {\exp_args:Nc\toks_set:Nf}
\def_new:Npn \toks_set:cn {\exp_args:Nc\toks_set:Nn}
\def_new:Npn \toks_set:co {\exp_args:Nc\toks_set:No}
\def_new:Npn \toks_set:cx {\exp_args:Nc\toks_set:Nx}
```

\toks_gset:Nn
\toks_gset:No
\toks_gset:Nx
\toks_gset:cn
\toks_gset:co
\toks_gset:cx

These functions are the global variants of the above.

```
⟨*check⟩
\def_new:Npn \toks_gset:Nn #1{\chk_global:N #1\pref_global:D#1}
⟨/check⟩
⟨−check⟩ \let_new:NN \toks_gset:Nn\pref_global:D
\def_new:Npn \toks_gset:No {\exp_args:NNo \toks_gset:Nn}
\def_new:Npn \toks_gset:Nx {\exp_args:NNx \toks_gset:Nn}
\def_new:Npn \toks_gset:cn {\exp_args:Nc \toks_gset:Nn}
\def_new:Npn \toks_gset:co {\exp_args:Nc \toks_gset:No}
\def_new:Npn \toks_gset:cx {\exp_args:Nc \toks_gset:Nx}
```

\toks_set_eq:NN
\toks_set_eq:Nc
\toks_set_eq:cN
\toks_set_eq:cc
\toks_gset_eq:NN
\toks_gset_eq:Nc
\toks_gset_eq:cN
\toks_gset_eq:cc

`\toks_set_eq:NN`⟨*toks1*⟩⟨*toks2*⟩ copies the contents of ⟨*toks2*⟩ in ⟨*toks1*⟩.

```
⟨*check⟩
\def_new:Npn\toks_set_eq:NN#1#2{
   \chk_local:N#1
   \chk_var_or_const:N#2
   #1#2}
\def_new:Npn\toks_gset_eq:NN#1#2{
   \chk_global:N#1
   \chk_var_or_const:N#2
   \pref_global:D#1#2}
⟨/check⟩
⟨−check⟩ \let_new:NN \toks_set_eq:NN  \use_noop:
```

```
⟨−check⟩ \let_new:NN \toks_gset_eq:NN \pref_global:D
\def_new:Npn \toks_set_eq:Nc {\exp_args:NNc\toks_set_eq:NN}
\def_new:Npn \toks_set_eq:cN {\exp_args:Nc\toks_set_eq:NN}
\def_new:Npn \toks_set_eq:cc {\exp_args:Ncc\toks_set_eq:NN}
\def_new:Npn \toks_gset_eq:Nc {\exp_args:NNc\toks_gset_eq:NN}
\def_new:Npn \toks_gset_eq:cN {\exp_args:Nc\toks_gset_eq:NN}
\def_new:Npn \toks_gset_eq:cc {\exp_args:Ncc\toks_gset_eq:NN}
```

\toks_if_empty_p:N
\toks_if_empty_p:c
\toks_if_empty:NTF
\toks_if_empty:cTF
\toks_if_empty:NT
\toks_if_empty:cT
\toks_if_empty:NF
\toks_if_empty:cF

\toks_if_empty:NTF⟨*toks*⟩⟨*true code*⟩⟨*false code*⟩ tests if a token register is empty and executes either ⟨*true code*⟩ or ⟨*false code*⟩. This test had the advantage of being expandable. Otherwise one has to do an x type expansion in order to prevent problems with parameter tokens.

```
\def_new:Npn\toks_if_empty_p:N#1{
  \if:w \tlist_if_empty_p:o{\toks_use:N #1}
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn{toks_if_empty:N}#1{\if:w \toks_if_empty_p:N #1}
\def_new:Npn\toks_if_empty:cTF{\exp_args:Nc\toks_if_empty:NTF}
\def_new:Npn\toks_if_empty:cT{\exp_args:Nc\toks_if_empty:NT}
\def_new:Npn\toks_if_empty:cF{\exp_args:Nc\toks_if_empty:NF}
```

\toks_if_eq:NNTF
\toks_if_eq:NNT
\toks_if_eq:NNF
\toks_if_eq:NcTF
\toks_if_eq:NcT
\toks_if_eq:NcF
\toks_if_eq:cNTF
\toks_if_eq:cNT
\toks_if_eq:cNF
\toks_if_eq:ccTF
\toks_if_eq:ccT
\toks_if_eq:ccF
\toks_if_eq_p:NN
\toks_if_eq_p:cN
\toks_if_eq_p:Nc
\toks_if_eq_p:cc

This function test whether two token registers contain the same.

```
\def_new:NNn \toks_if_eq:NNTF 2 {
  \tlist_if_eq:xxTF{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:NNn \toks_if_eq:NNT 2 {
  \tlist_if_eq:xxT{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:NNn \toks_if_eq:NNF 2 {
  \tlist_if_eq:xxF{\toks_use:N #1}{\toks_use:N #2}
}
\def_new:Npn \toks_if_eq:NcTF {\exp_args:NNc \toks_if_eq:NNTF}
\def_new:Npn \toks_if_eq:NcT {\exp_args:NNc \toks_if_eq:NNT}
\def_new:Npn \toks_if_eq:NcF {\exp_args:NNc \toks_if_eq:NNF}
\def_new:Npn \toks_if_eq:cNTF {\exp_args:Nc \toks_if_eq:NNTF}
\def_new:Npn \toks_if_eq:cNT {\exp_args:Nc \toks_if_eq:NNT}
\def_new:Npn \toks_if_eq:cNF {\exp_args:Nc \toks_if_eq:NNF}
\def_new:Npn \toks_if_eq:ccTF {\exp_args:Ncc \toks_if_eq:NNTF}
\def_new:Npn \toks_if_eq:ccT {\exp_args:Ncc \toks_if_eq:NNT}
\def_new:Npn \toks_if_eq:ccF {\exp_args:Ncc \toks_if_eq:NNF}
\def_new:NNn \toks_if_eq_p:NN 2 {
  \tlist_if_eq_p:xx {\toks_use:N #1} {\toks_use:N #2}
}
\def_new:Npn \toks_if_eq_p:cN {\exp_args:Nc \toks_if_eq_p:NN}
\def_new:Npn \toks_if_eq_p:Nc {\exp_args:NNc \toks_if_eq_p:NN}
\def_new:Npn \toks_if_eq_p:cc {\exp_args:Ncc \toks_if_eq_p:NN}
```

\l_tmpa_toks
\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks

Some scratch register ...

```
\tex_toksdef:D \l_tmpa_toks = 255
⟨initex⟩\seq_put_right:Nn \g_toks_allocation_seq {255}
\toks_new:N \l_tmpb_toks
\toks_new:N \l_tmpc_toks
\toks_new:N \g_tmpa_toks
\toks_new:N \g_tmpb_toks
\toks_new:N \g_tmpc_toks
```

`\c_empty_toks` And here is a constant, which is a (permanently) empty token register.

```
\toks_new:N \c_empty_toks
```

`ks_remove_extra_brace_group:N` `ove_extra_brace_group_aux:NNw` Small function for removing an extra brace group if present. Hmm, not really needed anymore.

```
\def_new:Npn \toks_remove_extra_brace_group:N #1{
  \exp_after:NN \toks_remove_extra_brace_group_aux:NNw
  \exp_after:NN \toks_set:Nn \exp_after:NN #1
  \toks_use:N#1\q_nil
}
\def_long_new:Npn\toks_remove_extra_brace_group_aux:NNw #1#2#3\q_nil{#1#2{#3}}
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

19 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.

19.1 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*⟩. A normal line break in such messages can be achieved with `\iow_newline:` from the l3iow module.

19.2 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.

19.2.1 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.

19.2.2 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.

19.3 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.

19.4 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.

19.5 Variables and constants

`\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.

19.6 The implementation

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3basics}
⟨package⟩\RequirePackage{l3tlp}
⟨package⟩\RequirePackage{l3expan}
⟨package⟩\RequirePackage{l3num}
⟨package⟩\RequirePackage{l3io}
⟨package⟩\RequirePackage{l3int}
⟨package⟩\RequirePackage{l3toks}
⟨package⟩\RequirePackage{l3token}
⟨*initex | package⟩
```

19.6.1 Code to be moved to other modules

`\g_file_curr_name_tlp` This variable is used to store the name of the file currently being processed. It should be part of the code that defines the higher level I/O commands.

```
\tlp_new:Nn \g_file_curr_name_tlp {no~file}
```

`\err_message:x` The LaTeX3 name for a TeX primitive. This should perhaps move to `l3names.dtx`.

```
\let_new:NN \err_message:x \tex_errmessage:D
```

`\text_put_sp:` `\text_put_four_sp:` We need these functions for certain error and warning messages right away. They put one and four spaces into the message stream.

```
\def_new:Npn \text_put_sp: {~}
\def_new:Npn \text_put_four_sp: {\text_put_sp: \text_put_sp:
                                 \text_put_sp: \text_put_sp: }
```

`\cmd_arg_list_build` This macro takes a digit as its argument and creates a string of `#` characters and argument numbers, such as `##1##2##3`. This list can then later be used in defining a new macro. To do this it locally uses a count register and a token register.

```
\def:Npn\cmd_arg_list_build#1{
```

First we need to make sure that the token register we will be using for temporary storage is empty.

```
  \toks_clear:N\l_tmpb_toks
```

Then we can store the argument in a count register that will be decremented untill it's value is zero. Beacuse of the use of the result of this macro, the argument needs to be between 1 and 9; this could be tested, but such a test is not (yet) added.

```
  \int_set:Nn \l_tmpa_int {#1}
  \int_while:nNnT \l_tmpa_int > \c_zero {
```

In the loop we first add the value of our counter to the contents of the token register;

```
    \toks_put_left:No \l_tmpb_toks {\the_internal:D\l_tmpa_int}
```

and precede it with two hash marks.

```
    \toks_put_left:Nn \l_tmpb_toks {##}
```

Now the count register is decremented and another iteration will follow so long as zero isn't reached.

```
    \int_decr:N\l_tmpa_int
  }
```

Finally the contents of the token register needs to be copied as the expansion of a local variable.

```
  \def:Npx\l_cmd_arg_list{\the_internal:D\l_tmpb_toks}
}
```

`\cmd_declare:Nnn` This macro is a first replacement for LaTeX 2_ε's `\newcommand`. It takes the name of a new macro as its first argument, the number of arguments for the new macro is taken as the second argument.

```
\def:Npn\cmd_declare:Nnn#1[#2]{
  \cmd_arg_list_build{#2}
  \exp_args:NNO\def:Npn#1\l_cmd_arg_list
}
```

`\io_show_file_lineno:` A function to add the number of the line and the name of the file to a message as an indication of where the message was triggered.

```
\def_new:Npn \io_show_file_lineno:{
  on~line~\the_internal:D\tex_inputlineno:D\text_put_sp:~of~
  file~\g_file_curr_name_tlp}
```

19.6.2 Variables and constants

`\g_err_help_toks` A token register to store the help text for an error message in.

```
\let:NwN \g_err_help_toks \tex_errhelp:D
```

`\l_err_label_token` This will hold the current error label.

```
\def_new:Npn \l_err_label_token {}
```

`\tex_errorcontextlines:D` Since we are producing our own error and help messages we can turn off the nasty stack information coming from TeX's stomach.

```
\int_set:Nn\tex_errorcontextlines:D\c_minus_one
```

19.6.3 Displaying the information

Here we define the fairly low level commands needed to communicate with the user.

`\err_info:nn` `\err_warn:nn` Write a message to the log file (`\err_info:nn`) or to both the log file and the terminal (`\err_warn:nn`).

```
\def_new:Npn \err_info:nn #1#2{
```

Make sure that the *continuation* is part of `\err_newline:`.

```
  \def:Npn\err_newline:{\iow_newline:#2}
```

Then write the message.

```
  \io_put_log:x {#1~\io_show_file_lineno:}}
\def_new:Npn \err_warn:nn #1#2{
  \def:Npn\err_newline:{\iow_newline:#2}
  \io_put_term:x {#1~\io_show_file_lineno:}}
```

`\err_info_noline:nn` `\err_warn_noline:nn` These variants of the above two functions don't add the linenumber to the message.

```
\def_new:Npn \err_info_noline:nn #1#2{
  \def:Npn\err_newline:{\iow_newline:#2}
  \io_put_log:x {#1}}
\def_new:Npn \err_warn_noline:nn #1#2{
  \def:Npn\err_newline:{\iow_newline:#2}
  \io_put_term:x {#1}}
```

`\err_interrupt:NNw` `\err_interrupt:NNw` is the function that is called when some error occurs in the code. It takes at least two arguments, the ⟨*errfile*⟩ which is a token list that holds the name of the file where the error message should be fetched from, and the label to identify the error message in the error file. However, it may have additional arguments that are picked up by the error handler extracted from the error file. This is specified in the third argument to `\err_interrupt_new:NNNnnn`.

```
\def_new:Npn \err_interrupt:NNw #1#2{\let:NwN \l_err_label_token #2
    \group_begin:
```

For some reason we get some `\par`s into the file if we use the current definition of `\iow_long_unxepanded:N` to write the messages. This is probably a consequence of using token registers to prohibit the expansion of code.

```
      \let:NwN \par\use_noop:
```

We want to ensure that we are not in programmer's mode (no spaces) but we want to switch on internal naming conventions.

```
      \CodeStop
      \NamesStart:
```

We better clear all short references that are active, otherwise we may get surprising results.

```
      %\clearshortrefmaps
      \tex_input:D #1~\err_display_aux:w}
```

\err_fatal:nn
\err_fatal_noline:nn

Write a message to the log file and to the terminal.

```
\def_new:Npn \err_fatal:nn #1#2{
```

Make sure that the *continuation* is part of `\err_newline`.

```
  \def:Npn\err_newline:{\iow_newline:#2}
```

Then write the message.

```
  \io_put_term:x {#1~\io_show_file_lineno:}
```

Finally abort the LaTeX run.

```
  \tex_end:D
  }
```

A variant that doesn't include the line number where the error occured.

```
\def_new:Npn \err_fatal_noline:nn #1#2{
  \def:Npn\err_newline:{\iow_newline:#2}
  \io_put_term:x {#1}
  \tex_end:D
  }
```

\err_newline:

`\err_newline:` is used to introduce a new line in an error message. I would like to use `\\` but this would mean redefinition which should be avoided to make the error message the last action before control is given to the user (otherwise something like `\group_end:` would interfere with insertions/deletions by the user). `\err_newline:` will be redefined by the various functions displaying messages to include the correct *continuation*.

```
\def_new:Npn \err_newline: {^^J}
```

19.6.4 Dealing with the error file

This section contains code that combines Michaels original thoughts on the the subject with Denys' further ideas.

\c_iow_err_stream

Error messages are logged using the output stream `\c_iow_err_stream`.

```
\iow_new:N \c_iow_err_stream
```

\g_err_curr_fname

A nick name for the currently open error file. It is empty if no error file is currently open.

```
\tlp_new:Nn \g_err_curr_fname{}
```

\err_file_new:Nn

This function defines a new error file. The first argument is a token list which should hold the name of the error file, the second argument is the name of the error file. The token list should be a constant defined by this function.

```
\def_new:Npn \err_file_new:Nn #1#2{
  \tlp_if_empty:NF\g_err_curr_fname
     {\err_latex_bug:x{Unclosed~error~file~`\g_err_curr_fname'}}
  \iow_open:Nn \c_iow_err_stream {#2}
  \err_kernel_info:n{Errorfile~`#2'~opened~for~output}
  \tlp_gset:Nn \g_err_curr_fname{#2}
  \tlp_new:Nn #1{#2}}
```

`\err_file_close:N` This function closes the current error file.

```
\def_new:Npn \err_file_close:N#1{
```

Before we close the stream, we write out a final error handler that catches mismatch within error message labels and their calls. Actually this should be integrated into `\err_file_new:Nn`, too.

```
  \tlp_if_eq:NNF#1\g_err_curr_fname
     {\err_latex_bug:x{You~closed~the~wrong~error~file~`#1'.~
         Open~is~`\g_err_curr_fname'.}}
  \iow_long_unexpanded:Nn \c_iow_err_stream {\err_latex_bug:x{Didn't~find~the~
        correct~error~message~to~show.\iow_newline:
        Was~searching~for~a~function~
        with~the~following~meaning:\iow_newline:
          \token_to_string:N\token_to_meaning:N
          \token_to_string:N\l_err_label_token}
```

The `\group_end:` here matches the one from `\err_interrupt:NNw` that is used to hide changes to `\par` etc.

```
      \group_end:}
  \iow_close:N \c_iow_err_stream
  \err_kernel_info:n{Errorfile~`\g_err_curr_fname'~closed}
  \tlp_gset_eq:NN\g_err_curr_fname\c_empty_tlp
}
```

19.6.5 Declaring an error message in the error file

`\err_interrupt_new:NNNnnn` This function declares a new error message. `\err_interrupt_new:NNNnnn` ⟨*errfile*⟩ ⟨*errlabel*⟩ ⟨*argno*⟩ ⟨*errmsg*⟩ ⟨*helpmsg*⟩ ⟨*code*⟩. That error message is fetched from the error file ⟨*errfile*⟩. The label to search for is ⟨*errlabel*⟩, the error handler has ⟨*argno*⟩ number of arguments (actually ⟨*argno*⟩ + 3 since the ⟨*errmsg*⟩, ⟨*helpmsg*⟩ and ⟨*code*⟩ are also arguments), and ⟨*errmsg*⟩ is the message to display, ⟨*helpmsg*⟩ is the message that is displayed when the user enters `h`, while ⟨*code*⟩ is extra code to perform when the error occurs. ⟨*code*⟩ is perhaps not necessary, we will see.

We have to check that the label associated with the error message is unique. This means that its replacement text (labels are simply arbitrary functions) is different from the replacement text of any other label in the same error set.

```
\def_new:Npn \err_interrupt_new:NNNnnn #1{
```

Both ⟨*errmsg*⟩ and ⟨*code*⟩ might contain hashmarks denoting arguments to the error handler.

```
  \group_begin: \char_set_catcode:nn{`\#}{12}
```

We also have to check that output goes to the correct error file.

```
  \if_meaning:NN#1\g_err_curr_fname
  \else:
      \err_latex_bug:x{Error~text~goes~to~wrong~err~file:~
          `\g_err_curr_fname'~is~open~but~you~requested~
```

```
          `#1'}
  \fi:
  \err_interrupt_new_aux:w}
\def_long_new:Npn \err_interrupt_new_aux:w #1#2#3#4#5{
  \iow_long_unexpanded:Nn \c_iow_err_stream
        {\err_msgline_aux:NNnnn #1#2{#3}{#4}{#5}\use_noop:}
  \group_end:}
```

`\err_msgline_aux:NNnnn` This function is executed when an error file is read back by `\err_interrupt:NNw`. It compares its first argument against `\l_err_label_token` using `\if_meaning:NN` and if this fails the function does nothing; otherwise it defines `\err_display_aux:w` in a way that it will pick up the arguments (if any) from the code and generates a suitable error message.

```
\def_new:Npn \err_msgline_aux:NNnnn #1#2#3#4#5{
  \if_meaning:NN#1\l_err_label_token
```

At the moment we simply use the old LaTeX error code and `\renewcommand` to generate the error handler. After displaying the error message we insert error code this can be manipulated by the user with the deletion/insertion facility of TeX's error mechanism.

The `\group_end:` at the very beginning matches the `\group_begin:` when the file starts.

```
        \cmd_declare:Nnn\err_display_aux:w [#2]{
           \group_end:
           \toks_gset:Nx\g_err_help_toks{#4}
           \io_put_term:x{LaTeX~error~\io_show_file_lineno:.\iow_newline:
               \text_put_sp:\text_put_four_sp: \text_put_sp:
               See~LaTeX~manual~for~explanation.\iow_newline:
               \text_put_sp:\text_put_four_sp: \text_put_sp:
               Type~\text_put_sp: H~<return>~\text_put_sp: for~
               immediate~help.}
           \err_message:x{#3}
           #5}
        \tex_endinput:D
   \fi:}
```

`\err_display_aux:w` We should make sure that this function is definable.

```
\def_new:Npn \err_display_aux:w {}
```

19.6.6 Kernel specific functions

`\err_kernel_interrupt:Nw` `\err_kernel_interrupt:Nw` is just the abbreviation to read from the standard system error file.

```
\def_new:Npn \err_kernel_interrupt:Nw {\err_interrupt:NNw \c_kernel_err_tlp}
```

`rr_kernel_interrupt_new:NNnnn` To ease the coding in case of system messages that should all go to one and the same error file (if!) we also have the following function.

```
\def_new:Npn \err_kernel_interrupt_new:NNnnn {
        \err_interrupt_new:NNNnnn \c_kernel_err_tlp}
```

\err_kernel_info:n
\err_kernel_warn:n
\err_kernel_fatal:n
\err_kernel_info_noline:n
\err_kernel_warn_noline:n
\err_kernel_fatal_noline:n

These variants are specific for the LaTeX kernel.

```
\def_new:Npn \err_kernel_info:n #1 {
  \err_info:nn {LaTeX~Info:~#1}
               {\text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
\def_new:Npn \err_kernel_warn:n #1 {
  \err_warn:nn {LaTeX~Warning:~#1}
               {\text_put_sp:\text_put_sp:\text_put_sp:
                \text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
\def_new:Npn \err_kernel_fatal:n #1 {
  \err_fatal:nn {LaTeX~Fatal:~#1}
                {\text_put_sp:
                 \text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
\def_new:Npn \err_kernel_info_noline:n #1 {
  \err_info_noline:nn {LaTeX~Info:~#1}
               {\text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
\def_new:Npn \err_kernel_warn_noline:n #1 {
  \err_warn_noline:nn {LaTeX~Warning:~#1}
               {\text_put_sp:\text_put_sp:\text_put_sp:
                \text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
\def_new:Npn \err_kernel_fatal_noline:n #1 {
  \err_fatal_noline:nn {LaTeX~Fatal:~#1}
                {\text_put_sp:
                 \text_put_four_sp:\text_put_four_sp:\text_put_four_sp:}
  }
⟨/initex | package⟩
```

At a later stage variants may be provided for what in LaTeX 2_ε used to be called document classes and packages.

\c_kernel_err_tlp

Most error messages will go to the system error file; it's name is stored in `\c_kernel_err_tlp`.

```
⟨initex⟩\err_file_new:Nn \c_kernel_err_tlp {ltxkernel.err}
⟨package⟩\err_file_new:Nn \c_kernel_err_tlp {l3in2e.err}
```

The code below is a temporary implementation of a few of LaTeX209 error messages with the new syntax. They are only included in the package as the LaTeX3 kernel will certainly have it's own error message definitions that differ from LaTeX 2_ε's way of signalling errors. These primarily serve as an example on how to use this concept of dealing with errors.

First we declare a couple of helper macros that contain texts that are used frequently throughout LaTeX.

```
⟨*package⟩
\def:Npn\err_help_ignored: {
  Your~command~was~ignored.\iow_newline:
  Type \text_put_sp: I~<command>~<return>
  \text_put_sp: to~replace~it~with~another~command,\iow_newline:
  or~\text_put_sp: <return> \text_put_sp: to~continue~without~it.}
```

```
\def:Npn\err_help_textlost: {
  You've~lost~some~text.\text_put_sp: \err_help_return_or_X:}

\def:Npn\err_help_return_or_X: {
  Try~typing\text_put_sp: <return>
  \text_put_sp: to~proceed.\iow_newline:
  If~that~doesn't~work,~type
  \text_put_sp: X~<return>\text_put_sp: to~quit.}

\def:Npn\err_help_trouble: {
  You're~ in~ trouble~ here.
  \text_put_sp:\err_help_return_or_X:}
```

Below are the definitions of the complete messages

```
\err_kernel_interrupt_new:NNnnn\cs_free_p:N{1}
   {Command~name~`\tex_string:D#1'~already~used}
   {You~tried~to~define~a~command~which~already~has~
     a~meaning.\iow_newline:
     If~you~really~want~to~redefine~it~try~
     \token_to_string:N\cmd_declare:Nnn\text_put_sp:
     next~time.\iow_newline:
     For~this~run~I~will~ignore~your~definition.}
   {}

\err_kernel_interrupt_new:NNnnn\newline{0}
   {There's~no~line~here~to~end}
   {You~tried~to~end~a~line~at~a~place~where~I~thought~
     we~were~already~between~paragraphs.}
   {}

\err_kernel_interrupt_new:NNnnn\newcnt{0}
   {No~such~counter}
   {The~counter~name~mentioned~in~the~operation~is~not~
     known~to~me.\iow_newline:
     Check~the~spelling.}
   {}

\err_kernel_interrupt_new:NNnnn\nodocument{0}
    {Missing~\token_to_string:N\begin{document}}
    {\err_help_trouble:}
    {}

\err_kernel_interrupt_new:NNnnn\badmath{0}
    {Bad~math~environment~delimiter}
    {\err_help_ignored:}
    {}

\err_kernel_interrupt_new:NNnnn\toodeep{0}
    {Too~deeply~nested}
    {\err_help_trouble:}
    {}
```

```
\err_kernel_interrupt_new:NNnnn\badpoptabs{0}
    {\token_to_string:N\pushtabs \text_put_sp:
      and~\token_to_string:N\poptabs
      \text_put_sp: don't~match}
    {\err_help_trouble:}
    {}

\err_kernel_interrupt_new:NNnnn\badtab{0}
    {Undefined~tab~position}
    {\err_help_trouble:}
    {}

\err_kernel_interrupt_new:NNnnn\preamerr{}
    {\if_case:w #1~Illegal~character\or:
      Missing~@-exp\or: Missing~p-arg\fi:\text_put_sp:
      in~array~arg}
    {\err_help_trouble:}
    {}

\err_kernel_interrupt_new:NNnnn\badlinearg{}
    {Bad~\token_to_string:N\line
      \text_put_sp: or~\token_to_string:N\vector
      \text_put_sp: argument}
    {\err_help_textlost:}
    {}

\err_kernel_interrupt_new:NNnnn\parmoderr{0}
    {Not~in~outer~par~mode}
    {\err_help_textlost:}
    {}

\err_kernel_interrupt_new:NNnnn\fltovf{0}
    {Too~many~unprocessed~floats}
    {\err_help_textlost:}
    {}

\err_kernel_interrupt_new:NNnnn\badcrerr{0}
    {Bad~use~of~\token_to_string:N\\}
    {\err_help_return_or_X:}
    {}

\err_kernel_interrupt_new:NNnnn\noitemerr{0}
    {Something's~wrong--perhaps~a~missing~
      \token_to_string:N\item}
    {\err_help_return_or_X:}
    {}

\err_kernel_interrupt_new:NNnnn\notprerr{0}
    {Can~be~used~only~in~preamble}
    {\err_help_ignored:}
    {}

\err_file_close:N\c_kernel_err_tlp
⟨package⟩
```

Show token usage:

```
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

20 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_`.

20.1 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_p:N`
`\box_if_empty_p:c`
`\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_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c
\box_set_to_last:N ⟨*box*⟩
Sets ⟨*box*⟩ equal to the previous box \R_last_box and removes \R_last_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_last_box
`\R_last_box` is a read-only box register. You can set other boxes to this box, which will then be removed from the current list.

20.2 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_begin:c`
`\hbox_set_inline_end:`
`\hbox_gset_inline_begin:N`
`\hbox_gset_inline_begin:c`
`\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`.

20.3 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_ht:nn {⟨dimen⟩} ⟨contents⟩`
\vbox_to_zero:n `\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`.

20.4 The Implementation

Announce and ensure that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package&!check⟩\RequirePackage{l3prg,l3token}\par
⟨package & check⟩\RequirePackage{l3chk}\par
⟨*initex | package⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

20.4.1 Generic boxes

\box_new:N
\box_new_l:N
\box_new:c

Defining a new ⟨*box*⟩ register.

```
⟨*initex⟩
\alloc_setup_type:nnn {box} \c_zero \c_max_register_num
```

Now, remember that `\box255` has a special role in TEX, it shouldn't be allocated...

```
\seq_put_right:Nn \g_box_allocation_seq {255}
\def_new:Npn \box_new:N   #1 {\alloc_reg:NnNN g {box} \tex_mathchardef:D #1}
\def_new:Npn \box_new_l:N #1 {\alloc_reg:NnNN l {box} \tex_mathchardef:D #1}
⟨/initex⟩
```

When we run on top of LaTeX, we just use its allocation mechanism.

```
⟨package⟩\let_new:NN \box_new:N \newbox
\def_new:Npn \box_new:c {\exp_args:Nc \box_new:N}
```

`\if_hbox:N` `\if_vbox:N` `\if_box_empty:N` The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

```
\let_new:NN \if_hbox:N        \tex_ifhbox:D
\let_new:NN \if_vbox:N        \tex_ifvbox:D
\let_new:NN \if_box_empty:N   \tex_ifvoid:D
```

`\box_if_empty_p:N` `\box_if_empty_p:c` `\box_if_empty:NTF` `\box_if_empty:cTF` `\box_if_empty:NT` `\box_if_empty:cT` `\box_if_empty:NF` `\box_if_empty:cF` Testing if a ⟨*box*⟩ is empty/void.

```
\def_new:Npn \box_if_empty_p:N #1{
  \if_box_empty:N #1 \c_true \else: \c_false \fi:}
\def_new:Npn \box_if_empty_p:c {\exp_args:Nc \box_if_empty_p:N}
\def_test_function_new:npn {box_if_empty:N}#1{\if_box_empty:N #1}
\def_new:Npn \box_if_empty:cTF {\exp_args:Nc \box_if_empty:NTF}
\def_new:Npn \box_if_empty:cT {\exp_args:Nc \box_if_empty:NT}
\def_new:Npn \box_if_empty:cF {\exp_args:Nc \box_if_empty:NF}
```

`\box_set_eq:NN` `\box_set_eq:cN` `\box_set_eq:Nc` `\box_set_eq:cc` Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```
\def_new:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2}
\def_new:Npn \box_set_eq:cN {\exp_args:Nc \box_set_eq:NN}
\def_new:Npn \box_set_eq:Nc {\exp_args:NNc \box_set_eq:NN}
\def_new:Npn \box_set_eq:cc {\exp_args:Ncc \box_set_eq:NN}
```

`\box_gset_eq:NN` `\box_gset_eq:cN` `\box_gset_eq:Nc` `\box_gset_eq:cc` Global version of the above.

```
\def_new:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN}
\def_new:Npn \box_gset_eq:cN {\exp_args:Nc \box_gset_eq:NN}
\def_new:Npn \box_gset_eq:Nc {\exp_args:NNc \box_gset_eq:NN}
\def_new:Npn \box_gset_eq:cc {\exp_args:Ncc \box_gset_eq:NN}
```

`\R_last_box` A different name for this read-only primitive.

```
\let_new:NN \R_last_box \tex_lastbox:D
```

`\box_set_to_last:N` `\box_set_to_last:c` `\box_gset_to_last:N` `\box_gset_to_last:c` Set a box to the previous box.

```
\def_new:Npn \box_set_to_last:N #1{\tex_setbox:D#1\R_last_box}
\def_new:Npn \box_set_to_last:c {\exp_args:Nc \box_set_to_last:N}
\def_new:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N}
\def_new:Npn \box_gset_to_last:c {\exp_args:Nc \box_gset_to_last:N}
```

`\box_move_left:nn` `\box_move_right:nn` `\box_move_up:nn` `\box_move_down:nn` Move box material in different directions.

```
\def_long_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1}{#2}}
\def_long_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1}{#2}}
\def_long_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1}{#2}}
\def_long_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1}{#2}}
```

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

Clear a ⟨*box*⟩ register.

```
\def_new:Npn \box_clear:N #1{\box_set_eq:NN #1 \c_empty_box }
\def_new:Npn \box_clear:c {\exp_args:Nc \box_clear:N }
\def_new:Npn \box_gclear:N {\pref_global:D\box_clear:N}
\def_new:Npn \box_gclear:c {\exp_args:Nc \box_gclear:c }
```

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

Accessing the height, depth, and width of a ⟨*box*⟩ register.

```
\let_new:NN \box_ht:N \tex_ht:D
\def_new:Npn \box_ht:c {\exp_args:Nc \box_ht:N}
\let_new:NN \box_dp:N \tex_dp:D
\def_new:Npn \box_dp:c {\exp_args:Nc \box_dp:N}
\let_new:NN \box_wd:N \tex_wd:D
\def_new:Npn \box_wd:c {\exp_args:Nc \box_wd:N}
```

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

Using a ⟨*box*⟩. This is just TEX primitives with meaningful names.

```
\let_new:NN \box_use_clear:N \tex_box:D
\def_new:Npn \box_use_clear:c {\exp_args:Nc \box_use_clear:N}
\let_new:NN \box_use:N \tex_copy:D
\def_new:Npn \box_use:c {\exp_args:Nc \box_use:N}
```

\box_show:N
\box_show:c

Show the contents of a box and write it into the log file.

```
\let:NN \box_show:N \tex_showbox:D
\def_new:Npn \box_show:c {\exp_args:Nc \box_show:N}
```

\c_empty_box
\l_tmpa_box
\l_tmpb_box

We allocate some ⟨*box*⟩ registers here (and borrow a few from LATEX).

```
⟨package⟩\let:NN \c_empty_box \voidb@x
⟨package⟩\let_new:NN \l_tmpa_box \@tempboxa
⟨initex⟩\box_new:N \c_empty_box
⟨initex⟩\box_new:N \l_tmpa_box
\box_new:N \l_tmpb_box
```

20.4.2 Vertical boxes

\vbox:n

Put a vertical box directly into the input stream.

```
\def_new:Npn \vbox:n {\tex_vbox:D \scan_stop:}
```

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

Storing material in a vertical box with a natural height.

```
\def_long_new:Npn \vbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_vbox:D {#2}}
\def_new:Npn \vbox_set:cn {\exp_args:Nc \vbox_set:Nn}
\def_new:Npn \vbox_gset:Nn  {\pref_global:D \vbox_set:Nn}
\def_new:Npn \vbox_gset:cn {\exp_args:Nc \vbox_gset:Nn}
```

\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

Storing material in a vertical box with a specified height.

```
\def_long_new:Npn \vbox_set_to_ht:Nnn #1#2#3 {
```

```
  \tex_setbox:D #1 \tex_vbox:D to #2 {#3}}
\def_new:Npn \vbox_set_to_ht:cnn{\exp_args:Nc \vbox_set_to_ht:Nnn }
\def_new:Npn \vbox_gset_to_ht:Nnn {\pref_global:D \vbox_set_to_ht:Nnn }
\def_new:Npn \vbox_gset_to_ht:cnn{\exp_args:Nc \vbox_gset_to_ht:Nnn }
\def_new:Npn \vbox_gset_to_ht:ccn {\exp_args:Ncc \vbox_gset_to_ht:Nnn}
```

`\vbox_set_inline_begin:N` `\vbox_set_inline_end:` `\vbox_gset_inline_begin:N` `\vbox_set_inline_end:` Storing material in a vertical box. This type is useful in environment definitions.

```
\def_new:Npn \vbox_set_inline_begin:N  #1 {
  \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
\let_new:NN \vbox_set_inline_end: \c_group_end_token
\def_new:Npn \vbox_gset_inline_begin:N {
  \pref_global:D \vbox_set_inline_begin:N }
\let_new:NN \vbox_gset_inline_end: \c_group_end_token
```

`\vbox_to_ht:nn` `\vbox_to_zero:n` Put a vertical box directly into the input stream.

```
\def_long_new:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}}
\def_long_new:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}}
```

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```
\def_new:Npn \vbox_set_split_to_ht:NNn #1#2#3{
  \tex_setbox:D #1 \tex_vsplit:D #2 to #3
}
```

`\vbox_unpack:N` `\vbox_unpack:c` `\vbox_unpack_clear:N` `\vbox_unpack_clear:c` Unpacking a box and if requested also clear it.

```
\let_new:NN \vbox_unpack:N \tex_unvcopy:D
\def_new:Npn \vbox_unpack:c {\exp_args:Nc \vbox_unpack:N}
\let_new:NN \vbox_unpack_clear:N \tex_unvbox:D
\def_new:Npn \vbox_unpack_clear:c {\exp_args:Nc \vbox_unpack_clear:N}
```

20.4.3 Horizontal boxes

`\hbox:n` Put a horizontal box directly into the input stream.

```
\def_new:Npn \hbox:n {\tex_hbox:D \scan_stop:}
```

`\hbox_set:Nn` `\hbox_set:cn` `\hbox_gset:Nn` `\hbox_gset:cn` Assigning the contents of a box to be another box. This clears the second box globally (that's how TEX does it).

```
\def_long_new:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}}
\def_new:Npn \hbox_set:cn {\exp_args:Nc \hbox_set:Nn}
\def_new:Npn \hbox_gset:Nn  {\pref_global:D \hbox_set:Nn}
\def_new:Npn \hbox_gset:cn {\exp_args:Nc \hbox_gset:Nn}
```

`\hbox_set_to_wd:Nnn` `\hbox_set_to_wd:cnn` `\hbox_gset_to_wd:Nnn` `\hbox_gset_to_wd:cnn` Storing material in a horizontal box with a specified width.

```
\def_long_new:Npn \hbox_set_to_wd:Nnn #1#2#3 {
  \tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3}}
\def_new:Npn \hbox_set_to_wd:cnn{\exp_args:Nc \hbox_set_to_wd:Nnn }
\def_new:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn }
\def_new:Npn \hbox_gset_to_wd:cnn{\exp_args:Nc \hbox_gset_to_wd:Nnn }
```

\hbox_set_inline_begin:N
\hbox_set_inline_begin:c
\hbox_set_inline_end:
\hbox_gset_inline_begin:N
\hbox_gset_inline_begin:c
\hbox_set_inline_end:

Storing material in a horizontal box. This type is useful in environment definitions.

```
\def_new:Npn \hbox_set_inline_begin:N  #1 {
  \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
\def:Npn \hbox_set_inline_begin:c {\exp_args:Nc
  \hbox_set_inline_begin:N}
\let_new:NN \hbox_set_inline_end: \c_group_end_token
\def_new:Npn \hbox_gset_inline_begin:N {
  \pref_global:D \hbox_set_inline_begin:N }
\def:Npn \hbox_gset_inline_begin:c {\exp_args:Nc
  \hbox_gset_inline_begin:N }
\let_new:NN \hbox_gset_inline_end: \c_group_end_token
```

\hbox_to_wd:nn
\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
\def_long_new:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}}
\def_long_new:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}}
```

\hbox_unpack:N
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\let_new:NN \hbox_unpack:N \tex_unhcopy:D
\def_new:Npn \hbox_unpack:c {\exp_args:Nc \hbox_unpack:N}
\let_new:NN \hbox_unpack_clear:N \tex_unhbox:D
\def_new:Npn \hbox_unpack_clear:c {\exp_args:Nc \hbox_unpack_clear:N}
```

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

21 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.

21.1 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*⟩.

21.2 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3num}
⟨package⟩\RequirePackage{l3io}
⟨package⟩\RequirePackage{l3seq}
⟨package⟩\RequirePackage{l3int}
```

It might speed up the processing of documents when certain parts of the document style file are 'precompiled' and stored in a separate file.

`\c_cs_dump_stream` We need to allocate an output stream in order to be able to write the precompiled code out. Stream number for the dump.

```
⟨*initex | package⟩
⟨*precompile⟩
\iow_new:N\c_cs_dump_stream
```

`\g_cs_dump_name_tlp` This ⟨*tlp*⟩ is used to store the name of the file.

```
\tlp_new:Nn\g_cs_dump_name_tlp{}
```

\g_cs_dump_seq While processing the documentstyle we build up a list of control sequence names to be dumped. For this purpose we use the \g_cs_dump_seq sequence.

```
\seq_new:N\g_cs_dump_seq
```

\cs_record_name:N
\cs_record_name:c These functions mark a control sequence for dumping into a precompiled style.

When the trace 'module' is included in the code we also write information about the control sequence into a .dmp file.

```
\def_new:Npn\cs_record_name:N#1{
⟨*trace⟩
\seq_gput_left:Nn
  \g_cs_trace_seq#1
⟨/trace⟩
\seq_gput_left:Nn
  \g_cs_dump_seq#1}
\def_new:Npn\cs_record_name:c{\exp_args:Nc\cs_record_name:N}
```

As you can see \cs_dump: wording this pre-dates LaTeX 2ε *Frank*

When a document style calls \cs_dump: it triggers this code to write all the precompilation information out to a file.

Before dumping, we write a message to the terminal informing the 'user' of this fact.

```
\def_new:Npn\cs_dump:{
\iow_expanded_term:n{Precompiling~style~into~(\g_cs_dump_name_tlp)}
\iow_open:Nn\c_cs_dump_stream{\g_cs_dump_name_tlp}
```

The first thing we write on a 'dump' file is a command that allows us to use * in control sequences. We also need to be able to write to (and read from) the file internal control sequences, containing _ and :.

```
\iow_expanded:Nn\c_cs_dump_stream
{\group_begin:
\tex_catcode:D`\token_to_string:N\*=11\scan_stop:
\token_to_string:N\CodeStart
}
\seq_map_inline:Nn
\g_cs_dump_seq
{\tex_message:D{.}
\iow_expanded:Nn\c_cs_dump_stream
```

We use a direct \gdef:Npn to disable any type of local/global check on the pointers.

```
      {\exp_not:n{\gdef:Npn ##1}
          {\tlp_to_str:N##1}}
}
```

We also need to remember the current values of the \g_gen_sym_num and \g_ggen_sym_num counters to allow further updates after a database was dumped.

```
\iow_expanded:Nn \c_cs_dump_stream {\exp_not:n{\num_gset:Nn
                  \g_gen_sym_num}
                  {\num_use:N\g_gen_sym_num}^^J
\exp_not:n{\num_gset:Nn \g_ggen_sym_num}
```

```
                {\num_use:N\g_ggen_sym_num}}
\iow_expanded:Nn
\c_cs_dump_stream
{\group_end:}
\iow_close:N\c_cs_dump_stream
\tex_message:D{~finished}
}
⟨/precompile⟩
```

`\cs_load_dump:n` A function to read a precompiled file into memory and skip until a `\cs_dump:` command is found. If no such file is found, processing continues and a subsequent `\cs_dump:` command will then create the dump file.

```
\def_new:Npn\cs_load_dump:n#1{
\file_not_found:nTF{#1.cmp}
⟨*precompile⟩
{\tlp_gset:Nn\g_cs_dump_name_tlp{#1.cmp}}
⟨/precompile⟩
⟨−precompile⟩ {\tex_errmessage:D{Cannot~ dump~ with~ this~ format}}
{\input{#1.cmp}
\let:NN\cs_dump:\fi:
\if_false:}}
```

`\g_gen_sym_num` `\g_ggen_sym_num` Two counters to make up new local or global *short* names in pointer structures like the LDB. We use a fake counters since operations with them are seldom.

```
\num_new:N\g_gen_sym_num \num_gset:Nn\g_gen_sym_num{0}
\num_new:N\g_ggen_sym_num \num_gset:Nn\g_ggen_sym_num{0}
```

`\cs_gen_sym:N` `\cs_ggen_sym:N` We need to be able to generate control sequences on the fly. They will exist of a prefix, either `l*` or `g*`, followed by the value of the counter `\g_gen_sym_num` (`\g_ggen_sym_num`) in roman numeral representation. The generated control sequence is locally stored in the token that was passed in `#1`.

```
\def_new:Npn\cs_gen_sym:N#1{
\num_gincr:N\g_gen_sym_num
\tlp_set:Nc#1{l*\tex_romannumeral:D\num_use:N\g_gen_sym_num}
⟨*precompile⟩
\exp_after:NN\cs_record_name:N#1
⟨/precompile⟩
```

We still want to define the initial value for the new symbol globally to make sure that during compilation something is written to the output file.

```
\exp_after:NN\tlp_clear_new:N#1}
```

The global variant

```
\def_new:Npn\cs_ggen_sym:N#1{
\num_gincr:N\g_ggen_sym_num
\tlp_set:Nc#1{g*\tex_romannumeral:D\num_use:N\g_ggen_sym_num}
⟨*precompile⟩
\exp_after:NN\cs_record_name:N#1
⟨/precompile⟩
\exp_after:NN\tlp_clear_new:N#1}
```

`\g_cs_trace_seq` A sequence which holds the control sequence names that are to be dumped. They are stored together with their meaning.

ATTENTION: as we currently don't distribute allocation routines for primitive registers this code will have no effect!

```
⟨*trace⟩
\seq_new:N\g_cs_trace_seq
```

`\g_register_trace_seq` Sequence holding the register names to be dumped with their corresponding values.

ATTENTION: as we currently don't distribute allocation routines for primitive registers this code will have no effect!

```
\seq_new:N\g_register_trace_seq
```

`\cs_record_meaning:N` Function marking a control sequence for dumping with meaning.

```
\def:Npn\cs_record_meaning:N#1{
\seq_gput_left:Nn
  \g_cs_trace_seq#1}
```

`\register_record_name:N` Function marking a register for dumping with value.

```
\def:Npn\register_record_name:N#1{
\seq_gput_left:Nn
  \g_register_trace_seq#1}
```

`\dumpLaTeXstate` The function `\dumpLaTeXstate` is used to write control sequences and registers, together with their meaning or value in the `.dmp` file. We write informational messages to the terminal during the dump.

ATTENTION: as we currently don't distribute allocation routines for primitive registers this part of the code will dump nothing unless `\register_record_name:N` is explicitly used.

```
\def_new:Npn\dumpLaTeXstate#1{
  \iow_expanded_term:n{Dumping~commands~into~(#1.dmp)}
  \iow_open:Nn\c_cs_dump_stream{#1.dmp}
  \seq_map_inline:Nn
       \g_cs_trace_seq
       {\tex_message:D{.}
        \iow_expanded:Nn\c_cs_dump_stream
             {\token_to_string:N##1~
              \token_to_meaning:N##1}
       }
  \tex_message:D{~registers}
  \seq_map_inline:Nn
       \g_register_trace_seq
       {\tex_message:D{.}
        \iow_expanded:Nn\c_cs_dump_stream
             {\token_to_string:N##1
              \the_internal:D##1}
       }
```

```
\tex_message:D{~finished}
}
```
⟨/trace⟩
⟨/initex | package⟩

Show token usage:

⟨*showmemory⟩
```
\showMemUsage
```
⟨/showmemory⟩

22 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_`.

The documentation needs some updating.

22.1 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.[10]

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

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`.

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

22.2 Recursion

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

`\q_recursion_tail` This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop` This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`

`\quark_if_recursion_tail_stop:n {`⟨*list element*⟩`}`
`\quark_if_recursion_tail_stop:N` ⟨*list element*⟩

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

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

```
\tlp_set:Nn \l_tmpa_tlp { ⟨list element⟩ }
```

\quark_if_recursion_tail_stop_do:Nn
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on

```
\quark_if_recursion_tail_stop_do:nn
{⟨list element⟩} { ⟨post action⟩ }
\quark_if_recursion_tail_stop_do:Nn
⟨list element⟩ { ⟨post action⟩ }
```

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

22.3 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.

22.4 The Implementation

We start by ensuring that the required packages are loaded. We check for `l3expan` since this a basic package that is essential for use of any higher-level package.

```
⟨package⟩\ProvidesExplPackage
⟨package⟩  {\filename}{\filedate}{\fileversion}{\filedescription}
⟨package⟩\RequirePackage{l3expan}\par
⟨*initex | package⟩
```

`\quark_new:N` Allocate a new quark.

```
\def_new:Npn \quark_new:N #1{\tlp_new:Nn #1{#1}}
```

`\q_stop` `\q_no_value` `\q_nil` `\q_stop` is often used as a marker in parameter text, `\q_no_value` is the canonical missing value, and `\q_nil` represents a nil pointer in some data structures.

```
\quark_new:N \q_stop
\quark_new:N \q_no_value
\quark_new:N \q_nil
```

`\q_error` `\q_mark` We need two additional quarks. `\q_error` delimits the end of the computation for purposes of error recovery. `\q_mark` is used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

```
\quark_new:N\q_error
\quark_new:N\q_mark
```

\q_recursion_tail
\q_recursion_stop

Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
\quark_new:N\q_recursion_tail
\quark_new:N\q_recursion_stop
```

uark_if_recursion_tail_stop:n
uark_if_recursion_tail_stop:N
uark_if_recursion_tail_stop:o

When doing recursions it is easy to spend a lot of time testing if we found the end marker. To avoid this, we use a recursion end marker every time we do this kind of task. Also, if the recursion end marker is found, we wrap things up and finish.

```
\def_long_new:Npn \quark_if_recursion_tail_stop:n #1 {
  \exp_after:NN\if_meaning:NN
    \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
    \exp_after:NN \use_none_delimit_by_q_recursion_stop:w
  \fi:
}
\def_long_new:Npn \quark_if_recursion_tail_stop:N #1 {
  \if_meaning:NN#1\q_recursion_tail
    \exp_after:NN \use_none_delimit_by_q_recursion_stop:w
  \fi:
}
\def_new:Npn \quark_if_recursion_tail_stop:o{
  \exp_args:No\quark_if_recursion_tail_stop:n
}
```

_if_recursion_tail_stop_do:nn
_if_recursion_tail_stop_do:Nn
_if_recursion_tail_stop_do:on

```
\def_long_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2 {
  \exp_after:NN\if_meaning:NN
    \quark_if_recursion_tail_aux:w #1?\q_nil\q_recursion_tail\q_recursion_tail
    \exp_after:NN \use_arg_i_delimit_by_q_recursion_stop:nw
  \else:
    \exp_after:NN\use_none:n
  \fi:
  {#2}
}
\def_long_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2 {
  \if_meaning:NN #1\q_recursion_tail
    \exp_after:NN \use_arg_i_delimit_by_q_recursion_stop:nw
  \else:
    \exp_after:NN\use_none:n
  \fi:
  {#2}
}
\def_new:Npn \quark_if_recursion_tail_stop_do:on{
  \exp_args:No\quark_if_recursion_tail_stop_do:nn
}
```

quark_if_recursion_tail_aux:w
delimit_by_q_recursion_stop:w
elimit_by_q_recursion_stop:nw

Helper macros for picking up the first token of a list to see if it is `\q_recursion_tail` and to stop the recursion.

```
\def_long_new:Npn \quark_if_recursion_tail_aux:w
  #1#2\q_nil\q_recursion_tail{#1}
\def_long_new:Npn\use_none_delimit_by_q_recursion_stop:w
  #1\q_recursion_stop {}
\def_long_new:Npn\use_arg_i_delimit_by_q_recursion_stop:nw
  #1#2\q_recursion_stop {#1}
```

\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_p:n
\quark_if_no_value:nTF
\quark_if_no_value:nT
\quark_if_no_value:nF
\quark_if_no_value:nFT

Here we test if we found a special quark as the first argument.

```
\def_long_test_function_new:npn {quark_if_no_value:N} #1 {
```

We better start with \q_no_value as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like aabc instead of a single token.[11]

```
    \if_meaning:NN\q_no_value#1}
\def_long_new:Npn \quark_if_no_value_p:N #1{
  \if_meaning:NN \q_no_value #1 \c_true
  \else: \c_false \fi:
}
```

We also provide an n type. If run under a sufficiently new pdfε-TEX, it uses a built-in primitive for string comparisons, otherwise it uses the slower \str_if_eq_var_p:nf function. In the latter case it would be faster to use a temporary token list pointer but it would render the function non-expandable. Using the pdfε-TEX primitive is the preferred approach. Note that we have to add a manual space token in the first part of the comparison, otherwise it is gobbled by \str_if_eq_var_p:nf. The reason for using this function instead of \str_if_eq_p:nn is that a sequence like ␣\q_no_value will test equal to \q_no_value using the latter test function and unfortunately this example turned up in one application.

```
\cs_if_really_free:cTF{pdf_strcmp:D}{
  \def_long_new:Npn \quark_if_no_value_p:n #1{
    \if:w \exp_args:No \str_if_eq_var_p:nf
⟨package⟩       {\token_to_string:N\q_no_value\space}
⟨initex⟩      {\token_to_string:N\q_no_value\text_put_sp:}
      {\tlist_to_str:n{#1}}
      \c_true
    \else:
      \c_false
    \fi:
  }
}
{
  \def_long_new:Npn \quark_if_no_value_p:n #1{
    \if_num:w
    \pdf_strcmp:D {\exp_not:N \q_no_value}{\exp_not:n{#1}}=\c_zero
    \c_true \else: \c_false \fi:
  }
}
\def_long_test_function_new:npn {quark_if_no_value:n} #1 {
  \if:w \quark_if_no_value_p:n{#1}}
```

[11] It may still loop in special circumstances however!

We also define a version where the true and false code is ordered differently.

```
\def_long:Npn \quark_if_no_value:nFT #1{
  \if:w \quark_if_no_value_p:n{#1}
      \exp_after:NN\use_arg_ii:nn
    \else:
      \exp_after:NN\use_arg_i:nn
    \fi:
}
```

`\quark_if_nil_p:N` A function to check for the presence of `\q_nil`.
`\quark_if_nil:NTF`
`\quark_if_nil:NT`
`\quark_if_nil:NF`

```
\def_long_new:Npn \quark_if_nil_p:N #1{
  \if_meaning:NN \q_nil #1 \c_true
  \else: \c_false \fi:
}
\def_long_test_function_new:npn {quark_if_nil:N}#1{
  \if_meaning:NN\q_nil#1}
```

`\quark_if_nil_p:n` A function to check for the presence of `\q_nil`.
`\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`

```
\cs_if_really_free:cTF{pdf_strcmp:D}{
  \def_long_new:Npn \quark_if_nil_p:n #1{
    \if:w \exp_args:No \str_if_eq_var_p:nf
⟨package⟩       {\token_to_string:N\q_nil\space}
⟨initex⟩       {\token_to_string:N\q_nil\text_put_sp:}
      {\tlist_to_str:n{#1}}
      \c_true
    \else:
      \c_false
    \fi:
  }
}
{
  \def_long_new:Npn \quark_if_nil_p:n #1{
    \if_num:w
    \pdf_strcmp:D {\exp_not:N \q_nil}{\exp_not:n{#1}}=\c_zero
    \c_true \else: \c_false \fi:
  }
}
\def_long_test_function_new:npn {quark_if_nil:n} #1 {
  \if:w \quark_if_nil_p:n{#1}}
\def_new:Npn \quark_if_nil_p:o{\exp_args:No\quark_if_nil_p:n}
\def_new:Npn \quark_if_nil:oTF{\exp_args:No\quark_if_nil:nTF}
\def_new:Npn \quark_if_nil:oT {\exp_args:No\quark_if_nil:nT}
\def_new:Npn \quark_if_nil:oF {\exp_args:No\quark_if_nil:nF}
```

Show token usage:

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

23 Control structures

23.1 Choosing modes

`\mode_if_vertical_p:`
`\mode_if_vertical:TF`
`\mode_if_vertical:T`
`\mode_if_vertical:F` `\mode_if_vertical:TF {⟨true code⟩} {⟨false code⟩}`

Determines if TEX is in vertical mode or not and executes either ⟨*true code*⟩ or ⟨*false code*⟩ accordingly.

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF`
`\mode_if_horizontal:T`
`\mode_if_horizontal:F` `\mode_if_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_if_inner_p:`
`\mode_if_inner:TF`
`\mode_if_inner:T`
`\mode_if_inner:F` `\mode_if_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_if_math:TF`
`\mode_if_math:T`
`\mode_if_math:F` `\mode_if_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.

23.1.1 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.

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

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

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

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

23.3 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. Parsing `\iffalse` and `\iftrue` tokens can be quite tricky at times so the easiest is to simply 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 `\predicate` are expandable and expect the input to also be fully expandable. More generic constructs do not contain `predicate` in their names.

23.3.1 The boolean data type

`\bool_new:N`
`\bool_new:c`
`\bool_new:N` ⟨*bool*⟩
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.

23.3.2 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.

\predicate_p:n
\predicate:nTF
\predicate:nT
\predicate:nF

\predicate:nTF {⟨*list of predicates*⟩} {⟨*true*⟩} {⟨*false*⟩}

The functions evaluate the truth value of ⟨*list of predicates*⟩ where each predicate is separated by `&&` or `||` denoting logical And and Or functions. Minimal evaluation is carried out so that whenever a truth value cannot be changed anymore, the remainding tests are not carried out. Hence

```
\predicate_p:n{
  \int_compare_p:nNn 1=1 &&
  \predicate_p:n {
    \int_compare_p:nNn 2=3 ||
    \int_compare_p:nNn 4=4 ||
    \int_compare_p:nNn 1=\error % is skipped
  } &&
  \int_compare_p:nNn 2=2
}
```

returns ⟨*true*⟩.

\predicate_not_p:n

\predicate_not_p:n {⟨*list of predicates*⟩}

`\predicate_not_p:n` reverses the truth value of its argument. Thus

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

ultimately returns ⟨*true*⟩.

23.3.3 Case switches

\prg_case_int:nnn {⟨*integer expr*⟩} {
{⟨*integer expr*$_1$⟩}{⟨*code*$_1$⟩}{⟨*integer expr*$_2$⟩}{⟨*code*$_2$⟩}
...{⟨*integer expr*$_n$⟩}{⟨*code*$_n$⟩}
\prg_case_int:nnn } {⟨*else case*⟩}

This function evaluates the first ⟨*integer expr*⟩ and then compares it to the values found in the list. Thus the expression

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

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

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

\prg_case_int:nnn {⟨*dim expr*⟩} {
{⟨*dim expr*$_1$⟩}{⟨*code*$_1$⟩}{⟨*dim expr*$_2$⟩}{⟨*code*$_2$⟩}
...{⟨*dim expr*$_n$⟩}{⟨*code*$_n$⟩}
\prg_case_dim:nnn } {⟨*else case*⟩}

This function works just like `\prg_case_int:nnn` except it works for ⟨*dim*⟩ registers.

\prg_case_str:nnn {⟨*string*⟩} {
{⟨*string*$_1$⟩}{⟨*code*$_1$⟩}{⟨*string*$_2$⟩}{⟨*code*$_2$⟩}
...{⟨*string*$_n$⟩}{⟨*code*$_n$⟩}
\prg_case_str:nnn } {⟨*else case*⟩}

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

The function is expandable[12] and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

23.3.4 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`.

[12] Provided you use pdfTeX v1.30 or later

23.4 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 }
```

23.5 The Implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{l3quark}
\RequirePackage{l3toks}
\RequirePackage{l3int}
⟨/package⟩
⟨*initex | package⟩
```

23.5.1 Choosing modes

\mode_if_vertical_p:
\mode_if_vertical:TF
\mode_if_vertical:T
\mode_if_vertical:F

For testing vertical mode.

```
\def_new:Npn \mode_if_vertical_p: {
  \if_mode_vertical: \c_true \else: \c_false\fi:}
\def_test_function_new:npn{mode_if_vertical:}{\if_mode_vertical:}
```

\mode_if_horizontal_p: For testing horizontal mode.
\mode_if_horizontal:TF
\mode_if_horizontal:T
\mode_if_horizontal:F

```
\def_new:Npn \mode_if_horizontal_p: {
  \if_mode_horizontal: \c_true \else: \c_false\fi:}
\def_test_function_new:npn{mode_if_horizontal:}{\if_mode_horizontal:}
```

\mode_if_inner_p: For testing inner mode.
\mode_if_inner:TF
\mode_if_inner:T
\mode_if_inner:F

```
\def_new:Npn \mode_if_inner_p: {
  \if_mode_inner: \c_true \else: \c_false\fi:}
\def_test_function_new:npn{mode_if_inner:}{\if_mode_inner:}
```

\mode_if_math:TF For testing math mode. Uses the kern-save \scan_align_safe_stop:.
\mode_if_math:T
\mode_if_math:F

```
\def_test_function_new:npn{mode_if_math:} {
  \scan_align_safe_stop:  \if_mode_math: }
```

Alignment safe grouping and scanning

\group_align_safe_begin:
\group_align_safe_end:

TEX's alignment structures present many problems. As Knuth says himself in *TEX: The Program*: "It's sort of a miracle whenever \halign or \valign work, [...]" One problem relates to commands that internally issues a \cr but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a & with category code 4 we will get some sort of weird error message because the underlying \tex_futurelet:D will store the token at the end of the alignment template. This could be a $\&_4$ giving a message like `! Misplaced \cr.` or even worse: it could be the \endtemplate token causing even more trouble! To solve this we have to open a special group so that TEX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The TEXbook*...

```
\def_new:Npn \group_align_safe_begin: {
  \if_false:{\fi:\if_num:w`}=\c_zero\fi:}
\def_new:Npn \group_align_safe_end:   {\if_num:w`{=\c_zero}\fi:}
```

\scan_align_safe_stop:

When TEX is in the beginning of an align cell (right after the \cr) it is in a somewhat strange mode as it is looking ahead to find an \tex_omit:D or \tex_noalign:D and hasn't looked at the preamble yet. Thus an \tex_ifmmode:D test will always fail unless we insert \scan_stop: to stop TEX's scanning ahead. On the other hand we don't want to insert a \scan_stop: every time as that will destroy kerning between letters[13] Unfortunately there is no way to detect if we're in the beginning of an alignment cell as they have different characteristics depending on column number etc. However we *can* detect if we're in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that \scan_stop: is only inserted iff a) we're in the outer part of an alignment cell and b) the last node *wasn't* a char node or a ligature node.

```
\def_new:Npn \scan_align_safe_stop: {
  \num_compare:nNnT \etex_currentgrouptype:D = \c_six
```

[13] Unless we enforce an extra pass with an appropriate value of \pretolerance.

```
  {
    \num_compare:nNnF \etex_lastnodetype:D = \c_zero
    {
      \num_compare:nNnF \etex_lastnodetype:D = \c_seven
        \scan_stop:
    }
  }
}
```

23.5.2 Making n copies

\prg_replicate:nn
\prg_replicate_aux:N
\prg_replicate_first_aux:N

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. Finally we must ensure that the cascade comes to a peaceful end so we make it so that the original csname TEX is creating is simply `\use_noop:` expanding to nothing.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it will severely affect the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use. An alternative approach is to create a string of `m`'s with `\int_to_roman:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```
\def_new:Npn \prg_replicate:nn #1{
  \cs:w use_noop:
  \exp_after:NN\prg_replicate_first_aux:N
  \int_use:N \int_eval:n{#1} \cs_end:
  \cs_end:
}
\def_new:Npn \prg_replicate_aux:N#1{
  \cs:w prg_replicate_#1:n\prg_replicate_aux:N
}
\def_new:Npn \prg_replicate_first_aux:N#1{
  \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
}
```

Then comes all the functions that do the hard work of inserting all the copies.

```
\def_new:Npn      \prg_replicate_ :n #1{}% no, this is not a typo!
\def_long_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
\def_long_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}#1}
```

```
\def_long_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1}#1#1}
\def_long_new:cpn {prg_replicate_3:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1}
\def_long_new:cpn {prg_replicate_4:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1}
\def_long_new:cpn {prg_replicate_5:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_6:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_7:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_8:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_9:n}#1{
  \cs_end:{#1#1#1#1#1#1#1#1#1}#1#1#1#1#1#1#1#1#1}
```

Users shouldn't ask for something to be replicated once or even not at all but...

```
\def_long_new:cpn {prg_replicate_first_0:n}#1{\cs_end: }
\def_long_new:cpn {prg_replicate_first_1:n}#1{\cs_end: #1}
\def_long_new:cpn {prg_replicate_first_2:n}#1{\cs_end: #1#1}
\def_long_new:cpn {prg_replicate_first_3:n}#1{\cs_end: #1#1#1}
\def_long_new:cpn {prg_replicate_first_4:n}#1{\cs_end: #1#1#1#1}
\def_long_new:cpn {prg_replicate_first_5:n}#1{\cs_end: #1#1#1#1#1}
\def_long_new:cpn {prg_replicate_first_6:n}#1{\cs_end: #1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_first_7:n}#1{\cs_end: #1#1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_first_8:n}#1{\cs_end: #1#1#1#1#1#1#1#1}
\def_long_new:cpn {prg_replicate_first_9:n}#1{\cs_end: #1#1#1#1#1#1#1#1#1}
```

`\prg_stepwise_function:nnnN`
`g_stepwise_function_incr:nnnN`
`g_stepwise_function_decr:nnnN`

A stepwise function. Firstly we check the direction of the steps `#2` since that will depend on which test we should use. If the step is positive we use a greater than test, otherwise a less than test. If the test comes out true exit, otherwise perform `#4`, add the step to `#1` and try again with this new value of `#1`.

```
\def_long_new:NNn \prg_stepwise_function:nnnN 2{
  \num_compare:nNnTF{#2}<\c_zero
  {\exp_args:No\prg_stepwise_function_decr:nnnN }
  {\exp_args:No\prg_stepwise_function_incr:nnnN }
  {\int_use:N\int_eval:n{#1}}{#2}
}
\def_long_new:NNn \prg_stepwise_function_incr:nnnN 4{
  \num_compare:nNnF {#1}>{#3}
  {
    #4{#1}
    \exp_args:No \prg_stepwise_function_incr:nnnN
    {\int_use:N\int_eval:n{#1 + #2}}
    {#2}{#3}{#4}
  }
}
\def_long_new:NNn \prg_stepwise_function_decr:nnnN 4{
  \num_compare:nNnF {#1}<{#3}
  {
    #4{#1}
    \exp_args:No \prg_stepwise_function_decr:nnnN
```

```
    {\int_use:N\int_eval:n{#1 + #2}}
    {#2}{#3}{#4}
  }
}
```

\g_prg_inline_level_int
\prg_stepwise_inline:nnnn
prg_stepwise_inline_decr:nnnn
prg_stepwise_inline_incr:nnnn

This function uses the same approach as for instance `\clist_map_inline:Nn` to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed csname. Must make assignments global when we maintain our own stack.

```
\int_new:N\g_prg_inline_level_int
\def_long_new:NNn\prg_stepwise_inline:nnnn 4{
  \int_gincr:N \g_prg_inline_level_int
  \gdef:cpn{prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}##1{#4}
  \num_compare:nNnTF {#2}<\c_zero
  {\exp_args:Nco \prg_stepwise_inline_decr:Nnnn }
  {\exp_args:Nco \prg_stepwise_inline_incr:Nnnn }
  {prg_stepwise_inline_\int_use:N\g_prg_inline_level_int :n}
  {\int_use:N\int_eval:n{#1}} {#2} {#3}
  \int_gdecr:N \g_prg_inline_level_int
}
\def_long_new:NNn \prg_stepwise_inline_incr:Nnnn 4{
  \num_compare:nNnF {#2}>{#4}
  {
    #1{#2}
    \exp_args:NNo \prg_stepwise_inline_incr:Nnnn #1
    {\int_use:N\int_eval:n{#2 + #3}} {#3}{#4}
  }
}
\def_long_new:NNn \prg_stepwise_inline_decr:Nnnn 4{
  \num_compare:nNnF {#2}<{#4}
  {
    #1{#2}
    \exp_args:NNo \prg_stepwise_inline_decr:Nnnn #1
    {\int_use:N\int_eval:n{#2 + #3}} {#3}{#4}
  }
}
```

\prg_stepwise_variable:nnnNn
_stepwise_variable_decr:nnnNn
_stepwise_variable_incr:nnnNn

Almost the same as above. Just store the value in `#4` and execute `#5`.

```
\def_long_new:NNn \prg_stepwise_variable:nnnNn 2 {
  \num_compare:nNnTF {#2}<\c_zero
  {\exp_args:No\prg_stepwise_variable_decr:nnnNn}
  {\exp_args:No\prg_stepwise_variable_incr:nnnNn}
  {\int_use:N\int_eval:n{#1}}{#2}
}
\def_long_new:NNn \prg_stepwise_variable_incr:nnnNn 5 {
  \num_compare:nNnF {#1}>{#3}
  {
    \def:Npn #4{#1} #5
    \exp_args:No \prg_stepwise_variable_incr:nnnNn
    {\int_use:N\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
  }
}
```

```
\def_long_new:NNn \prg_stepwise_variable_decr:nnnNn 5 {
  \num_compare:nNnF {#1}<{#3}
  {
    \def:Npn #4{#1} #5
    \exp_args:No \prg_stepwise_variable_decr:nnnNn
    {\int_use:N\int_eval:n{#1 + #2}}{#2}{#3}#4{#5}
  }
}
```

23.5.3 Booleans

For normal booleans we set them to either `\c_true` or `\c_false` and then use `\if:w` to choose the right branch. The functions return either the TF, T, or F case *after* ending the `\if:w`. We only define the `N` versions here as the `c` versions can easily be constructed with the expansion module.

`\bool_new:N` `\bool_new:c` `\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`

Defining and setting a boolean is easy.

```
\def_new:Npn \bool_new:N #1   { \let_new:NN #1 \c_false }
\def_new:Npn \bool_new:c #1   { \let_new:cN {#1} \c_false }
\def_new:Npn \bool_set_true:N   #1 { \let:NN  #1 \c_true }
\def_new:Npn \bool_set_true:c   #1 { \let:cN  {#1} \c_true }
\def_new:Npn \bool_set_false:N  #1 { \let:NN  #1 \c_false }
\def_new:Npn \bool_set_false:c  #1 { \let:cN  {#1} \c_false }
\def_new:Npn \bool_gset_true:N   #1 { \glet:NN  #1 \c_true }
\def_new:Npn \bool_gset_true:c   #1 { \glet:cN  {#1} \c_true }
\def_new:Npn \bool_gset_false:N  #1 { \glet:NN  #1 \c_false }
\def_new:Npn \bool_gset_false:c  #1 { \glet:cN  {#1} \c_false }
```

`\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`

Setting a boolean to another is also pretty easy.

```
\let_new:NN \bool_set_eq:NN \let:NN
\let_new:NN \bool_set_eq:Nc \let:Nc
\let_new:NN \bool_set_eq:cN \let:cN
\let_new:NN \bool_set_eq:cc \let:cc
\let_new:NN \bool_gset_eq:NN \glet:NN
\let_new:NN \bool_gset_eq:Nc \glet:Nc
\let_new:NN \bool_gset_eq:cN \glet:cN
\let_new:NN \bool_gset_eq:cc \glet:cc
```

`\l_tmpa_bool` `\g_tmpa_bool`

A few booleans just if you need them.

```
\bool_new:N \l_tmpa_bool
\bool_new:N \g_tmpa_bool
```

`\bool_if:NTF` `\bool_if:NT` `\bool_if:NF` `\bool_if:cTF` `\bool_if:cT` `\bool_if:cF`

Straight forward here.

```
\def_test_function_new:npn{bool_if:N}#1{\if:w #1}
\def_new:Npn \bool_if:cTF{\exp_args:Nc\bool_if:NTF}
\def_new:Npn \bool_if:cT{\exp_args:Nc\bool_if:NT}
\def_new:Npn \bool_if:cF{\exp_args:Nc\bool_if:NF}
```

\bool_if_p:N
\bool_if_p:c

We also make a predicate function for the `bool` data type but since we use `\c_true` and `\c_false` it's rather simple... Not that there's anything wrong in simplicity – on the contrary!

```
\def_new:Npn \bool_if_p:N #1 { #1 }
\let_new:NN \bool_if_p:c \cs_use:c
```

\bool_whiledo:NT
\bool_whiledo:cT
\bool_whiledo:NF
\bool_whiledo:cF

A `while` loop where the boolean is tested before executing the statement. The `NT` version executes the `T` part as long as the boolean is true while the `NF` version executes the `F` part as long as the boolean is false.

```
\def_long_new:Npn \bool_whiledo:NT #1 #2 {
  \bool_if:NT #1 {#2 \bool_whiledo:NT #1 {#2}}
}
\def_new:Npn \bool_whiledo:cT{\exp_args:Nc\bool_whiledo:NT}
\def_long_new:Npn \bool_whiledo:NF #1 #2 {
  \bool_if:NF #1 {#2 \bool_whiledo:NF #1 {#2}}
}
\def_new:Npn \bool_whiledo:cF{\exp_args:Nc\bool_whiledo:NF}
```

\bool_dowhile:NT
\bool_dowhile:cT
\bool_dowhile:NF
\bool_dowhile:cF

A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```
\def_long_new:Npn \bool_dowhile:NT #1 #2 {
  #2 \bool_if:NT #1 {\bool_dowhile:NT #1 {#2}}
}
\def_new:Npn \bool_dowhile:cT{\exp_args:Nc\bool_dowhile:NT}
\def_long_new:Npn \bool_dowhile:NF #1 #2 {
  #2 \bool_if:NF #1 {\bool_dowhile:NF #1 {#2}}
}
\def_new:Npn \bool_dowhiledo:cF{\exp_args:Nc\bool_dowhile:cF}
```

\bool_double_if:NNnnnn
\bool_double_if:cNnnnn
\bool_double_if:Ncnnnn
\bool_double_if:ccnnnn

Execute `#3` iff TT, `#4` iff TF, `#5` iff FT and `#6` iff FF. The name isn't that great but I'll have to think about that. Ideally it should be something with `TF` since only one of the cases is executed but we haven't got any naming scheme for this kind of thing so for now I'll just stick to simple `nnnn`.

```
\def_new:Npn \bool_double_if:NNnnnn#1#2{
  \if_case:w \num_eval:w #1\scan_stop:
    \if_case:w \num_eval:w #2\scan_stop:
      \exp_after:NN\exp_after:NN\exp_after:NN \use_arg_i:nnnn
    \else:
      \exp_after:NN\exp_after:NN\exp_after:NN \use_arg_ii:nnnn
    \fi:
  \else:
    \if_case:w \num_eval:w #2\scan_stop:
      \exp_after:NN\exp_after:NN\exp_after:NN \use_arg_iii:nnnn
    \else:
      \exp_after:NN\exp_after:NN\exp_after:NN \use_arg_iv:nnnn
    \fi:
  \fi:
}
\def_new:Npn \bool_double_if:cNnnnn{\exp_args:Nc\bool_double_if:NNnnnn}
\def_new:Npn \bool_double_if:Ncnnnn{\exp_args:NNc\bool_double_if:NNnnnn}
\def_new:Npn \bool_double_if:ccnnnn{\exp_args:Ncc\bool_double_if:NNnnnn}
```

23.5.4 Generic testing

\prg_whiledo:nT
\prg_whiledo:nF
\prg_dowhile:nT
\prg_dowhile:nF

We provide these four generic while loops. `#1` is a test function and for the `T` functions it should be a test function ending with just the true case. Similar for the `F` types.

```
\def_long_new:Npn \prg_whiledo:nT #1#2{
  #1 {#2 \prg_whiledo:nT {#1}{#2}}
}
\def_long_new:Npn \prg_whiledo:nF #1#2{
  #1 {#2 \prg_whiledo:nF {#1}{#2}}
}
\def_long_new:Npn \prg_dowhile:nT #1#2{
  #2 #1 {\prg_dowhile:nT {#1}{#2}}
}
\def_long_new:Npn \prg_dowhile:nF #1#2{
  #2 #1 {\prg_dowhile:nF {#1}{#2}}
}
```

\predicate_p:n
\predicate:nTF
\predicate:nT
\predicate:nF
\predicate_auxi:NN
\predicate_auxii:NNN
\predicate_88_0:w
\predicate_88_1:w
\predicate_II_0:w
\predicate_II_1:w
\predicate_02_0:w
\predicate_02_1:w

Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages. The function evaluates predicates from left to right, expanding them to 00 and 01 resp., which leads to six different situations of tokens in the input stream:

`00&&` Current truth value is true, logical And seen, continue to see if next is also true.

`01&&` Current truth value is false, logical And seen, break the scanning and return ⟨*false*⟩.

`00||` Current truth value is true, logical Or seen, break the scanning and return ⟨*true*⟩.

`01||` Current truth value is false, logical Or seen, continue to see if a later predicate is true.

`0002` Current truth value is true, end marker seen, return ⟨*true*⟩.

`0102` Current truth value is false, end marker seen, return ⟨*false*⟩.

To accomplish this we pre-expand the predicate list using `f` type expansion which leads to `00` or `01`, possibly with a sequence of unfinished `\else: \c_false \fi:` or similar after it, which we remove using the same trick. We also carry over the truth value of the evaluated predicate. The expansion stops when it sees the end marker or `&&` or `||` (assuming these are not active characters at the programming level).

```
\def_long_new:Npn \predicate_p:n #1{
  \group_align_safe_begin:
  \exp_after:NN \predicate_auxi:NN
  \int_to_roman:w-`\q #1  02\scan_stop:
}
\def_long_test_function_new:npn {predicate:n}#1{
  \group_align_safe_begin:
  \if:w \exp_after:NN \predicate_auxi:NN
  \int_to_roman:w-`\q #1  02\scan_stop:
}
\def_new:Npn \predicate_auxi:NN 0 #1{
  \exp_after:NN \predicate_auxii:NNN \exp_after:NN #1
  \int_to_roman:w-`\q
}
```

After removing trailing conditionals we call a macro for the case we are in (see list above).

```
\def_new:Npn \predicate_auxii:NNN #1#2#3{
  \cs_use:c{predicate_#2#3_#1:w} }
\def_new:cpn{predicate_&&_0:w}{
  \exp_after:NN \predicate_auxi:NN\int_to_roman:w-`\q
}
\def_long_new:cpn{predicate_&&_1:w} #1 02\scan_stop:{
  \group_align_safe_end: 01}
\def_long_new:cpn{predicate_||_0:w} #1 02\scan_stop:{
  \group_align_safe_end: 00}
\def_new:cpn{predicate_||_1:w}{
  \exp_after:NN \predicate_auxi:NN\int_to_roman:w-`\q
}
\def_new:cpn{predicate_02_0:w}\scan_stop:{ \group_align_safe_end: 00 }
\def_new:cpn{predicate_02_1:w}\scan_stop:{ \group_align_safe_end: 01 }
```

`\predicate_not_p:n` The `not` variant just reverses the outcome of `\predicate_p:n`.

```
\def_long_new:Npn \predicate_not_p:n #1{
  \if:w \predicate_p:n{#1} \c_false \else: \c_true \fi:
}
```

23.5.5 Case switch

`\prg_case_int:nnn` `\prg_case_int_aux:nnn` This case switch is in reality quite simple. It takes three arguments:

1. An integer expression you wish to find.
2. A list of pairs of `{`⟨*integer expr*⟩`}{`⟨*code*⟩`}`. The list can be as long as is desired and ⟨*integer expr*⟩ can be negative.
3. The code to be executed if the value wasn't found.

We don't need the else case here yet, so leave it dangling in the input stream.

```
\def_long:Npn \prg_case_int:nnn #1 #2 {
```

We will be parsing on `#1` for each step so we might as well evaluate it first in case it is complicated.

```
  \exp_args:No \prg_case_int_aux:nnn {\num_value:w \int_eval:n{#1}} #2
```

The `?` below is just so there are enough arguments when we reach the end. And it made you look. `;-)`

```
  \q_recursion_tail ? \q_recursion_stop
}
\def_long_new:Npn \prg_case_int_aux:nnn #1#2#3{
```

If we reach the end, return the else case. We just remove braces.

```
  \quark_if_recursion_tail_stop_do:nn{#2}{\use_arg_i:n}
```

Otherwise we compare (which evaluates #2 for us)

```
\num_compare:nNnTF{#1}={#2}
```

If true, we want to remove the remainder of the list, the else case and then execute the code specified. Why not use #3\use_none:n? Because if we are doing f style expansion, we will get leftovers. If the test was false, we try the next pair, carrying the #1.

```
  { \use_arg_i_delimit_by_q_recursion_stop:nw {\use_arg_i:nn{#3}} }
  { \prg_case_int_aux:nnn {#1}}
}
```

\prg_case_dim:nnn
\prg_case_dim_aux:nnn

Same as \prg_case_dim:nnn except it is for ⟨*dim*⟩ registers.

```
\def_long:Npn \prg_case_dim:nnn #1 #2 {
  \exp_args:No \prg_case_dim_aux:nnn {\dim_use:N \dim_eval:n{#1}} #2
  \q_recursion_tail ? \q_recursion_stop
}
\def_long_new:Npn \prg_case_dim_aux:nnn #1#2#3{
  \quark_if_recursion_tail_stop_do:nn{#2}{\use_arg_i:n}
  \dim_compare:nNnTF{#1}={#2}
  { \use_arg_i_delimit_by_q_recursion_stop:nw {\use_arg_i:nn{#3}} }
  { \prg_case_dim_aux:nnn {#1}}
}
```

\prg_case_str:nnn
\prg_case_str_aux:nnn

Same as \prg_case_dim:nnn except it is for strings.

```
\def_long:Npn \prg_case_str:nnn #1 #2 {
  \prg_case_str_aux:nnn {#1} #2
  \q_recursion_tail ? \q_recursion_stop
}
\def_long_new:Npn \prg_case_str_aux:nnn #1#2#3{
  \quark_if_recursion_tail_stop_do:nn{#2}{\use_arg_i:n}
  \tlist_if_eq:xxTF{#1}{#2}
  { \use_arg_i_delimit_by_q_recursion_stop:nw {\use_arg_i:nn{#3}} }
  { \prg_case_str_aux:nnn {#1}}
}
```

23.5.6 Sorting

\prg_define_quicksort:nnn

#1 is the name, #2 and #3 are the tokens enclosing the argument. For the somewhat strange ⟨*clist*⟩ type which doesn't enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```
\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}
```

defines the user function \seq_quicksort:n and furthermore expects to use the two functions \seq_quicksort_compare:nnTF which compares the items and \seq_quicksort_function:n which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the seq type a sequence is a token list pointer, so one additionally has to define

```
\def:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}
```

For details on the implementation see "Sorting in TEX's Mouth" by Bernd Raichle. Firstly we define the function for parsing the ininital list and then the braced list afterwards.

```
\def_new:NNn \prg_define_quicksort:nnn 3 {
  \def_long:cNx{#1_quicksort:n}1{
    \exp_not:c{#1_quicksort_start_partition:w} ##1
    \exp_not:n{#2\q_nil#3\q_stop}
  }
  \def_long:cNx{#1_quicksort_braced:n}1{
    \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
    \exp_not:N\q_nil\exp_not:N\q_stop
  }
  \def_long:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
    \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
    \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{}{}
  }
  \def_long:cNx {#1_quicksort_start_partition_braced:n} 1 {
    \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{}{}
  }
```

Now for doing the partitions.

```
  \def_long:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
      \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
      \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
  \def_long:cNx {#1_quicksort_do_partition_i_braced:nnnn} 4 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
      \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
      \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
  \def_long:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
      \exp_not:c{#1_quicksort_partition_less_i:nnnn}
      \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
  \def_long:cNx {#1_quicksort_do_partition_ii_braced:nnnn} 4 {
    \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
```

```
    {
      \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
      \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
      \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
    }
    {##1}{##2}{##3}{##4}
  }
```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```
  \def_long:cNx {#1_quicksort_partition_less_i:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{{##4}##3}}
  \def_long:cNx {#1_quicksort_partition_less_ii:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3{##4}}}
  \def_long:cNx {#1_quicksort_partition_greater_i:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{{##4}##2}{##3}}
  \def_long:cNx {#1_quicksort_partition_greater_ii:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2{##4}}{##3}}
  \def_long:cNx {#1_quicksort_partition_less_i_braced:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{{##4}##3}}
  \def_long:cNx {#1_quicksort_partition_less_ii_braced:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3{##4}}}
  \def_long:cNx {#1_quicksort_partition_greater_i_braced:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{{##4}##2}{##3}}
  \def_long:cNx {#1_quicksort_partition_greater_ii_braced:nnnn} 4{
    \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2{##4}}{##3}}
```

Finally, the big kahuna! This is where the sub-lists are sorted.

```
  \def_long:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
    \exp_not:c{#1_quicksort_braced:n}{##2}
    \exp_not:c{#1_quicksort_function:n}{##1}
    \exp_not:c{#1_quicksort_braced:n}{##3}
  }
}
```

`\prg_quicksort:n` A simple version. Sorts a list of tokens, uses the function `\prg_quicksort_compare:nnTF` to compare items, and places the function `\prg_quicksort_function:n` in front of each of them.

```
\prg_define_quicksort:nnn {prg}{}{}
```

`\prg_quicksort_function:n`
`\prg_quicksort_compare:nnTF`

```
\let:NN \prg_quicksort_function:n \ERROR
\let:NN \prg_quicksort_compare:nnTF \ERROR
```

That's it (for now).

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

24 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 `tlist`s 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.

24.1 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.

`\char_set_mathcode:nn`
`\char_set_mathcode:w`
`\char_gset_mathcode:nn`
`\char_gset_mathcode:w`
`\char_value_mathcode:n`
`\char_value_mathcode:w`
`\char_show_value_mathcode:n`
`\char_show_value_mathcode:w`

`\char_set_mathcode:nn {`⟨*char*⟩`} {`⟨*number*⟩`}`
`\char_set_mathcode:w` ⟨*char*⟩ `=` ⟨*number*⟩
`\char_value_mathcode:n {`⟨*char*⟩`}`
`\char_show_value_mathcode:n {`⟨*char*⟩`}`

Set the math code for ⟨*char*⟩.

TEXhackers note: `\char_set_mathcode:w` is the TEX primitive `\mathcode` renamed.

24.2 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.

24.2.1 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.

24.3 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.

24.3.1 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.

24.4 Implementation

First a few required packages to get this going.

```
4487 ⟨package⟩\ProvidesExplPackage
4488 ⟨package⟩   {\filename}{\filedate}{\fileversion}{\filedescription}
4489 ⟨package⟩\RequirePackage{l3prg}
4490 ⟨package⟩\RequirePackage{l3int}
4491 ⟨∗initex | package⟩
```

24.4.1 Character tokens

```
\char_set_catcode:w
\char_set_catcode:nn
\char_value_catcode:w
\char_value_catcode:n
\char_show_value_catcode:w
\char_show_value_catcode:n
4492 \let_new:NN \char_set_catcode:w \tex_catcode:D
4493 \def_new:Npn \char_set_catcode:nn #1#2{
4494   \char_set_catcode:w #1 = \int_eval:n{#2}
4495 }
4496 \def_new:Npn \char_value_catcode:w {\int_use:N\tex_catcode:D}
4497 \def_new:Npn \char_value_catcode:n #1{\char_value_catcode:w \int_eval:n{#1}}
4498 \def_new:Npn \char_show_value_catcode:w {\tex_showthe:D\tex_catcode:D}
4499 \def_new:Npn \char_show_value_catcode:n #1{
4500   \char_show_value_catcode:w \int_eval:n{#1}}
```

\char_set_mathcode:w
\char_set_mathcode:nn
\char_gset_mathcode:w
\char_gset_mathcode:nn
\char_value_mathcode:w
\char_value_mathcode:n
\char_show_value_mathcode:w
\char_show_value_mathcode:n

Math codes.

```
\let_new:NN \char_set_mathcode:w \tex_mathcode:D
\def_new:Npn \char_set_mathcode:nn #1#2{
  \char_set_mathcode:w #1 = \int_eval:n{#2}
}
\def_protected_new:Npn \char_gset_mathcode:w {\pref_global:D\tex_mathcode:D}
\def_new:Npn \char_gset_mathcode:nn #1#2{
  \char_gset_mathcode:w #1 = \int_eval:n{#2}
}
\def_new:Npn \char_value_mathcode:w {\int_use:N\tex_mathcode:D}
\def_new:Npn \char_value_mathcode:n #1{\char_value_mathcode:w \int_eval:n{#1}}
\def_new:Npn \char_show_value_mathcode:w {\tex_showthe:D\tex_mathcode:D}
\def_new:Npn \char_show_value_mathcode:n #1{
  \char_show_value_mathcode:w \int_eval:n{#1}}
```

\char_set_lccode:w
\char_set_lccode:nn
\char_value_lccode:w
\char_value_lccode:n
\char_show_value_lccode:w
\char_show_value_lccode:n

```
\let_new:NN \char_set_lccode:w \tex_lccode:D
\def_new:Npn \char_set_lccode:nn #1#2{
  \char_set_lccode:w #1 = \int_eval:n{#2}
}
\def_new:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
\def_new:Npn \char_value_lccode:n #1{\char_value_lccode:w \int_eval:n{#1}}
\def_new:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}
\def_new:Npn \char_show_value_lccode:n #1{
  \char_show_value_lccode:w \int_eval:n{#1}}
```

\char_set_uccode:w
\char_set_uccode:nn
\char_value_uccode:w
\char_value_uccode:n
\char_show_value_uccode:w
\char_show_value_uccode:n

```
\let_new:NN \char_set_uccode:w \tex_uccode:D
\def_new:Npn \char_set_uccode:nn #1#2{
  \char_set_uccode:w #1 = \int_eval:n{#2}
}
\def_new:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}
\def_new:Npn \char_value_uccode:n #1{\char_value_uccode:w \int_eval:n{#1}}
\def_new:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
\def_new:Npn \char_show_value_uccode:n #1{
  \char_show_value_uccode:w \int_eval:n{#1}}
```

\char_set_sfcode:w
\char_set_sfcode:nn
\char_value_sfcode:w
\char_value_sfcode:n
\char_show_value_sfcode:w
\char_show_value_sfcode:n

```
\let_new:NN \char_set_sfcode:w \tex_sfcode:D
\def_new:Npn \char_set_sfcode:nn #1#2{
  \char_set_sfcode:w #1 = \int_eval:n{#2}
}
\def_new:Npn \char_value_sfcode:w {\int_use:N\tex_uccode:D}
\def_new:Npn \char_value_sfcode:n #1{\char_value_sfcode:w \int_eval:n{#1}}
\def_new:Npn \char_show_value_sfcode:w {\tex_showthe:D\tex_sfcode:D}
\def_new:Npn \char_show_value_sfcode:n #1{
  \char_show_value_sfcode:w \int_eval:n{#1}}
```

24.4.2 Generic tokens

`\token_new:Nn` Creates a new token.

```
\def_new:Npn \token_new:Nn #1#2{\glet_new:NN #1#2}
```

`\c_group_begin_token` `\c_group_end_token` `\c_math_shift_token` `\c_alignment_tab_token` `\c_parameter_token` `\c_math_superscript_token` `\c_math_subscript_token` `\c_space_token` `\c_letter_token` `\c_other_char_token` `\c_active_char_token` We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```
\let_new:NN \c_group_begin_token {
\let_new:NN \c_group_end_token }
\group_begin:
\char_set_catcode:nn{`\*}{3}
\token_new:Nn \c_math_shift_token {*}
\char_set_catcode:nn{`\*}{4}
\token_new:Nn \c_alignment_tab_token {*}
\token_new:Nn \c_parameter_token {#}
\token_new:Nn \c_math_superscript_token {^}
\char_set_catcode:nn{`\*}{8}
\token_new:Nn \c_math_subscript_token {*}
\token_new:Nn \c_space_token {~}
\token_new:Nn \c_letter_token {a}
\token_new:Nn \c_other_char_token {1}
\char_set_catcode:nn{`\*}{13}
\token_new:Nn \c_active_char_token {*}
\group_end:
```

`\token_if_group_begin_p:N` `\token_if_group_begin:NTF` `\token_if_group_begin:NT` `\token_if_group_begin:NF` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
\def_new:Npn \token_if_group_begin_p:N #1{
  \if_catcode:w \exp_not:N #1\c_group_begin_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_group_begin:N} #1{
  \if:w\token_if_group_begin_p:N #1}
```

`\token_if_group_end_p:N` `\token_if_group_end:NTF` `\token_if_group_end:NT` `\token_if_group_end:NF` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
\def_new:Npn \token_if_group_end_p:N #1{
  \if_catcode:w \exp_not:N #1\c_group_end_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_group_end:N} #1{
  \if:w\token_if_group_end_p:N #1}
```

\token_if_math_shift_p:N
\token_if_math_shift:NTF
\token_if_math_shift:NT
\token_if_math_shift:NF

Check if token is a math shift token. We use the constant \c_math_shift_token for this.

```
\def_new:Npn \token_if_math_shift_p:N #1{
  \if_catcode:w \exp_not:N #1\c_math_shift_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_math_shift:N} #1{
  \if:w\token_if_math_shift_p:N#1}
```

\token_if_alignment_tab_p:N
\token_if_alignment_tab:NTF
\token_if_alignment_tab:NT
\token_if_alignment_tab:NF

Check if token is an alignment tab token. We use the constant \c_alignment_tab_token for this.

```
\def_new:Npn \token_if_alignment_tab_p:N #1{
  \if_catcode:w \exp_not:N #1\c_alignment_tab_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_alignment_tab:N} #1{
  \if:w\token_if_alignment_tab_p:N#1}
```

\token_if_parameter_p:N
\token_if_parameter:NTF
\token_if_parameter:NT
\token_if_parameter:NF

Check if token is a parameter token. We use the constant \c_parameter_token for this. We have to trick TEX a bit to avoid an error message.

```
\def_new:Npn \token_if_parameter_p:N #1{
  \exp_after:NN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_parameter:N} #1{
  \if:w\token_if_parameter_p:N#1}
```

token_if_math_superscript_p:N
token_if_math_superscript:NTF
\token_if_math_superscript:NT
\token_if_math_superscript:NF

Check if token is a math superscript token. We use the constant \c_math_superscript_token for this.

```
\def_new:Npn \token_if_math_superscript_p:N #1{
  \if_catcode:w \exp_not:N #1\c_math_superscript_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_math_superscript:N} #1{
  \if:w\token_if_math_superscript_p:N #1}
```

\token_if_math_subscript_p:N
\token_if_math_subscript:NTF
\token_if_math_subscript:NT
\token_if_math_subscript:NF

Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.

```
\def_new:Npn \token_if_math_subscript_p:N #1{
  \if_catcode:w \exp_not:N #1\c_math_subscript_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_math_subscript:N} #1{
  \if:w\token_if_math_subscript_p:N #1}
```

`\token_if_space_p:N`
`\token_if_space:NTF`
`\token_if_space:NT`
`\token_if_space:NF`

Check if token is a space token. We use the constant `\c_space_token` for this.

```
\def_new:Npn \token_if_space_p:N #1{
  \if_catcode:w \exp_not:N #1\c_space_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_space:N} #1{
  \if:w\token_if_space_p:N #1}
```

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

Check if token is a letter token. We use the constant `\c_letter_token` for this.

```
\def_new:Npn \token_if_letter_p:N #1{
  \if_catcode:w \exp_not:N #1\c_letter_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_letter:N} #1{
  \if:w\token_if_letter_p:N #1}
```

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

Check if token is an other char token. We use the constant `\c_other_char_token` for this.

```
\def_new:Npn \token_if_other_char_p:N #1{
  \if_catcode:w \exp_not:N #1\c_other_char_token
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_other_char:N} #1{
  \if:w\token_if_other_char_p:N #1}
```

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

Check if token is an active char token. We use the constant `\c_active_char_token` for this.

```
\def_new:Npn \token_if_active_char_p:N #1{
  \if_catcode:w \exp_not:N #1\c_active_char_token
    \c_true
```

```
    \else:
      \c_false
    \fi:
}
\def_test_function_new:npn {token_if_active_char:N} #1{
  \if:w\token_if_active_char_p:N #1}
```

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

Check if the tokens `#1` and `#2` have same meaning.

```
\def_new:Npn \token_if_eq_meaning_p:NN #1#2 {
  \if_meaning:NN  #1  #2
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_eq_meaning:NN}#1#2{
  \if_meaning:NN  #1  #2}
```

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

Check if the tokens `#1` and `#2` have same category code.

```
\def_new:Npn \token_if_eq_catcode_p:NN #1#2 {
  \if_catcode:w \exp_not:N #1 \exp_not:N #2
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_eq_catcode:NN}#1#2{
  \if:w\token_if_eq_catcode_p:NN#1#2}
```

`\token_if_eq_charcode_p:NN` `\token_if_eq_charcode:NNTF` `\token_if_eq_charcode:NNT` `\token_if_eq_charcode:NNF`

Check if the tokens `#1` and `#2` have same character code.

```
\def_new:Npn \token_if_charcode_eq_p:NN #1#2 {
  \if_charcode:w \exp_not:N #1 \exp_not:N #2
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_eq_charcode:NN}#1#2{
  \if:w\token_if_eq_charcode_p:NN#1#2}
```

`\token_if_macro_p:N` `\token_if_macro_p_aux:w` `\token_if_macro:NTF` `\token_if_macro:NT` `\token_if_macro:NF`

When a token is a macro, `\token_to_meaning:N` will always output something like `\long macro:#1->#1` so we simply check to see if the meaning contains `->`. Argument `#2` in the code below will be empty if the string `->` isn't present, proof that the token was not a macro (which is why we reverse the emptiness test). However this function will fail on its own auxiliary function (and a few other private functions as well) but that should certainly never be a problem!

```
\def_new:Npn \token_if_macro_p:N #1 {
  \exp_after:NN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_nil
}
```

```
\def_new:Npn \token_if_macro_p_aux:w #1 -> #2 \q_nil{
  \if:w \tlist_if_empty_p:n{#2} \c_false \else: \c_true \fi:
}
\def_test_function_new:npn {token_if_macro:N} #1{\if:w\token_if_macro_p:N#1}
```

`\token_if_cs_p:N`
`\token_if_cs:NTF`
`\token_if_cs:NT`
`\token_if_cs:NF`

Check if token has same catcode as a control sequence. We use `\scan_stop:` for this.

```
\def_new:Npn \token_if_cs_p:N {\token_if_eq_catcode_p:NN \scan_stop:}
\def_test_function_new:npn {token_if_cs:N} #1{
  \if:w \token_if_eq_catcode_p:NN \scan_stop: #1}
```

`\token_if_expandable_p:N`
`\token_if_expandable:NTF`
`\token_if_expandable:NT`
`\token_if_expandable:NF`

Check if token is expandable. We use the fact that TEX will temporarily convert `\exp_not:N` ⟨*token*⟩ into `\scan_stop:` if ⟨*token*⟩ is expandable.

```
\def_new:Npn \token_if_expandable_p:N #1{
  \exp_after:NN \if_token_eq:NN \exp_not:N #1 \scan_stop:
    \c_true
  \else:
    \c_false
  \fi:
}
\def_test_function_new:npn {token_if_expandable:N} #1{
  \if:w\token_if_expandable_p:N#1}
```

`\token_if_chardef_p:N`
`\token_if_chardef_p_aux:w`
`\token_if_mathchardef_p:N`
`\token_if_mathchardef_p_aux:w`
`\token_if_int_register_p:N`
`\token_if_int_register_p_aux:w`
`\token_if_skip_register_p:N`
`\token_if_skip_register_p_aux:w`
`\token_if_dim_register_p:N`
`\token_if_dim_register_p_aux:w`
`\token_if_toks_register_p:N`
`\token_if_toks_register_p_aux:w`
`\token_if_protected_macro_p:N`
`\token_if_protected_macro_p_aux:w`
`\token_if_long_macro_p:N`
`\token_if_long_macro_p_aux:w`
`\token_if_protected_long_macro_p:N`
`\token_if_protected_long_macro_p_aux:w`

Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below...

```
\group_begin:
  \char_set_lccode:nn {`\X}{`\n}
  \char_set_lccode:nn {`\Y}{`\t}
  \char_set_lccode:nn {`\Z}{`\d}
  \char_set_lccode:nn {`\?}{`\\}
  \tlist_map_inline:nn{\X\Y\Z\M\T\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
    {\char_set_catcode:nn {`#1}{12}}
```

We convert the token list to lowercase and restore the catcode and lowercase code changes.

```
\tlist_to_lowercase:n{
  \group_end:
```

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since TEX thinks of such tokens as hexadecimal so it stores them as `\char"`⟨*hex number*⟩ or `\mathchar"`⟨*hex number*⟩.

```
\def_new:Npn \token_if_chardef_p:N #1 {
  \exp_after:NN \token_if_chardef_p_aux:w
  \token_to_meaning:N #1?CHAR"\q_nil
}
\def_new:Npn \token_if_chardef_p_aux:w #1?CHAR"#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
\def_new:Npn \token_if_mathchardef_p:N #1 {
```

```
  \exp_after:NN \token_if_mathchardef_p_aux:w
  \token_to_meaning:N #1?MAYHCHAR"\q_nil
}
\def_new:Npn \token_if_mathchardef_p_aux:w #1?MAYHCHAR"#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Integer registers are a little more difficult since they expand to `\count`⟨*number*⟩ and there is also a primitive `\countdef`. So we have to check for that primitive as well.

```
\def:Npn \token_if_int_register_p:N #1{
  \if_meaning:NN \tex_countdef:D #1
    \c_false
  \else:
    \exp_after:NN \token_if_int_register_p_aux:w
      \token_to_meaning:N #1?COUXY\q_nil
  \fi:
}
\def_new:Npn \token_if_int_register_p_aux:w #1?COUXY#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Skip registers are done the same way as the integer registers.

```
\def:Npn \token_if_skip_register_p:N #1{
  \if_meaning:NN \tex_skipdef:D #1
  \c_false
  \else:
  \exp_after:NN \token_if_skip_register_p_aux:w
  \token_to_meaning:N #1?SKIP\q_nil
  \fi:
}
\def_new:Npn \token_if_skip_register_p_aux:w #1?SKIP#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Dim registers. No news here

```
\def:Npn \token_if_dim_register_p:N #1{
  \if_meaning:NN \tex_dimendef:D #1
    \c_false
  \else:
    \exp_after:NN \token_if_dim_register_p_aux:w
    \token_to_meaning:N #1?ZIMEX\q_nil
  \fi:
}
\def_new:Npn \token_if_dim_register_p_aux:w #1?ZIMEX#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Toks registers. Ho-hum.

```
\def:Npn \token_if_toks_register_p:N #1{
  \if_meaning:NN \tex_toksdef:D #1
    \c_false
```

```
  \else:
    \exp_after:NN \token_if_toks_register_p_aux:w
    \token_to_meaning:N #1?YOKS\q_nil
  \fi:
}
\def_new:Npn \token_if_toks_register_p_aux:w #1?YOKS#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Protected macros.

```
\def_new:Npn \token_if_protected_macro_p:N #1 {
  \exp_after:NN \token_if_protected_macro_p_aux:w
  \token_to_meaning:N #1?PROYECYEZ~MACRO\q_nil
}
\def_new:Npn \token_if_protected_macro_p_aux:w #1?PROYECYEZ~MACRO#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Long macros.

```
\def_new:Npn \token_if_long_macro_p:N #1 {
  \exp_after:NN \token_if_long_macro_p_aux:w
  \token_to_meaning:N #1?LOXG~MACRO\q_nil
}
\def_new:Npn \token_if_long_macro_p_aux:w #1?LOXG~MACRO#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Finally protected long macros where we for once don't have to add an extra test since there is no primitive for the combined prefixes.

```
\def_new:Npn \token_if_protected_long_macro_p:N #1 {
  \exp_after:NN \token_if_protected_long_macro_p_aux:w
  \token_to_meaning:N #1?PROYECYEZ~?LOXG~MACRO\q_nil
}
\def_new:Npn \token_if_protected_long_macro_p_aux:w #1
  ?PROYECYEZ~?LOXG~MACRO#2\q_nil{
  \tlist_if_empty_p:n{#1}
}
```

Finally the `\tlist_to_lowercase:n` ends!

```
}
```

`\token_if_chardef:NTF`
`\token_if_chardef:NT`
`\token_if_chardef:NF`

```
\def_test_function_new:npn {token_if_chardef:N} {\if:w \token_if_chardef_p:N}
```

`\token_if_mathchardef:NTF`
`\token_if_mathchardef:NT`
`\token_if_mathchardef:NF`

```
\def_test_function_new:npn {token_if_mathchardef:N} {
  \if:w \token_if_mathchardef_p:N}
```

```
\token_if_long_macro:NTF
\token_if_long_macro:NT
\token_if_long_macro:NF
```
```
4797 \def_test_function_new:npn {token_if_long_macro:N} {
4798   \if:w \token_if_long_macro_p:N}
```

```
\token_if_protected_macro:NTF
\token_if_protected_macro:NT
\token_if_protected_macro:NF
```
```
4799 \def_test_function_new:npn {token_if_protected_macro:N} {
4800   \if:w \token_if_protected_macro_p:N}
```

```
n_if_protected_long_macro:NTF
en_if_protected_long_macro:NT
en_if_protected_long_macro:NF
```
```
4801 \def_test_function_new:npn {token_if_protected_long_macro:N} {
4802   \if:w \token_if_protected_long_macro_p:N}
```

```
\token_if_dim_register:NTF
\token_if_dim_register:NT
\token_if_dim_register:NF
```
```
4803 \def_test_function_new:npn {token_if_dim_register:N} {
4804   \if:w \token_if_dim_register_p:N}
```

```
\token_if_skip_register:NTF
\token_if_skip_register:NT
\token_if_skip_register:NF
```
```
4805 \def_test_function_new:npn {token_if_skip_register:N} {
4806   \if:w \token_if_skip_register_p:N}
```

```
\token_if_int_register:NTF
\token_if_int_register:NT
\token_if_int_register:NF
```
```
4807 \def_test_function_new:npn {token_if_int_register:N} {
4808   \if:w \token_if_int_register_p:N}
```

```
\token_if_toks_register:NTF
\token_if_toks_register:NT
\token_if_toks_register:NF
```
```
4809 \def_test_function_new:npn {token_if_toks_register:N} {
4810   \if:w \token_if_toks_register_p:N}
```

We do not provide a function for testing if a control sequence is "outer" since we don't use that in LaTeX3.

```
_prefix_arg_replacement_aux:w
    \token_get_prefix_spec:N
       \token_get_arg_spec:N
\token_get_replacement_spec:N
```

In the xparse package we sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```
4811 \group_begin:
4812 \char_set_lccode:nn {`\?}{`\:}
4813 \char_set_catcode:nn{`\M}{12}
4814 \char_set_catcode:nn{`\A}{12}
4815 \char_set_catcode:nn{`\C}{12}
4816 \char_set_catcode:nn{`\R}{12}
4817 \char_set_catcode:nn{`\O}{12}
```

```
\tlist_to_lowercase:n{
  \group_end:
  \def_new:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_nil#4{
    #4{#1}{#2}{#3}
  }
  \def_new:Npn\token_get_prefix_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:NN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_arg_i:nnn
    }{\scan_stop:}
  }
  \def_new:Npn\token_get_arg_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:NN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_arg_ii:nnn
    }{\scan_stop:}
  }
  \def_new:Npn\token_get_replacement_spec:N #1{
    \token_if_macro:NTF #1{
      \exp_after:NN \token_get_prefix_arg_replacement_aux:w
      \token_to_meaning:N #1\q_nil\use_arg_iii:nnn
    }{\scan_stop:}
  }
}
```

Useless code: because we can!

`\token_if_primitive_p:N`
`\token_if_primitive_p_aux:N`
`\token_if_primitive:NTF`
`\token_if_primitive:NT`
`\token_if_primitive:NF`

It is rather hard to determine if a token is a primitive. First we can check if it is a control sequence or active character. If either, we check if it is a macro. Then we can go through a tedious process of testing for different register types... I don't actually think this function is useful but you never know.

```
\def_new:Npn \token_if_primitive_p:N #1{
  \if:w \token_if_cs_p:N #1\scan_stop:
    \if:w \token_if_macro_p:N #1
      \c_false
    \else:
      \token_if_primitive_p_aux:N #1
    \fi:
  \else:
    \if:w \token_if_active_p:N #1
      \if:w \token_if_macro_p:N #1
        \c_false
      \else:
        \token_if_primitive_p_aux:N #1
      \fi:
    \else:
      \c_false
    \fi:
  \fi:
}
\def_new:Npn \token_if_primitive_p_aux:N #1{
  \if:w \token_if_chardef_p:N #1 \c_false
```

```
  \else:
    \if:w \token_if_mathchardef_p:N #1 \c_false
    \else:
      \if:w \token_if_int_register_p:N #1 \c_false
      \else:
        \if:w \token_if_skip_register_p:N #1 \c_false
        \else:
          \if:w \token_if_dim_register_p:N #1 \c_false
          \else:
            \if:w   \token_if_toks_register_p:N #1 \c_false
            \else:
```

We made it!

```
              \c_true
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  \fi:
}
\def_test_function_new:npn {token_if_primitive:N} #1{
  \if:w\token_if_primitive_p:N#1}
```

24.4.3 Peeking ahead at the next token

`\l_peek_token` `\g_peek_token` `\l_peek_search_token` We define some other tokens which will initially be the character `?`.

```
\token_new:Nn \l_peek_token {?}
\token_new:Nn \g_peek_token {?}
\token_new:Nn \l_peek_search_token {?}
```

`\peek_after:NN` `\peek_gafter:NN` `\peek_after:NN` takes two argument where the first is a function acting on `\l_peek_token` and the second is the next token in the input stream which `\l_peek_token` is set equal to. `\peek_gafter:NN` does the same globally to `\g_peek_token`.

```
\def_new:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
\def_new:Npn \peek_gafter:NN {
  \pref_global:D \tex_futurelet:D \g_peek_token
}
```

For normal purposes there are four main cases:

1. peek at the next token.
2. peek at the next non-space token.
3. peek at the next token and remove it.
4. peek at the next non-space token and remove it.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the ⟨*true*⟩ and ⟨*false*⟩ cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a ⟨*toks*⟩ register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tlp` `\l_peek_false_tlp` Two dedicated token list pointers that store the true and false cases.

```
\tlp_new:Nn \l_peek_true_tlp {}
\tlp_new:Nn \l_peek_false_tlp {}
```

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```
\def_new:Npn \peek_tmp:w{}
```

`\l_peek_search_tlp` We also use this token list pointer for storing the token we want to compare. This turns out to be useful.

```
\tlp_new:Nn \l_peek_search_tlp{}
```

`\peek_token_generic:NNTF` `#1` is the function to execute (obey or ignore spaces, etc.), `#2` is the special token we're looking for, and `#3` and `#4` are the ⟨*true*⟩ and ⟨*false*⟩ branches.

```
\def_long_new:Npn \peek_token_generic:NNTF #1#2#3#4{
  \let:NN \l_peek_search_token #2
  \tlp_set:Nn \l_peek_search_tlp {#2}
  \tlp_set:Nx \l_peek_true_tlp {\exp_not:n{\group_align_safe_end: #3}}
  \tlp_set:Nx \l_peek_false_tlp {\exp_not:n{\group_align_safe_end: #4}}
  \group_align_safe_begin:
    \peek_after:NN #1
}
```

`eek_token_remove_generic:NNTF` If we want to be able to remove any character from the input stream we might as well do it the same way for all characters so we define this as little differently from above.

```
\def_long_new:Npn \peek_token_remove_generic:NNTF #1#2#3#4{
  \let:NN \l_peek_search_token #2
  \tlp_set:Nn \l_peek_search_tlp {#2}
  \tlp_set:Nx \l_peek_true_aux_tlp { \exp_not:n{ #3 } }
  \tlp_set_eq:NN \l_peek_true_tlp \c_peek_true_remove_next_tlp
  \tlp_set:Nx \l_peek_false_tlp {\exp_not:n{\group_align_safe_end: #4}}
  \group_align_safe_begin:
    \peek_after:NN #1
}
```

\l_peek_true_aux_tlp
\l_peek_true_remove_next_tlp

Two token list pointers to help with removing the character from the input stream.

```
\tlp_new:Nn \l_peek_true_aux_tlp {}
\tlp_new:Nn \c_peek_true_remove_next_tlp {\group_align_safe_end:
  \tex_afterassignment:D \l_peek_true_aux_tlp \let:NN \peek_tmp:w
}
```

\peek_execute_braches_meaning:
\peek_execute_braches_catcode:
\peek_execute_braches_charcode:
\peek_execute_branches_charcode_aux:NN

There are three major tests between tokens in TEX: meaning, catcode and charcode. Hence we define three basic test functions that set in after the ignoring phase is over and done with.

```
\def_new:Npn \peek_execute_branches_meaning: {
  \if_meaning:NN \l_peek_token \l_peek_search_token
    \exp_after:NN \l_peek_true_tlp
  \else:
    \exp_after:NN \l_peek_false_tlp
  \fi:
}
\def_new:Npn \peek_execute_branches_catcode: {
  \if_catcode:w \exp_not:N\l_peek_token \exp_not:N\l_peek_search_token
    \exp_after:NN \l_peek_true_tlp
  \else:
    \exp_after:NN \l_peek_false_tlp
  \fi:
}
```

For the charcode version we do things a little differently. We want to check the token directly but if we do this we face problems if the next thing in the input stream is a braced group or a space token. The braced group would be read as a complete argument and the space would be gobbled by TEX's argument reading routines. Hence we test for both of these and if one of them is found we just execute the false result directly since no one should ever try to use the charcode function for searching for \c_group_begin_token or \c_space_token.

```
\def_new:Npn \peek_execute_branches_charcode: {
  \predicate:nTF {
    \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
    \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
  }
  { \l_peek_false_tlp  }
```

Otherwise we call a small auxiliary function that just grabs the next token. We can do that because it really is a single token; we just have insert it again afterwards. Also we stored the token we were looking for in the token list pointer \l_peek_search_tlp so we unpack it again for this function.

```
  { \exp_after:NN \peek_execute_branches_charcode_aux:NN \l_peek_search_tlp }
}
```

Then we just do the usual \if_charcode:w comparison. We also remember to insert #2 again after executing the true or false branches.

```
\def_long_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
  \if_charcode:w \exp_not:N #1\exp_not:N#2
```

```
    \exp_after:NN \l_peek_true_tlp
  \else:
    \exp_after:NN \l_peek_false_tlp
  \fi:
  #2
}
```

\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF

Here we use meaning comparison with \if_meaning:NN.

```
\def_new:Npn \peek_meaning:NTF {
  \peek_token_generic:NNTF \peek_execute_branches_meaning:
}
\def_new:Npn \peek_meaning_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_meaning:
  \peek_token_generic:NNTF \peek_ignore_spaces_execute_branches:
}
\def_new:Npn \peek_meaning_remove:NTF {
  \peek_token_remove_generic:NNTF \peek_execute_branches_meaning:
}
\def_new:Npn \peek_meaning_remove_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_meaning:
  \peek_token_remove_generic:NNTF \peek_ignore_spaces_execute_branches:
}
```

\peek_catcode:NTF
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF

Here we use catcode comparison with \if_catcode:w.

```
\def_new:Npn \peek_catcode:NTF {
  \peek_token_generic:NNTF \peek_execute_branches_catcode:
}
\def_new:Npn \peek_catcode_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_catcode:
  \peek_token_generic:NNTF \peek_ignore_spaces_execute_branches:
}
\def_new:Npn \peek_catcode_remove:NTF {
  \peek_token_remove_generic:NNTF \peek_execute_branches_catcode:
}
\def_new:Npn \peek_catcode_remove_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_catcode:
  \peek_token_remove_generic:NNTF \peek_ignore_spaces_execute_branches:
}
```

\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

Here we use charcode comparison with \if_charcode:w.

```
\def_new:Npn \peek_charcode:NTF {
  \peek_token_generic:NNTF \peek_execute_branches_charcode:
}
\def_new:Npn \peek_charcode_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_charcode:
  \peek_token_generic:NNTF \peek_ignore_spaces_execute_branches:
}
\def_new:Npn \peek_charcode_remove:NTF {
  \peek_token_remove_generic:NNTF \peek_execute_branches_charcode:
}

```

```
\def_new:Npn \peek_charcode_remove_ignore_spaces:NTF {
  \let:NN \peek_execute_branches: \peek_execute_branches_charcode:
  \peek_token_remove_generic:NNTF \peek_ignore_spaces_execute_branches:
}
```

`\peek_ignore_spaces_aux:`
`\peek_ignore_spaces_execute_branches:`

Throw away a space token and search again. We could define this in a more devious way where the auxiliary function gobbles the space token but then what do we do if we decide that a certain function should ignore more than one specific token? For example someone might find it interesting to define a `\peek_` function that ignores `a`'s and `b`'s! Or maybe different kinds of "funny spaces"... Therefore I have decided to use this version which uses `\tex_afterassignment:D` to call the auxiliary function after the next token has been removed by `\let:NN`. That way it is easily extensible.

```
\def_new:Npn \peek_ignore_spaces_aux: {
  \peek_after:NN \peek_ignore_spaces_execute_branches:
}
\def_new:Npn \peek_ignore_spaces_execute_branches: {
  \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
  {  \tex_afterassignment:D \peek_ignore_spaces_aux:
     \let:NN \peek_tmp:w
  }
  \peek_execute_branches:
}
```

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

25 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.

25.1 Implementation

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{l3quark}
\RequirePackage{l3toks}
\RequirePackage{l3io}
\RequirePackage{l3prop}
\RequirePackage{l3int}
\RequirePackage{l3token}
⟨/package⟩
⟨*initex | package⟩
```

There are two kinds of information, namely information which is *immediate* like a section title and then there's *deferred* information like page numbers. Each reference type belong to one of these categories, which we save internally as the property lists `\g_xref_all_curr_immediate_fields_plist` and `\g_xref_all_curr_deferred_fields_plist` and the reference type ⟨*xyz*⟩ exists as the key-info pair `\xref_`⟨*xyz*⟩`_key {\l_xref_curr_`⟨*xyz*⟩`_tlp}` on one of these lists. This way each new entry type is just added as another key-info pair.

When the cross references are generated at the beginning of the document each will turn into a control sequence. Thus `\label{mylab}` will internally refer to the property list `\g_xref_mylab_plist`.

The extraction of values from this property list can be done in several different ways but we want to keep the operation expandable. Therefore we use a dedicated function for each type of cross reference, which looks like this:

```
\xref_get_value_xyz_aux:w -> #1 \xref_xyz_key #2#3\q_nil{#2}
```

This will throw away all the bits we don't need. In case `xyz` is the first on the `mylab` property list `#1` is empty, if it's the last key-info pair `#3` is empty. The value of the field can be extracted with the function `\xref_get_value:nn` where the first argument is the type and the second the label name so here it would be `\xref_get_value:nn {xyz} {mylab}`.

`\g_xref_all_curr_immediate_fields_plist`
`\g_xref_all_curr_deferred_fields_plist`

The two main property lists for storing information. They contain key-info pairs for all known types.

```
\prop_new:N \g_xref_all_curr_immediate_fields_plist
\prop_new:N \g_xref_all_curr_deferred_fields_plist
```

`\xref_new:nn`
`\xref_deferred_new:nn`
`\xref_new_aux:nnn`

Setting up a new cross reference type is fairly straight forward when we follow the game plan mentioned earlier.

```
\def_new:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
\def_new:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
\def_new:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
\prop_gput:ccx {g_xref_all_curr_ #1 _fields_plist}
{ xref_ #2 _key }
{ \exp_not:c {l_xref_curr_#2_tlp }}
```

Then define the key to be a protected macro.[14]

```
\def_protected:cpn { xref_#2_key }{}
\tlp_new:cn{l_xref_curr_#2_tlp}{#3}
```

Now for the function extracting the value of a reference. We could do this with a simple `\prop_if_in` thing put since we want to do things in an expandable way we make a separate grabber for each type—this is also faster. The grabber function can be defined by using an intricate construction of `\exp_after:NN` and other goodies but I prefer readable code. The end result for the input `xyz` is

```
\def:Npn\xref_get_value_xyz_aux:w #1\xref_xyz_key #2#3\q_nil{#2}
```

```
\toks_set:Nx \l_tmpa_toks {
  \exp_not:n { \def:cpn {xref_get_value_#2_aux:w} ##1 }
  \exp_not:c { xref_#2_key }
}
\toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
}
```

`\xref_get_value:nn` Getting the correct value for a given label-type pair is a matter of connecting the correct grabber functions and property list.

```
\def_new:Npn \xref_get_value:nn #1#2 {
  \cs_if_really_free:cTF{g_xref_#2_plist}
  {??}
  {
```

This next expansion may look a little weird but it isn't if you think about it!

```
    \exp_args:NcNc \exp_after:NN {xref_get_value_#1_aux:w}
    \prop_use:N {g_xref_#2_plist}
```

Better put in the stop marker.

```
    \q_nil
  }
}
\def:NNn \exp_after:cc 2 {
  \exp_after:NN \exp_after:NN
  \cs:w #1\exp_after:NN\cs_end: \cs:w #2\cs_end:
}
```

`\xref_define_label:nn` `\xref_define_label_aux:nn` Define the property list for each label. We better do this in two steps because the special catcode regime is in effect and since some of the info fields are very likely to contain actual text, we better make sure spaces aren't ignored! As for the meaning of other characters

[14] We could also set it equal to `\scan_stop:` but this just feels "cleaner".

then it is a possibility to also have a field containing catcode instructions which can then be activated with `\etex_scantokens:D`.

```
\def_protected_new:Npn \xref_define_label:nn {
  \group_begin:
    \char_set_catcode:nn {`\ }\c_ten
    \xref_define_label_aux:nn
}
```

If the label is already taken we have a multiply defined label and we should do something about it. For now we don't do anything spectacular.

```
\def_new:Npn \xref_define_label_aux:nn #1#2 {
    \cs_if_really_free:cTF{g_xref_#1_plist}
    {\prop_new:c{g_xref_#1_plist}}{\WARNING}
    \toks_gset:cn{g_xref_#1_plist}{#2}
  \group_end:
}
```

`\xref_set_label:n` Then the generic command for setting a label. We expand the immediate labels fully before calling the write function but make sure the deferred fields aren't expanded just yet. Due to property lists being implemented as token list registers we must expand the 'immediate' fields twice.

```
\def:Npn \xref_set_label:n #1{
  \def:Npx \tmp:w{\prop_use:N\g_xref_all_curr_immediate_fields_plist}
  \exp_args:NNx\iow_deferred_expanded:Nn \xref_write{
    \xref_define_label:nn {#1} {
      \tmp:w
      \prop_use:N \g_xref_all_curr_deferred_fields_plist
    }
  }
}
```

`\xref_write` A stream for writing cross references although they do not require to be in a separate file.

```
\iow_new:N \xref_write
```

That's it (for now).

```
⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩

⟨*testfile⟩
\documentclass{article}
\usepackage{l3xref}
\ExplSyntaxOn
\def:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
\def:Npn \DefineCrossReferences {
  \group_begin:
    \NamesStart
```

```
    \InputIfFileExists{\jobname.xref}{}{}
  \group_end:
}
\AtBeginDocument{\DefineCrossReferences\startrecording}

\xref_new:nn {name}{}
\def:Npn \setname{\tlp_set:Nn\l_xref_curr_name_tlp}
\def:Npn \getname{\xref_get_value:nn{name}}

\xref_deferred_new:nn {page}{\thepage}
\def:Npn \getpage{\xref_get_value:nn{page}}

\xref_deferred_new:nn {valuepage}{\number\value{page}}
\def:Npn \getvaluepage{\xref_get_value:nn{valuepage}}

\let:NN \setlabel \xref_set_label:n

\ExplSyntaxOff
\begin{document}
\pagenumbering{roman}

Text\setname{This is a name}\setlabel{testlabel1}.  More
text\setname{This is another name}\setlabel{testlabel2}. \clearpage

Text\setname{This is a third name}\setlabel{testlabel3}. More
text\setname{Hello World!}\setlabel{testlabel4}. \clearpage

\pagenumbering{arabic}

Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
  6}\setlabel{testlabel6}. \clearpage

Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
  8}\setlabel{testlabel8}. \clearpage

Now let's extract some values. \getname{testlabel1} on page
\getpage{testlabel1} with value \getvaluepage{testlabel1}.

Now let's extract some values. \getname{testlabel 7} on page
\getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
\end{document}
```
⟨/testfile⟩

26 Infix notation arithmetic

This is pretty much a straight adaption of the calc package and as such has same syntax for the ⟨*calc expression*⟩. However, there are some noticeable differences.

- The calc expression is expanded fully, which means there are no problems with unfinished conditionals. However, the contents of `\widthof` etc. is not expanded at all. This includes uses in traditional LaTeX as in the array package, which tries

to do an `\edef` several times. The code used in l3calc provides self-protection for these cases.

- Muskip registers are supported although they can only be used in `\ratio` if already evaluating a muskip expression. For the other three register types, you can use points.
- All results are rounded, not truncated. More precisely, the primitive TEX operations `\divide` and `\multiply` are not used. The only instance where one will observe an effect is when dividing integers.

This version of l3calc is a now a complete replacement for the original calc package providing the same functionality and will prevent the original calc package from loading.

`\calc_int_set:Nn`
`\calc_int_gset:Nn`
`\calc_int_add:Nn`
`\calc_int_gadd:Nn`
`\calc_int_sub:Nn`
`\calc_int_gsub:Nn`
`\calc_int_set:Nn` ⟨*int*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*int*⟩ or sets ⟨*int*⟩ to it. These operations can also be global.

`\calc_dim_set:Nn`
`\calc_dim_gset:Nn`
`\calc_dim_add:Nn`
`\calc_dim_gadd:Nn`
`\calc_dim_sub:Nn`
`\calc_dim_gsub:Nn`
`\calc_dim_set:Nn` ⟨*dim*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*dim*⟩ or sets ⟨*dim*⟩ to it. These operations can also be global.

`\calc_skip_set:Nn`
`\calc_skip_gset:Nn`
`\calc_skip_add:Nn`
`\calc_skip_gadd:Nn`
`\calc_skip_sub:Nn`
`\calc_skip_gsub:Nn`
`\calc_skip_set:Nn` ⟨*skip*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*skip*⟩ or sets ⟨*skip*⟩ to it. These operations can also be global.

`\calc_muskip_set:Nn`
`\calc_muskip_gset:Nn`
`\calc_muskip_add:Nn`
`\calc_muskip_gadd:Nn`
`\calc_muskip_sub:Nn`
`\calc_muskip_gsub:Nn`
`\calc_muskip_set:Nn` ⟨*muskip*⟩ {⟨*calc expression*⟩}

Evaluates ⟨*calc expression*⟩ and either adds or subtracts it from ⟨*muskip*⟩ or sets ⟨*muskip*⟩ to it. These operations can also be global.

`\calc_calculate_box_size:nnn` `\calc_calculate_box_size:nnn {⟨dim-set⟩} {⟨item 1⟩ ⟨item 2⟩ ... ⟨item n⟩} {⟨contents⟩}`

Sets ⟨*contents*⟩ in a temporary box `\l_tmpa_box`. Then ⟨*dim-set*⟩ is put in front of a loop that inserts `+`⟨*item*$_i$⟩ in front of `\l_tmpa_box` and this is evaluated. For instance, if we wanted to determine the total height of the text `xyz` and store it in `\l_tmpa_dim`, we would call it as.

```
\calc_calculate_box_size:nnn
  {\dim_set:Nn\l_tmpa_dim}{\box_ht:N\box_dp:N}{xyz}
```

Similarly, if we wanted the difference between height and depth, we could call it as

```
\calc_calculate_box_size:nnn
  {\dim_set:Nn\l_tmpa_dim}{\box_ht:N{-\box_dp:N}}{xyz}
```

26.1 The Implementation

Since this is basically a re-worked version of the calc package, I haven't bothered with too many comments except for in the places where this package differs. This may (and should) change at some point.

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{l3int}
\RequirePackage{l3skip}
\RequirePackage{l3box}
⟨/package⟩
⟨*initex | package⟩
```

`\l_calc_expression_tlp` `\g_calc_A_register` `\l_calc_B_register` `\l_calc_current_type_int`

Here we define some registers and pointers we will need.

```
\tlp_new:Nn\l_calc_expression_tlp{}
\def_new:Npn \g_calc_A_register{}
\def_new:Npn \l_calc_B_register{}
\int_new:N \l_calc_current_type_int
```

`\g_calc_A_int` `\l_calc_B_int` `\l_calc_C_int` `\g_calc_A_dim` `\l_calc_B_dim` `\l_calc_C_dim` `\g_calc_A_skip` `\l_calc_B_skip` `\l_calc_C_skip` `\g_calc_A_muskip` `\l_calc_B_muskip` `\l_calc_C_muskip`

For each type of register we will need three registers to do our manipulations.

```
\int_new:N \g_calc_A_int
\int_new:N \l_calc_B_int
\int_new:N \l_calc_C_int
\dim_new:N \g_calc_A_dim
```

```
\dim_new:N \l_calc_B_dim
\dim_new:N \l_calc_C_dim
\skip_new:N \g_calc_A_skip
\skip_new:N \l_calc_B_skip
\skip_new:N \l_calc_C_skip
\muskip_new:N \g_calc_A_muskip
\muskip_new:N \l_calc_B_muskip
\muskip_new:N \l_calc_C_muskip
```

`\calc_assign_generic:NNNNnn` The generic function. `#1` is a number denoting which type we are doing. (0=int, 1=dim, 2=skip, 3=muskip), `#2` = temp register A, `#3` = temp register B, `#4` is a function acting on `#5` which is the register to be set. `#6` is the calc expression. We do a little extra work so that `\real` and `\ratio` can still be used by the user.

```
\def_long_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
  \let:NN\g_calc_A_register#2
  \let:NN\l_calc_B_register#3
  \int_set:Nn \l_calc_current_type_int {#1}
  \group_begin:
    \let:NN \real \calc_real:n
    \let:NN \ratio\calc_ratio:nn
    \tlp_set:Nx\l_calc_expression_tlp{#6}
    \exp_after:NN
  \group_end:
  \exp_after:NN\calc_open:w\exp_after:NN(\l_calc_expression_tlp !
  \pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:
  #4{#5}\l_calc_B_register
}
```

A simpler version relying on `\real` and `\ratio` having our definition is

```
\def_long_new:Npn \calc_assign_generic:NNNNnn#1#2#3#4#5#6{
  \let:NN\g_calc_A_register#2\let:NN\l_calc_B_register#3
  \int_set:Nn \l_calc_current_type_int {#1}
  \tlp_set:Nx\l_calc_expression_tlp{#6}
  \exp_after:NN\calc_open:w\exp_after:NN(\l_calc_expression_tlp !
  \pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:
  #4{#5}\l_calc_B_register
}
```

`\calc_int_set:Nn` `\calc_int_gset:Nn` `\calc_int_add:Nn` `\calc_int_gadd:Nn` `\calc_int_sub:Nn` `\calc_int_gsub:Nn` Here are the individual versions for the different register types. First integer registers.

```
\def_new:Npn\calc_int_set:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_set:Nn
}
\def_new:Npn\calc_int_gset:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gset:Nn
}
\def_new:Npn\calc_int_add:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_add:Nn
}
```

```
\def_new:Npn\calc_int_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gadd:Nn
}
\def_new:Npn\calc_int_sub:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_sub:Nn
}
\def_new:Npn\calc_int_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_zero\g_calc_A_int\l_calc_B_int\int_gsub:Nn
}
```

\calc_dim_set:Nn
\calc_dim_gset:Nn
\calc_dim_add:Nn
\calc_dim_gadd:Nn
\calc_dim_sub:Nn
\calc_dim_gsub:Nn

Dimens.

```
\def_new:Npn\calc_dim_set:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_set:Nn
}
\def_new:Npn\calc_dim_gset:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gset:Nn
}
\def_new:Npn\calc_dim_add:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_add:Nn
}
\def_new:Npn\calc_dim_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_dim\l_calc_B_dim\dim_gadd:Nn
}
\def_new:Npn\calc_dim_sub:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_sub:Nn
}
\def_new:Npn\calc_dim_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_one\g_calc_A_int\l_calc_B_int\dim_gsub:Nn
}
```

\calc_skip_set:Nn
\calc_skip_gset:Nn
\calc_skip_add:Nn
\calc_skip_gadd:Nn
\calc_skip_sub:Nn
\calc_skip_gsub:Nn

Skips.

```
\def_new:Npn\calc_skip_set:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_set:Nn
}
\def_new:Npn\calc_skip_gset:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gset:Nn
}
\def_new:Npn\calc_skip_add:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_add:Nn
}
\def_new:Npn\calc_skip_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gadd:Nn
}
\def_new:Npn\calc_skip_sub:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_sub:Nn
}
\def_new:Npn\calc_skip_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_two\g_calc_A_skip\l_calc_B_skip\skip_gsub:Nn
}
```

\calc_muskip_set:Nn
\calc_muskip_gset:Nn
\calc_muskip_add:Nn
\calc_muskip_gadd:Nn
\calc_muskip_sub:Nn
\calc_muskip_gsub:Nn

Muskips.

```
\def_new:Npn\calc_muskip_set:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_set:Nn
}
\def_new:Npn\calc_muskip_gset:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gset:Nn
}
\def_new:Npn\calc_muskip_add:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_add:Nn
}
\def_new:Npn\calc_muskip_gadd:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gadd:Nn
}
\def_new:Npn\calc_muskip_sub:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_add:Nn
}
\def_new:Npn\calc_muskip_gsub:Nn{
  \calc_assign_generic:NNNNnn\c_three\g_calc_A_muskip\l_calc_B_muskip
    \muskip_gadd:Nn
}
```

`\calc_pre_scan:N` In case we found one of the special operations, this should just be executed.

```
\def_new:Npn \calc_pre_scan:N #1{
  \if_meaning:NN(#1
    \exp_after:NN\calc_open:w
  \else:
    \if_meaning:NN \calc_textsize:Nn #1
    \else:
      \if_meaning:NN \calc_maxmin_operation:Nnn #1
      \else:
```

`\calc_numeric:` uses a primitive assignment so doesn't care about these dangling `\fi:`s.

```
        \calc_numeric:
      \fi:
    \fi:
  \fi:
  #1}
```

`\calc_open:w`

```
\def_new:Npn \calc_open:w({
  \group_begin:\group_execute_after:N\calc_init_B:
  \group_begin:\group_execute_after:N\calc_init_B:
  \calc_pre_scan:N
}
```

`\calc_init_B:`
`\calc_numeric:`
`\calc_close:`

```
\def_new:Npn\calc_init_B:{\l_calc_B_register\g_calc_A_register}
\def_new:Npn\calc_numeric:{
  \tex_afterassignment:D\calc_post_scan:N
  \pref_global:D\g_calc_A_register
}
\def_new:Npn\calc_close:{
  \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
  \group_end:\pref_global:D\g_calc_A_register\l_calc_B_register
  \calc_post_scan:N}
```

`\calc_post_scan:N` Look at what token we have and decide where to go.

```
\def_new:Npn\calc_post_scan:N#1{
  \if_meaning:NN#1!\let:NN\calc_next:w\group_end: \else:
    \if_meaning:NN#1+\let:NN\calc_next:w\calc_add: \else:
      \if_meaning:NN#1-\let:NN\calc_next:w\calc_subtract: \else:
        \if_meaning:NN#1*\let:NN\calc_next:w\calc_multiply:N \else:
          \if_meaning:NN#1/\let:NN\calc_next:w\calc_divide:N \else:
            \if_meaning:NN#1)\let:NN\calc_next:w\calc_close: \else:
              \if_meaning:NN#1\scan_stop:\let:NN\calc_next:w\calc_post_scan:N
              \else:
```

If we get here, there is an error but let's also disable `\calc_next:w` since it is otherwise undefined. No need to give extra errors just for that.

```
                \let:NN \calc_next:w \use_noop:
                \calc_error:N#1
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  \fi:
  \calc_next:w}
```

`\calc_multiply:N` `\calc_divide:N` The switches for multiplication and division.

```
\def_new:Npn \calc_multiply:N #1{
  \if_meaning:NN \calc_maxmin_operation:Nnn #1
    \let:NN \calc_next:w \calc_maxmin_multiply:
  \else:
    \if_meaning:NN \calc_ratio_multiply:nn #1
      \let:NN \calc_next:w \calc_ratio_multiply:nn
    \else:
      \if_meaning:NN \calc_real_evaluate:nn #1
        \let:NN \calc_next:w \calc_real_multiply:n
      \else:
        \def:Npn \calc_next:w{\calc_multiply: #1}
      \fi:
    \fi:
  \fi:
  \calc_next:w
}
```

```
\def_new:Npn \calc_divide:N #1{
  \if_meaning:NN \calc_maxmin_operation:Nnn #1
    \let:NN \calc_next:w \calc_maxmin_divide:
  \else:
    \if_meaning:NN \calc_ratio_multiply:nn #1
      \let:NN \calc_next:w \calc_ratio_divide:nn
    \else:
      \if_meaning:NN \calc_real_evaluate:nn #1
        \let:NN \calc_next:w \calc_real_divide:n
      \else:
        \def:Npn \calc_next:w{\calc_divide: #1}
      \fi:
    \fi:
  \fi:
  \calc_next:w
}
```

\calc_generic_add:N
\calc_add:
\calc_subtract:
\calc_add_A_to_B:
\calc_subtract_A_from_B:

Here is how we add and subtract.

```
\def_new:Npn\calc_generic_add_or_subtract:N#1{
  \group_end:
  \pref_global:D\g_calc_A_register\l_calc_B_register\group_end:
  \group_begin:\group_execute_after:N#1\group_begin:
  \group_execute_after:N\calc_init_B:
  \calc_pre_scan:N}
\def_new:Npn\calc_add:{\calc_generic_add_or_subtract:N\calc_add_A_to_B:}
\def_new:Npn\calc_subtract:{
  \calc_generic_add_or_subtract:N\calc_subtract_A_from_B:}
```

Don't use `\tex_advance:D` since it allows overflows.

```
\def_new:Npn\calc_add_A_to_B:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register + \g_calc_A_register\scan_stop:
}
\def_new:Npn\calc_subtract_A_from_B:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register - \g_calc_A_register\scan_stop:
}
```

_generic_multiply_or_divide:N
\calc_multiply_B_by_A:
\calc_divide_B_by_A:
\calc_multiply:
\calc_divide:

And here is how we multiply and divide. Note that we do not use the primitive TEX operations but the expandable operations provided by ε-TEX. This means that all results are rounded not truncated!

```
\def_new:Npn\calc_generic_multiply_or_divide:N#1{
  \group_end:
  \group_begin:
  \let:NN\g_calc_A_register\g_calc_A_int
  \let:NN\l_calc_B_register\l_calc_B_int
  \int_zero:N \l_calc_current_type_int
  \group_execute_after:N#1\calc_pre_scan:N
}
\def_new:Npn\calc_multiply_B_by_A:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register*\g_calc_A_int\scan_stop:
}
\def_new:Npn\calc_divide_B_by_A:{
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register/\g_calc_A_int\scan_stop:
}
\def_new:Npn\calc_multiply:{
  \calc_generic_multiply_or_divide:N\calc_multiply_B_by_A:}
\def_new:Npn\calc_divide:{
  \calc_generic_multiply_or_divide:N\calc_divide_B_by_A:}
```

`\calc_calculate_box_size:nnn` `\calc_calculate_box_size_aux:n` Put something in a box and measure it. `#1` is a list of `\box_ht:N` etc., `#2` should be `\dim_set:Nn`⟨*dim register*⟩ or `\dim_gset:Nn`⟨*dim register*⟩ and `#3` is the contents.

```
\def_long_new:Npn \calc_calculate_box_size:nnn #1#2#3{
  \hbox_set:Nn \l_tmpa_box {{#3}}
  #2{\c_zero_dim \tlist_map_function:nN{#1}\calc_calculate_box_size_aux:n}
}
```

Helper for calculating the final dimension.

```
\def:Npn \calc_calculate_box_size_aux:n#1{ + #1\l_tmpa_box}
```

`\calc_textsize:Nn` Now we can define `\calc_textsize:Nn`.

```
\def_protected_long:Npn \calc_textsize:Nn#1#2{
  \group_begin:
  \let:NN\calc_widthof_aux:n\box_wd:N
  \let:NN\calc_heightof_aux:n\box_ht:N
  \let:NN\calc_depthof_aux:n\box_dp:N
  \def:Npn\calc_totalheightof_aux:n{\box_ht:N\box_dp:N}
  \exp_args:No\calc_calculate_box_size:nnn{#1}
  {\dim_gset:Nn\g_calc_A_register}
```

Restore the four user commands here since there might be a recursive call.

```
  {
    \let:NN \calc_depthof_aux:n \calc_depthof_auxi:n
    \let:NN \calc_widthof_aux:n \calc_widthof_auxi:n
    \let:NN \calc_heightof_aux:n \calc_heightof_auxi:n
    \let:NN \calc_totalheightof_aux:n \calc_totalheightof_auxi:n
    #2
  }
  \group_end:
  \calc_post_scan:N
}
```

`\calc_ratio_multiply:nn` `\calc_ratio_divide:nn` Evaluate a ratio. If we were already evaluation a ⟨*muskip*⟩ register, the ratio is probably also done with this type and we'll have to convert them to regular points.

```
\def_protected_long:Npn\calc_ratio_multiply:nn#1#2{
  \group_end:\group_begin:
  \if_num:w\l_calc_current_type_int < \c_three
    \calc_dim_set:Nn\l_calc_B_int{#1}
    \calc_dim_set:Nn\l_calc_C_int{#2}
  \else:
    \calc_dim_muskip:Nn{\l_calc_B_int\etex_mutoglue:D}{#1}
    \calc_dim_muskip:Nn{\l_calc_C_int\etex_mutoglue:D}{#2}
  \fi:
```

Then store the ratio as a fraction, which we just pass on.

```
  \gdef:Npx\calc_calculated_ratio:{
    \int_use:N\l_calc_B_int/\int_use:N\l_calc_C_int
  }
  \group_end:
```

Here we set the new value of `\l_calc_B_register` and remember to evaluate it as the correct type. Note that the intermediate calculation is a scaled operation (meaning the intermediate value is 64-bit) so we don't get into trouble when first multiplying by a large number and then dividing.

```
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register*\calc_calculated_ratio:\scan_stop:
  \group_begin:
  \calc_post_scan:N}
```

Division is just flipping the arguments around.

```
\def_long_new:Npn \calc_ratio_divide:nn#1#2{\calc_ratio_multiply:nn{#2}{#1}}
```

`\calc_real_evaluate:nn` `\calc_real_multiply:n` `\calc_real_divide:n` Although we could define the `\real` function as a subcase of `\ratio`, this is horribly inefficient since we just want to convert the decimal to a fraction.

```
\def_protected_new:Npn\calc_real_evaluate:nn #1#2{
  \group_end:
  \l_calc_B_register
  \if_case:w\l_calc_current_type_int
  \etex_numexpr:D\or:
  \etex_dimexpr:D\or:
  \etex_glueexpr:D\or:
  \etex_muexpr:D\fi:
  \l_calc_B_register *
    \tex_number:D \dim_eval:n{#1pt}/
    \tex_number:D\dim_eval:n{#2pt}
  \scan_stop:
  \group_begin:
  \calc_post_scan:N}
\def_new:Npn \calc_real_multiply:n #1{\calc_real_evaluate:nn{#1}{1}}
\def_new:Npn \calc_real_divide:n {\calc_real_evaluate:nn{1}}
```

\calc_maxmin_operation:Nnn
\calc_maxmin_generic:Nnn
\calc_maxmin_div_or_mul:NNnn
\calc_maxmin_multiply:
\calc_maxmin_multiply:

The max and min functions.

```
\def_protected_long:Npn\calc_maxmin_operation:Nnn#1#2#3{
  \group_begin:
  \calc_maxmin_generic:Nnn#1{#2}{#3}
  \group_end:
  \calc_post_scan:N
}
```

`#1` is either `>` or `<` and was expanded into this initially.

```
\def_protected_long_new:Npn \calc_maxmin_generic:Nnn#1#2#3{
  \group_begin:
  \if_case:w\l_calc_current_type_int
    \calc_int_set:Nn\l_calc_C_int{#2}%
    \calc_int_set:Nn\l_calc_B_int{#3}%
    \pref_global:D\g_calc_A_register
    \if_num:w\l_calc_C_int#1\l_calc_B_int
    \l_calc_C_int\else:\l_calc_B_int\fi:
  \or:
    \calc_dim_set:Nn\l_calc_C_dim{#2}%
    \calc_dim_set:Nn\l_calc_B_dim{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_dim#1\l_calc_B_dim
    \l_calc_C_dim\else:\l_calc_B_dim\fi:
  \or:
    \calc_skip_set:Nn\l_calc_C_skip{#2}%
    \calc_skip_set:Nn\l_calc_B_skip{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_skip#1\l_calc_B_skip
    \l_calc_C_skip\else:\l_calc_B_skip\fi:
  \else:
    \calc_muskip_set:Nn\l_calc_C_muskip{#2}%
    \calc_muskip_set:Nn\l_calc_B_muskip{#3}%
    \pref_global:D\g_calc_A_register
    \if_dim:w\l_calc_C_muskip#1\l_calc_B_muskip
    \l_calc_C_muskip\else:\l_calc_B_muskip\fi:
```

```
  \fi:
  \group_end:
}
\def_long_new:Npn\calc_maxmin_div_or_mul:NNnn#1#2#3#4{
  \group_end:
  \group_begin:
  \int_zero:N\l_calc_current_type_int
  \group_execute_after:N#1
  \calc_maxmin_generic:Nnn#2{#3}{#4}
  \group_end:
  \group_begin:
  \calc_post_scan:N
}
\def_new:Npn\calc_maxmin_multiply:{
  \calc_maxmin_div_or_mul:NNnn\calc_multiply_B_by_A:}
\def_new:Npn\calc_maxmin_divide:  {
  \calc_maxmin_div_or_mul:NNnn\calc_divide_B_by_A:}
```

`\calc_error:N` The error message.

```
\def_new:Npn\calc_error:N#1{
  \PackageError{calc}
  {`\token_to_string:N#1'~ invalid~ at~ this~ point}
  {I~ expected~ to~ see~ one~ of:~ +~ -~ *~ /~ )}
}
```

26.2 Higher level commands

The various operations allowed.

`\calc_maxof:nn` `\calc_minof:nn` `\maxof` `\minof` Max and min operations

```
\def_long_new:Npn \calc_maxof:nn#1#2{
  \calc_maxmin_operation:Nnn > \exp_not:n{{#1}{#2}}
}
\def_long_new:Npn \calc_minof:nn#1#2{
  \calc_maxmin_operation:Nnn < \exp_not:n{{#1}{#2}}
}
\let:NN \maxof \calc_maxof:nn
\let:NN \minof \calc_minof:nn
```

`\calc_widthof:n` `\calc_widthof_aux:n` `\calc_widthof_auxi:n` `\calc_heightof:n` `\calc_heightof_aux:n` `\calc_heightof_auxi:n` `\calc_depthof:n` `\calc_depthof_aux:n` `\calc_depthof_auxi:n` `\calc_totalheightof:n` `\calc_totalheightof_aux:n` `\calc_totalheightof_auxi:n` Text dimension commands.

```
\def_long_new:Npn \calc_widthof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_widthof_aux:n\exp_not:n{{#1}}
}
\def_long_new:Npn \calc_heightof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
}
\def_long_new:Npn \calc_depthof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
}
```

```
\def_long_new:Npn \calc_totalheightof:n#1{
  \calc_textsize:Nn \exp_not:N\calc_totalheightof_aux:n \exp_not:n{{#1}}
}
\def_long_new:Npn \calc_widthof_aux:n #1{
  \exp_not:N\calc_widthof_aux:n\exp_not:n{{#1}}
}
\let_new:NN \calc_widthof_auxi:n \calc_widthof_aux:n
\def_long_new:Npn \calc_depthof_aux:n #1{
  \exp_not:N\calc_depthof_aux:n\exp_not:n{{#1}}
}
\let_new:NN \calc_depthof_auxi:n \calc_depthof_aux:n
\def_long_new:Npn \calc_heightof_aux:n #1{
  \exp_not:N\calc_heightof_aux:n\exp_not:n{{#1}}
}
\let_new:NN \calc_heightof_auxi:n \calc_heightof_aux:n
\def_long_new:Npn \calc_totalheightof_aux:n #1{
  \exp_not:N\calc_totalheightof_aux:n\exp_not:n{{#1}}
}
\let_new:NN \calc_totalheightof_auxi:n \calc_totalheightof_aux:n
```

`\calc_ratio:nn` `\calc_real:n` Ratio and real.

```
\def_long_new:Npn \calc_ratio:nn#1#2{
  \calc_ratio_multiply:nn\exp_not:n{{#1}{#2}}}
\def_new:Npn \calc_real:n {\calc_real_evaluate:nn}
```

We can implement real and ratio without actually using these names. We'll see.

`\widthof` `\heightof` `\depthof` `\totalheightof` `\ratio` `\real` User commands.

```
\let:NN \depthof\calc_depthof:n
\let:NN \widthof\calc_widthof:n
\let:NN \heightof\calc_heightof:n
\let:NN \totalheightof\calc_totalheightof:n
%%\let:NN \ratio\calc_ratio:nn
%%\let:NN \real\calc_real:n
```

`\setlength` `\gsetlength` `\addtolength` `\gaddtolength`

```
\def_protected:Npn \setlength{\calc_skip_set:Nn}
\def_protected:Npn \gsetlength{\calc_skip_gset:Nn}
\def_protected:Npn \addtolength{\calc_skip_add:Nn}
\def_protected:Npn \gaddtolength{\calc_skip_gadd:Nn}
```

`\calc_setcounter:nn` `\calc_addtocounter:nn` `\calc_stepcounter:n` Document commands for LaTeX 2_ε counters. Also add support for amstext. Note that when l3breqn is used, `\mathchoice` will no longer need this switch as the argument is only executed once.

`\setcounter` `\addtocounter` `\stepcounter` `\calc_chk_document_counter:nn`

```
\newif\iffirstchoice@    \firstchoice@true
\def_protected:Npn \calc_setcounter:nn#1#2{
  \calc_chk_document_counter:nn{#1}{
    \exp_args:Nc\calc_int_gset:Nn {c@#1}{#2}
  }
```

```
}
\def_protected:Npn \calc_addtocounter:nn#1#2{
  \iffirstchoice@
  \calc_chk_document_counter:nn{#1}{
    \exp_args:Nc\calc_int_gadd:Nn {c@#1}{#2}
  }
  \fi:
}
\def_protected:Npn \calc_stepcounter:n#1{
  \iffirstchoice@
  \calc_chk_document_counter:nn{#1}{
    \int_gincr:c {c@#1}
    \group_begin:
      \let:NN \@elt\@stpelt \cs_use:c{cl@#1}
    \group_end:
  }
  \fi:
}
\def_new:Npn \calc_chk_document_counter:nn#1{
  \cs_if_free:cTF{c@#1}{\@nocounterr {#1}}
}
\let:NN \setcounter \calc_setcounter:nn
\let:NN \addtocounter \calc_addtocounter:nn
\let:NN \stepcounter \calc_stepcounter:n
\AtBeginDocument{
  \let:NN \setcounter \calc_setcounter:nn
  \let:NN \addtocounter \calc_addtocounter:nn
  \let:NN \stepcounter \calc_stepcounter:n
}
```

Prevent the usual calc from loading.

```
⟨package⟩\def:cpn{ver@calc.sty}{2005/08/06}

⟨/initex | package⟩
⟨*showmemory⟩
\showMemUsage
⟨/showmemory⟩
```

Change History

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

C

D

E

F

G

H

I

M

N

Q

R

S

T