The LaTeX3 Sources

The LaTeX3 Project*

April 23, 2012

Abstract

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

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

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

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

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

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

Contents

Part I

Introduction to expl3 and this document

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

1 Naming functions and variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`g` Parameters whose value should only be set globally.

`l` Parameters whose value should only be set locally.

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

`bool` Either true or false.

`box` Box register.

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

`clist` Comma separated list.

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

`dim` “Rigid” lengths.

`fp` floating-point values;

`int` Integer-valued count register.

`prop` Property list.

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

`skip` “Rubber” lengths.

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

`tl` Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

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

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

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

2 Documentation conventions

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

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

`\seq_new:N` ⟨*sequence*⟩

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

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

`\cs_to_str:N` ★

`\cs_to_str:N` ⟨*cs*⟩

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

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

`\seq_map_function:NN` ☆

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

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

`\xetex_if_engine:TF` ⋆

`\xetex_if_engine:TF {⟨true code⟩} {⟨false code⟩}`

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

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

`\l_tmpa_tl`

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

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

`\token_to_str:N` ⋆

`\token_to_str:N ⟨token⟩`

The normal description text.

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

3 Formal language conventions which apply generally

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

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

Part II
The l3bootstrap package
Bootstrap code

4 Using the LaTeX3 modules

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

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

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

`\ExplSyntaxNamesOn`
`\ExplSyntaxNamesOff`

`\ExplSyntaxNamesOn` ⟨*code*⟩ `\ExplSyntaxNamesOff`

The `\ExplSyntaxOn` function switches to a category code regimé in which the colon (`:`) and underscore (`_`) are treated as "letters", thus allowing access to the names of code functions and variables. In contrast to `\ExplSyntaxOn`, using `\ExplSyntaxNamesOn` does not cause spaces to be ignored. The `\ExplSyntaxNamesOff` reverts to the document category code regimé.

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

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

These functions act broadly in the same way as the LaTeX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as LaTeX 2_ε provides in turning on `\makeatletter` within package and class code.)

`\GetIdInfo`

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

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

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

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

Part III

The l3names package
Namespace for primitives

5 Setting up the LaTeX3 programming language

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

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

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

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

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

`\pdftex_...` Introduced by pdfTeX;

`\xetex_...` Introduced by XeTeX;

`\luatex_...` Introduced by LuaTeX.

Part IV

The l3basics package
Basic definitions

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

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

6 No operation functions

`\prg_do_nothing:` ⋆

`\prg_do_nothing:`

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

`\scan_stop:`

`\scan_stop:`

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

7 Grouping material

`\group_begin:`
`\group_end:`

`\group_begin:`
`\group_end:`

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

`\group_insert_after:N`

`\group_insert_after:N` ⟨*token*⟩

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

8 Control sequences and functions

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

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

8.1 Defining functions

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

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

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

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

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

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

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

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

8.2 Defining new functions using primitive parameter text

`\cs_new:Npn`
`\cs_new:(cpn|Npx|cpx)`

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

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

`\cs_new_nopar:Npn`
`\cs_new_nopar:(cpn|Npx|cpx)`

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

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

`\cs_new_protected:Npn`
`\cs_new_protected:(cpn|Npx|cpx)`

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

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

`\cs_new_protected_nopar:Npn`
`\cs_new_protected_nopar:(cpn|Npx|cpx)`

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

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

`\cs_set:Npn`
`\cs_set:(cpn|Npx|cpx)`

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

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

`\cs_set_nopar:Npn`
`\cs_set_nopar:(cpn|Npx|cpx)`

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

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

`\cs_set_protected:Npn`
`\cs_set_protected:(cpn|Npx|cpx)`

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

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

`\cs_set_protected_nopar:Npn`
`\cs_set_protected_nopar:(cpn|Npx|cpx)`

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

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

`\cs_gset:Npn`
`\cs_gset:(cpn|Npx|cpx)`

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

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

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:(cpn|Npx|cpx)`

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

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

`\cs_gset_protected:Npn`
`\cs_gset_protected:(cpn|Npx|cpx)`

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

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

`\cs_gset_protected_nopar:Npn`
`\cs_gset_protected_nopar:(cpn|Npx|cpx)`

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

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

8.3 Defining new functions using the signature

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Updated: 2012-01-14

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

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

8.4 Copying control sequences

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

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

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

`\cs_new_eq:NN` ⟨*cs 1*⟩ ⟨*cs 2*⟩
`\cs_new_eq:NN` ⟨*cs 1*⟩ ⟨*token*⟩

Globally creates ⟨*control sequence 1*⟩ and sets it to have the same meaning as ⟨*control sequence 2*⟩ or ⟨*token*⟩. The second control sequence may subsequently be altered without affecting the copy.

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

`\cs_set_eq:NN` ⟨*cs 1*⟩ ⟨*cs 2*⟩
`\cs_set_eq:NN` ⟨*cs 1*⟩ ⟨*token*⟩

Sets ⟨*control sequence 1*⟩ to have the same meaning as ⟨*control sequence 2*⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence 1*⟩ is restricted to the current TeX group level.

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

`\cs_gset_eq:NN` ⟨*cs 1*⟩ ⟨*cs 2*⟩
`\cs_gset_eq:NN` ⟨*cs 1*⟩ ⟨*token*⟩

Globally sets ⟨*control sequence 1*⟩ to have the same meaning as ⟨*control sequence 2*⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence 1*⟩ is *not* restricted to the current TeX group level: the assignment is global.

8.5 Deleting control sequences

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

`\cs_undefine:N`
`\cs_undefine:c`

Updated: 2011-09-15

`\cs_undefine:N` ⟨*control sequence*⟩

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

8.6 Showing control sequences

\cs_meaning:N ⋆
\cs_meaning:c ⋆

Updated: 2011-12-22

\cs_meaning:N ⟨*control sequence*⟩

This function expands to the *meaning* of the ⟨*control sequence*⟩ control sequence. This will show the ⟨*replacement text*⟩ for a macro.

TeXhackers note: This is TeX's \meaning primitive. The c variant correctly reports undefined arguments.

\cs_show:N
\cs_show:c

Updated: 2011-12-22

\cs_show:N ⟨*control sequence*⟩

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

TeXhackers note: This is the TeX primitive \show.

8.7 Converting to and from control sequences

\use:c ⋆

\use:c {⟨*control sequence name*⟩}

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

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

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

and

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

would be equivalent to

```
\abc
```

after two expansions of \use:c.

\cs:w ⋆
\cs_end: ⋆

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

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

TeXhackers note: These are the TeX primitives \csname and \endcsname.

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

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

and

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

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

`\cs_to_str:N` ⋆

`\cs_to_str:N {⟨control sequence⟩}`

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

9 Using or removing tokens and arguments

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

`\use:n` ⋆
`\use:(nn|nnn|nnnn)` ⋆

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

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

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

will result in the input stream containing

```
abc { def }
```

i.e. only the outer braces will be removed.

`\use_i:nn` ⋆
`\use_ii:nn` ⋆

`\use_i:nn {`$\langle arg_1 \rangle$`} {`$\langle arg_2 \rangle$`}`

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

`\use_i:nnn` ⋆
`\use_ii:nnn` ⋆
`\use_iii:nnn` ⋆

`\use_i:nnn {`$\langle arg_1 \rangle$`} {`$\langle arg_2 \rangle$`} {`$\langle arg_3 \rangle$`}`

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

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

`\use_i:nnnn {`$\langle arg_1 \rangle$`} {`$\langle arg_2 \rangle$`} {`$\langle arg_3 \rangle$`} {`$\langle arg_4 \rangle$`}`

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

`\use_i_ii:nnn` ⋆

`\use_i_ii:nnn {`$\langle arg_1 \rangle$`} {`$\langle arg_2 \rangle$`} {`$\langle arg_3 \rangle$`}`

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

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

will result in the input stream containing

```
abc { def }
```

i.e. the outer braces will be removed and the third group will be removed.

`\use_none:n` ⋆
`\use_none:(nn|nnn|nnnn|nnnnn|nnnnnn|nnnnnnn|nnnnnnnn|nnnnnnnnn)` ⋆

`\use_none:n {`$\langle group_1 \rangle$`}`

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

\use:x

Updated: 2011-12-31

\use:x {⟨*expandable tokens*⟩}

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

9.1 Selecting tokens from delimited arguments

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

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

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

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

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

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

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

9.2 Decomposing control sequences

\cs_get_arg_count_from_signature:N ⋆

\cs_get_arg_count_from_signature:N ⟨*function*⟩

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

\cs_get_function_name:N ⋆

\cs_get_function_name:N ⟨*function*⟩

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

\cs_get_function_signature:N ⋆

\cs_get_function_signature:N ⟨*function*⟩

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

`\cs_split_function:NN` ⋆ `\cs_split_function:NN` ⟨*function*⟩ ⟨*processor*⟩

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

10 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

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

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

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

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

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

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

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

`\cs_if_free_p:N`

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

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

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

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

`\c_true_bool`
`\c_false_bool`

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

10.1 Tests on control sequences

`\cs_if_eq_p:NN` ⋆
`\cs_if_eq:NNTF` ⋆

`\cs_if_eq_p:NN {⟨cs1⟩} {⟨cs2⟩}`
`\cs_if_eq:NNTF {⟨cs1⟩} {⟨cs2⟩} {⟨true code⟩} {⟨false code⟩}`

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

`\cs_if_exist_p:N` ⋆
`\cs_if_exist_p:c` ⋆
`\cs_if_exist:NTF` ⋆
`\cs_if_exist:cTF` ⋆

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

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

`\cs_if_free_p:N` ⋆
`\cs_if_free_p:c` ⋆
`\cs_if_free:NTF` ⋆
`\cs_if_free:cTF` ⋆

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

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

10.2 Testing string equality

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

\str_if_eq_p:nn {⟨*tl₁*⟩} {⟨*tl₂*⟩}
\str_if_eq:nnTF {⟨*tl₁*⟩} {⟨*tl₂*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\str_if_eq_p:xx { abc } { \tl_to_str:n { abc } }
```

is logically `true`. All versions of these functions are fully expandable (including those involving an `x`-type expansion).

10.3 Engine-specific conditionals

\luatex_if_engine_p: ★
\luatex_if_engine:TF ★

Updated: 2011-09-06

\luatex_if_luatex:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects is the document is being compiled using LuaTeX.

\pdftex_if_engine_p: ★
\pdftex_if_engine:TF ★

Updated: 2011-09-06

\pdftex_if_engine:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects is the document is being compiled using pdfTeX.

\xetex_if_engine_p: ★
\xetex_if_engine:TF ★

Updated: 2011-09-06

\xetex_if_engine:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects is the document is being compiled using XeTeX.

10.4 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: ⋆
\or: ⋆
\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. `\or:` is used in case switches, see l3int for more.

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

\if_meaning:w ⋆

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

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

TEXhackers note: This is TEX's `\ifx`.

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

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

These conditionals 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 ⋆
\if_cs_exist:w ⋆

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

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

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

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

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

11 Internal kernel functions

\chk_if_exist_cs:N
\chk_if_exist_cs:c

`\chk_if_exist_cs:N` ⟨*cs*⟩

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

\chk_if_free_cs:N
\chk_if_free_cs:c

`\chk_if_free_cs:N` ⟨*cs*⟩

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

12 Experimental functions

\cs_if_exist_use:NTF ⋆
\cs_if_exist_use:cTF ⋆

New: 2011-10-10

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

If the ⟨*control sequence*⟩ exists, leave it in the input stream, followed by the ⟨*true code*⟩ (unbraced). Otherwise, leave the ⟨*false*⟩ code in the input stream. For example,

```
\cs_set:Npn \mypkg_use_character:N #1
  { \cs_if_exist_use:cF { mypkg_#1:n } { \mypkg_default:N #1 } }
```

calls the function `\mypkg_#1:n` if it exists, and falls back to a default action otherwise. This could also be done (more slowly) using `\prg_case_str:xxn`.

TeXhackers note: The `c` variants do not introduce the ⟨*control sequence*⟩ in the hash table if it is not there.

Part V

The l3expan package
Argument expansion

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

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

13 Defining new variants

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

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

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

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

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

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

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

14 Methods for defining variants

`\cs_generate_variant:Nn`

Updated: 2011-09-15

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

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

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

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

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

would generate the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the ⟨*parent control sequence*⟩ is already defined. If the ⟨*parent control sequence*⟩ is protected then the new sequence will also be protected. The ⟨*variant*⟩ is created globally, as is any `\exp_args:N`⟨*variant*⟩ function needed to carry out the expansion.

15 Introducing the variants

The available internal functions for argument expansion come in two 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 expandably, *i.e.*, functions of this type will not work correctly in arguments that are itself subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing which is not fast. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in TeX registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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

The `f` type is so special that it deserves an example. Let's pretend we want to set `\aaa` equal to the control sequence stemming from turning `b \l_tmpa_tl b` into a control sequence. Furthermore we want to store the execution of it in a ⟨*tl var*⟩. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straightforward approach is

```
\tl_set:No \l_tmpb_tl  {\cs_set_eq:Nc \aaa { b \l_tmpa_tl b } }
```

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

```
\tl_set:Nf \l_tmpb_tl {\cs_set_eq:Nc \aaa { b \l_tmpa_tl b } }
```

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

```
\cs_set_nopar:Npn \tl_set:Nf { \exp_args:NNf \tl_set:Nn }
```

If you use this type of expansion in conditional processing then you should stick to using `TF` type functions only as it does not try to finish any `\if... \fi:` itself!

16 Manipulating the first argument

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

`\exp_args:No` ⋆

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

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

`\exp_args:Nc` ⋆
`\exp_args:cc` ⋆

`\exp_args:Nc` ⟨*function*⟩ {⟨*tokens*⟩}

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

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

`\exp_args:NV` ⋆ `\exp_args:NV` ⟨*function*⟩ ⟨*variable*⟩

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

`\exp_args:Nv` ⋆ `\exp_args:Nv` ⟨*function*⟩ {⟨*tokens*⟩}

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

`\exp_args:Nf` ⋆ `\exp_args:Nf` ⟨*function*⟩ {⟨*tokens*⟩}

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

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

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

17 Manipulating two arguments

`\exp_args:NNo` ⋆
`\exp_args:(NNc|NNv|NNV|NNf|Nco|Ncf|Ncc|NVV)` ⋆

`\exp_args:NNc` ⟨*token1*⟩ ⟨*token2*⟩ {⟨*tokens*⟩}

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

`\exp_args:Nno` ⋆
`\exp_args:(NnV|Nnf|Noo|Nof|Noc|Nff|Nfo|Nnc)` ⋆

Updated: 2012-01-14

`\exp_args:Noo` ⟨*token*⟩ {⟨$tokens_1$⟩} {⟨$tokens_2$⟩}

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

\exp_args:NNx
\exp_args:(Nnx|Ncx|Nox|Nxo|Nxx)

\exp_args:NNx ⟨*token1*⟩ ⟨*token2*⟩ {⟨*tokens*⟩}

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

18 Manipulating three arguments

\exp_args:NNNo ⋆
\exp_args:(NNNV|Nccc|NcNc|NcNo|Ncco) ⋆

\exp_args:NNNo ⟨*token1*⟩ ⟨*token2*⟩ ⟨*token3*⟩ {⟨*tokens*⟩}

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

\exp_args:NNoo ⋆
\exp_args:(NNno|Nnno|Nnnc|Nooo) ⋆

\exp_args:NNNo ⟨*token1*⟩ ⟨*token2*⟩ ⟨*token3*⟩ {⟨*tokens*⟩}

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

\exp_args:NNnx
\exp_args:(NNox|Nnnx|Nnox|Noox|Ncnx|Nccx)

\exp_args:NNnx ⟨*token1*⟩ ⟨*token2*⟩ {⟨$tokens_1$⟩} {⟨$tokens_2$⟩}

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

19 Unbraced expansion

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

Updated: 2012-02-12

\exp_last_unbraced:Nno ⟨*token*⟩ ⟨*tokens1*⟩ ⟨*tokens2*⟩

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

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

\exp_last_unbraced:Nx

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

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

\exp_last_two_unbraced:Noo ⋆

\exp_last_two_unbraced:Noo ⟨*token*⟩ ⟨*tokens1*⟩ {⟨*tokens*$_2$⟩}

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

\exp_after:wN ⋆

\exp_after:wN ⟨*token1*⟩ ⟨*token2*⟩

Carries out a single expansion of ⟨*token2*⟩ (which may consume arguments) prior to the expansion of ⟨*token1*⟩. If ⟨*token2*⟩ is a TeX primitive, it will be executed rather than expanded, while if ⟨*token2*⟩ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that ⟨*token1*⟩ may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TeX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate \exp_arg:N function.

TeXhackers note: This is the TeX primitive \expandafter renamed.

20 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves will not appear after the expansion has completed.

`\exp_not:N` ★

`\exp_not:N` ⟨*token*⟩

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

TeXhackers note: This is the TeX `\noexpand` primitive.

`\exp_not:c` ★

`\exp_not:c {`⟨*tokens*⟩`}`

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

`\exp_not:n` ★

`\exp_not:n {`⟨*tokens*⟩`}`

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

TeXhackers note: This is the ε-TeX `\unexpanded` primitive.

`\exp_not:V` ★

`\exp_not:V` ⟨*variable*⟩

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

`\exp_not:v` ★

`\exp_not:v {`⟨*tokens*⟩`}`

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

`\exp_not:o` ★

`\exp_not:o {`⟨*tokens*⟩`}`

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

`\exp_not:f` ★

`\exp_not:f {`⟨*tokens*⟩`}`

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

`\exp_stop_f:` ★

Updated: 2011-06-03

`\function:f` ⟨*tokens*⟩ `\exp_stop_f:` ⟨*more tokens*⟩

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

21 Internal functions and variables

`\l_exp_internal_tl`

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

`\exp_eval_register:N` ⋆
`\exp_eval_register:c` ⋆

`\exp_eval_register:N` ⟨*variable*⟩

These functions evaluates a ⟨*variable*⟩ as part of a `V` or `v` expansion (respectively), preceded by `\c_zero` which stops the expansion of a previous `\romannumeral`. A ⟨*variable*⟩ might exist as one of two things: a parameter-less non-long, non-protected macro or a built-in TeX register such as `\count`.

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

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

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

`\cs_generate_internal_variant:n`

`\cs_generate_internal_variant:n` ⟨*arg spec*⟩

Tests if the function `\exp_args:N`⟨*arg spec*⟩ exists, and defines it if it does not. The ⟨*arg spec*⟩ should be a series of one or more of the letters `N`, `c`, `n`, `o`, `V`, `v`, `f` and `x`.

Part VI

The **l3prg** package
Control structures

Conditional processing in LaTeX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The typical states returned are ⟨*true*⟩ and ⟨*false*⟩ but other states are possible, say an ⟨*error*⟩ state for erroneous input, *e.g.*, text as input in a function comparing integers.

LaTeX3 has two forms of conditional flow processing based on these states. The firs form is predicate functions that turn the returned state into a boolean ⟨*true*⟩ or ⟨*false*⟩. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean ⟨*true*⟩ or ⟨*false*⟩ values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure.

22 Defining a set of conditional functions

`\prg_new_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\prg_set_conditional:Npnn`
`\prg_set_conditional:Nnn`

Updated: 2012-02-06

`\prg_new_conditional:Npnn \`⟨*name*⟩`:`⟨*arg spec*⟩ ⟨*parameters*⟩ `{`⟨*conditions*⟩`}` `{`⟨*code*⟩`}`
`\prg_new_conditional:Nnn \`⟨*name*⟩`:`⟨*arg spec*⟩ `{`⟨*conditions*⟩`}` `{`⟨*code*⟩`}`

These functions create a family of conditionals using the same `{`⟨*code*⟩`}` to perform the test created. Those conditionals are expandable if ⟨*code*⟩ is. The `new` versions will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

`\prg_new_protected_conditional:Npnn`
`\prg_new_protected_conditional:Nnn`
`\prg_set_protected_conditional:Npnn`
`\prg_set_protected_conditional:Nnn`

Updated: 2012-02-06

`\prg_new_protected_conditional:Npnn \`⟨*name*⟩`:`⟨*arg spec*⟩ ⟨*parameters*⟩
`{`⟨*conditions*⟩`}` `{`⟨*code*⟩`}`
`\prg_new_protected_conditional:Nnn \`⟨*name*⟩`:`⟨*arg spec*⟩
`{`⟨*conditions*⟩`}` `{`⟨*code*⟩`}`

These functions create a family of protected conditionals using the same `{`⟨*code*⟩`}` to perform the test created. The ⟨*code*⟩ does not need to be expandable. The `new` version will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version will not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `T`, `F` and `TF` (not `p`).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\`⟨*name*⟩`_p:`⟨*arg spec*⟩ — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function will not work properly for `protected` conditionals.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`T` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in this additional argument will be left on the input stream only if the test is `true`.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`F` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*false branch*⟩ code in this additional argument will be left on the input stream only if the test is `false`.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`TF` — a function with two more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in the first additional argument will be left on the input stream if the test is `true`, while the ⟨*false branch*⟩ code in the second argument will be left on the input stream if the test is `false`.

The ⟨*code*⟩ of the test may use ⟨*parameters*⟩ as specified by the second argument to `\prg_-set_conditional:Npnn`: this should match the ⟨*argument specification*⟩ but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (*cf.* `\cs_new:Nn`, *etc.*). Within the ⟨*code*⟩, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Nnn \foo_if_bar:NN { p , T , TF }
  {
    \if_meaning:w \l_tmpa_tl #1
      \prg_return_true:
    \else:
      \if_meaning:w \l_tmpa_tl #2
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the ⟨*conditions*⟩ list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch, failing to do so will result in an error if that branch is executed.

`\prg_new_eq_conditional:NNn`
`\prg_set_eq_conditional:NNn`

`\prg_new_eq_conditional:NNn \⟨name1⟩:⟨arg spec1⟩ \⟨name2⟩:⟨arg spec2⟩ {⟨conditions⟩}`

These functions copies a family of conditionals. The `new` version will check for existing definitions (*cf.* `\cs_new:Npn`) whereas the `set` version will not (*cf.* `\cs_set:Npn`). The conditionals copied are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

`\prg_return_true:` ⋆
`\prg_return_false:` ⋆

`\prg_return_true:`
`\prg_return_false:`

These functions define the logical state at the end of a conditional. As such, they should appear within the code for a conditional statement generated by `\prg_set_conditional:Npnn`, *etc.*

23 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which will generally mean being constructed from predicate functions, possibly nested).

`\bool_new:N`
`\bool_new:c`

`\bool_new:N ⟨boolean⟩`

Creates a new ⟨*boolean*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*boolean*⟩ will initially be `false`.

`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

`\bool_set_false:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `false`.

`\bool_set_true:N`
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`

`\bool_set_true:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `true`.

\bool_set_eq:NN
\bool_set_eq:(cN|Nc|cc)
\bool_gset_eq:NN
\bool_gset_eq:(cN|Nc|cc)

\bool_set_eq:NN ⟨*boolean1*⟩ ⟨*boolean2*⟩

Sets the content of ⟨*boolean1*⟩ equal to that of ⟨*boolean2*⟩.

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

\bool_set:Nn ⟨*boolean*⟩ {⟨*boolexpr*⟩}

Evaluates the ⟨*boolean expression*⟩ as described for `\bool_if:n(TF)`, and sets the ⟨*boolean*⟩ variable to the logical truth of this evaluation.

\bool_if_p:N ⋆
\bool_if_p:c ⋆
\bool_if:N*TF* ⋆
\bool_if:c*TF* ⋆

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

Tests the current truth of ⟨*boolean*⟩, and continues expansion based on this result.

\bool_show:N
\bool_show:c

New: 2012-02-09

\bool_show:N ⟨*boolean*⟩

Displays the logical truth of the ⟨*boolean*⟩ on the terminal.

\bool_show:n

New: 2012-02-09

\bool_show:n {⟨*boolean expression*⟩}

Displays the logical truth of the ⟨*boolean expression*⟩ on the terminal.

\bool_if_exist_p:N ⋆
\bool_if_exist_p:c ⋆
\bool_if_exist:N*TF* ⋆
\bool_if_exist:c*TF* ⋆

New: 2012-03-03

\bool_if_exist_p:N ⟨*boolean*⟩
\bool_if_exist:NTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\l_tmpa_bool

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_bool

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

24 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean ⟨*true*⟩ or ⟨*false*⟩ values, it seems only fitting that we also provide a parser for ⟨*boolean expressions*⟩.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean ⟨*true*⟩ or ⟨*false*⟩. It supports the logical operations And, Or and Not as the well-known infix operators `&&`, `||` and `!`. In addition to this, parentheses can be used to isolate sub-expressions. For example,

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

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

`\bool_if_p:n` ⋆
`\bool_if:nTF` ⋆

`\bool_if_p:n {⟨boolean expression⟩}`
`\bool_if:nTF {⟨boolean expression⟩} {⟨true code⟩} {⟨false code⟩}`

Tests the current truth of ⟨*boolean expression*⟩, and continues expansion based on this result. The ⟨*boolean expression*⟩ should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` ("And"), `||` ("Or"), `!` ("Not") and parentheses. Minimal evaluation is used in the processing, so that once a result is defined there is not further expansion of the tests. For example

```
\bool_if_p:n
  {
   \int_compare_p:nNn { 1 } = { 1 }
   &&
   (
     \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 } = { 4 } )
}
```

will be `true` and will not evaluate `\int_compare_p:nNn { 1 } = { \error }`. The logical Not applies to the next single predicate or group. As shown above, this means that any predicates requiring an argument have to be given within parentheses.

`\bool_not_p:n` ⋆

`\bool_not_p:n {⟨boolean expression⟩}`

Function version of `!(`⟨*boolean expression*⟩`)` within a boolean expression.

`\bool_xor_p:nn` ⋆

`\bool_xor_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₁⟩}`

Implements an "exclusive or" operation between two boolean expressions. There is no infix operation for this logical operator.

25 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_until_do:Nn ☆
\bool_until_do:cn ☆

`\bool_until_do:Nn {⟨boolean⟩} {⟨code⟩}`

This function firsts checks the logical value of the ⟨*boolean*⟩. If it is `false` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean*⟩ is re-evaluated. The process will then loop until the ⟨*boolean*⟩ is `true`.

\bool_while_do:Nn ☆
\bool_while_do:cn ☆

`\bool_while_do:Nn {⟨boolean⟩} {⟨code⟩}`

This function firsts checks the logical value of the ⟨*boolean*⟩. If it is `true` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean*⟩ is re-evaluated. The process will then loop until the ⟨*boolean*⟩ is `false`.

\bool_until_do:nn ☆

`\bool_until_do:nn {⟨boolean expression⟩} {⟨code⟩}`

This function firsts checks the logical value of the ⟨*boolean expression*⟩ (as described for `\bool_if:nTF`). If it is `false` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean expression*⟩ is re-evaluated. The process will then loop until the ⟨*boolean expression*⟩ is `true`.

\bool_while_do:nn ☆

`\bool_while_do:nn {⟨boolean expression⟩} {⟨code⟩}`

This function firsts checks the logical value of the ⟨*boolean expression*⟩ (as described for `\bool_if:nTF`). If it is `true` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean expression*⟩ is re-evaluated. The process will then loop until the ⟨*boolean expression*⟩ is `false`.

26 Switching by case

For cases where a number of cases need to be considered a family of case-selecting functions are available.

\prg_case_int:nnn ★

Updated: 2011-09-17

```
\prg_case_int:nnn {⟨test integer expression⟩}
  {
    {⟨intexpr case1⟩} {⟨code case1⟩}
    {⟨intexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨intexpr case_n⟩} {⟨code case_n⟩}
  }
  {⟨else case⟩}
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream. If none of the tests are `true` then the `else code` will be left in the input stream.

As an example of `\prg_case_int:nnn`:

```
\prg_case_int:nnn
  { 2 * 5 }
  {
    { 5 }       { Small }
    { 4 + 6 }   { Medium }
    { -2 * 10 } { Negative }
  }
  { No idea! }
```

will leave "`Medium`" in the input stream.

\prg_case_dim:nnn ⋆

Updated: 2011-07-06

```
\prg_case_dim:nnn {⟨test dimension expression⟩}
  {
    {⟨dimexpr case1⟩} {⟨code case1⟩}
    {⟨dimexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨dimexpr case_n⟩} {⟨code case_n⟩}
  }
  {⟨else case⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream. If none of the tests are `true` then the `else code` will be left in the input stream.

\prg_case_str:nnn ⋆
\prg_case_str:(onn|xxn) ⋆

Updated: 2011-09-17

```
\prg_case_str:nnn {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string case_n⟩} {⟨code case_n⟩}
  }
  {⟨else case⟩}
```

This function compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩. If the two are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. If none of the tests are `true` then the `else code` will be left in the input stream. The `xx` variant fully expands ⟨*strings*⟩ before comparing them, but does not expand the corresponding ⟨*code*⟩. It is fully expandable, in the same way as the underlying `\str_if_eq:xxTF` test.

\prg_case_tl:Nnn ⋆
\prg_case_tl:cnn ⋆

Updated: 2011-09-17

```
\prg_case_tl:Nnn ⟨test token list variable⟩
  {
    ⟨token list variable case1⟩ {⟨code case1⟩}
    ⟨token list variable case2⟩ {⟨code case2⟩}
    ...
    ⟨token list variable case_n⟩ {⟨code case_n⟩}
  }
  {⟨else case⟩}
```

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for `\tl_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. If none of the tests are `true` then the `else code` will be left in the input stream.

27 Producing n copies

\prg_replicate:nn ⋆

Updated: 2011-07-04

```
\prg_replicate:nn {⟨integer expression⟩} {⟨tokens⟩}
```

Evaluates the ⟨*integer expression*⟩ (which should be zero or positive) and creates the resulting number of copies of the ⟨*tokens*⟩. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

\prg_stepwise_function:nnnN ☆

Updated: 2011-09-06

```
\prg_stepwise_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩}
⟨function⟩
```

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). Thus ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\prg_stepwise_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]

\prg_stepwise_inline:nnnn

Updated: 2011-09-06

```
\prg_stepwise_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}
```

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*code*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). Thus the ⟨*code*⟩ should define a function of one argument (`#1`).

\prg_stepwise_variable:nnnNn

Updated: 2011-09-06

`\prg_stepwise_variable:nnnNn`
`{⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

28 Detecting TeX's mode

\mode_if_horizontal_p: ⋆
\mode_if_horizontal:*TF* ⋆

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF {⟨true code⟩} {⟨false code⟩}`

Detects if TeX is currently in horizontal mode.

\mode_if_inner_p: ⋆
\mode_if_inner:*TF* ⋆

`\mode_if_inner_p:`
`\mode_if_inner:TF {⟨true code⟩} {⟨false code⟩}`

Detects if TeX is currently in inner mode.

\mode_if_math_p: ⋆
\mode_if_math:*TF* ⋆

Updated: 2011-09-05

`\mode_if_math:TF {⟨true code⟩} {⟨false code⟩}`

Detects if TeX is currently in maths mode.

\mode_if_vertical_p: ⋆
\mode_if_vertical:*TF* ⋆

`\mode_if_vertical_p:`
`\mode_if_vertical:TF {⟨true code⟩} {⟨false code⟩}`

Detects if TeX is currently in vertical mode.

29 Internal programming functions

\group_align_safe_begin: ⋆
\group_align_safe_end: ⋆

Updated: 2011-08-11

```
\group_align_safe_begin:
...
\group_align_safe_end:
```

These functions are used to enclose material in a TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

`\scan_align_safe_stop:`

Updated: 2011-09-06

`\scan_align_safe_stop:`

Stops TEX's scanner looking for expandable control sequences at the beginning of an alignment cell. This function is required, for example, to obtain the expected output when testing `\mode_if_math:TF` at the start of a math array cell: placing `\scan_align_safe_stop:` before `\mode_if_math:TF` will give the correct result. This function does not destroy any kerning if used in other locations, but *does* render functions non-expandable.

TEXhackers note: This is a protected version of `\prg_do_nothing:`, which therefore stops TEX's scanner in the circumstances described without producing any affect on the output.

`\prg_variable_get_scope:N` ⋆

`\prg_variable_get_scope:N` ⟨*variable*⟩

Returns the scope (`g` for global, blank otherwise) for the ⟨*variable*⟩.

`\prg_variable_get_type:N` ⋆

`\prg_variable_get_type:N` ⟨*variable*⟩

Returns the type of ⟨*variable*⟩ (`tl`, `int`, *etc.*)

`\if_predicate:w` ⋆

`\if_predicate:w` ⟨*predicate*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the ⟨*predicate*⟩ but to make the coding clearer this should be done through `\if_bool:N`.)

`\if_bool:N` ⋆

`\if_bool:N` ⟨*boolean*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

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

`\prg_break_point:n` ⋆

`\prg_break_point:n` ⟨*tokens*⟩

Used to mark the end of a recursion or mapping: the functions `\prg_map_break:` and `\prg_map_break:n` use this to break out of the loop. After the loop ends, the ⟨*tokens*⟩ are inserted into the input stream. This occurs even if the the break functions are *not* applied: `\prg_break_point:n` is functionally-equivalent in these cases to `\use:n`.

`\prg_map_break:` ⋆
`\prg_map_break:n` ⋆

```
\prg_map_break:n {⟨user code⟩}
...
\prg_break_point:n {⟨ending code⟩}
```

Breaks a recursion in mapping contexts, inserting in the input stream the ⟨*user code*⟩ after the ⟨*ending code*⟩ for the loop.

Part VII
The l3quark package
Quarks

30 Introduction to quarks and scan marks

Two special types of constants in LaTeX3 are "quarks" and "scan marks". By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`. *Scan marks are an experimental feature.*

30.1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, with the most command use case as the 'stop token' (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
  { <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

30.2 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence will never expand in an expansion context and will be (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

31 Defining quarks

`\quark_new:N`

`\quark_new:N` ⟨*quark*⟩

Creates a new ⟨*quark*⟩ which expands only to ⟨*quark*⟩. The ⟨*quark*⟩ will be defined globally, and an error message will be raised if the name was already taken.

`\q_stop`

Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

`\q_mark`

Used as a marker for delimited arguments when `\q_stop` is already in use.

Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to `\q_stop`, which is only ever used as a delimiter).

`\q_no_value`

A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a "return" value by functions such as `\prop_get:NnN` if there is no data to return.

32 Quark tests

The method used to define quarks means that the single token (`N`) tests are faster than the multi-token (`n`) tests. The later should therefore only be used when the argument can definitely take more than a single token.

`\quark_if_nil_p:N` ★
`\quark_if_nil:NTF` ★

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

Tests if the ⟨*token*⟩ is equal to `\q_nil`.

`\quark_if_nil_p:n` ★
`\quark_if_nil_p:(o|V)` ★
`\quark_if_nil:nTF` ★
`\quark_if_nil:(o|V)TF` ★

`\quark_if_nil_p:n` {⟨*token list*⟩}
`\quark_if_nil:nTF` {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_nil` (distinct from ⟨*token list*⟩ being empty or containing `\q_nil` plus one or more other tokens).

\quark_if_no_value_p:N ⋆
\quark_if_no_value_p:c ⋆
\quark_if_no_value:NTF ⋆
\quark_if_no_value:cTF ⋆

\quark_if_no_value_p:N ⟨*token*⟩
\quark_if_no_value:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is equal to \q_no_value.

\quark_if_no_value_p:n ⋆
\quark_if_no_value:nTF ⋆

\quark_if_no_value_p:n {⟨*token list*⟩}
\quark_if_no_value:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only \q_no_value (distinct from ⟨*token list*⟩ being empty or containing \q_no_value plus one or more other tokens).

33 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. Can you guess why the documentation for this quark requires us to write the control sequence with the wrong slash before it?

\q_recursion_stop

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

\quark_if_recursion_tail_stop:N

\quark_if_recursion_tail_stop:N ⟨*token*⟩

Tests if ⟨*token*⟩ contains only the marker \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items.

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o

Updated: 2011-09-06

\quark_if_recursion_tail_stop:n {⟨*token list*⟩}

Tests if the ⟨*token list*⟩ contains only \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items.

\quark_if_recursion_tail_stop_do:Nn

\quark_if_recursion_tail_stop_do:Nn ⟨*token*⟩ {⟨*insertion*⟩}

Tests if ⟨*token*⟩ contains only the marker \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on

Updated: 2011-09-06

\quark_if_recursion_tail_stop_do:nn {⟨*token list*⟩} {⟨*insertion*⟩}

Tests if the ⟨*token list*⟩ contains only \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_break:N
\quark_if_recursion_tail_break:n

\quark_if_recursion_tail_break:n {⟨*token list*⟩}

Tests if ⟨*token list*⟩ contains only \q_recursion_tail, and if so terminates the recursion using \prg_map_break:. The recursion end should be marked by \prg_break_point:n.

34 Scan marks

\scan_new:N

\scan_new:N ⟨*scan mark*⟩

Creates a new ⟨*scan mark*⟩ which is set equal to \scan_stop:. The ⟨*scan mark*⟩ will be defined globally, and an error message will be raised if the name was already taken by another scan mark.

\s_stop

Used at the end of a set of instructions, as a marker that can be jumped to using \use_none_delimit_by_s_stop:w.

\use_none_delimit_by_s_stop:w

\use_none_delimit_by_s_stop:w ⟨*tokens*⟩ \s_stop

Removes the ⟨*tokens*⟩ and \s_stop from the input stream. This leads to a low-level TeX error if \s_stop is absent.

35 Internal quark functions

\use_none_delimit_by_q_recursion_stop:w

\use_none_delimit_by_q_recursion_stop:w ⟨*tokens*⟩ \q_recursion_stop

Used to prematurely terminate a recursion using \q_recursion_stop as the end marker, removing any remaining ⟨*tokens*⟩ from the input stream.

\use_i_delimit_by_q_recursion_stop:nw

\use_i_delimit_by_q_recursion_stop:nw {⟨*insertion*⟩} ⟨*tokens*⟩ \q_recursion_stop

Used to prematurely terminate a recursion using \q_recursion_stop as the end marker, removing any remaining ⟨*tokens*⟩ from the input stream. The ⟨*insertion*⟩ is then made into the input stream after the end of the recursion.

Part VIII

The l3token package
Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such 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 by a single control sequence is a "token list variable" `tl var`. Functions for these two types are found in the l3tl module.

36 All possible tokens

Let us start by reviewing every case that a given token can fall into. It is very important to distinguish two aspects of a token: its meaning, and what it looks like.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three for the same internal operation of TeX, namely the primitive testing the next two characters for equality of their character code. They behave identically in many situations. However, TeX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below will take everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

37 Character tokens

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

`\char_set_catcode_letter:N` ⟨*character*⟩

Sets the category code of the ⟨*character*⟩ to that indicated in the function name. Depending on the current category code of the ⟨*token*⟩ the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```
\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
```

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

Sets the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. This version can be used to set up characters which cannot otherwise be given (*cf.* the `N`-type variants). The assignment is local.

`\char_set_catcode:nn`

`\char_set_catcode:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

These functions set the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. The first ⟨*integer expression*⟩ is the character code and the second is the category code to apply. The setting applies within the current TeX group. In general, the symbolic functions `\char_set_catcode_`⟨*type*⟩ should be preferred, but there are cases where these lower-level functions may be useful.

`\char_value_catcode:n` ⋆

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

Expands to the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_catcode:n`

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

Displays the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_lccode:nn`

`\char_set_lcode:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

This function set up the behaviour of ⟨*character*⟩ when found inside `\tl_to_lowercase:n`, such that ⟨*character1*⟩ will be converted into ⟨*character2*⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX `‘`⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour
\char_set_lccode:nn { ‘\A } { ‘\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TeX group.

`\char_value_lccode:n` ⋆

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

Expands to the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_lccode:n`

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

Displays the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_uccode:nn`

`\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

This function set up the behaviour of ⟨*character*⟩ when found inside `\tl_to_uppercase:n`, such that ⟨*character1*⟩ will be converted into ⟨*character2*⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX ‘⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_uccode:nn { `\a } { `\A } % Standard behaviour
\char_set_uccode:nn { `\A } { `\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

`\char_value_uccode:n` ⋆

`\char_value_uccode:n {⟨integer expression⟩}`

Expands to the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_uccode:n`

`\char_show_value_uccode:n {⟨integer expression⟩}`

Displays the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_mathcode:nn`

`\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

This function sets up the math code of ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

`\char_value_mathcode:n` ⋆

`\char_value_mathcode:n {⟨integer expression⟩}`

Expands to the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_mathcode:n`

`\char_show_value_mathcode:n {⟨integer expression⟩}`

Displays the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_sfcode:nn`

`\char_set_sfcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

This function sets up the space factor for the ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

`\char_value_sfcode:n` ⋆

`\char_value_sfcode:n {⟨integer expression⟩}`

Expands to the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_sfcode:n`

`\char_show_value_sfcode:n {⟨integer expression⟩}`

Displays the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\l_char_active_seq`

New: 2012-01-23

Used to track which tokens will require special handling at the document level as they are of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single active character. Active tokens should be added to the sequence when they are defined for general document use.

`\l_char_special_seq`

New: 2012-01-23

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories ⟨*letter*⟩ (catcode 11) or ⟨*other*⟩ (catcode 12). Each entry in the sequence consists of a single escaped token, for example `\\` for the backslash or `\{` for an opening brace.Escaped tokens should be added to the sequence when they are defined for general document use.

38 Generic tokens

`\token_new:Nn`

`\token_new:Nn ⟨token1⟩ {⟨token2⟩}`

Defines ⟨*token1*⟩ to globally be a snapshot of ⟨*token2*⟩. This will be an implicit representation of ⟨*token2*⟩.

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

`\c_catcode_letter_token`
`\c_catcode_other_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

`\c_catcode_active_tl`

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

39 Converting tokens

\token_to_meaning:N ⋆

\token_to_meaning:N ⟨*token*⟩

Inserts the current meaning of the ⟨*token*⟩ into the input stream as a series of characters of category code 12 (other). This will be the primitive TeX description of the ⟨*token*⟩, thus for example both functions defined by \cs_set_nopar:Npn and token list variables defined using \tl_new:N will be described as macros.

TeXhackers note: This is the TeX primitive \meaning.

\token_to_str:N ⋆
\token_to_str:c ⋆

\token_to_str:N ⟨*token*⟩

Converts the given ⟨*token*⟩ into a series of characters with category code 12 (other). The current escape character will be the first character in the sequence, although this will also have category code 12 (the escape character is part of the ⟨*token*⟩). This function requires only a single expansion.

TeXhackers note: \token_to_str:N is the TeX primitive \string renamed.

40 Token conditionals

\token_if_group_begin_p:N ⋆
\token_if_group_begin:N*TF* ⋆

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

Tests if ⟨*token*⟩ has the category code of a begin group token ({ when normal TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_group_end_p:N ⋆
\token_if_group_end:N*TF* ⋆

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

Tests if ⟨*token*⟩ has the category code of an end group token (} when normal TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_math_toggle_p:N ⋆
\token_if_math_toggle:N*TF* ⋆

\token_if_math_toggle_p:N ⟨*token*⟩
\token_if_math_toggle:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a math shift token ($ when normal TeX category codes are in force).

\token_if_alignment_p:N ⋆
\token_if_alignment:N*TF* ⋆

\token_if_alignment_p:N ⟨*token*⟩
\token_if_alignment:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an alignment token (& when normal TeX category codes are in force).

\token_if_parameter_p:N ⋆
\token_if_parameter:NTF ⋆

\token_if_parameter_p:N ⟨*token*⟩
\token_if_alignment:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a macro parameter token (# when normal TeX category codes are in force).

\token_if_math_superscript_p:N ⋆
\token_if_math_superscript:NTF ⋆

\token_if_math_superscript_p:N ⟨*token*⟩
\token_if_math_superscript:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a superscript token (^ when normal TeX category codes are in force).

\token_if_math_subscript_p:N ⋆
\token_if_math_subscript:NTF ⋆

\token_if_math_subscript_p:N ⟨*token*⟩
\token_if_math_subscript:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a subscript token (_ when normal TeX category codes are in force).

\token_if_space_p:N ⋆
\token_if_space:NTF ⋆

\token_if_space_p:N ⟨*token*⟩
\token_if_space:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

\token_if_letter_p:N ⋆
\token_if_letter:NTF ⋆

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

Tests if ⟨*token*⟩ has the category code of a letter token.

\token_if_other_p:N ⋆
\token_if_other:NTF ⋆

\token_if_other_p:N ⟨*token*⟩
\token_if_other:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an "other" token.

\token_if_active_p:N ⋆
\token_if_active:NTF ⋆

\token_if_active_p:N ⟨*token*⟩
\token_if_active:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an active character.

\token_if_eq_catcode_p:NN ⋆
\token_if_eq_catcode:NNTF ⋆

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

Tests if the two ⟨*tokens*⟩ have the same category code.

\token_if_eq_charcode_p:NN ⋆
\token_if_eq_charcode:NNTF ⋆

\token_if_eq_charcode_p:NN ⟨*token1*⟩ ⟨*token2*⟩
\token_if_eq_charcode:NNTF ⟨*token1*⟩ ⟨*token2*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same character code.

\token_if_eq_meaning_p:NN ⋆
\token_if_eq_meaning:NNTF ⋆

\token_if_eq_meaning_p:NN ⟨token1⟩ ⟨token2⟩
\token_if_eq_meaning:NNTF ⟨token1⟩ ⟨token2⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the two ⟨*tokens*⟩ have the same meaning when expanded.

\token_if_macro_p:N ⋆
\token_if_macro:NTF ⋆

Updated: 2011-05-23

\token_if_macro_p:N ⟨token⟩
\token_if_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is a TEX macro.

\token_if_cs_p:N ⋆
\token_if_cs:NTF ⋆

\token_if_cs_p:N ⟨token⟩
\token_if_cs:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is a control sequence.

\token_if_expandable_p:N ⋆
\token_if_expandable:NTF ⋆

\token_if_expandable_p:N ⟨token⟩
\token_if_expandable:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is expandable. This test returns ⟨*false*⟩ for an undefined token.

\token_if_long_macro_p:N ⋆
\token_if_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_long_macro_p:N ⟨token⟩
\token_if_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is a long macro.

\token_if_protected_macro_p:N ⋆
\token_if_protected_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_macro_p:N ⟨token⟩
\token_if_protected_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is a protected macro: a macro which is both protected and long will return logical `false`.

\token_if_protected_long_macro_p:N ⋆
\token_if_protected_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_long_macro_p:N ⟨token⟩
\token_if_protected_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token*⟩ is a protected long macro.

\token_if_chardef_p:N ⋆
\token_if_chardef:NTF ⋆

Updated: 2012-01-20

\token_if_chardef_p:N ⟨token⟩
\token_if_chardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

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

TEXhackers note: Booleans, boxes and small integer constants are implemented as chardefs.

`\token_if_mathchardef_p:N` ⋆
`\token_if_mathchardef:NTF` ⋆

Updated: 2012-01-20

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

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

`\token_if_dim_register_p:N` ⋆
`\token_if_dim_register:NTF` ⋆

Updated: 2012-01-20

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

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

`\token_if_int_register_p:N` ⋆
`\token_if_int_register:NTF` ⋆

Updated: 2012-01-20

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

Tests if the ⟨*token*⟩ is defined to be a integer register.

TEXhackers note: Constant integers may be implemented as integer registers, chardefs, or mathchardefs depending on their value.

`\token_if_muskip_register_p:N` ⋆
`\token_if_muskip_register:NTF` ⋆

New: 2012-02-15

`\token_if_muskip_register_p:N` ⟨*token*⟩
`\token_if_muskip_register:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a muskip register.

`\token_if_skip_register_p:N` ⋆
`\token_if_skip_register:NTF` ⋆

Updated: 2012-01-20

`\token_if_skip_register_p:N` ⟨*token*⟩
`\token_if_skip_register:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

`\token_if_toks_register_p:N` ⋆
`\token_if_toks_register:NTF` ⋆

Updated: 2012-01-20

`\token_if_toks_register_p:N` ⟨*token*⟩
`\token_if_toks_register:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a toks register (not used byLATEX3).

`\token_if_primitive_p:N` ⋆
`\token_if_primitive:NTF` ⋆

Updated: 2011-05-23

`\token_if_primitive_p:N` ⟨*token*⟩
`\token_if_primitive:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is an engine primitive.

41 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the "peek" functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

`\peek_after:Nw`

`\peek_after:Nw ⟨function⟩ ⟨token⟩`

Locally sets the test variable `\l_peek_token` equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ will remain in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\peek_gafter:Nw`

`\peek_gafter:Nw ⟨function⟩ ⟨token⟩`

Globally sets the test variable `\g_peek_token` equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ will remain in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\l_peek_token`

Token set by `\peek_after:Nw` and available for testing as described above.

`\g_peek_token`

Token set by `\peek_gafter:Nw` and available for testing as described above.

`\peek_catcode:NTF`

Updated: 2011-07-02

`\peek_catcode:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_catcode_ignore_spaces:NTF`

Updated: 2011-07-02

`\peek_catcode_ignore_spaces:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are ignored by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_catcode_remove:NTF

Updated: 2011-07-02

\peek_catcode_remove:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_catcode_remove_ignore_spaces:NTF

Updated: 2011-07-02

\peek_catcode_remove_ignore_spaces:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are ignored by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode:NTF

Updated: 2011-07-02

\peek_charcode:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_charcode_ignore_spaces:NTF

Updated: 2011-07-02

\peek_charcode_ignore_spaces:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are ignored by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_charcode_remove:NTF

Updated: 2011-07-02

\peek_charcode_remove:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode_remove_ignore_spaces:NTF

Updated: 2011-07-02

\peek_charcode_remove_ignore_spaces:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are ignored by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_meaning:NTF`

Updated: 2011-07-02

`\peek_meaning:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_meaning_ignore_spaces:NTF`

Updated: 2011-07-02

`\peek_meaning_ignore_spaces:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are ignored by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_meaning_remove:NTF`

Updated: 2011-07-02

`\peek_meaning_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_meaning_remove_ignore_spaces:NTF`

Updated: 2011-07-02

`\peek_meaning_remove_ignore_spaces:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are ignored by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

42 Decomposing a macro definition

These functions decompose TeX macros into their constituent parts: if the ⟨*token*⟩ passed is not a macro then no decomposition can occur. In the later case, all three functions leave `\scan_stop:` in the input stream.

`\token_get_arg_spec:N` ⋆

`\token_get_arg_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function will leave the primitive TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1 y #2 }
```

will leave `#1#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` will be left in the input stream

TeXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

`\token_get_replacement_spec:N` ⋆

`\token_get_replacement_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function will leave the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

will leave `x#1 y#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` will be left in the input stream

`\token_get_prefix_spec:N` ⋆

`\token_get_prefix_spec:N` ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function will leave the TeX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

```
\cs_set:Npn \next #1#2 { x #1~y #2 }
```

will leave `\long` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` will be left in the input stream

43 Experimental token functions

`\char_set_active:Npn`
`\char_set_active:Npx`

New: 2011-12-27

`\char_set_active:Npn` ⟨*char*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Makes ⟨*char*⟩ an active character to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed. The ⟨*char*⟩ is made active within the current TeX group level, and the definition is also local.

\char_gset_active:Npn
\char_gset_active:Npx

New: 2011-12-27

\char_gset_active:Npn ⟨*char*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Makes ⟨*char*⟩ an active character to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed. The ⟨*char*⟩ is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the ⟨*char*⟩ is again made active).

\char_set_active_eq:NN

New: 2011-12-27

\char_set_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Makes ⟨*char*⟩ an active character equivalent in meaning to the ⟨*function*⟩ (which may itself be an active character). The ⟨*char*⟩ is made active within the current TeX group level, and the definition is also local.

\char_gset_active_eq:NN

New: 2011-12-27

\char_gset_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Makes ⟨*char*⟩ an active character equivalent in meaning to the ⟨*function*⟩ (which may itself be an active character). The ⟨*char*⟩ is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the ⟨*char*⟩ is again made active).

\peek_N_type:*TF*

New: 2011-08-14

\peek_N_type:TF {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream can be safely grabbed as an N-type argument. The test will be ⟨*false*⟩ if the next ⟨*token*⟩ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), and ⟨*true*⟩ in all other cases. Note that a ⟨*true*⟩ result ensures that the next ⟨*token*⟩ is a valid N-type argument. However, if the next ⟨*token*⟩ is for instance \c_space_token, the test will take the ⟨*false*⟩ branch, even though the next ⟨*token*⟩ is in fact a valid N-type argument. The ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

Part IX
The l3int package
Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* ("`int expr`").

44 Integer expressions

`\int_eval:n` ⋆

`\int_eval:n {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩, expanding any integer and token list variables within the ⟨*expression*⟩ to their content (without requiring `\int_use:N`/`\tl_use:N`) and applying the standard mathematical rules. For example both

```
\int_eval:n { 5 +  4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
 \tl_new:N  \l_my_tl
 \tl_set:Nn \l_my_tl { 5 }
 \int_new:N  \l_my_int
 \int\set:Nn \l_my_int { 4 }
\int_eval:n { \l_my_tl +  \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

both evaluate to -6. The `{`⟨*integer expression*⟩`}` may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. After two expansions, `\int_eval:n` yields a ⟨*integer denotation*⟩ which is left in the input stream. This is *not* an ⟨*internal integer*⟩, and therefore requires suitable termination if used in a TeX-style integer assignment.

`\int_abs:n` ⋆

`\int_abs:n {⟨integer expression⟩}`

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

`\int_div_round:nn` ⋆

`\int_div_round:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then calculates the result of dividing the first value by the second, rounding any remainder. Ties are rounded away from zero. Note that this is identical to using `/` directly in an ⟨*integer expression*⟩. The result is left in the input stream as a ⟨*integer denotation*⟩ after two expansions.

`\int_div_truncate:nn` ⋆

Updated: 2012-02-09

`\int_div_truncate:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

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

`\int_max:nn` ⋆
`\int_min:nn` ⋆

`\int_max:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`
`\int_min:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

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

`\int_mod:nn` ⋆

`\int_mod:nn {`⟨*intexpr*$_1$⟩`} {`⟨*intexpr*$_2$⟩`}`

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

45 Creating and initialising integers

`\int_new:N`
`\int_new:c`

`\int_new:N` ⟨*integer*⟩

Creates a new ⟨*integer*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*integer*⟩ will initially be equal to 0.

`\int_const:Nn`
`\int_const:cn`

Updated: 2011-10-22

`\int_const:Nn` ⟨*integer*⟩ `{`⟨*integer expression*⟩`}`

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ will be set globally to the ⟨*integer expression*⟩.

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

`\int_zero:N` ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0.

`\int_zero_new:N`
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

New: 2011-12-13

`\int_zero_new:N` ⟨*integer*⟩

Ensures that the ⟨*integer*⟩ exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the ⟨*integer*⟩ set to zero.

`\int_set_eq:NN`
`\int_set_eq:(cN|Nc|cc)`
`\int_gset_eq:NN`
`\int_gset_eq:(cN|Nc|cc)`

`\int_set_eq:NN` ⟨*integer1*⟩ ⟨*integer2*⟩

Sets the content of ⟨*integer1*⟩ equal to that of ⟨*integer2*⟩.

\int_if_exist_p:N ⋆
\int_if_exist_p:c ⋆
\int_if_exist:NTF ⋆
\int_if_exist:cTF ⋆

New: 2012-03-03

\int_if_exist_p:N ⟨int⟩
\int_if_exist:NTF ⟨int⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨*int*⟩ is currently defined. This does not check that the ⟨*int*⟩ really is an integer variable.

46 Setting and incrementing integers

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

Updated: 2011-10-22

\int_add:Nn ⟨integer⟩ {⟨integer expression⟩}

Adds the result of the ⟨*integer expression*⟩ to the current content of the ⟨*integer*⟩.

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

\int_decr:N ⟨integer⟩

Decreases the value stored in ⟨*integer*⟩ by 1.

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c

\int_incr:N ⟨integer⟩

Increases the value stored in ⟨*integer*⟩ by 1.

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

Updated: 2011-10-22

\int_set:Nn ⟨integer⟩ {⟨integer expression⟩}

Sets ⟨*integer*⟩ to the value of ⟨*integer expression*⟩, which must evaluate to an integer (as described for `\int_eval:n`).

\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Updated: 2011-10-22

\int_sub:Nn ⟨integer⟩ {⟨integer expression⟩}

Subtracts the result of the ⟨*integer expression*⟩ to the current content of the ⟨*integer*⟩.

47 Using integers

\int_use:N ⋆
\int_use:c ⋆

Updated: 2011-10-22

`\int_use:N ⟨integer⟩`

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

TeXhackers note: `\int_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

48 Integer expression conditionals

\int_compare_p:nNn ⋆
\int_compare:nNn*TF* ⋆

```
\int_compare_p:nNn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩}
\int_compare:nNnTF
  {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*integer expressions*⟩ as described for `\int_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

\int_compare_p:n ⋆
\int_compare:n*TF* ⋆

```
\int_compare_p:n { ⟨intexpr1⟩ ⟨relation⟩ ⟨intexpr2⟩ }
\int_compare:nTF
  { ⟨intexpr1⟩ ⟨relation⟩ ⟨intexpr2⟩ }
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*integer expressions*⟩ as described for `\int_eval:n`. The two results are then compared using the ⟨*relation*⟩:

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

\int_if_even_p:n ⋆
\int_if_even:n*TF* ⋆
\int_if_odd_p:n ⋆
\int_if_odd:n*TF* ⋆

```
\int_if_odd_p:n {⟨integer expression⟩}
\int_if_odd:nTF {⟨integer expression⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*integer expression*⟩ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

49 Integer expression loops

\int_do_while:nNnn ☆

\int_do_while:nNnn
{⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is false.

\int_do_until:nNnn ☆

\int_do_until:nNnn
{⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is true.

\int_until_do:nNnn ☆

\int_until_do:nNnn
{⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nNnTF. If the test is false then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is true.

\int_while_do:nNnn ☆

\int_while_do:nNnn {⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nNnTF. If the test is true then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is false.

\int_do_while:nn ☆

\int_do_while:nn
{ ⟨*intexpr1*⟩ ⟨*relation*⟩ ⟨*intexpr2*⟩ } {⟨*code*⟩}

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is false.

\int_do_until:nn ☆

\int_do_until:nn
{ ⟨*intexpr1*⟩ ⟨*relation*⟩ ⟨*intexpr2*⟩ } {⟨*code*⟩}

Evaluates the relationship between the two ⟨*integer expressions*⟩ as described for \int_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is true.

`\int_until_do:nn` ☆

```
\int_until_do:nn
  { ⟨intexpr1⟩ ⟨relation⟩ ⟨intexpr2⟩ } {⟨code⟩}
```

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

`\int_while_do:nn` ☆

```
\int_while_do:nn          { ⟨intexpr1⟩ ⟨relation⟩ ⟨intexpr2⟩ } {⟨code⟩}
```

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

50 Formatting integers

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

`\int_to_arabic:n` ★

Updated: 2011-10-22

```
\int_to_arabic:n {⟨integer expression⟩}
```

Places the value of the ⟨*integer expression*⟩ in the input stream as digits, with category code 12 (other).

`\int_to_alph:n` ★
`\int_to_Alph:n` ★

Updated: 2011-09-17

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

Evaluates the ⟨*integer expression*⟩ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places `a` in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as `z` and

```
\int_to_alph:n { 27 }
```

is converted to `aa`. For conversions using other alphabets, use `\int_convert_to_-symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified.

`\int_to_symbols:nnn` ⋆

Updated: 2011-09-17

```
\int_to_symbols:nnn
  {⟨integer expression⟩} {⟨total symbols⟩}
  ⟨value to symbol mapping⟩
```

This is the low-level function for conversion of an ⟨*integer expression*⟩ into a symbolic form (which will often be letters). The ⟨*total symbols*⟩ available should be given as an integer expression. Values are actually converted to symbols according to the ⟨*value to symbol mapping*⟩. This should be given as ⟨*total symbols*⟩ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_convert_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        ...
        { 26 } { z }
      }
  }
```

`\int_to_binary:n` ⋆

Updated: 2011-09-17

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

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

`\int_to_hexadecimal:n` ⋆

Updated: 2011-09-17

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

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

`\int_to_octal:n` ⋆

Updated: 2011-09-17

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

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

`\int_to_base:nn` ⋆

Updated: 2011-09-17

`\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}`

Calculates the value of the ⟨*integer expression*⟩ and converts it into the appropriate representation in the ⟨*base*⟩; the later may be given as an integer expression. For bases greater than 10 the higher "digits" are represented by the upper case letters from the English alphabet. The maximum ⟨*base*⟩ value is 36.

TEXhackers note: This is a generic version of `\int_to_binary:n`, *etc.*

\int_to_roman:n ☆
\int_to_Roman:n ☆

Updated: 2011-10-22

\int_to_roman:n {⟨*integer expression*⟩}

Places the value of the ⟨*integer expression*⟩ in the input stream as Roman numerals, either lower case (\int_to_roman:n) or upper case (\int_to_Roman:n). The Roman numerals are letters with category code 11 (letter).

51 Converting from other formats to integers

\int_from_alph:n ⋆

\int_from_alpa:n {⟨*letters*⟩}

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

\int_from_binary:n ⋆

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

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream.

\int_from_hexadecimal:n ⋆

\int_from_binary:n {⟨*hexadecimal number*⟩}

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

\int_from_octal:n ⋆

\int_from_octal:n {⟨*octal number*⟩}

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream.

\int_from_roman:n ⋆

\int_from_roman:n {⟨*roman numeral*⟩}

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

\int_from_base:nn ⋆

\int_from_base:nn {⟨*number*⟩} {⟨*base*⟩}

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

52 Viewing integers

\int_show:N
\int_show:c

\int_show:N ⟨*integer*⟩

Displays the value of the ⟨*integer*⟩ on the terminal.

\int_show:n

New: 2011-11-22

\int_show:n ⟨*integer expression*⟩

Displays the result of evaluating the ⟨*integer expression*⟩ on the terminal.

53 Constant integers

\c_minus_one
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand

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

\c_max_int

The maximum value that can be stored as an integer.

\c_max_register_int

Maximum number of registers.

54 Scratch integers

\l_tmpa_int
\l_tmpb_int
\l_tmpc_int

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

\g_tmpa_int
\g_tmpb_int

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

55 Internal functions

\int_get_digits:n ★

`\int_get_digits:n ⟨value⟩`

Parses the ⟨*value*⟩ to leave the absolute ⟨*value*⟩ in the input stream. This may therefore be used to remove multiple sign tokens from the ⟨*value*⟩ (which may be symbolic).

\int_get_sign:n ☆

`\int_get_sign:n ⟨value⟩`

Parses the ⟨*value*⟩ to leave a single sign token (either `+` or `-`) in the input stream. This may therefore be used to sanitise sign tokens from the ⟨*value*⟩ (which may be symbolic).

\int_to_letter:n ★

Updated: 2011-09-17

`\int_to_letter:n ⟨integer value⟩`

For ⟨*integer values*⟩ from 0 to 9, leaves the ⟨*value*⟩ in the input stream unchanged. For ⟨*integer values*⟩ from 10 to 35, leaves the appropriate upper case letter (from the standard English alphabet) in the input stream: for example, 10 is converted to `A`, 11 to `B`, *etc.*

\int_to_roman:w ★

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

Converts ⟨*integer*⟩ to it lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *are* expanded by this process. Negative ⟨*integer*⟩ values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

TeXhackers note: This is the TeX primitive `\romannumeral` renamed.

`\if_num:w` ★
`\if_int_compare:w` ★

```
\if_num:w ⟨integer1⟩ ⟨relation⟩ ⟨integer2⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Compare two integers using ⟨*relation*⟩, which must be one of `=`, `<` or `>` with category code 12. The `\else:` branch is optional.

> **TEXhackers note:** These are both names for the TEX primitive `\ifnum`.

`\if_case:w` ★
`\or:` ★

```
\if_case:w ⟨integer⟩ ⟨case0⟩
  \or: ⟨case1⟩
  \or: ...
  \else: ⟨default⟩
\fi:
```

Selects a case to execute based on the value of the ⟨*integer*⟩. The first case (⟨*case0*⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨*case1*⟩) if the ⟨*integer*⟩ is 1, *etc.* The ⟨*integer*⟩ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

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

`\int_value:w` ★

```
\int_value:w ⟨integer⟩
\int_value:w ⟨tokens⟩ ⟨optional space⟩
```

Expands ⟨*tokens*⟩ until an ⟨*integer*⟩ is formed. One space may be gobbled in the process.

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

`\int_eval:w` ★
`\int_eval_end:` ★

```
\int_eval:w ⟨intexpr⟩ \int_eval_end:
```

Evaluates ⟨*integer expression*⟩ as described for `\int_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when `\int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

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

`\if_int_odd:w` ★

```
\if_int_odd:w ⟨tokens⟩ ⟨optional space⟩
  ⟨true code⟩
\else:
  ⟨true code⟩
\fi:
```

Expands ⟨*tokens*⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨*integer*⟩ is odd. If so, ⟨*true code*⟩ is executed. The `\else:` branch is optional.

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

Part X

The l3skip package Dimensions and skips

LaTeX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

56 Creating and initialising `dim` variables

`\dim_new:N`
`\dim_new:c`

`\dim_new:N` ⟨*dimension*⟩

Creates a new ⟨*dimension*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*dimension*⟩ will initially be equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`
New: 2012-03-05

`\dim_const:Nn` ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ will be set globally to the ⟨*dimension expression*⟩.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N` ⟨*dimension*⟩

Sets ⟨*dimension*⟩ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`
New: 2012-01-07

`\dim_zero_new:N` ⟨*dimension*⟩

Ensures that the ⟨*dimension*⟩ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the ⟨*dimension*⟩ set to zero.

`\dim_if_exist_p:N` ⋆
`\dim_if_exist_p:c` ⋆
`\dim_if_exist:NTF` ⋆
`\dim_if_exist:cTF` ⋆
New: 2012-03-03

`\dim_if_exist_p:N` ⟨*dimension*⟩
`\dim_if_exist:NTF` ⟨*dimension*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

57 Setting dim variables

\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn

Updated: 2011-10-22

\dim_add:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Adds the result of the ⟨*dimension expression*⟩ to the current content of the ⟨*dimension*⟩.

\dim_set:Nn
\dim_set:cn
\dim_gset:Nn
\dim_gset:cn

Updated: 2011-10-22

\dim_set:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Sets ⟨*dimension*⟩ to the value of ⟨*dimension expression*⟩, which must evaluate to a length with units.

\dim_set_eq:NN
\dim_set_eq:(cN|Nc|cc)
\dim_gset_eq:NN
\dim_gset_eq:(cN|Nc|cc)

\dim_set_eq:NN ⟨*dimension1*⟩ ⟨*dimension2*⟩

Sets the content of ⟨*dimension1*⟩ equal to that of ⟨*dimension2*⟩.

\dim_set_max:Nn
\dim_set_max:cn
\dim_gset_max:Nn
\dim_gset_max:cn

Updated: 2012-02-06

\dim_set_max:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Compares the current value of the ⟨*dimension*⟩ with that of the ⟨*dimension expression*⟩, and sets the ⟨*dimension*⟩ to the larger of these two value.

\dim_set_min:Nn
\dim_set_min:cn
\dim_gset_min:Nn
\dim_gset_min:cn

Updated: 2012-02-06

\dim_set_min:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Compares the current value of the ⟨*dimension*⟩ with that of the ⟨*dimension expression*⟩, and sets the ⟨*dimension*⟩ to the smaller of these two value.

\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

Updated: 2011-10-22

\dim_sub:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Subtracts the result of the ⟨*dimension expression*⟩ to the current content of the ⟨*dimension*⟩.

58 Utilities for dimension calculations

\dim_abs:n ★

Updated: 2011-10-22

\dim_abs:n {⟨*dimexpr*⟩}

Converts the ⟨*dimexpr*⟩ to its absolute value, leaving the result in the input stream as an ⟨*dimension denotation*⟩.

`\dim_ratio:nn` ⋆

Updated: 2011-10-22

`\dim_ratio:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}`

Parses the two ⟨*dimension expressions*⟩ and converts the ratio of the two to a form suitable for use inside a ⟨*dimension expression*⟩. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
  { 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ration expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

will display `327680/655360` on the terminal.

59 Dimension expression conditionals

`\dim_compare_p:nNn` ⋆
`\dim_compare:nNnTF` ⋆

```
\dim_compare_p:nNn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}
\dim_compare:nNnTF
  {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*dimension expressions*⟩ as described for `\dim_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

`\dim_compare_p:n` ⋆
`\dim_compare:nTF` ⋆

```
\dim_compare_p:n { ⟨dimexpr1⟩ ⟨relation⟩ ⟨dimexpr2⟩ }
\dim_compare:nTF
  { ⟨dimexpr1⟩ ⟨relation⟩ ⟨dimexpr2⟩ }
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*dimension expressions*⟩ as described for `\dim_eval:n`. The two results are then compared using the ⟨*relation*⟩:

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

60 Dimension expression loops

`\dim_do_while:nNnn` ☆

`\dim_do_while:nNnn {`⟨*dimexpr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dimexpr*$_2$⟩`} {`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is `false`.

`\dim_do_until:nNnn` ☆

`\dim_do_until:nNnn {`⟨*dimexpr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dimexpr*$_2$⟩`} {`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is `true`.

`\dim_until_do:nNnn` ☆

`\dim_until_do:nNnn {`⟨*dimexpr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dimexpr*$_2$⟩`} {`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `true`.

`\dim_while_do:nNnn` ☆

`\dim_while_do:nNnn {`⟨*dimexpr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dimexpr*$_2$⟩`} {`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `false`.

`\dim_do_while:nn` ☆

`\dim_do_while:nn {` ⟨*dimexpr1*⟩ ⟨*relation*⟩ ⟨*dimexpr2*⟩ `} {`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is `false`.

`\dim_do_until:nn` ☆

`\dim_do_until:nn {` ⟨*dimexpr1*⟩ ⟨*relation*⟩ ⟨*dimexpr2*⟩ `} {`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test will be repeated, and a loop will occur until the test is `true`.

`\dim_until_do:nn` ☆

`\dim_until_do:nn {` ⟨*dimexpr1*⟩ ⟨*relation*⟩ ⟨*dimexpr2*⟩ `} {`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nTF`. If the test is `false` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `true`.

\dim_while_do:nn ☆

\dim_while_do:nn { ⟨*dimexpr1*⟩ ⟨*relation*⟩ ⟨*dimexpr2*⟩ } {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nTF`. If the test is `true` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `false`.

61 Using dim expressions and variables

\dim_eval:n ★

Updated: 2011-10-22

\dim_eval:n {⟨*dimension expression*⟩}

Evaluates the ⟨*dimension expression*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*dimension denotation*⟩ after two expansions. This will be expressed in points (`pt`), and will require suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal dimension*⟩.

\dim_use:N ★
\dim_use:c ★

\dim_use:N ⟨*dimension*⟩

Recovers the content of a ⟨*dimension*⟩ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

62 Viewing dim variables

\dim_show:N
\dim_show:c

\dim_show:N ⟨*dimension*⟩

Displays the value of the ⟨*dimension*⟩ on the terminal.

\dim_show:n

New: 2011-11-22

\dim_show:n ⟨*dimension expression*⟩

Displays the result of evaluating the ⟨*dimension expression*⟩ on the terminal.

63 Constant dimensions

\c_max_dim

The maximum value that can be stored as a dimension or skip (these are equivalent).

\c_zero_dim

A zero length as a dimension or a skip (these are equivalent).

64 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
\l_tmpc_dim

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

\g_tmpa_dim
\g_tmpb_dim

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

65 Creating and initialising skip variables

\skip_new:N
\skip_new:c

`\skip_new:N ⟨skip⟩`

Creates a new ⟨*skip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*skip*⟩ will initially be equal to 0 pt.

\skip_const:Nn
\skip_const:cn

New: 2012-03-05

`\skip_const:Nn ⟨skip⟩ {⟨skip expression⟩}`

Creates a new constant ⟨*skip*⟩ or raises an error if the name is already taken. The value of the ⟨*skip*⟩ will be set globally to the ⟨*skip expression*⟩.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

`\skip_zero:N ⟨skip⟩`

Sets ⟨*skip*⟩ to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

New: 2012-01-07

`\skip_zero_new:N ⟨skip⟩`

Ensures that the ⟨*skip*⟩ exists globally by applying `\skip_new:N` if necessary, then applies `\skip_(g)zero:N` to leave the ⟨*skip*⟩ set to zero.

\skip_if_exist_p:N ⋆
\skip_if_exist_p:c ⋆
\skip_if_exist:NTF ⋆
\skip_if_exist:cTF ⋆

New: 2012-03-03

`\skip_if_exist_p:N ⟨skip⟩`
`\skip_if_exist:NTF ⟨skip⟩ {⟨true code⟩} {⟨false code⟩}`

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

66 Setting skip variables

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn

Updated: 2011-10-22

`\skip_add:Nn ⟨skip⟩ {⟨skip expression⟩}`

Adds the result of the ⟨*skip expression*⟩ to the current content of the ⟨*skip*⟩.

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Updated: 2011-10-22

`\skip_set:Nn ⟨skip⟩ {⟨skip expression⟩}`

Sets ⟨*skip*⟩ to the value of ⟨*skip expression*⟩, which must evaluate to a length with units and may include a rubber component (for example `1 cm plus 0.5 cm`.

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

`\skip_set_eq:NN ⟨skip1⟩ ⟨skip2⟩`

Sets the content of ⟨*skip1*⟩ equal to that of ⟨*skip2*⟩.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Updated: 2011-10-22

`\skip_sub:Nn ⟨skip⟩ {⟨skip expression⟩}`

Subtracts the result of the ⟨*skip expression*⟩ to the current content of the ⟨*skip*⟩.

67 Skip expression conditionals

\skip_if_eq_p:nn ⋆
\skip_if_eq:nnTF ⋆

```
\skip_if_eq_p:nn {⟨skipexpr1⟩} {⟨skipexpr2⟩}
\dim_compare:nTF
  {⟨skip expr1⟩} {⟨skip expr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*skip expressions*⟩ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

\skip_if_infinite_glue_p:n ⋆
\skip_if_infinite_glue:nTF ⋆

Updated: 2012-03-05

```
\skip_if_infinite_glue_p:n {⟨skipexpr⟩}
\skip_if_infinite_glue:nTF {⟨skipexpr⟩} {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*skip expression*⟩ as described for `\skip_eval:n`, and then tests if this contains an infinite stretch or shrink component (or both).

\skip_if_finite_p:n ★
\skip_if_finite:nTF ★

New: 2012-03-05

\skip_if_finite_p:n {⟨*skipexpr*⟩}
\skip_if_finite:nTF {⟨*skipexpr*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Evaluates the ⟨*skip expression*⟩ as described for \skip_eval:n, and then tests if all of its components are finite.

68 Using skip expressions and variables

\skip_eval:n ★

Updated: 2011-10-22

\skip_eval:n {⟨*skip expression*⟩}

Evaluates the ⟨*skip expression*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring \skip_use:N/\tl_use:N) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*glue denotation*⟩ after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal glue*⟩.

\skip_use:N ★
\skip_use:c ★

\skip_use:N ⟨*skip*⟩

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of \skip_eval:n).

TeXhackers note: \skip_use:N is the TeX primitive \the: this is one of several LaTeX3 names for this primitive.

69 Viewing skip variables

\skip_show:N
\skip_show:c

\skip_show:N ⟨*skip*⟩

Displays the value of the ⟨*skip*⟩ on the terminal.

\skip_show:n

New: 2011-11-22

\skip_show:n ⟨*skip expression*⟩

Displays the result of evaluating the ⟨*skip expression*⟩ on the terminal.

70 Constant skips

\c_max_skip

The maximum value that can be stored as a dimension or skip (these are equivalent).

\c_zero_skip

A zero length as a dimension or a skip (these are equivalent).

71 Scratch skips

`\l_tmpa_skip`
`\l_tmpb_skip`
`\l_tmpc_skip`

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

`\g_tmpa_skip`
`\g_tmpb_skip`

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

72 Creating and initialising muskip variables

`\muskip_new:N`
`\muskip_new:c`

`\muskip_new:N ⟨muskip⟩`

Creates a new ⟨*muskip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*muskip*⟩ will initially be equal to 0 mu.

`\muskip_const:Nn`
`\muskip_const:cn`

New: 2012-03-05

`\muskip_const:Nn ⟨muskip⟩ {⟨muskip expression⟩}`

Creates a new constant ⟨*muskip*⟩ or raises an error if the name is already taken. The value of the ⟨*muskip*⟩ will be set globally to the ⟨*muskip expression*⟩.

`\muskip_zero:N`
`\muskip_zero:c`
`\muskip_gzero:N`
`\muskip_gzero:c`

`\skip_zero:N ⟨muskip⟩`

Sets ⟨*muskip*⟩ to 0 mu.

`\muskip_zero_new:N`
`\muskip_zero_new:c`
`\muskip_gzero_new:N`
`\muskip_gzero_new:c`

New: 2012-01-07

`\muskip_zero_new:N ⟨muskip⟩`

Ensures that the ⟨*muskip*⟩ exists globally by applying **`\muskip_new:N`** if necessary, then applies **`\muskip_(g)zero:N`** to leave the ⟨*muskip*⟩ set to zero.

`\muskip_if_exist_p:N` ⋆
`\muskip_if_exist_p:c` ⋆
`\muskip_if_exist:NTF` ⋆
`\muskip_if_exist:cTF` ⋆

New: 2012-03-03

`\muskip_if_exist_p:N ⟨muskip⟩`
`\muskip_if_exist:NTF ⟨muskip⟩ {⟨true code⟩} {⟨false code⟩}`

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

73 Setting muskip variables

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn

Updated: 2011-10-22

\muskip_add:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Adds the result of the ⟨*muskip expression*⟩ to the current content of the ⟨*muskip*⟩.

\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn

Updated: 2011-10-22

\muskip_set:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Sets ⟨*muskip*⟩ to the value of ⟨*muskip expression*⟩, which must evaluate to a math length with units and may include a rubber component (for example `1 mu plus 0.5 mu`.

\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)

\muskip_set_eq:NN ⟨*muskip1*⟩ ⟨*muskip2*⟩

Sets the content of ⟨*muskip1*⟩ equal to that of ⟨*muskip2*⟩.

\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Updated: 2011-10-22

\muskip_sub:Nn ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Subtracts the result of the ⟨*muskip expression*⟩ to the current content of the ⟨*skip*⟩.

74 Using muskip expressions and variables

\muskip_eval:n ⋆

Updated: 2011-10-22

\muskip_eval:n {⟨*muskip expression*⟩}

Evaluates the ⟨*muskip expression*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\muskip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*muglue denotation*⟩ after two expansions. This will be expressed in `mu`, and will require suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal muglue*⟩.

\muskip_use:N ⋆
\muskip_use:c ⋆

\muskip_use:N ⟨*muskip*⟩

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\muskip_eval:n`).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

75 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:(c|n)

Updated: 2011-10-22

`\skip_horizontal:N` ⟨*skip*⟩
`\skip_horizontal:n` {⟨*skipexpr*⟩}

Inserts a horizontal ⟨*skip*⟩ into the current list.

> **TEXhackers note:** `\skip_horizontal:N` is the TEX primitive `\hskip` renamed.

\skip_vertical:N
\skip_vertical:(c|n)

Updated: 2011-10-22

`\skip_vertical:N` ⟨*skip*⟩
`\skip_vertical:n` {⟨*skipexpr*⟩}

Inserts a vertical ⟨*skip*⟩ into the current list.

> **TEXhackers note:** `\skip_vertical:N` is the TEX primitive `\vskip` renamed.

76 Viewing muskip variables

\muskip_show:N
\muskip_show:c

`\muskip_show:N` ⟨*muskip*⟩

Displays the value of the ⟨*muskip*⟩ on the terminal.

\muskip_show:n

New: 2011-11-22

`\muskip_show:n` ⟨*muskip expression*⟩

Displays the result of evaluating the ⟨*muskip expression*⟩ on the terminal.

77 Internal functions

\if_dim:w

```
\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen1⟩
  ⟨true code⟩
\else:
  ⟨false⟩
\fi:
```

Compare two dimensions. The ⟨*relation*⟩ is one of `<`, `=` or `>` with category code 12.

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

\dim_eval:w ★
\dim_eval_end: ★

`\dim_eval:w` ⟨*dimexpr*⟩ `\dim_eval_end:`

Evaluates ⟨*dimension expression*⟩ as described for `\dim_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when `\dim_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\dim_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

> **TEXhackers note:** This is the ε-TEX primitive `\dimexpr`.

78 Experimental skip functions

`\skip_split_finite_else_action:nnNN`

Updated: 2011-10-22

`\skip_split_finite_else_action:nnNN {⟨skipexpr⟩} {⟨action⟩}`
`⟨dimen1⟩ ⟨dimen2⟩`

Checks if the ⟨*skipexpr*⟩ 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 0 pt and place `#2` into the input stream: this is usually an error or warning message of some sort.

79 Internal functions

`\dim_strip_bp:n` ★
`\dim_strip_pt:n` ★

New: 2011-11-11

`\dim_strip_bp:n {⟨dimension expression⟩}`
`\dim_strip_pt:n {⟨dimension expression⟩}`

Evaluates the ⟨*dimension expression*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The magnitude of the result, expressed in big points (`bp`) or points (`pt`), will be left in the input stream with *no units*. If the decimal part of the magnitude is zero, this will be omitted.

If the `{`⟨*dimension expression*⟩`}` contains additional units, these will be ignored, so for example

```
\dim_strip_pt:n { 1 bp pt }
```

will leave `1.00374` in the input stream (*i.e.* the magnitude of one “big point” when converted to points).

Part XI
The l3tl package
Token lists

TEX works with tokens, and LATEX3 therefore provides a number of functions to deal with token lists. Token lists may be present directly in the argument to a function:

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

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

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

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

A token list can be seen either as a list of "items", or a list of "tokens". An item is whatever `\use_none:n` grabs as its argument: either a single token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or , `{`, or `}` (assuming normal TEX category codes). Thus for example

```
{ Hello } ~ world
```

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

80 Creating and initialising token list variables

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

`\tl_new:N` ⟨*tl var*⟩

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

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

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

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

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

`\tl_clear:N` ⟨*tl var*⟩

Clears all entries from the ⟨*tl var*⟩ within the scope of the current TEX group.

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

\tl_clear_new:N ⟨*tl var*⟩

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

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

\tl_set_eq:NN ⟨*tl var1*⟩ ⟨*tl var2*⟩

Sets the content of ⟨*tl var1*⟩ equal to that of ⟨*tl var2*⟩.

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

New: 2012-03-03

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

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

81 Adding data to token list variables

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

\tl_set:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

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

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

\tl_put_left:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

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

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

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

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

82 Modifying token list variables

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

Updated: 2011-08-11

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

Replaces the first (leftmost) occurrence of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain {, } or # (assuming normal TeX category codes).

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

Updated: 2011-08-11

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

Replaces all occurrences of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain {, } or # (assuming normal TeX category codes). As this function operates from left to right, the pattern ⟨*old tokens*⟩ may remain after the replacement (see \tl_remove_all:Nn for an example). The assignment is restricted to the current TeX group.

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

Updated: 2011-08-11

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

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain {, } or # (assuming normal TeX category codes).

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

Updated: 2011-08-11

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

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain {, } or # (assuming normal TeX category codes). As this function operates from left to right, the pattern ⟨*tokens*⟩ may remain after the removal, for instance,

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

will result in \l_tmpa_tl containing abcd.

83 Reassigning token list category codes

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

Updated: 2011-12-18

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

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, applying the category code régime specified in the ⟨*setup*⟩ before carrying out the assignment. This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. See also \tl_rescan:nn.

\tl_rescan:nn

Updated: 2011-12-18

\tl_rescan:nn {⟨*setup*⟩} {⟨*tokens*⟩}

Rescans ⟨*tokens*⟩ applying the category code régime specified in the ⟨*setup*⟩, and leaves the resulting tokens in the input stream. See also \tl_set_rescan:Nnn.

84 Reassigning token list character codes

\tl_to_lowercase:n

\tl_to_lowercase:n {⟨*tokens*⟩}

Works through all of the ⟨*tokens*⟩, replacing each character with the lower case equivalent as defined by \char_set_lccode:nn. Characters with no defined lower case character code are left unchanged. This process does not alter the category code assigned to the ⟨*tokens*⟩.

TeXhackers note: This is the TeX primitive \lowercase renamed. As a result, this function takes place on execution and not on expansion.

\tl_to_uppercase:n

\tl_to_uppercase:n {⟨*tokens*⟩}

Works through all of the ⟨*tokens*⟩, replacing each character with the upper case equivalent as defined by \char_set_uccode:nn. Characters with no defined lower case character code are left unchanged. This process does not alter the category code assigned to the ⟨*tokens*⟩.

TeXhackers note: This is the TeX primitive \uppercase renamed. As a result, this function takes place on execution and not on expansion.

85 Token list conditionals

\tl_if_blank_p:n ★
\tl_if_blank_p:(V|o) ★
\tl_if_blank:n*TF* ★
\tl_if_blank:(V|o)*TF* ★

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

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

\tl_if_empty_p:N ★
\tl_if_empty_p:c ★
\tl_if_empty:N*TF* ★
\tl_if_empty:c*TF* ★

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

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

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

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

Tests if the ⟨*token list*⟩ is entirely empty (*i.e.* contains no tokens at all). All versions of these functions are fully expandable (including those involving an x-type expansion).

\tl_if_eq_p:NN ⋆
\tl_if_eq_p:(Nc|cN|cc) ⋆
\tl_if_eq:NN*TF* ⋆
\tl_if_eq:(Nc|cN|cc)*TF* ⋆

`\tl_if_eq_p:NN {⟨tl var1⟩} {⟨tl var2⟩}`
`\tl_if_eq:NNTF {⟨tl var1⟩} {⟨tl var2⟩} {⟨true code⟩} {⟨false code⟩}`

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

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq_p:NN \l_tmpa_tl \l_tmpb_tl
```

is logically `false`.

\tl_if_eq:nn*TF*

`\tl_if_eq:nnTF ⟨token list1⟩ {⟨token list2⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if ⟨*token list1*⟩ and ⟨*token list2*⟩ are equal, both in respect of character codes and category codes.

\tl_if_in:Nn*TF*
\tl_if_in:cn*TF*

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

Tests if the ⟨*token list*⟩ is found in the content of the ⟨*token list variable*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (assuming the usual TeX category codes apply).

\tl_if_in:nn*TF*
\tl_if_in:(Vn|on|no)*TF*

`\tl_if_in:nnTF {⟨token list1⟩} {⟨token list2⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if ⟨*token list2*⟩ is found inside ⟨*token list1*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (assuming the usual TeX category codes apply).

\tl_if_single_p:N ⋆
\tl_if_single_p:c ⋆
\tl_if_single:N*TF* ⋆
\tl_if_single:c*TF* ⋆

Updated: 2011-08-13

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

Tests if the content of the ⟨*tl var*⟩ consists of a single item, *i.e.* is either a single normal token (excluding spaces, and brace tokens) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has length 1 according to `\tl_length:N`.

\tl_if_single_p:n ⋆
\tl_if_single:n*TF* ⋆

Updated: 2011-08-13

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

Tests if the token list has exactly one item, *i.e.* is either a single normal token or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has length 1 according to `\tl_length:n`.

\tl_if_single_token_p:n ⋆
\tl_if_single_token:n*TF* ⋆

New: 2011-08-11

`\tl_if_single_token_p:n {⟨token list⟩}`
`\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single "normal" token. Token groups (`{...}`) are not single tokens.

86 Mapping to token lists

`\tl_map_function:NN` ☆
`\tl_map_function:cN` ☆

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

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

`\tl_map_function:nN` ☆

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

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

`\tl_map_inline:Nn`
`\tl_map_inline:cn`

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

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

`\tl_map_inline:nn`

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

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

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

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

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

`\tl_map_variable:nNn`

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

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

`\tl_map_break:` ☆

`\tl_map_break:`

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

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

Use outside of a `\tl_map_...` scenario will lead low level TEX errors.

87 Using token lists

`\tl_to_str:N` ★
`\tl_to_str:c` ★

`\tl_to_str:N` ⟨*tl var*⟩

Converts the content of the ⟨*tl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream.

`\tl_to_str:n` ★

`\tl_to_str:n {`⟨*tokens*⟩`}`

Converts the given ⟨*tokens*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream. Note that this function requires only a single expansion.

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

`\tl_use:N` ★
`\tl_use:c` ★

`\tl_use:N` ⟨*tl var*⟩

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

88 Working with the content of token lists

`\tl_length:n` ★
`\tl_length:(V|o)` ★

Updated: 2011-08-13

`\tl_length:n {`⟨*tokens*⟩`}`

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

\tl_length:N ⋆
\tl_length:c ⋆

Updated: 2011-08-13

\tl_length:N {⟨*tl var*⟩}

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

\tl_reverse:n ⋆
\tl_reverse:(V|o) ⋆

Updated: 2012-01-08

\tl_reverse:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨*item1*⟩⟨*item2*⟩⟨*item3*⟩ ... ⟨$item_n$⟩ becomes ⟨$item_n$⟩... ⟨*item3*⟩⟨*item2*⟩⟨*item1*⟩. This process will preserve unprotected space within the ⟨*token list*⟩. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \tl_reverse_items:n. See also \tl_reverse:N.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

Updated: 2012-01-08

\tl_reverse:N {⟨*tl var*⟩}

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that ⟨*item1*⟩⟨*item2*⟩⟨*item3*⟩ ... ⟨$item_n$⟩ becomes ⟨$item_n$⟩... ⟨*item3*⟩⟨*item2*⟩⟨*item1*⟩. This process will preserve unprotected spaces within the ⟨*token list variable*⟩. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also \tl_reverse:n.

\tl_reverse_items:n ⋆

New: 2012-01-08

\tl_reverse_items:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that {⟨$item_1$⟩}{⟨$item_2$⟩}{⟨$item_3$⟩} ... {⟨$item_n$⟩} becomes {⟨$item_n$⟩} ... {⟨$item_3$⟩}{⟨$item_2$⟩}{⟨$item_1$⟩}. This process will remove any unprotected space within the ⟨*token list*⟩. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider \tl_reverse:n or \tl_reverse_tokens:n.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_trim_spaces:n ⋆

New: 2011-07-09
Updated: 2011-08-13

\tl_trim_spaces:n ⟨*token list*⟩

Removes any leading and trailing explicit space characters from the ⟨*token list*⟩ and leaves the result in the input stream. This process requires two expansions.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c

New: 2011-07-09

`\tl_trim_spaces:N` ⟨*tl var*⟩

Removes any leading and trailing explicit space characters from the content of the ⟨*tl var*⟩.

89 The first token from a token list

Functions which deal with either only the very first token of a token list or everything except the first token.

\tl_head:N ⋆
\tl_head:(n|V|v|f) ⋆

Updated: 2012-02-08

`\tl_head:n {`⟨*tokens*⟩`}`

Leaves in the input stream the first non-space token from the ⟨*tokens*⟩. Any leading space tokens will be discarded, and thus for example

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

and

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

will both leave `a` in the input stream. An empty list of ⟨*tokens*⟩ or one which consists only of space (category code 10) tokens will result in `\tl_head:n` leaving nothing in the input stream.

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

\tl_head:w ⋆

`\tl_head:w` ⟨*tokens*⟩ `\q_stop`

Leaves in the input stream the first non-space token from the ⟨*tokens*⟩. An empty list of ⟨*tokens*⟩ or one which consists only of space (category code 10) tokens will result in an error, and thus ⟨*tokens*⟩ must *not* be "blank" as determined by `\tl_if_blank:n(TF)`. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

`\tl_tail:N` ★
`\tl_tail:(n|V|v|f)` ★

Updated: 2012-02-08

`\tl_tail:n {⟨tokens⟩}`

Discards the all leading space tokens and the first non-space token in the ⟨*tokens*⟩, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { abc }
```

and

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

will both leave `bc` in the input stream. An empty list of ⟨*tokens*⟩ or one which consists only of space (category code 10) tokens will result in `\tl_tail:n` leaving nothing in the input stream.

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

`\tl_tail:w` ★

`\tl_tail:w {⟨tokens⟩} \q_stop`

Discards the all leading space tokens and the first non-space token in the ⟨*tokens*⟩, and leaves the remaining tokens in the input stream. An empty list of ⟨*tokens*⟩ or one which consists only of space (category code 10) tokens will result in an error, and thus ⟨*tokens*⟩ must *not* be "blank" as determined by `\tl_if_blank:n(TF)`. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_tail:n` should be preferred if the number of expansions is not critical.

`\str_head:n` ★
`\str_tail:n` ★

New: 2011-08-10

`\str_head:n {⟨tokens⟩}`
`\str_tail:n {⟨tokens⟩}`

Converts the ⟨*tokens*⟩ into a string, as described for `\tl_to_str:n`. The `\str_head:n` function then leaves the first character of this string in the input stream. The `\str_tail:n` function leaves all characters except the first in the input stream. The first character may be a space. If the ⟨*tokens*⟩ argument is entirely empty, nothing is left in the input stream.

`\tl_if_head_eq_catcode_p:nN` ★
`\tl_if_head_eq_catcode:nNTF` ★

Updated: 2011-08-10

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

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same category code as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, its head is considered to be `\q_nil`, and the test will be true if ⟨*test token*⟩ is a control sequence.

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

Updated: 2011-08-10

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

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same character code as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, its head is considered to be `\q_nil`, and the test will be true if ⟨*test token*⟩ is a control sequence.

\tl_if_head_eq_meaning_p:nN ⋆
\tl_if_head_eq_meaning:nNTF ⋆

Updated: 2011-08-10

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

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same meaning as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, its head is considered to be `\q_nil`, and the test will be true if ⟨*test token*⟩ has the same meaning as `\q_nil`.

\tl_if_head_group_p:n ⋆
\tl_if_head_group:nTF ⋆

Updated: 2011-08-11

```
\tl_if_head_group_p:n {⟨token list⟩}
\tl_if_head_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

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

\tl_if_head_N_type_p:n ⋆
\tl_if_head_N_type:nTF ⋆

New: 2011-08-11

```
\tl_if_head_N_type_p:n {⟨token list⟩}
\tl_if_head_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

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

\tl_if_head_space_p:n ⋆
\tl_if_head_space:nTF ⋆

Updated: 2011-08-11

```
\tl_if_head_space_p:n {⟨token list⟩}
\tl_if_head_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit space character (with category code 10 and character code 32). If ⟨*token list*⟩ starts with an implicit token such as `\c_space_token`, the test will yield false, as well as if the argument is empty. This function is useful to implement actions on token lists on a token by token basis.

TEXhackers note: When TEX reads a character of category code 10 for the first time, it is converted to an explicit space token, with character code 32, regardless of the initial character code. "Funny" spaces with a different category code, can be produced using `\lowercase`. Explicit spaces are also produced as a result of `\token_to_str:N`, `\tl_to_str:n`, etc.

90 Viewing token lists

\tl_show:N
\tl_show:c

\tl_show:N ⟨*tl var*⟩

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

TEXhackers note: \tl_show:N is the TEX primitive \show.

\tl_show:n

\tl_show:n ⟨*token list*⟩

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

TEXhackers note: \tl_show:n is the ε-TEX primitive \showtokens.

91 Constant token lists

\c_job_name_tl

Updated: 2011-08-18

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 is set by TEX and should not be overwritten by the package.

\c_empty_tl

Constant that is always empty.

\c_space_tl

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

92 Scratch token lists

\l_tmpa_tl
\l_tmpb_tl

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_tl
\g_tmpb_tl

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

93 Experimental token list functions

\tl_reverse_tokens:n ⋆

New: 2012-01-08

\tl_reverse_tokens:n {⟨*tokens*⟩}

This function, which works directly on TeX tokens, reverses the order of the ⟨*tokens*⟩: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, \tl_reverse_tokens:n {a~{b()}} leaves {)(b}~a in the input stream. This function requires two steps of expansion.

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

\tl_length_tokens:n ⋆

New: 2011-08-11

\tl_length_tokens:n {⟨*tokens*⟩}

Counts the number of TeX tokens in the ⟨*tokens*⟩ and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the length of a~{bc} is 6. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_expandable_uppercase:n ⋆
\tl_expandable_lowercase:n ⋆

New: 2012-01-08

\tl_expandable_uppercase:n {⟨*tokens*⟩}
\tl_expandable_lowercase:n {⟨*tokens*⟩}

The \tl_expandable_uppercase:n function works through all of the ⟨*tokens*⟩, replacing characters in the range a–z (with arbitrary category code) by the corresponding letter in the range A–Z, with category code 11 (letter). Similarly, \tl_expandable_lowercase:n replaces characters in the range A–Z by letters in the range a–z, and leaves other tokens unchanged. This function requires two steps of expansion.

TeXhackers note: Begin-group and end-group characters are normalized and become { and }, respectively. The result is returned within the \unexpanded primitive (\exp_not:n), which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_item:nn ⋆
\tl_item:(Nn|cn) ⋆

New: 2011-11-21
Updated: 2012-01-08

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

Indexing items in the ⟨*token list*⟩ from 0 on the left, this function will evaluate the ⟨*integer expression*⟩ and leave the appropriate item from the ⟨*token list*⟩ in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then thr function expands to nothing.

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

94 Internal functions

\q_tl_act_mark
\q_tl_act_stop

Quarks which are only used for the particular purposes of `\tl_act_...` functions.

Part XII

The l3seq package Sequences and stacks

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

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

95 Creating and initialising sequences

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

`\seq_new:N` ⟨*sequence*⟩

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

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

`\seq_clear:N` ⟨*sequence*⟩

Clears all items from the ⟨*sequence*⟩.

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

`\seq_clear_new:N` ⟨*sequence*⟩

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

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

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

Sets the content of ⟨*sequence1*⟩ equal to that of ⟨*sequence2*⟩.

`\seq_set_split:Nnn`
`\seq_gset_split:Nnn`

New: 2011-08-15
Updated: 2011-12-07

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

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

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

\seq_concat:NNN ⟨*sequence1*⟩ ⟨*sequence2*⟩ ⟨*sequence3*⟩

Concatenates the content of ⟨*sequence2*⟩ and ⟨*sequence3*⟩ together and saves the result in ⟨*sequence1*⟩. The items in ⟨*sequence2*⟩ will be placed at the left side of the new sequence.

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

New: 2012-03-03

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

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

96 Appending data to sequences

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

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

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

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

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

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

97 Recovering items from sequences

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

\seq_get_left:NN
\seq_get_left:cN

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

Stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_get_right:NN
\seq_get_right:cN

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

Stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_pop_left:NN
\seq_pop_left:cN

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

Pops the left-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_gpop_left:NN
\seq_gpop_left:cN

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

Pops the left-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_pop_right:NN
\seq_pop_right:cN

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

Pops the right-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in in the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_gpop_right:NN
\seq_gpop_right:cN

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

Pops the right-most item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local. If ⟨*sequence*⟩ is empty an error will be raised.

98 Modifying sequences

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

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

\seq_remove_duplicates:N ⟨*sequence*⟩

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

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

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

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

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

99 Sequence conditionals

\seq_if_empty_p:N ⋆
\seq_if_empty_p:c ⋆
\seq_if_empty:NTF ⋆
\seq_if_empty:cTF ⋆

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

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

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

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

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

100 Mapping to sequences

\seq_map_function:NN ☆
\seq_map_function:cN ☆

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

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

\seq_map_inline:Nn
\seq_map_inline:cn

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

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

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

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

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

\seq_map_break: ☆

\seq_map_break:

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

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

Use outside of a \seq_map_... scenario will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro \prg_break_point:n before further items are taken from the input stream. This will depend on the design of the mapping function.

\seq_map_break:n ☆

\seq_map_break:n {⟨*tokens*⟩}

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

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

Use outside of a \seq_map_... scenario will lead to low level TEX errors.

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

101 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

\seq_get:NN
\seq_get:cN

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

Reads the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_pop:NN
\seq_pop:cN

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

Pops the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_gpop:NN
\seq_gpop:cN

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

Pops the top item from a ⟨*sequence*⟩ into the ⟨*token list variable*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty an error will be raised.

\seq_push:Nn
\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gpush:Nn
\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

\seq_push:Nn ⟨*sequence*⟩ {⟨*item*⟩}

Adds the {⟨*item*⟩} to the top of the ⟨*sequence*⟩.

102 Viewing sequences

\seq_show:N
\seq_show:c

\seq_show:N ⟨*sequence*⟩

Displays the entries in the ⟨*sequence*⟩ in the terminal.

103 Experimental sequence functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

\seq_get_left:NN*TF*
\seq_get_left:cN*TF*

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

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

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

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

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

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

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

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

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

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

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

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

`\seq_length:N` ⋆
`\seq_length:c` ⋆

`\seq_length:N` ⟨*sequence*⟩

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

\seq_item:Nn ★
\seq_item:cn ★

Updated: 2012-01-08

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

Indexing items in the ⟨*sequence*⟩ from 0 at the top (left), this function will evaluate the ⟨*integer expression*⟩ and leave the appropriate item from the sequence in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the sequence. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*sequence*⟩ (as calculated by \seq_length:N) then the function will expand to nothing.

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

\seq_use:N ☆
\seq_use:c ☆

\seq_use:N ⟨*sequence*⟩

Places each ⟨*item*⟩ in the ⟨*sequence*⟩ in turn in the input stream. This occurs in an expandable fashion, and is implemented as a mapping. This means that the process may be prematurely terminated using \seq_map_break: or \seq_map_break:n. The ⟨*items*⟩ in the ⟨*sequence*⟩ will be used from left (top) to right (bottom).

\seq_mapthread_function:NNN ☆
\seq_mapthread_function:(NcN|cNN|ccN) ☆

\seq_mapthread_function:NNN ⟨*seq1*⟩ ⟨*seq2*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every pair of items ⟨*seq1-item*⟩–⟨*seq2-item*⟩ from the two sequences, returning items from both sequences from left to right. The ⟨*function*⟩ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

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

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

Sets the ⟨*sequence*⟩ within the current TeX group to be equal to the content of the ⟨*comma-list*⟩.

\seq_reverse:N
\seq_greverse:N

New: 2011-11-22
Updated: 2011-11-24

\seq_reverse:N ⟨*sequence*⟩

Reverses the order of items in the ⟨*sequence*⟩, and assigns the result to ⟨*sequence*⟩, locally or globally according to the variant chosen.

\seq_set_filter:NNn
\seq_gset_filter:NNn

New: 2011-12-22

`\seq_set_filter:NNn` ⟨*sequence1*⟩ ⟨*sequence2*⟩ {⟨*inline boolexpr*⟩}

Evaluates the ⟨*inline boolexpr*⟩ for every ⟨*item*⟩ stored within the ⟨*sequence2*⟩. The ⟨*inline boolexpr*⟩ will receive the ⟨*item*⟩ as `#1`. The sequence of all ⟨*items*⟩ for which the ⟨*inline boolexpr*⟩ evaluated to `true` is assigned to ⟨*sequence1*⟩.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TeX errors.

\seq_set_map:NNn
\seq_gset_map:NNn

New: 2011-12-22

`\seq_set_map:NNn` ⟨*sequence1*⟩ ⟨*sequence2*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence2*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. The sequence resulting from `x`-expanding ⟨*inline function*⟩ applied to each ⟨*item*⟩ is assigned to ⟨*sequence1*⟩. As such, the code in ⟨*inline function*⟩ should be expandable.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TeX errors.

104 Internal sequence functions

\seq_if_empty_err_break:N

`\seq_if_empty_err_break:N` ⟨*sequence*⟩

Tests if the ⟨*sequence*⟩ is empty, and if so issues an error message before skipping over any tokens up to `\prg_break_point:n`. This function is used to avoid more serious errors which would otherwise occur if some internal functions were applied to an empty ⟨*sequence*⟩.

\seq_item:n ⋆

`\seq_item:n` ⟨*item*⟩

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

\seq_push_item_def:n
\seq_push_item_def:x

`\seq_push_item_def:n` {⟨*code*⟩}

Saves the definition of `\seq_item:n` and redefines it to accept one parameter and expand to ⟨*code*⟩. This function should always be balanced by use of `\seq_pop_item_def:`.

\seq_pop_item_def:

`\seq_pop_item_def:`

Restores the definition of `\seq_item:n` most recently saved by `\seq_push_item_def:n`. This function should always be used in a balanced pair with `\seq_push_item_def:n`.

\seq_break: ⋆

`\seq_break:`

Used to terminate sequence functions by gobbling all tokens up to `\prg_break_point:n`. This function is a copy of `\seq_map_break:`, but is used in situations which are not mappings.

`\seq_break:n` ⋆

`\seq_break:n {⟨tokens⟩}`

Used to terminate sequence functions by gobbling all tokens up to `\prg_break_point:n`, then inserting the ⟨*tokens*⟩ before continuing reading the input stream. This function is a copy of `\seq_map_break:n`, but is used in situations which are not mappings.

Part XIII

The l3clist package
Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

will leave `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces.

105 Creating and initialising comma lists

`\clist_new:N`
`\clist_new:c`

`\clist_new:N` ⟨*comma list*⟩

Creates a new ⟨*comma list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*comma list*⟩ will initially contain no items.

`\clist_clear:N`
`\clist_clear:c`
`\clist_gclear:N`
`\clist_gclear:c`

`\clist_clear:N` ⟨*comma list*⟩

Clears all items from the ⟨*comma list*⟩.

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

`\clist_clear_new:N` ⟨*comma list*⟩

Ensures that the ⟨*comma list*⟩ exists globally by applying `\clsit_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

`\clist_set_eq:NN`
`\clist_set_eq:(cN|Nc|cc)`
`\clist_gset_eq:NN`
`\clist_gset_eq:(cN|Nc|cc)`

`\clist_set_eq:NN` ⟨*comma list1*⟩ ⟨*comma list2*⟩

Sets the content of ⟨*comma list1*⟩ equal to that of ⟨*comma list2*⟩.

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

\clist_concat:NNN ⟨*comma list1*⟩ ⟨*comma list2*⟩ ⟨*comma list3*⟩

Concatenates the content of ⟨*comma list2*⟩ and ⟨*comma list3*⟩ together and saves the result in ⟨*comma list1*⟩. The items in ⟨*comma list2*⟩ will be placed at the left side of the new comma list.

\clist_if_exist_p:N ★
\clist_if_exist_p:c ★
\clist_if_exist:NTF ★
\clist_if_exist:cTF ★

New: 2012-03-03

\clist_if_exist_p:N ⟨*comma list*⟩
\clist_if_exist:NTF ⟨*comma list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

106 Adding data to comma lists

\clist_set:Nn
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)

New: 2011-09-06

\clist_set:Nn ⟨*comma list*⟩ {⟨*item1*⟩,...,⟨*item$_n$*⟩}

Sets ⟨*comma list*⟩ to contain the ⟨*items*⟩, removing any previous content from the variable. Spaces are removed from both sides of each item.

\clist_put_left:Nn
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)

Updated: 2011-09-05

\clist_put_left:Nn ⟨*comma list*⟩ {⟨*item1*⟩,...,⟨*item$_n$*⟩}

Appends the ⟨*items*⟩ to the left of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

\clist_put_right:Nn
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)

Updated: 2011-09-05

\clist_put_right:Nn ⟨*comma list*⟩ {⟨*item1*⟩,...,⟨*item$_n$*⟩}

Appends the ⟨*items*⟩ to the right of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

107 Using comma lists

\clist_use:N ⋆
\clist_use:c ⋆

\clist_use:N ⟨*comma list*⟩

Places the ⟨*comma list*⟩ directly into the input stream, including the commas, thus treating it as a ⟨*token list*⟩.

108 Modifying comma lists

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

\clist_remove_duplicates:N
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

\clist_remove_duplicates:N ⟨*comma list*⟩

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

TEXhackers note: This function iterates through every item in the ⟨*comma list*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it will not work if any of the items in the ⟨*comma list*⟩ contains `{`, `}`, or `#` (assuming the usual TEX category codes apply).

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn

Updated: 2011-09-06

\clist_remove_all:Nn ⟨*comma list*⟩ {⟨*item*⟩}

Removes every occurrence of ⟨*item*⟩ from the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TEXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TEX category codes apply).

109 Comma list conditionals

\clist_if_empty_p:N ⋆
\clist_if_empty_p:c ⋆
\clist_if_empty:N*TF* ⋆
\clist_if_empty:c*TF* ⋆

\clist_if_empty_p:N ⟨*comma list*⟩
\clist_if_empty:NTF ⟨*comma list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

`\clist_if_eq_p:NN` ⋆
`\clist_if_eq_p:(Nc|cN|cc)` ⋆
`\clist_if_eq:NNTF` ⋆
`\clist_if_eq:(Nc|cN|cc)TF` ⋆

`\clist_if_eq_p:NN` ⟨*clist₁*⟩ ⟨*clist₂*⟩
`\clist_if_eq:NNTF` ⟨*clist₁*⟩ ⟨*clist₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*comma lists*⟩ and is logically `true` if the two contain the same list of entries in the same order.

`\clist_if_in:NnTF`
`\clist_if_in:(NV|No|cn|cV|co|nn|nV|no)TF`

Updated: 2011-09-06

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

Tests if the ⟨*item*⟩ is present in the ⟨*comma list*⟩. In the case of an `n`-type ⟨*comma list*⟩, spaces are stripped from each item, but braces are not removed. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields `false`.

TeXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TeX category codes apply), and should not contain `,` nor start or end with a space.

110 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

When the comma list is given explicitly, as an `n`-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if your comma list that is being mapped is `{a␣,␣{b␣},␣,{},␣{c},}` then the arguments passed to the mapped function are '`a`', '`{b␣}`', an empty argument, and '`c`'.

When the comma list is given as an `N`-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using `n`-type comma lists.

`\clist_map_function:NN` ☆
`\clist_map_function:(cN|nN)` ☆

`\clist_map_function:NN` ⟨*comma list*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*comma list*⟩. The ⟨*function*⟩ will receive one argument for each iteration. The ⟨*items*⟩ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`. One mapping may be nested inside another.

`\clist_map_inline:Nn`
`\clist_map_inline:(cn|nn)`

`\clist_map_inline:Nn` ⟨*comma list*⟩ {⟨*inline function*⟩}

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

`\clist_map_variable:NNn`
`\clist_map_variable:(cNn|nNn)`

`\clist_map_variable:NNn` ⟨*comma list*⟩ ⟨*tl var.*⟩ {⟨*function using tl var.*⟩}

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

`\clist_map_break:` ☆

`\clist_map_break:`

Used to terminate a `\clist_map_...` function before all entries in the ⟨*comma list*⟩ have been processed. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\clist_map_...` scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\prg_break_point:n` before further items are taken from the input stream. This will depend on the design of the mapping function.

`\clist_map_break:n` ☆ `\clist_map_break:n {⟨tokens⟩}`

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

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\clist_map_...` scenario will lead to low level TeX errors.

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

111 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

`\clist_get:NN`
`\clist_get:cN`

`\clist_get:NN ⟨comma list⟩ ⟨token list variable⟩`

Stores the left-most item from a ⟨*comma list*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*comma list*⟩. The ⟨*token list variable*⟩ is assigned locally.

`\clist_get:NN`
`\clist_get:cN`

`\clist_get:NN ⟨comma list⟩ ⟨token list variable⟩`

Stores the right-most item from a ⟨*comma list*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*comma list*⟩. The ⟨*token list variable*⟩ is assigned locally.

`\clist_pop:NN`
`\clist_pop:cN`
Updated: 2011-09-06

`\clist_pop:NN ⟨comma list⟩ ⟨token list variable⟩`

Pops the left-most item from a ⟨*comma list*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*token list variable*⟩. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN ⟨*comma list*⟩ ⟨*token list variable*⟩

Pops the left-most item from a ⟨*comma list*⟩ into the ⟨*token list variable*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*token list variable*⟩. The ⟨*comma list*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local.

\clist_push:Nn
\clist_push:(NV|No|Nx|cn|cV|co|cx)
\clist_gpush:Nn
\clist_gpush:(NV|No|Nx|cn|cV|co|cx)

\clist_push:Nn ⟨*comma list*⟩ {⟨*items*⟩}

Adds the {⟨*items*⟩} to the top of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

112 Viewing comma lists

\clist_show:N
\clist_show:c

\clist_show:N ⟨*comma list*⟩

Displays the entries in the ⟨*comma list*⟩ in the terminal.

\clist_show:n

\clist_show:n {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

113 Scratch comma lists

\l_tmpa_clist
\l_tmpb_clist

New: 2011-09-06

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

\g_tmpa_clist
\g_tmpb_clist

New: 2011-09-06

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

114 Experimental comma list functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

\clist_length:N ⋆
\clist_length:(c|n) ⋆

New: 2011-06-25
Updated: 2011-09-06

\clist_length:N ⟨*comma list*⟩

Leaves the number of items in the ⟨*comma list*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*comma list*⟩ will include those which are duplicates, *i.e.* every item in a ⟨*comma list*⟩ is unique.

\clist_item:Nn ★
\clist_item:(cn|nn) ★

Updated: 2012-01-08

\clist_item:Nn ⟨*comma list*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*comma list*⟩ from 0 at the top (left), this function will evaluate the ⟨*integer expression*⟩ and leave the appropriate item from the comma list in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*comma list*⟩ (as calculated by \clist_length:N) then the function will expand to nothing.

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

\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

Updated: 2011-08-31

\clist_set_from_seq:NN ⟨*comma list*⟩ ⟨*sequence*⟩

Sets the ⟨*comma list*⟩ to be equal to the content of the ⟨*sequence*⟩. Items which contain either spaces or commas are surrounded by braces.

\clist_const:Nn
\clist_const:(Nx|cn|cx)

New: 2011-11-26

\clist_const:Nn ⟨*clist var*⟩ {⟨*comma list*⟩}

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

\clist_if_empty_p:n ★
\clist_if_empty:n*TF* ★

New: 2011-12-07

\clist_if_empty_p:n {⟨*comma list*⟩}
\clist_if_empty:nTF {⟨*comma list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*comma list*⟩ is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list {~,~,,~} (without outer braces) is empty, while {~,{},} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

115 Internal comma-list functions

\clist_trim_spaces:n ☆

New: 2011-07-09

\clist_trim_spaces:n {⟨*comma list*⟩}

Removes leading and trailing spaces from each ⟨*item*⟩ in the ⟨*comma list*⟩, leaving the resulting modified list in the input stream. This is used by the functions which add data into a comma list.

Part XIV

The l3prop package
Property lists

LaTeX3 implements a "property list" data type, which contain an unordered list of entries each of which consists of a ⟨*key*⟩ and an associated ⟨*value*⟩. The ⟨*key*⟩ and ⟨*value*⟩ may both be any ⟨*balanced text*⟩. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique ⟨*key*⟩: if an entry is added to a property list which already contains the ⟨*key*⟩ then the new entry will overwrite the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the keys module.

116 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` ⟨*property list*⟩

Creates a new ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property lists*⟩ will initially contain no entries.

`\prop_clear:N`
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

`\prop_clear:N` ⟨*property list*⟩

Clears all entries from the ⟨*property list*⟩.

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

`\prop_clear_new:N` ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

`\prop_set_eq:NN`
`\prop_set_eq:(cN|Nc|cc)`
`\prop_gset_eq:NN`
`\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN` ⟨*property list1*⟩ ⟨*property list2*⟩

Sets the content of ⟨*property list1*⟩ equal to that of ⟨*property list2*⟩.

117 Adding entries to property lists

\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)

\prop_put:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with \tl_to_str:n, meaning that category codes are ignored. If the ⟨*key*⟩ is already present in the ⟨*property list*⟩, the existing entry is overwritten by the new ⟨*value*⟩.

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn

\prop_put_if_new:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then no action is taken. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then a new entry is added. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with \tl_to_str:n, meaning that category codes are ignored.

118 Recovering values from property lists

\prop_get:NnN
\prop_get:(NVN|NoN|cnN|cVN|coN)

Updated: 2011-08-28

\prop_get:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ will contain the special marker \q_no_value. The ⟨*token list variable*⟩ is set within the current TeX group. See also \prop_get:NnNTF.

\prop_pop:NnN
\prop_pop:(NoN|cnN|coN)

Updated: 2011-08-18

\prop_pop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ will contain the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. Both assignments are local.

\prop_gpop:NnN
\prop_gpop:(NoN|cnN|coN)

Updated: 2011-08-18

\prop_gpop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*token list variable*⟩ will contain the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. The ⟨*property list*⟩ is modified globally, while the assignment of the ⟨*token list variable*⟩ is local.

119 Modifying property lists

\prop_del:Nn
\prop_del:(NV|cn|cV)
\prop_gdel:Nn
\prop_gdel:(NV|cn|cV)

\prop_del:Nn ⟨*property list*⟩ {⟨*key*⟩}

Deletes the entry listed under ⟨*key*⟩ from the ⟨*property list*⟩ which may be accessed. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it. The deletion is restricted to the current TEX group.

120 Property list conditionals

\prop_if_exist_p:N ⋆
\prop_if_exist_p:c ⋆
\prop_if_exist:N*TF* ⋆
\prop_if_exist:c*TF* ⋆

New: 2012-03-03

\prop_if_exist_p:N ⟨*property list*⟩
\prop_if_exist:NTF ⟨*property list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\prop_if_empty_p:N ⋆
\prop_if_empty_p:c ⋆
\prop_if_empty:N*TF* ⋆
\prop_if_empty:c*TF* ⋆

\prop_if_empty_p:N ⟨*property list*⟩
\prop_if_empty:NTF ⟨*property list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*property list*⟩ is empty (containing no entries).

\prop_if_in_p:Nn ⋆
\prop_if_in_p:(NV|No|cn|cV|co) ⋆
\prop_if_in:Nn*TF* ⋆
\prop_if_in:(NV|No|cn|cV|co)*TF* ⋆

Updated: 2011-09-15

\prop_if_in:NnTF ⟨*property list*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the ⟨*property list*⟩, making the comparison using the method described by \str_if_eq:nnTF.

TEXhackers note: This function iterates through every key–value pair in the ⟨*property list*⟩ and is therefore slower than using the non-expandable \prop_get:NnNTF.

121 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnNTF
\prop_get:(NVN|NoN|cnN|cVN|coN)TF

Updated: 2011-08-28

\prop_get:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩
{⟨true code⟩} {⟨false code⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, stores the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*property list*⟩. The ⟨*token list variable*⟩ is assigned locally.

\prop_pop:NnNTF
\prop_pop:cnNTF

New: 2011-08-18

\prop_pop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩
{⟨true code⟩} {⟨false code⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*property list*⟩. Both the ⟨*property list*⟩ and the ⟨*token list variable*⟩ are assigned locally.

122 Mapping to property lists

\prop_map_function:NN ☆
\prop_map_function:cN ☆

\prop_map_function:NN ⟨property list⟩ ⟨function⟩

Applies ⟨*function*⟩ to every ⟨*entry*⟩ stored in the ⟨*property list*⟩. The ⟨*function*⟩ will receive two argument for each iteration: the ⟨*key*⟩ and associated ⟨*value*⟩. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon.

\prop_map_inline:Nn
\prop_map_inline:cn

\prop_map_inline:Nn ⟨property list⟩ {⟨inline function⟩}

Applies ⟨*inline function*⟩ to every ⟨*entry*⟩ stored within the ⟨*property list*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*key*⟩ as #1 and the ⟨*value*⟩ as #2. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon.

\prop_map_break: ☆

\prop_map_break:

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario will lead low level TeX errors.

\prop_map_break:n ☆

\prop_map_break:n {⟨*tokens*⟩}

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario will lead low level TeX errors.

123 Viewing property lists

\prop_show:N
\prop_show:c

\prop_show:N ⟨*property list*⟩

Displays the entries in the ⟨*property list*⟩ in the terminal.

124 Experimental property list functions

This section contains functions which may or may not be retained, depending on how useful they are found to be.

\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

New: 2011-08-18

\prop_gpop:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream and leaves the ⟨*token list variable*⟩ unchanged. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*property list*⟩. The ⟨*property list*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\prop_map_tokens:Nn ☆
\prop_map_tokens:cn ☆

New: 2011-08-18

\prop_map_tokens:Nn ⟨*property list*⟩ {⟨*code*⟩}

Analogue of \prop_map_function:NN which maps several tokens instead of a single function. The ⟨*code*⟩ receives each key–value pair in the ⟨*property list*⟩ as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

will expand to the value corresponding to mykey: for each pair in \l_my_prop the function \str_if_eq:nnT receives mykey, the ⟨*key*⟩ and the ⟨*value*⟩ as its three arguments. For that specific task, \prop_get:Nn is faster.

\prop_get:Nn ⋆
\prop_get:cn ⋆

Updated: 2012-01-08

`\prop_get:Nn` ⟨*property list*⟩ {⟨*key*⟩}

Expands to the ⟨*value*⟩ corresponding to the ⟨*key*⟩ in the ⟨*property list*⟩. If the ⟨*key*⟩ is missing, this has an empty expansion.

TeXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*value*⟩ will not expand further when appearing in an x-type argument expansion.

125 Internal property list functions

\q_prop

The internal token used to separate out property list entries, separating both the ⟨*key*⟩ from the ⟨*value*⟩ and also one entry from another.

\c_empty_prop

A permanently-empty property list used for internal comparisons.

\prop_split:Nnn

`\prop_split:Nnn` ⟨*property list*⟩ {⟨*key*⟩} {⟨*code*⟩}

Splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three groups: the ⟨*extract*⟩ of ⟨*property list*⟩ before the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*⟩ of the ⟨*property list*⟩ after the ⟨*value*⟩. The first ⟨*extract*⟩ retains the internal structure of a property list. The second is only missing the leading separator `\q_prop`. This ensures that the concatenation of the two ⟨*extracts*⟩ is a property list. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the second group will contain the marker `\q_no_value` and the third is empty. Once the split has occurred, the ⟨*code*⟩ is inserted followed by the three groups: thus the ⟨*code*⟩ should properly absorb three arguments. The ⟨*key*⟩ comparison takes place as described for `\str_if_eq:nn`.

\prop_split:NnTF

`\prop_split:NnTF` ⟨*property list*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three groups: the ⟨*extract*⟩ of ⟨*property list*⟩ before the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*⟩ of the ⟨*property list*⟩ after the ⟨*value*⟩. The first ⟨*extract*⟩ retains the internal structure of a property list. The second is only missing the leading separator `\q_prop`. This ensures that the concatenation of the two ⟨*extracts*⟩ is a property list. If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then the ⟨*true code*⟩ is left in the input stream, followed by the three groups: thus the ⟨*true code*⟩ should properly absorb three arguments. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the ⟨*false code*⟩ is left in the input stream, with no trailing material. The ⟨*key*⟩ comparison takes place as described for `\str_if_eq:nn`.

Part XV
The l3box package
Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

126 Creating and initialising boxes

`\box_new:N`
`\box_new:c`

`\box_new:N` ⟨*box*⟩

Creates a new ⟨*box*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*box*⟩ will initially be void.

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

`\box_clear:N` ⟨*box*⟩

Clears the content of the ⟨*box*⟩ by setting the box equal to `\c_void_box`.

`\box_clear_new:N`
`\box_clear_new:c`
`\box_gclear_new:N`
`\box_gclear_new:c`

`\box_clear_new:N` ⟨*box*⟩

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

`\box_set_eq:NN`
`\box_set_eq:(cN|Nc|cc)`
`\box_gset_eq:NN`
`\box_gset_eq:(cN|Nc|cc)`

`\box_set_eq:NN` ⟨*box1*⟩ ⟨*box2*⟩

Sets the content of ⟨*box1*⟩ equal to that of ⟨*box2*⟩.

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

`\box_set_eq_clear:NN` ⟨*box1*⟩ ⟨*box2*⟩

Sets the content of ⟨*box1*⟩ within the current TeX group equal to that of ⟨*box2*⟩, then clears ⟨*box2*⟩ globally.

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

`\box_gset_eq_clear:NN` ⟨*box1*⟩ ⟨*box2*⟩

Sets the content of ⟨*box1*⟩ equal to that of ⟨*box2*⟩, then clears ⟨*box2*⟩. These assignments are global.

\box_if_exist_p:N ⋆
\box_if_exist_p:c ⋆
\box_if_exist:NTF ⋆
\box_if_exist:cTF ⋆

New: 2012-03-03

\box_if_exist_p:N ⟨*box*⟩
\box_if_exist:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

127 Using boxes

\box_use:N
\box_use:c

\box_use:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting.

TeXhackers note: This is the TeX primitive `\copy`.

\box_use_clear:N
\box_use_clear:c

\box_use_clear:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting, then globally clears the content of the ⟨*box*⟩.

TeXhackers note: This is the TeX primitive `\box`.

\box_move_right:nn
\box_move_left:nn

\box_move_right:nn {⟨*dimexpr*⟩} {⟨*box function*⟩}

This function operates in vertical mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced horizontally by the given ⟨*dimexpr*⟩ from the reference point for typesetting, to the right or left as appropriate. The ⟨*box function*⟩ should be a box operation such as `\box_use:N \<box>` or a "raw" box specification such as `\vbox:n { xyz }`.

\box_move_up:nn
\box_move_down:nn

\box_move_up:nn {⟨*dimexpr*⟩} {⟨*box function*⟩}

This function operates in horizontal mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced vertical by the given ⟨*dimexpr*⟩ from the reference point for typesetting, up or down as appropriate. The ⟨*box function*⟩ should be a box operation such as `\box_use:N \<box>` or a "raw" box specification such as `\vbox:n { xyz }`.

128 Measuring and setting box dimensions

\box_dp:N
\box_dp:c

\box_dp:N ⟨*box*⟩

Calculates the depth (below the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

TeXhackers note: This is the TeX primitive `\dp`.

\box_ht:N
\box_ht:c

\box_ht:N ⟨*box*⟩

Calculates the height (above the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

TEXhackers note: This is the TEX primitive \ht.

\box_wd:N
\box_wd:c

\box_wd:N ⟨*box*⟩

Calculates the width of the ⟨*box*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

TEXhackers note: This is the TEX primitive \wd.

\box_set_dp:Nn
\box_set_dp:cn

Updated: 2011-10-22

\box_set_dp:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the depth (below the baseline) of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

\box_set_ht:Nn
\box_set_ht:cn

Updated: 2011-10-22

\box_set_ht:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the height (above the baseline) of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

\box_set_wd:Nn
\box_set_wd:cn

Updated: 2011-10-22

\box_set_wd:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

Set the width of the ⟨*box*⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

129 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TEX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

\box_resize:Nnn
\box_resize:cnn

New: 2011-09-02

\box_resize:Nnn ⟨*box*⟩ {⟨*x-size*⟩} {⟨*y-size*⟩}

Resize the ⟨*box*⟩ to ⟨*x-size*⟩ horizontally and ⟨*y-size*⟩ vertically (both of the sizes are dimension expressions). The ⟨*y-size*⟩ is the vertical size (height plus depth) of the box. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. Negative sizes will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged. The resizing applies within the current TEX group level.

This function is experimental

`\box_resize_to_ht_plus_dp:Nn`
`\box_resize_to_ht_plus_dp:cn`

New: 2011-09-02
Updated: 2011-10-22

`\box_resize_to_ht_plus_dp:Nn` ⟨*box*⟩ {⟨*y-size*⟩}

Resize the ⟨*box*⟩ to ⟨*y-size*⟩ vertically, scaling the horizontal size by the same amount (⟨*y-size*⟩ is a dimension expression). The ⟨*y-size*⟩ is the vertical size (height plus depth) of the box. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative size will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged. The resizing applies within the current TeX group level.

This function is experimental

`\box_resize_to_wd:Nn`
`\box_resize_to_wd:cn`

New: 2011-09-02
Updated: 2011-10-22

`\box_resize_to_wd:Nn` ⟨*box*⟩ {⟨*x-size*⟩}

Resize the ⟨*box*⟩ to ⟨*x-size*⟩ horizontally, scaling the vertical size by the same amount (⟨*x-size*⟩ is a dimension expression). The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative size will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged. The resizing applies within the current TeX group level.

This function is experimental

`\box_rotate:Nn`
`\box_rotate:cn`

New: 2011-09-02
Updated: 2011-10-22

`\box_rotate:Nn` ⟨*box*⟩ {⟨*angle*⟩}

Rotates the ⟨*box*⟩ by ⟨*angle*⟩ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the rotation is applied. The rotation applies within the current TeX group level.

This function is experimental

`\box_scale:Nnn`
`\box_scale:cnn`

New: 2011-09-02
Updated: 2011-10-22

`\box_scale:Nnn` ⟨*box*⟩ {⟨*x-scale*⟩} {⟨*y-scale*⟩}

Scales the ⟨*box*⟩ by factors ⟨*x-scale*⟩ and ⟨*y-scale*⟩ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the scaling is applied. Negative scalings will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged. The scaling applies within the current TeX group level.

This function is experimental

130 Viewing part of a box

\box_clip:N
\box_clip:c

New: 2011-11-13

\box_clip:N ⟨*box*⟩

Clips the ⟨*box*⟩ in the output so that only material inside the bounding box is displayed in the output. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the clipping is applied. The clipping applies within the current TEX group level.

This function is experimental

TEXhackers note: Clipping is implemented by the driver, and as such the full content of the box is places in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

\box_trim:Nnnnn
\box_trim:cnnnn

New: 2011-11-13

\box_trim:Nnnnn ⟨*box*⟩ {⟨*left*⟩} {⟨*bottom*⟩} {⟨*right*⟩} {⟨*top*⟩}

Adjusts the bounding box of the ⟨*box*⟩ ⟨*left*⟩ is removed from the left-hand edge of the bounding box, ⟨*right*⟩ from the right-hand edge and so fourth. All adjustments are ⟨*dimension expressions*⟩. Material output of the bounding box will still be displayed in the output unless \box_clip:N is subsequently applied. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the viewport operation is applied. The clipping applies within the current TEX group level.

This function is experimental

\box_viewport:Nnnnn
\box_viewport:cnnnn

New: 2011-11-13

\box_viewport:Nnnnn ⟨*box*⟩ {⟨*llx*⟩} {⟨*lly*⟩} {⟨*urx*⟩} {⟨*ury*⟩}

Adjusts the bounding box of the ⟨*box*⟩ such that it has lower-left co-ordinates (⟨*llx*⟩, ⟨*lly*⟩) and upper-right co-ordinates (⟨*urx*⟩, ⟨*ury*⟩). All four co-ordinate positions are ⟨*dimension expressions*⟩. Material output of the bounding box will still be displayed in the output unless \box_clip:N is subsequently applied. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the viewport operation is applied. The clipping applies within the current TEX group level.

This function is experimental

131 Box conditionals

\box_if_empty_p:N ⋆
\box_if_empty_p:c ⋆
\box_if_empty:NTF ⋆
\box_if_empty:cTF ⋆

\box_if_empty_p:N ⟨*box*⟩
\box_if_empty:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is a empty (equal to \c_empty_box).

\box_if_horizontal_p:N ⋆
\box_if_horizontal_p:c ⋆
\box_if_horizontal:NTF ⋆
\box_if_horizontal:cTF ⋆

\box_if_horizontal_p:N ⟨*box*⟩
\box_if_horizontal:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*box*⟩ is a horizontal box.

\box_if_vertical_p:N ⋆
\box_if_vertical_p:c ⋆
\box_if_vertical:NTF ⋆
\box_if_vertical:cTF ⋆

\box_if_vertical_p:N ⟨box⟩
\box_if_vertical:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨*box*⟩ is a vertical box.

132 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

\box_set_to_last:N ⟨box⟩

Sets the ⟨*box*⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨*box*⟩ will always be void as it is not possible to recover the last added item.

133 Constant boxes

\c_empty_box

This is a permanently empty box, which is neither set as horizontal nor vertical.

134 Scratch boxes

\l_tmpa_box
\l_tmpb_box

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

135 Viewing box contents

\box_show:N
\box_show:c

\box_show:N ⟨box⟩

Writes the contents of ⟨*box*⟩ to the log file.

TeXhackers note: This is a wrapper around the TeX primitive `\showbox`.

136 Horizontal mode boxes

\hbox:n

\hbox:n {⟨contents⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of natural width and then includes this box in the current list for typesetting.

TeXhackers note: This is the TeX primitive `\hbox`.

`\hbox_to_wd:nn`

`\hbox_to_wd:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of width ⟨*dimexpr*⟩ and then includes this box in the current list for typesetting.

`\hbox_to_zero:n`

`\hbox_to_zero:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width and then includes this box in the current list for typesetting.

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

`\hbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩.

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn`

`\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩.

`\hbox_overlap_right:n`

`\hbox_overlap_right:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material will protrude to the right of the insertion point.

`\hbox_overlap_left:n`

`\hbox_overlap_left:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material will protrude to the left of the insertion point.

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_set_end:`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_gset_end:`

`\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:`

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

`\hbox_unpack:N`
`\hbox_unpack:c`

`\hbox_unpack:N ⟨box⟩`

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

> **TEXhackers note:** This is the TEX primitive `\unhcopy`.

\hbox_unpack_clear:N
\hbox_unpack_clear:c

`\hbox_unpack_clear:N ⟨box⟩`

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The ⟨*box*⟩ is then cleared globally.

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

137 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box will have no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter will typically be non-zero.

\vbox:n

Updated: 2011-12-18

`\vbox:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting.

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

\vbox_top:n

Updated: 2011-12-18

`\vbox_top:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box will tbe equal to that of the *first* item added to the box.

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

\vbox_to_ht:nn

Updated: 2011-12-18

`\vbox_to_ht:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of height ⟨*dimexpr*⟩ and then includes this box in the current list for typesetting.

\vbox_to_zero:n

Updated: 2011-12-18

`\vbox_to_zero:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of zero height and then includes this box in the current list for typesetting.

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

Updated: 2011-12-18

`\vbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩.

\vbox_set_top:Nn
\vbox_set_top:cn
\vbox_gset_top:Nn
\vbox_gset_top:cn

Updated: 2011-12-18

\vbox_set_top:Nn ⟨*box*⟩ {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. The baseline of the box will tbe equal to that of the *first* item added to the box.

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

Updated: 2011-12-18

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

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dimexpr*⟩ and then stores the result inside the ⟨*box*⟩.

\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:

Updated: 2011-12-18

\vbox_begin:Nw ⟨*box*⟩ ⟨*contents*⟩ \vbox_set_end:

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\vbox_set_split_to_ht:NNn

Updated: 2011-10-22

\vbox_set_split_to_ht:NNn ⟨*box1*⟩ ⟨*box2*⟩ {⟨*dimexpr*⟩}

Sets ⟨*box1*⟩ to contain material to the height given by the ⟨*dimexpr*⟩ by removing content from the top of ⟨*box2*⟩ (which must be a vertical box).

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

\vbox_unpack:N
\vbox_unpack:c

\vbox_unpack:N ⟨*box*⟩

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

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

\vbox_unpack_clear:N
\vbox_unpack_clear:c

\vbox_unpack:N ⟨*box*⟩

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The ⟨*box*⟩ is then cleared globally.

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

138 Primitive box conditionals

\if_hbox:N ⋆

```
\if_hbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a horizontal box.

> **TeXhackers note:** This is the TeX primitive `\ifhbox`.

\if_vbox:N ⋆

```
\if_vbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a vertical box.

> **TeXhackers note:** This is the TeX primitive `\ifvbox`.

\if_box_empty:N ⋆

```
\if_box_empty:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is an empty (void) box.

> **TeXhackers note:** This is the TeX primitive `\ifvoid`.

139 Experimental box functions

\box_show:Nnn
\box_show:cnn
New: 2011-11-21

`\box_show:Nnn ⟨box⟩ ⟨int 1⟩ ⟨int 2⟩`

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*int 1*⟩ items of the box, and descending into ⟨*int 1*⟩ levels of nesting.

> **TeXhackers note:** This is a wrapper around the TeX primitives `\showbox`, `\showboxbreadth` and `\showboxdepth`.

\box_show_full:N
\box_show_full:c
New: 2011-11-22

`\box_show_full:N ⟨box⟩`

Display the contents of ⟨*box*⟩ in the terminal, showing all items in the box.

Part XVI

The l3coffins package Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

140 Creating and initialising coffins

\coffin_new:N
\coffin_new:c

New: 2011-08-17

`\coffin_new:N` ⟨*coffin*⟩

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

\coffin_clear:N
\coffin_clear:c

New: 2011-08-17

`\coffin_clear:N` ⟨*coffin*⟩

Clears the content of the ⟨*coffin*⟩ within the current TEX group level.

\coffin_set_eq:NN
\coffin_set_eq:(Nc|cN|cc)

New: 2011-08-17

`\coffin_set_eq:NN` ⟨*coffin1*⟩ ⟨*coffin2*⟩

Sets both the content and poles of ⟨*coffin1*⟩ equal to those of ⟨*coffin2*⟩ within the current TEX group level.

141 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current TEX group level.

\hcoffin_set:Nn
\hcoffin_set:cn

New: 2011-08-17
Updated: 2011-09-03

`\hcoffin_set:Nn` ⟨*coffin*⟩ {⟨*material*⟩}

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:

New: 2011-09-10

`\hcoffin_set:Nw` ⟨*coffin*⟩ ⟨*material*⟩ `\hcoffin_set_end:`

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn
\vcoffin_set:cnn

New: 2011-08-17
Updated: 2011-09-03

`\vcoffin_set:Nnn` ⟨*coffin*⟩ {⟨*width*⟩} {⟨*material*⟩}

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:

New: 2011-09-10

`\vcoffin_set:Nnw` ⟨*coffin*⟩ {⟨*width*⟩} ⟨*material*⟩ `\vcoffin_set_end:`

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn

New: 2011-08-17

`\coffin_set_horizontal_pole:Nnn` ⟨*coffin*⟩ {⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run horizontally through the ⟨*coffin*⟩. The ⟨*pole*⟩ will be located at the ⟨*offset*⟩ from the bottom edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression; this may include the terms `\TotalHeight`, `\Height`, `\Depth` and `\Width`, which will evaluate to the appropriate dimensions of the ⟨*coffin*⟩.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn

New: 2011-08-17

`\coffin_set_vertical_pole:Nnn` ⟨*coffin*⟩ {⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run vertically through the ⟨*coffin*⟩. The ⟨*pole*⟩ will be located at the ⟨*offset*⟩ from the left-hand edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression; this may include the terms `\TotalHeight`, `\Height`, `\Depth` and `\Width`, which will evaluate to the appropriate dimensions of the ⟨*coffin*⟩.

142 Coffin transformations

\coffin_resize:Nnn
\coffin_resize:cnn

New: 2011-09-02

`\coffin_resize:Nnn` ⟨*coffin*⟩ {⟨*width*⟩} {⟨*total-height*⟩}

Resized the ⟨*coffin*⟩ to ⟨*width*⟩ and ⟨*total-height*⟩, both of which should be given as dimension expressions. These may include the terms `\TotalHeight`, `\Height`, `\Depth` and `\Width`, which will evaluate to the appropriate dimensions of the ⟨*coffin*⟩.

This function is experimental.

\coffin_rotate:Nn
\coffin_rotate:cn

New: 2011-09-02

`\coffin_rotate:Nn` ⟨*coffin*⟩ {⟨*angle*⟩}

Rotates the ⟨*coffin*⟩ by the given ⟨*angle*⟩ (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will not result in the bounding box of the coffin growing unnecessarily.

\coffin_scale:Nnn
\coffin_scale:cnn

New: 2011-09-02

\coffin_scale:Nnn ⟨*coffin*⟩ {⟨*x-scale*⟩} {⟨*y-scale*⟩}

Scales the ⟨*coffin*⟩ by a factors ⟨*x-scale*⟩ and ⟨*y-scale*⟩ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

This function is experimental.

143 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)

\coffin_attach:NnnNnnnn
⟨*coffin1*⟩ {⟨*coffin1-pole1*⟩} {⟨*coffin1-pole2*⟩}
⟨*coffin2*⟩ {⟨*coffin2-pole1*⟩} {⟨*coffin2-pole2*⟩}
{⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function attaches ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin1*⟩ is not altered, *i.e.* ⟨*coffin2*⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle1*⟩, the point of intersection of ⟨*coffin1-pole1*⟩ and ⟨*coffin1-pole2*⟩, and ⟨*handle2*⟩, the point of intersection of ⟨*coffin2-pole1*⟩ and ⟨*coffin2-pole2*⟩. ⟨*coffin2*⟩ is then attached to ⟨*coffin1*⟩ such that the relationship between ⟨*handle1*⟩ and ⟨*handle2*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)

\coffin_join:NnnNnnnn
⟨*coffin1*⟩ {⟨*coffin1-pole1*⟩} {⟨*coffin1-pole2*⟩}
⟨*coffin2*⟩ {⟨*coffin2-pole1*⟩} {⟨*coffin2-pole2*⟩}
{⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function joins ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin1*⟩ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating ⟨*handle1*⟩, the point of intersection of ⟨*coffin1-pole1*⟩ and ⟨*coffin1-pole2*⟩, and ⟨*handle2*⟩, the point of intersection of ⟨*coffin2-pole1*⟩ and ⟨*coffin2-pole2*⟩. ⟨*coffin2*⟩ is then attached to ⟨*coffin1*⟩ such that the relationship between ⟨*handle1*⟩ and ⟨*handle2*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

\coffin_typeset:Nnnnn ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩}
{⟨*x-offset*⟩} {⟨*y-offset*⟩}

Typesetting is carried out by first calculating ⟨*handle*⟩, the point of intersection of ⟨*pole1*⟩ and ⟨*pole2*⟩. The coffin is then typeset such that the relationship between the current reference point in the document and the ⟨*handle*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the "parent" coffin is the current insertion point.

144 Measuring coffins

\coffin_dp:N
\coffin_dp:c

\coffin_dp:N ⟨*coffin*⟩

Calculates the depth (below the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

\coffin_ht:N
\coffin_ht:c

\coffin_ht:N ⟨*coffin*⟩

Calculates the height (above the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

\coffin_wd:N
\coffin_wd:c

\coffin_wd:N ⟨*coffin*⟩

Calculates the width of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

145 Coffin diagnostics

\coffin_display_handles:cn
\coffin_display_handles:cn

Updated: 2011-09-02

\coffin_display_handles:Nn ⟨*coffin*⟩ {⟨*colour*⟩}

This function first calculates the intersections between all of the ⟨*poles*⟩ of the ⟨*coffin*⟩ to give a set of ⟨*handles*⟩. It then prints the ⟨*coffin*⟩ at the current location in the source, with the position of the ⟨*handles*⟩ marked on the coffin. The ⟨*handles*⟩ will be labelled as part of this process: the locations of the ⟨*handles*⟩ and the labels are both printed in the ⟨*colour*⟩ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn

Updated: 2011-09-02

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*colour*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole1*⟩ and ⟨*pole2*⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ will be labelled as part of this process: the location of the ⟨*handle*⟩ and the label are both printed in the ⟨*colour*⟩ specified.

\coffin_show_structure:N
\coffin_show_structure:c

Updated: 2012-01-01

\coffin_show_structure:N ⟨*coffin*⟩

This function shows the structural information about the ⟨*coffin*⟩ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

Part XVII
The l3color package Colour support

This module provides support for colour in LaTeX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

146 Colour in boxes

Controlling the colour of text in boxes requires a small number of control functions, so that the boxed material uses the colour at the point where it is set, rather than where it is used.

`\color_group_begin:`
`\color_group_end:`

New: 2011-09-03

```
\color_group_begin:
...
\color_group_end:
```

Creates a colour group: one used to "trap" colour settings.

`\color_ensure_current:`

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use the foreground colour at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin:` ...`\color_group_end:` group.

Part XVIII
The l3msg package
Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The l3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by l3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

147 Creating new messages

All messages have to be created before they can be used. All message setting is local, with the general assumption that messages will be managed as part of module set up outside of any TeX grouping.

The text of messages will automatically by wrapped to the length available in the console. As a result, formatting is only needed where it will help to show meaning. In particular, `\\` may be used to force a new line and `\␣` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the LaTeX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow only those messages from the `submodule` to be filtered out.

`\msg_new:nnnn`
`\msg_new:nnn`

Updated: 2011-08-16

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a ⟨*message*⟩ for a given ⟨*module*⟩. The message will be defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. An error will be raised if the ⟨*message*⟩ already exists.

\msg_set:nnnn
\msg_set:nnn
\msg_gset:nnnn
\msg_gset:nnn

`\msg_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a ⟨*message*⟩ for a given ⟨*module*⟩. The message will be defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used.

\msg_if_exist_p:nn ★
\msg_if_exist:nnTF ★

New: 2012-03-03

`\msg_if_exist_p:nn {⟨module⟩} {⟨message⟩}`
`\msg_if_exist:nnTF {⟨module⟩} {⟨message⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

148 Contextual information for messages

\msg_line_context: ☆

`\msg_line_context:`

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

\msg_line_number: ★

`\msg_line_number:`

Prints the current line number when a message is given.

\c_msg_return_text_tl

Standard text to indicate that the user should try pressing ⟨*return*⟩ to continue. The standard definition reads:

```
Try typing <return> to proceed.

If that doesn't work, type X <return> to quit.
```

\c_msg_trouble_text_tl

Standard text to indicate that the more errors are likely and that aborting the run is advised. The standard definition reads:

```
More errors will almost certainly follow:
the LaTeX run should be aborted.
```

\msg_fatal_text:n ★

`\msg_fatal_text:n {⟨module⟩}`

Produces the standard text:

```
Fatal <module> error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_critical_text:n` ★

`\msg_critical_text:n {`⟨*module*⟩`}`

Produces the standard text:

```
Critical <module> error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_error_text:n` ★

`\msg_error_text:n {`⟨*module*⟩`}`

Produces the standard text:

```
<module> error
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_warning_text:n` ★

`\msg_warning_text:n {`⟨*module*⟩`}`

Produces the standard text:

```
<module> warning
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

`\msg_info_text:n` ★

`\msg_info_text:n {`⟨*module*⟩`}`

Produces the standard text:

```
<module> info
```

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included.

149 Issuing messages

Messages behave differently depending on the message class. A number of standard message classes are supplied, but more can be created.

When issuing messages, any arguments passed should use `\tl_to_str:n` or `\token_to_str:N` to prevent unwanted expansion of the material.

`\msg_class_set:nn`

Updated: 2012-04-12

`\msg_class_set:nn {`⟨*class*⟩`} {`⟨*code*⟩`}`

Sets a ⟨*class*⟩ to output a message, using ⟨*code*⟩ to process the message text. The ⟨*class*⟩ should be a text value, while the ⟨*code*⟩ may be any arbitrary material. The ⟨*code*⟩ will receive 6 arguments: the module name (`#1`), the message name (`#2`) and the four arguments taken by the message text (`#3` to `#6`).

The kernel defines several common message classes. The following describes the standard behaviour of each class if no redirection of the class or message is active. In all cases, the message may be issued supplying 0 to 4 arguments. The code will ensure that there an no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

\msg_fatal:nnxxxx
\msg_fatal:(nnxxx|nnxx|nnx|nn)

`\msg_fatal:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TEX run will halt.

\msg_critical:nnxxxx
\msg_critical:(nnxxx|nnxx|nnx|nn)

`\msg_critical:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing the message reading the current input file will stop. This may halt the TEX run (if the current file is the main file) or may abort reading a sub-file.

\msg_error:nnxxxx
\msg_error:(nnxxx|nnxx|nnx|nn)

`\msg_error:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue.

\msg_warning:nnxxxx
\msg_warning:(nnxxx|nnxx|nnx|nn)

`\msg_warning:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text will be added to the log file, but the TEX run will not be interrupted.

\msg_info:nnxxxx
\msg_info:(nnxxx|nnxx|nnx|nn)

`\msg_info:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file.

\msg_log:nnxxxx
\msg_log:(nnxxx|nnxx|nnx|nn)

`\msg_log:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file: the output is briefer than `\msg_info:nnxxxx`.

\msg_none:nnxxxx
\msg_none:(nnxxx|nnxx|nnx|nn)

`\msg_none:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

150 Redirecting messages

Each message has a "name", which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this will raise an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

`\msg_redirect_class:nn`

Updated: 2012-04-12

`\msg_redirect_class:nn {⟨class one⟩} {⟨class two⟩}`

Changes the behaviour of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩. Multiple redirections are possible. Redirection to a missing class or infinite loops will raise errors when the messages are used, rather than at the point of redirection.

`\msg_redirect_module:nnn`

Updated: 2012-04-12

`\msg_redirect_module:nnn {⟨module⟩} {⟨class one⟩} {⟨class two⟩}`

Redirects message of ⟨*class one*⟩ for ⟨*module*⟩ to act as though they were from ⟨*class two*⟩. Messages of ⟨*class one*⟩ from sources other than ⟨*module*⟩ are not affected by this redirection. This function can be used to make some messages "silent" by default. For example, all of the `warning` messages of ⟨*module*⟩ could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

`\msg_redirect_name:nnn`

Updated: 2012-04-12

`\msg_redirect_name:nnn {⟨module⟩} {⟨message⟩} {⟨class⟩}`

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages. This function can be used to make a selected message "silent" without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

151 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

\msg_newline: ⋆
\msg_two_newlines: ⋆

`\msg_newline:`

Forces a new line in a message. This is a low-level function, which will not include any additional printing information in the message: contrast with `\\` in messages. The `two` version adds two lines.

\msg_interrupt:xxx

`\msg_interrupt:xxx {⟨first line⟩} {⟨text⟩} {⟨extra text⟩}`

Interrupts the TeX run, issuing a formatted message comprising ⟨*first line*⟩ and ⟨*text*⟩ laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!...............................................
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length. The user may then request more information, at which stage the ⟨*extra text*⟩ will be shown in the terminal in the format

```
|'''''''''''''''''''''''''''''''''''''''''''''''
|  <extra text>
|...............................................
```

where the ⟨*extra text*⟩ will be wrapped to fit within the current line length.

\msg_log:x

`\msg_log:x {⟨text⟩}`

Writes to the log file with the ⟨*text*⟩ laid out in the format

```
................................................
. <text>
................................................
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length.

\msg_term:x

`\msg_term:x {⟨text⟩}`

Writes to the terminal and log file with the ⟨*text*⟩ laid out in the format

```
************************************************
* <text>
************************************************
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length.

152 Kernel-specific functions

Messages from LaTeX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

`\msg_kernel_new:nnnn`
`\msg_kernel_new:nnn`

Updated: 2011-08-16

`\msg_kernel_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message will be defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used. An error will be raised if the ⟨*message*⟩ already exists.

`\msg_kernel_set:nnnn`
`\msg_kernel_set:nnn`

`\msg_kernel_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message will be defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (`#1` to `#4`) can be used: these will be supplied at the time the message is used. The parameters will be expanded when the message is used.

`\msg_kernel_fatal:nnxxxx`
`\msg_kernel_fatal:(nnxxx|nnxx|nnx|nn)`

`\msg_kernel_fatal:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TeX run will halt. Cannot be redirected.

`\msg_kernel_error:nnxxxx`
`\msg_kernel_error:(nnxxx|nnxx|nnx|nn)`

`\msg_kernel_error:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

`\msg_kernel_warning:nnxxxx`
`\msg_kernel_warning:(nnxxx|nnxx|nnx|nn)`

`\msg_kernel_warning:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text will be added to the log file, but the TeX run will not be interrupted.

`\msg_kernel_info:nnxxxx`
`\msg_kernel_info:(nnxxx|nnxx|nnx|nn)`

`\msg_kernel_info:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues kernel ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file.

153 Expandable errors

In a few places, the LaTeX3 kernel needs to produce errors in an expansion only context. This must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

`\msg_expandable_kernel_error:nnnnnn` ⋆
`\msg_expandable_kernel_error:(nnnnn|nnnn|nnn|nn)` ⋆

New: 2011-11-23

`\msg_expandable_kernel_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues an error, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The resulting string must be shorter than a line, otherwise it will be cropped.

`\msg_expandable_error:n` ⋆

New: 2011-08-11
Updated: 2011-08-13

`\msg_expandable_error:n {⟨error message⟩}`

Issues an "Undefined error" message from TeX itself, and prints the ⟨*error message*⟩. The ⟨*error message*⟩ must be short: it is cropped at the end of one line.

TeXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to TeX's prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

154 Internal l3msg functions

The following functions are used in several kernel modules.

`\msg_aux_use:nn`
`\msg_aux_use:nnxxxx`

`\msg_aux_use:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Prints the ⟨*message*⟩ from ⟨*module*⟩ in the terminal, without formatting.

`\msg_aux_show:x`

`\msg_aux_show:x {⟨formatted string⟩}`

Shows the ⟨*formatted string*⟩ on the terminal. After expansion, unless it is empty, the ⟨*formatted string*⟩ must contain `>`, and the part of ⟨*formatted string*⟩ before the first `>` is removed. Failure to do so causes low-level TeX errors.

`\msg_aux_show:Nnx`

`\msg_aux_show:Nnx ⟨variable⟩ {⟨module⟩} {⟨token list⟩}`

Auxiliary common to l3clist, l3prop and seq, which displays an appropriate message and the contents of the variable.

Part XIX
The l3keys package
Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. For the user, the system normally results in input of the form

```
\PackageControlMacro{
  key-one = value one,
  key-two = value two
}
```

or

```
\PackageMacro[
  key-one = value one,
  key-two = value two
]{argument}.
```

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { module }
  {
    key-one .code:n   = code including parameter #1,
    key-two .tl_set:N = \l_module_store_tl
  }
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { module }
  {
    key-one = value one,
    key-two = value two
  }
```

At a document level, `\keys_set:nn` will be used within a document function, for example

```
\DeclareDocumentCommand \SomePackageSetup { m }
  { \keys_set:nn { module } { #1 }  }
\DeclareDocumentCommand \SomePackageMacro { o m }
 {
   \group_begin:
```

```
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_-str:n`. As will be discussed in section 156, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module }
  {
    \l_module_tmp_tl .code:n = code
  }
```

will create a key called `\l_module_tmp_tl`, and not one called `key`.

155 Creating keys

`\keys_define:nn`

`\keys_define:nn {⟨module⟩} {⟨keyval list⟩}`

Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. The ⟨*module*⟩ name should be a text value, but there are no restrictions on the nature of the text. In practice the ⟨*module*⟩ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The ⟨*keyval list*⟩ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some~code~using~#1,
    keyname .value_required:
  }
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require exactly one argument. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary ⟨*key*⟩, which when used may be supplied with a ⟨*value*⟩. All key *definitions* are local.

`.bool_set:N`

`⟨key⟩ .bool_set:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to ⟨*value*⟩ (which must be either `true` or `false`). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*boolean*⟩ will be assigned locally.

`.bool_gset:N` ⟨*key*⟩ `.bool_gset:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to ⟨*value*⟩ (which must be either `true` or `false`). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*boolean*⟩ will be assigned globally.

`.bool_set_inverse:N` ⟨*key*⟩ `.bool_set_inverse:N = ⟨boolean⟩`

New: 2011-08-28

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to the logical inverse of ⟨*value*⟩ (which must be either `true` or `false`). If the ⟨*boolean*⟩ does not exist, it will be created at the point that the key is set up. The ⟨*boolean*⟩ will be assigned locally.

This property is experimental.

`.bool_gset_inverse:N` ⟨*key*⟩ `.bool_gset_inverse:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to the logical inverse of ⟨*value*⟩ (which must be either `true` or `false`). If the ⟨*boolean*⟩ does not exist, it will be created at the point that the key is set up. The ⟨*boolean*⟩ will be assigned globally.

This property is experimental.

`.choice:` ⟨*key*⟩ `.choice:`

Sets ⟨*key*⟩ to act as a choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 157.

`.choices:nn` ⟨*key*⟩ `.choices:nn ⟨choices⟩ ⟨code⟩`

New: 2011-08-21

Sets ⟨*key*⟩ to act as a choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 0). Choices are discussed in detail in section 157.

This property is experimental.

`.choice_code:n`
`.choice_code:x` ⟨*key*⟩ `.choice_code:n = ⟨code⟩`

Stores ⟨*code*⟩ for use when `.generate_choices:n` creates one or more choice sub-keys of the current key. Inside ⟨*code*⟩, `\l_keys_choice_tl` will expand to the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list given to `.generate_choices:n`. Choices are discussed in detail in section 157.

`.clist_set:N`
`.clist_set:c` ⟨*key*⟩ `.clist_set:N = ⟨comma list variable⟩`

New: 2011/09/11

Defines ⟨*key*⟩ to locally set ⟨*comma list variable*⟩ to ⟨*value*⟩. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created at the point that the key is set up.

`.clist_gset:N`
`.clist_gset:c` ⟨*key*⟩ `.clist_gset:N = ⟨comma list variable⟩`

New: 2011/09/11

Defines ⟨*key*⟩ to globally set ⟨*comma list variable*⟩ to ⟨*value*⟩. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created at the point that the key is set up.

`.code:n`
`.code:x`

⟨*key*⟩ `.code:n =` ⟨*code*⟩

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is used. The The ⟨*code*⟩ can include one parameter (`#1`), which will be the ⟨*value*⟩ given for the ⟨*key*⟩. The `x`-type variant will expand ⟨*code*⟩ at the point where the ⟨*key*⟩ is created.

`.default:n`
`.default:V`

⟨*key*⟩ `.default:n =` ⟨*default*⟩

Creates a ⟨*default*⟩ value for ⟨*key*⟩, which is used if no value is given. This will be used if only the key name is given, but not if a blank ⟨*value*⟩ is given:

```
\keys_define:nn { module }
  {
    key .code:n    = Hello~#1,
    key .default:n = World
  }
\keys_set:nn { module }
  {
    key = Fred, % Prints 'Hello Fred'
    key,        % Prints 'Hello World'
    key = ,     % Prints 'Hello '
  }
```

`.dim_set:N`
`.dim_set:c`

⟨*key*⟩ `.dim_set:N =` ⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*dimension*⟩ will be assigned locally.

`.dim_gset:N`
`.dim_gset:c`

⟨*key*⟩ `.dim_gset:N =` ⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*dimension*⟩ will be assigned globally.

`.fp_set:N`
`.fp_set:c`

⟨*key*⟩ `.fp_set:N =` ⟨*floating point*⟩

Defines ⟨*key*⟩ to set ⟨*floating point*⟩ to ⟨*value*⟩ (which must a floating point number). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*integer*⟩ will be assigned locally.

`.fp_gset:N`
`.fp_gset:c`

⟨*key*⟩ `.fp_gset:N =` ⟨*floating point*⟩

Defines ⟨*key*⟩ to set ⟨*floating-point*⟩ to ⟨*value*⟩ (which must a floating point number). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*integer*⟩ will be assigned globally.

`.generate_choices:n` ⟨*key*⟩ `.generate_choices:n = {`⟨*list*⟩`}`

This property will mark ⟨*key*⟩ as a multiple choice key, and will use the ⟨*list*⟩ to define the choices. The ⟨*list*⟩ should consist of a comma-separated list of choice names. Each choice will be set up to execute ⟨*code*⟩ as set using `.choice_code:n` (or `.choice_code:x`). Choices are discussed in detail in section 157.

`.int_set:N`
`.int_set:c`

⟨*key*⟩ `.int_set:N =` ⟨*integer*⟩

Defines ⟨*key*⟩ to set ⟨*integer*⟩ to ⟨*value*⟩ (which must be an integer expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*integer*⟩ will be assigned locally.

`.int_gset:N`
`.int_gset:c`

⟨*key*⟩ `.int_gset:N =` ⟨*integer*⟩

Defines ⟨*key*⟩ to set ⟨*integer*⟩ to ⟨*value*⟩ (which must be an integer expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*integer*⟩ will be assigned globally.

`.meta:n`
`.meta:x`

⟨*key*⟩ `.meta:n = {`⟨*keyval list*⟩`}`

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go. If ⟨*key*⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨*keys*⟩ for processing (as `#1`).

`.multichoice:`
New: 2011-08-21

⟨*key*⟩ `.multichoice:`

Sets ⟨*key*⟩ to act as a multiple choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 157.

This property is experimental.

`.multichoice:nn`
New: 2011-08-21

⟨*key*⟩ `.multichoice:nn` ⟨*choices*⟩ ⟨*code*⟩

Sets ⟨*key*⟩ to act as a multiple choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 0). Choices are discussed in detail in section 157.

This property is experimental.

`.skip_set:N`
`.skip_set:c`

⟨*key*⟩ `.skip_set:N =` ⟨*skip*⟩

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*skip*⟩ will be assigned locally.

`.skip_gset:N`
`.skip_gset:c`

⟨*key*⟩ `.skip_gset:N =` ⟨*skip*⟩

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*skip*⟩ will be assigned globally.

`.tl_set:N`
`.tl_set:c`

⟨*key*⟩ `.tl_set:N = ⟨token list variable⟩`

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩. If the variable does not exist, it will be created at the point that the key is set up. The ⟨*token list variable*⟩ will be assigned locally.

`.tl_gset:N`
`.tl_gset:c`

⟨*key*⟩ `.tl_gset:N = ⟨token list variable⟩`

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩. If the variable does not exist, it will be created at the point that the key is set up. The ⟨*token list variable*⟩ will be assigned globally.

`.tl_set_x:N`
`.tl_set_x:c`

⟨*key*⟩ `.tl_set_x:N = ⟨token list variable⟩`

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩, which will be subjected to an `x`-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*token list variable*⟩ will be assigned locally.

`.tl_gset_x:N`
`.tl_gset_x:c`

⟨*key*⟩ `.tl_gset_x:N = ⟨token list variable⟩`

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩, which will be subjected to an `x`-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it will be created at the point that the key is set up. The ⟨*token list variable*⟩ will be assigned globally.

`.value_forbidden:`

⟨*key*⟩ `.value_forbidden:`

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued.

`.value_required:`

⟨*key*⟩ `.value_required:`

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued.

156 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { module }
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `module/subgroup/key`.

As will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

157 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. "Multiple" choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { module }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of choices. Here, the keys can share the same code, and can be rapidly created using the `.choice_code:n` and `.generate_choices:n` properties:

```
\keys_define:nn { module }
  {
    key .choice_code:n =
      {
        You~gave~choice~'\int_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      },
    key .generate_choices:n =
      { choice-a, choice-b, choice-c }
  }
```

Following common computing practice, `\l_keys_choice_int` is indexed from 0 (as an offset), so that the value of `\l_keys_choice_int` for the first choice in a list will be zero.

The same approach is also implemented by the *experimental* property `.choices:nn`. This combines the functionality of `.choice_code:n` and `.generate_choices:n` into one property:

```
\keys_define:nn { module }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\int_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

Note that the `.choices:nn` property should *not* be mixed with use of `.generate_-choices:n`.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.generate_choice:` or `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 0.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { module }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.generate_choices:n` (*i.e.* anything might happen).

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```
\keys_define:nn { module }
  {
    key .multichoices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\int_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

and

```
\keys_define:nn { module }
  {
    key .multichoice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

are valid. The .multichoices:nn property causes \l_keys_choice_tl and \l_keys_-choice_int to be set in exactly the same way as described for .choices:nn.

When multiple choice keys are set, the value is treated as a comma-separated list:

```
\keys_set:nn { module }
  {
    key = { a , b , c } % 'key' defined as a multiple choice
  }
```

Each choice will be applied in turn, with the usual handling of unknown values.

158 Setting keys

\keys_set:nn
\keys_set:(nV|nv|no)

\keys_set:nn {⟨*module*⟩} {⟨*keyval list*⟩}

Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. The behaviour on finding an unknown key can be set by defining a special unknown key: this will be illustrated later.

If a key is not known, \keys_set:nn will look for a special unknown key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_tl'~to~'#1'.
  }
```

\l_keys_key_tl

When processing an unknown key, the name of the key is available as \l_keys_key_tl. Note that this will have been processed using \tl_to_str:n.

\l_keys_path_tl

When processing an unknown key, the path of the key used is available as \l_keys_-path_tl. Note that this will have been processed using \tl_to_str:n.

\l_keys_value_tl

When processing an unknown key, the value of the key is available as \l_keys_value_tl. Note that this will be empty if no value was given for the key.

159 Setting known keys only

The functionality described in this section is experimental and may be altered or removed, depending on feedback.

\keys_set_known:nnN
\keys_set_known:(nVN|nvN|noN)

New: 2011-08-23

\keys_set_known:nn {⟨*module*⟩} {⟨*keyval list*⟩} ⟨*clist*⟩

Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. Any keys which are unknown are not processed further by the parser. The key–value pairs for each *unknown* key name will be stored in the ⟨*clist*⟩.

160 Utility functions for keys

\keys_if_exist_p:nn ⋆
\keys_if_exist:nnTF ⋆

\keys_if_exist_p:nn ⟨*module*⟩ ⟨*key*⟩
\keys_if_exist:nnTF ⟨*module*⟩ ⟨*key*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ exists for ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩.

\keys_if_choice_exist_p:nn ⋆
\keys_if_choice_exist:nnTF ⋆

New: 2011-08-21

\keys_if_choice_exist_p:nnn ⟨*module*⟩ ⟨*key*⟩ ⟨*choice*⟩
\keys_if_choice_exist:nnnTF ⟨*module*⟩ ⟨*key*⟩ ⟨*choice*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*choice*⟩ is defined for the ⟨*key*⟩ within the ⟨*module*⟩,, *i.e.* if any code has been defined for ⟨*key*⟩/⟨*choice*⟩. The test is `false` if the ⟨*key*⟩ itself is not defined.

\keys_show:nn

\keys_show:nn {⟨*module*⟩} {⟨*key*⟩}

Shows the function which is used to actually implement a ⟨*key*⟩ for a ⟨*module*⟩.

161 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in especial circumstances you may wish to use a lower-level approach. The low-level parsing system converts a ⟨*key–value list*⟩ into ⟨*keys*⟩ and associated ⟨*values*⟩. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (*i.e* two arguments), and a second function if required for keys given without arguments (*i.e.* a single argument).

The parser does not double `#` tokens or expand any input. The tokens `=` and `,` are corrected so that the parser does not "miss" any due to category code changes. Spaces are removed from the ends of the keys and values. Values which are given in braces will have exactly one set removed, thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn`

Updated: 2011-09-08

`\keyval_parse:NNn` ⟨*function1*⟩ ⟨*function2*⟩ {⟨*key-value list*⟩}

Parses the ⟨*key–value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function1*⟩ should take one argument, while ⟨*function2*⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key–value list*⟩, ⟨*function1*⟩ will be used to process keys given with no value and ⟨*function2*⟩ will be used to process keys given with a value. The order of the ⟨*keys*⟩ in the ⟨*key–value list*⟩ will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

will be converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n  { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, and any *outer* set of braces are removed from the ⟨*value*⟩ as part of the processing.

Part XX
The l3file package
File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files TEX will attempt to locate them both the operating system path and entries in the TEX file database (most TEX systems use such a database). Thus the "current path" for TEX is somewhat broader than that for other programs.

162 File operation functions

`\g_file_current_name_tl`

Contains the name of the current LATEX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_job_name_tl` at the start of a LATEX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`

Updated: 2012-02-10

`\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`).

TEXhackers note: The ⟨*file name*⟩ may contain both literal items and expandable content, which should on full expansion be the desired file name. The expansion occurs when TEX searches for the file.

`\file_add_path:nN`

Updated: 2012-02-10

`\file_add_path:nN {⟨file name⟩} ⟨tl var⟩`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found sets the ⟨*tl var*⟩ the fully-qualified name of the file, *i.e.* the path and file name. If the file is not found then the ⟨*tl var*⟩ will contain the marker `\q_no_value`.

TEXhackers note: The ⟨*file name*⟩ may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names.

\file_input:n

Updated: 2012-02-17

`\file_input:n {⟨file name⟩}`

Searches for ⟨*file name*⟩ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error will be raised if the file is not found

TeXhackers note: The ⟨*file name*⟩ may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names.

\file_path_include:n

`\file_path_include:n {⟨path⟩}`

Adds ⟨*path*⟩ to the list of those used to search when reading files. The assignment is local.

\file_path_remove:n

`\file_path_remove:n {⟨path⟩}`

Removes ⟨*path*⟩ from the list of those used to search when reading files. The assignment is local.

\file_list:

`\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

162.1 Input–output stream management

As TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in LaTeX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

New: 2011-09-26
Updated: 2011-12-27

`\ior_new:Nn ⟨stream⟩`

Globally reserves the name of the ⟨*stream*⟩, either for reading or for writing as appropriate. The ⟨*stream*⟩ is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a ⟨*stream*⟩ which has not been opened will result in a TeX error.

\ior_open:Nn
\ior_open:cn

Updated: 2012-02-10

\ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a \ior_close:N instruction is given or the file ends.

TEXhackers note: The ⟨*file name*⟩ may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in \l_char_active_seq) will *not* be expanded, allowing the direct use of these in file names.

\iow_open:Nn
\iow_open:cn

Updated: 2012-02-09

\iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for writing using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a \iow_close:N instruction is given or the file ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

TEXhackers note: The ⟨*file name*⟩ may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in \l_char_active_seq) will *not* be expanded, allowing the direct use of these in file names.

\ior_close:N
\ior_close:c

Updated: 2011-12-27

\ior_close:N ⟨*stream*⟩

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmer.

\iow_close:N
\iow_close:c

Updated: 2011-12-27

\iow_close:N ⟨*stream*⟩

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmer.

\ior_list_streams:
\iow_list_streams:

\ior_list_streams:
\iow_list_streams:

Displays a list of the file names associated with each open stream: intended for tracking down problems.

163 Reading from files

\ior_to:NN
\ior_gto:NN

`\ior_to:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input ⟨*stream*⟩ and stores the result in the ⟨*token list*⟩ variable, locally or globally. If the ⟨*stream*⟩ is not open, input is requested from the terminal. The material read from the ⟨*stream*⟩ will be tokenized by TeX according to the category codes in force when the function is used.

TeXhackers note: This protected macro expands to the TeX primitives `\read` or `\global\read` along with the `to` keyword.

\ior_str_to:NN
\ior_str_gto:NN

`\ior_str_to:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

Functions that reads one line from the input ⟨*stream*⟩ and stores the result in the ⟨*token list*⟩ variable, locally or globally. If the ⟨*stream*⟩ is not open, input is requested from the terminal. The material read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space).

TeXhackers note: This protected macro expands to the ε-TeX primitives `\readline` or `\global\readline` along with the `to` keyword.

\ior_if_eof_p:N ⋆
\ior_if_eof:NTF ⋆

Updated: 2012-02-10

`\ior_if_eof_p:N` ⟨*stream*⟩
`\ior_if_eof:NTF` ⟨*stream*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test will also return a `true` value if the ⟨*stream*⟩ is not open.

164 Writing to files

\iow_now:Nn
\iow_now:Nx

`\iow_now:Nn` ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

TeXhackers note: `\iow_now:Nx` is a protected macro which expands to the two TeX primitives `\immediate\write`.

\iow_log:n
\iow_log:x

`\iow_log:n` {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

\iow_term:n
\iow_term:x

\iow_term:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

\iow_shipout:Nn
\iow_shipout:Nx

\iow_shipout:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The x-type variants expand the ⟨*tokens*⟩ at the point where the function is used but *not* when the resulting tokens are written to the ⟨*stream*⟩ (*cf.* \iow_shipout_x:Nn).

\iow_shipout_x:Nn
\iow_shipout_x:Nx

\iow_shipout_x:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The ⟨*tokens*⟩ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

> **TEXhackers note:** \iow_shipout_x:Nn is the TEX primitive \write renamed.

\iow_char:N ⋆

\iow_char:N ⟨*token*⟩

Inserts ⟨*token*⟩ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of \iow_now:Nn).

\iow_newline: ⋆

\iow_newline:

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of \iow_now:Nn).

165 Wrapping lines in output

`\iow_wrap:xnnnN`

Updated: 2011-09-21

`\iow_wrap:xnnnN {⟨text⟩} {⟨run-on text⟩} {⟨run-on length⟩} {⟨set up⟩} ⟨function⟩`

This function will wrap the ⟨*text*⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨*run-on text*⟩ will be inserted. The line length targeted will be the value of `\l_iow_line_length_int` minus the ⟨*run-on length*⟩. The later value should be the number of characters in the ⟨*run-on text*⟩. Additional functions may be added to the wrapping by using the ⟨*set up*⟩, which is executed before the wrapping takes place. The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which will typically be a wrapper around a writing operation. Within the ⟨*text*⟩,

- `\\` may be used to force a new line,
- `\ ` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_indent:n` may be used to indent a part of the message.

Both the wrapping process and the subsequent write operation will perform `x`-type expansion. For this reason, material which is to be written "as is" should be given as the argument to `\token_to_str:N` or `\tl_to_str:n` (as appropriate) within the ⟨*text*⟩. The output of `\iow_wrap:xnnnN` (*i.e.* the argument passed to the ⟨*function*⟩) will consist of characters of category code 12 (other) and 10 (space) only. This means that the output will *not* expand further when written to a file.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n {⟨text⟩}`

In the context of `\iow_wrap:xnnnN` (for instance in messages), indents ⟨*text*⟩ by four spaces. This function will not cause a line break, and only affects lines which start within the scope of the ⟨*text*⟩. In case the indented ⟨*text*⟩ should appear on separate lines from the surrounding text, use `\\` to force line breaks.

`\l_iow_line_length_int`

The maximum length of a line to be written by the `\iow_wrap:xxnnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeXlive and MiKTeX systems.

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, ("other"), and character code 32 (space).

166 Constant input–output streams

`\c_term_ior` Constant input stream for reading from the terminal. Reading from this stream using `\ior_to:NN` or similar will result in a prompt from TeX of the form

```
<tl>=
```

`\c_log_iow`
`\c_term_iow` Constant output streams for writing to the log and to the terminal (plus the log), respectively.

167 Experimental functions

`\ior_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn` ⟨*stream*⟩ `{`⟨*inline function*⟩`}`

Applies the ⟨*inline function*⟩ to ⟨*items*⟩ obtained by reading one or more lines (until an equal number of left and right braces are found) from the ⟨*stream*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*line*⟩ as `#1`.

`\ior_str_map_inline:nn`

New: 2012-02-11

`\ior_str_map_inline:Nn {`⟨*stream*⟩`} {`⟨*inline function*⟩`}`

Applies the ⟨*inline function*⟩ to every ⟨*line*⟩ in the ⟨*file*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*inline function*⟩ should consist of code which will receive the ⟨*line*⟩ as `#1`.

168 Internal file functions

`\g_file_stack_seq` Stores the stack of nested files loaded using `\file_input:n`. This is needed to restore the appropriate file name to `\g_file_current_name_tl` at the end of each file.

`\g_file_record_seq` Stores the name of every file loaded using `\file_input:n`. In contrast to `\g_file_stack_seq`, no items are ever removed from this sequence.

`\l_file_internal_name_tl` Used to return the full name of a file for internal use.

`\l_file_search_path_seq` The sequence of file paths to search when loading a file.

`\l_file_internal_saved_path_seq`

When loaded on top of LaTeX 2_ε, there is a need to save the search path so that `\input@path` can be used as appropriate.

`\l_file_internal_seq`

New: 2011-09-06

When loaded on top of LaTeX 2ε, there is a need to convert the comma lists `\input@path` and `\@filelist` to sequences.

169 Internal input–output functions

`\file_name_sanitize:nn`

New: 2012-02-09

`\file_name_sanitize:nn {⟨name⟩} {⟨tokens⟩}`

Exhaustively-expands the ⟨*name*⟩ with the exception of any category ⟨*active*⟩ (catcode 12) tokens, which are not expanded. The list of ⟨*active*⟩ tokens is taken from `\l_char_active_seq`. The ⟨*sanitized name*⟩ is then inserted (in braces) after the ⟨*tokens*⟩, which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

`\if_eof:w` ⋆

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

Tests if the ⟨*stream*⟩ returns "end of file", which is true for non-existent files. The `\else:` branch is optional.

> **TeXhackers note:** This is the TeX primitive `\ifeof`.

`\ior_open_unsafe:Nn`
`\ior_open_unsafe:No`
`\iow_open_unsafe:Nn`

New: 2012-01-23

`\ior_open_unsafe:Nn ⟨stream⟩ {⟨file name⟩}`

These functions have identical syntax to the generally-available versions without the `_unsafe` suffix. However, these functions do not take precautions against active characters in the ⟨*file name*⟩: they are therefore intended to be used by higher-level functions which have already fully expanded the ⟨*file name*⟩ and which need to perform multiple open or close operations. See for example the implementation of `\file_add_path:Nn`,

`\ior_raw_new:N`
`\ior_raw_new:c`

`\ior_raw_new:N ⟨stream⟩`

Directly allocates a new stream for reading, bypassing the stack system. This is to be used only when a new stream is required at a TeX level, when a new stream is requested by the stack itself.

`\iow_raw_new:N`
`\iow_raw_new:c`

`\iow_raw_new:N ⟨stream⟩`

Directly allocates a new stream for writing, bypassing the stack system. This is to be used only when a new stream is required at a TeX level, when a new stream is requested by the stack itself.

Part XXI

The l3fp package
Floating-point operations

A floating point number is one which is stored as a mantissa and a separate exponent. This module implements arithmetic using radix 10 floating point numbers. This means that the mantissa should be a real number in the range $1 \leq |x| < 10$, with the exponent given as an integer between -99 and 99. In the input, the exponent part is represented starting with an `e`. As this is a low-level module, error-checking is minimal. Numbers which are too large for the floating point unit to handle will result in errors, either from TeX or from LaTeX. The LaTeX code does not check that the input will not overflow, hence the possibility of a TeX error. On the other hand, numbers which are too small will be dropped, which will mean that extra decimal digits will simply be lost.

When parsing numbers, any missing parts will be interpreted as zero. So for example

```
\fp_set:Nn \l_my_fp { }
\fp_set:Nn \l_my_fp { . }
\fp_set:Nn \l_my_fp { - }
```

will all be interpreted as zero values without raising an error.

Operations which give an undefined result (such as division by 0) will not lead to errors. Instead special marker values are returned, which can be tested for using for example `\fp_if_undefined:N(TF)`. In this way it is possible to work with asymptotic functions without first checking the input. If these special values are carried forward in calculations they will be treated as 0.

Floating point numbers are stored in the `fp` floating point variable type. This has a standard range of functions for variable management.

170 Floating-point variables

`\fp_new:N`
`\fp_new:c`

`\fp_new:N` ⟨*floating point variable*⟩

Creates a new ⟨*floating point variable*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*floating point*⟩ will initially be set to `+0.000000000e0` (the zero floating point).

`\fp_const:Nn`
`\fp_const:cn`

`\fp_const:Nn` ⟨*floating point variable*⟩ {⟨*value*⟩}

Creates a new constant ⟨*floating point variable*⟩ or raises an error if the name is already taken. The value of the ⟨*floating point variable*⟩ will be set globally to the ⟨*value*⟩.

`\fp_set_eq:NN`
`\fp_set_eq:(cN|Nc|cc)`
`\fp_gset_eq:NN`
`\fp_gset_eq:(cN|Nc|cc)`

`\fp_set_eq:NN` ⟨*fp var1*⟩ ⟨*fp var2*⟩

Sets the value of ⟨*floating point variable1*⟩ equal to that of ⟨*floating point variable2*⟩.

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

\fp_zero:N ⟨*floating point variable*⟩

Sets the ⟨*floating point variable*⟩ to +0.000000000e0.

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

New: 2012-01-07

\fp_zero_new:N ⟨*floating point variable*⟩

Ensures that the ⟨*floating point variable*⟩ exists globally by applying \fp_new:N if necessary, then applies \fp_(g)zero:N to leave the ⟨*floating point variable*⟩ set to zero.

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn

\fp_set:Nn ⟨*floating point variable*⟩ {⟨*value*⟩}

Sets the ⟨*floating point variable*⟩ variable to ⟨*value*⟩.

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn

\fp_set_from_dim:Nn ⟨*floating point variable*⟩ {⟨*dimexpr*⟩}

Sets the ⟨*floating point variable*⟩ to the distance represented by the ⟨*dimension expression*⟩ in the units points. This means that distances given in other units are first converted to points before being assigned to the ⟨*floating point variable*⟩.

\fp_use:N ☆
\fp_use:c ☆

\fp_use:N ⟨*floating point variable*⟩

Inserts the value of the ⟨*floating point variable*⟩ into the input stream. The value will be given as a real number without any exponent part, and will always include a decimal point. For example,

```
\fp_new:Nn \test
\fp_set:Nn \test { 1.234 e 5 }
\fp_use:N \test
```

will insert 12345.00000 into the input stream. As illustrated, a floating point will always be inserted with ten significant digits given. Very large and very small values will include additional zeros for place value.

\fp_show:N
\fp_show:c

\fp_show:N ⟨*floating point variable*⟩

Displays the content of the ⟨*floating point variable*⟩ on the terminal.

\fp_if_exist_p:N ★
\fp_if_exist_p:c ★
\fp_if_exist:NTF ★
\fp_if_exist:cTF ★

New: 2012-03-03

\fp_if_exist_p:N ⟨*fp var*⟩
\fp_if_exist:NTF ⟨*fp var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*fp var*⟩ is currently defined. This does not check that the ⟨*fp var*⟩ really is a floating point variable.

171 Conversion of floating point values to other formats

It is useful to be able to convert floating point variables to other forms. These functions are expandable, so that the material can be used in a variety of contexts. The \fp_use:N function should also be consulted in this context, as it will insert the value of the floating point variable as a real number.

\fp_to_dim:N ☆
\fp_to_dim:c ☆

`\fp_to_dim:N` ⟨*floating point variable*⟩

Inserts the value of the ⟨*floating point variable*⟩ into the input stream converted into a dimension in points.

\fp_to_int:N ☆
\fp_to_int:c ☆

`\fp_to_int:N` ⟨*floating point variable*⟩

Inserts the integer value of the ⟨*floating point variable*⟩ into the input stream. The decimal part of the number will not be included, but will be used to round the integer.

\fp_to_tl:N ☆
\fp_to_tl:c ☆

`\fp_to_tl:N` ⟨*floating point variable*⟩

Inserts a representation of the ⟨*floating point variable*⟩ into the input stream as a token list. The representation follows the conventions of a pocket calculator:

Floating point value	Representation
1.234000000000e0	1.234
-1.234000000000e0	-1.234
1.234000000000e3	1234
1.234000000000e13	1234e13
1.234000000000e-1	0.1234
1.234000000000e-2	0.01234
1.234000000000e-3	1.234e-3

Notice that trailing zeros are removed in this process, and that numbers which do not require a decimal part do *not* include a decimal marker.

172 Rounding floating point values

The module can round floating point values to either decimal places or significant figures using the usual method in which exact halves are rounded up.

\fp_round_figures:Nn
\fp_round_figures:cn
\fp_ground_figures:Nn
\fp_ground_figures:cn

`\fp_round_figures:Nn` ⟨*floating point variable*⟩ {⟨*target*⟩}

Rounds the ⟨*floating point variable*⟩ to the ⟨*target*⟩ number of significant figures (an integer expression).

\fp_round_places:Nn
\fp_round_places:cn
\fp_ground_places:Nn
\fp_ground_places:cn

\fp_round_places:Nn ⟨*floating point variable*⟩ {⟨*target*⟩}

Rounds the ⟨*floating point variable*⟩ to the ⟨*target*⟩ number of decimal places (an integer expression).

173 Floating-point conditionals

\fp_if_undefined_p:N ⋆
\fp_if_undefined:NTF ⋆

\fp_if_undefined_p:N ⟨*fixed-point*⟩
\fp_if_undefined:NTF ⟨*fixed-point*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*floating point*⟩ is undefined (*i.e.* equal to the special \c_undefined_fp variable).

\fp_if_zero_p:N ⋆
\fp_if_zero:NTF ⋆

\fp_if_zero_p:N ⟨*fixed-point*⟩
\fp_if_zero:NTF ⟨*fixed-point*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*floating point*⟩ is equal to zero (*i.e.* equal to the special \c_zero_fp variable).

\fp_compare:nNnTF

\fp_compare:nNnTF
  {⟨*floating point1*⟩} ⟨*relation*⟩ {⟨*floating point2*⟩}
  {⟨*true code*⟩} {⟨*false code*⟩}

This function compared the two ⟨*floating point*⟩ values, which may be stored as fp variables, using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

\fp_compare:nTF

\fp_compare:nTF
  { ⟨*floating point1*⟩ ⟨*relation*⟩ ⟨*floating point2*⟩ }
  {⟨*true code*⟩} {⟨*false code*⟩}

This function compared the two ⟨*floating point*⟩ values, which may be stored as fp variables, using the ⟨*relation*⟩:

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

The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

174 Unary floating-point operations

The unary operations alter the value stored within an `fp` variable.

`\fp_abs:N`
`\fp_abs:c`
`\fp_gabs:N`
`\fp_gabs:c`

`\fp_abs:N ⟨floating point variable⟩`

Converts the ⟨*floating point variable*⟩ to its absolute value.

`\fp_neg:N`
`\fp_neg:c`
`\fp_gneg:N`
`\fp_gneg:c`

`\fp_neg:N ⟨floating point variable⟩`

Reverse the sign of the ⟨*floating point variable*⟩.

175 Floating-point arithmetic

Binary arithmetic operations act on the value stored in an `fp`, so for example

```
\fp_set:Nn \l_my_fp { 1.234 }
\fp_sub:Nn \l_my_fp { 5.678 }
```

sets `\l_my_fp` to the result of $1.234 - 5.678$ (*i.e.* -4.444).

`\fp_add:Nn`
`\fp_add:cn`
`\fp_gadd:Nn`
`\fp_gadd:cn`

`\fp_add:Nn ⟨floating point⟩ {⟨value⟩}`

Adds the ⟨*value*⟩ to the ⟨*floating point*⟩.

`\fp_sub:Nn`
`\fp_sub:cn`
`\fp_gsub:Nn`
`\fp_gsub:cn`

`\fp_sub:Nn ⟨floating point⟩ {⟨value⟩}`

Subtracts the ⟨*value*⟩ from the ⟨*floating point*⟩.

`\fp_mul:Nn`
`\fp_mul:cn`
`\fp_gmul:Nn`
`\fp_gmul:cn`

`\fp_mul:Nn ⟨floating point⟩ {⟨value⟩}`

Multiples the ⟨*floating point*⟩ by the ⟨*value*⟩.

`\fp_div:Nn`
`\fp_div:cn`
`\fp_gdiv:Nn`
`\fp_gdiv:cn`

`\fp_div:Nn ⟨floating point⟩ {⟨value⟩}`

Divides the ⟨*floating point*⟩ by the ⟨*value*⟩, making the assignment within the current TeX group level. If the ⟨*value*⟩ is zero, the ⟨*floating point*⟩ will be set to `\c_undefined_fp`.

176 Floating-point power operations

\fp_pow:Nn
\fp_pow:cn
\fp_gpow:Nn
\fp_gpow:cn

`\fp_pow:Nn ⟨floating point⟩ {⟨value⟩}`

Raises the ⟨*floating point*⟩ to the given ⟨*value*⟩. If the ⟨*floating point*⟩ is negative, then the ⟨*value*⟩ should be either a positive real number or a negative integer. If the ⟨*floating point*⟩ is positive, then the ⟨*value*⟩ may be any real value. Mathematically invalid operations such as 0^0 will give set the ⟨*floating point*⟩ to to `\c_undefined_fp`.

177 Exponential and logarithm functions

\fp_exp:Nn
\fp_exp:cn
\fp_gexp:Nn
\fp_gexp:cn

`\fp_exp:Nn ⟨floating point⟩ {⟨value⟩}`

Calculates the exponential of the ⟨*value*⟩ and assigns this to the ⟨*floating point*⟩.

\fp_ln:Nn
\fp_ln:cn
\fp_gln:Nn
\fp_gln:cn

`\fp_ln:Nn ⟨floating point⟩ {⟨value⟩}`

Calculates the natural logarithm of the ⟨*value*⟩ and assigns this to the ⟨*floating point*⟩.

178 Trigonometric functions

The trigonometric functions all work in radians. They accept a maximum input value of 100 000 000, as there are issues with range reduction and very large input values.

\fp_sin:Nn
\fp_sin:cn
\fp_gsin:Nn
\fp_gsin:cn

`\fp_sin:Nn ⟨floating point⟩ {⟨value⟩}`

Assigns the sine of the ⟨*value*⟩ to the ⟨*floating point*⟩. The ⟨*value*⟩ should be given in radians.

\fp_cos:Nn
\fp_cos:cn
\fp_gcos:Nn
\fp_gcos:cn

`\fp_cos:Nn ⟨floating point⟩ {⟨value⟩}`

Assigns the cosine of the ⟨*value*⟩ to the ⟨*floating point*⟩. The ⟨*value*⟩ should be given in radians.

\fp_tan:Nn
\fp_tan:cn
\fp_gtan:Nn
\fp_gtan:cn

`\fp_tan:Nn ⟨floating point⟩ {⟨value⟩}`

Assigns the tangent of the ⟨*value*⟩ to the ⟨*floating point*⟩. The ⟨*value*⟩ should be given in radians.

179 Constant floating point values

`\c_e_fp` The value of the base of natural numbers, e.

`\c_one_fp` A floating point variable with permanent value 1: used for speeding up some comparisons.

`\c_pi_fp` The value of π.

`\c_undefined_fp` A special marker floating point variable representing the result of an operation which does not give a defined result (such as division by 0).

`\c_zero_fp` A permanently zero floating point variable.

180 Notes on the floating point unit

As calculation of the elemental transcendental functions is computationally expensive compared to storage of results, after calculating a trigonometric function, exponent, *etc.* the module stored the result for reuse. Thus the performance of the module for repeated operations, most probably trigonometric functions, should be much higher than if the values were re-calculated every time they were needed.

Anyone with experience of programming floating point calculations will know that this is a complex area. The aim of the unit is to be accurate enough for the likely applications in a typesetting context. The arithmetic operations are therefore intended to provide ten digit accuracy with the last digit accurate to ± 1. The elemental transcendental functions may not provide such high accuracy in every case, although the design aim has been to provide 10 digit accuracy for cases likely to be relevant in typesetting situations. A good overview of the challenges in this area can be found in J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd edition, Birkhäuer Boston, New York, USA, 2006.

The internal representation of numbers is tuned to the needs of the underlying TeX system. This means that the format is somewhat different from that used in, for example, computer floating point units. Programming in TeX makes it most convenient to use a radix 10 system, using TeX `count` registers for storage and taking advantage where possible of delimited arguments.

Part XXII
The l3luatex package LuaTeX-specific functions

181 Breaking out to Lua

The LuaTeX engine provides access to the Lua programming language, and with it access to the "internals" of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX or XeTeX these will raise an error: use `\luatex_if_engine:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

`\lua_now:n` ⋆
`\lua_now:x` ⋆

`\lua_now:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which will include converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ immediately, and in an expandable manner.

TeXhackers note: `\lua_now:x` is the LuaTeX primitive `\directlua` renamed.

`\lua_shipout:n`
`\lua_shipout:x`

`\lua_shipout:x {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which will include converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ during the page-building routine: no TeX expansion of the ⟨*Lua input*⟩ will occur at this stage.

TeXhackers note: At a TeX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

\lua_shipout_x:n
\lua_shipout_x:x

\lua_shipout:n {⟨*token list*⟩}

The ⟨*token list*⟩ is first tokenized by TeX, which will include converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each \lua_shipout:n block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ during the page-building routine: the ⟨*Lua input*⟩ is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).

TeXhackers note: \lua_shipout_x:n is the LuaTeX primitive \latelua named using the LaTeX3 scheme.

At a TeX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

182 Category code tables

As well as providing methods to break out into Lua, there are places where additional LaTeX3 functions are provided by the LuaTeX engine. In particular, LuaTeX provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by \ExplSyntaxOn and ExplSyntaxOff when using the LuaTeX engine.

\cctab_new:N

\cctab_new:N ⟨*category code table*⟩

Creates a new category code table, initially with the codes as used by IniTeX.

\cctab_gset:Nn

\cctab_gset:Nn ⟨*category code table*⟩ {⟨*category code set up*⟩}

Sets the ⟨*category code table*⟩ to apply the category codes which apply when the prevailing regime is modified by the ⟨*category code set up*⟩. Thus within a standard code block the starting point will be the code applied by \c_code_cctab. The assignment of the table is global: the underlying primitive does not respect grouping.

\cctab_begin:N

\cctab_begin:N ⟨*category code table*⟩

Switches the category codes in force to those stored in the ⟨*category code table*⟩. The prevailing codes before the function is called are added to a stack, for use with \cctab_-end:.

\cctab_end:

\cctab_end:

Ends the scope of a ⟨*category code table*⟩ started using \cctab_begin:N, retuning the codes to those in force before the matching \cctab_begin:N was used.

\c_code_cctab

Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by \ExplSyntaxOn.

`\c_document_cctab` Category code table for a standard LaTeX document. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOff`.

`\c_initex_cctab` Category code table as set up by IniTeX.

`\c_other_cctab` Category code table where all characters have category code 12 (other).

`\c_str_cctab` Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIII
Implementation

183 l3bootstrap implementation

```
⟨*initex | package⟩
```

183.1 Format-specific code

The very first thing to do is to bootstrap the IniTeX system so that everything else will actually work. TeX does not start with some pretty basic character codes set up.

```
⟨*initex⟩
\catcode `\{ = 1 \relax
\catcode `\} = 2 \relax
\catcode `\# = 6 \relax
\catcode `\^ = 7 \relax
⟨/initex⟩
```

Tab characters should not show up in the code, but to be on the safe side.

```
⟨*initex⟩
\catcode `\^^I = 10 \relax
⟨/initex⟩
```

For LuaTeX the extra primitives need to be enabled before they can be use. No `\ifdefined` yet, so do it the old-fashioned way. The primitive `\strcmp` is simulated using some Lua code, which currently has to be applied to every job as the Lua code is not part of the format. Thanks to Taco Hoekwater for this code. The odd `\csname` business is needed so that the later deletion code will work.

```
⟨*initex⟩
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
\else
  \directlua
    {
```

```
      tex.enableprimitives('',tex.extraprimitives ())
      lua.bytecode[1] = function ()
        function strcmp (A, B)
          if A == B then
            tex.write("0")
          elseif A < B then
            tex.write("-1")
          else
            tex.write("1")
          end
        end
      end
      lua.bytecode[1]()
    }
  \everyjob\expandafter
    {\csname\detokenize{luatex_directlua:D}\endcsname{lua.bytecode[1]()}}
  \long\edef\pdfstrcmp#1#2%
     {%
       \expandafter\noexpand\csname\detokenize{luatex_directlua:D}\endcsname
         {%
           strcmp%
             (%
               "\noexpand\luaescapestring{#1}",%
               "\noexpand\luaescapestring{#2}"%
             )%
         }%
     }
\fi
⟨/initex⟩
```

183.2 Package-specific code

The package starts by identifying itself: the information itself is taken from the SVN `Id` string at the start of the source file.

```
⟨*package⟩
\ProvidesPackage{l3bootstrap}
  [%
    \ExplFileDate\space v\ExplFileVersion\space
    L3 Experimental bootstrap code%
  ]
⟨/package⟩
```

For LuaTEX the functionality of the `\pdfstrcmp` primitive needs to be provided: the pdftexmcds package is used to do this if necessary. At present, there is also a need to deal with some low-level allocation stuff that could usefully be added to `lualatex.ini`. As it is currently not, load Heiko Oberdiek's luatex package instead.

```
⟨*package⟩
\def\@tempa%
  {%
```

```
    \def\@tempa{}%
    \RequirePackage{luatex}%
    \RequirePackage{pdftexcmds}%
    \let\pdfstrcmp\pdf@strcmp
  }
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
\else
  \expandafter\@tempa
\fi
⟨/package⟩
```

\ExplSyntaxOff
\ExplSyntaxOn

Experimental syntax switching is set up here for the package-loading process. These are redefined in expl3 for the package and in l3final for the format.

```
⟨*package⟩
\protected\edef\ExplSyntaxOff
  {%
    \catcode   9 = \the\catcode   9\relax
    \catcode  32 = \the\catcode  32\relax
    \catcode  34 = \the\catcode  34\relax
    \catcode  38 = \the\catcode  38\relax
    \catcode  58 = \the\catcode  58\relax
    \catcode  94 = \the\catcode  94\relax
    \catcode  95 = \the\catcode  95\relax
    \catcode 124 = \the\catcode 124\relax
    \catcode 126 = \the\catcode 126\relax
    \endlinechar = \the\endlinechar\relax
    \chardef\csname\detokenize{l_expl_status_bool}\endcsname = 0 \relax
  }
\protected\edef\ExplSyntaxOn
  {
    \catcode 9   = 9  \relax
    \catcode 32  = 9  \relax
    \catcode 34  = 12 \relax
    \catcode 58  = 11 \relax
    \catcode 94  = 7  \relax
    \catcode 95  = 11 \relax
    \catcode 124 = 12 \relax
    \catcode 126 = 10 \relax
    \endlinechar = 32 \relax
    \chardef\csname\detokenize{l_expl_status_bool}\endcsname = 1 \relax
  }
⟨/package⟩
```

(*End definition for* `\ExplSyntaxOff` *and* `\ExplSyntaxOn`*. These functions are documented on page 6.*)

\l_expl_status_bool

The status for experimental code syntax: this is off at present. This code is used by both the package and the format.

```
\expandafter\chardef\csname\detokenize{l_expl_status_bool}\endcsname = 0 \relax
```

(*End definition for* `\l_expl_status_bool`*. This function is documented on page* **??**.)

183.3 Dealing with package-mode meta-data

\GetIdInfo
\GetIdInfoFull
\GetIdInfoAuxI
\GetIdInfoAuxII
\GetIdInfoAuxIII
\GetIdInfoAuxCVS
\GetIdInfoAuxSVN

Functions for collecting up meta-data from the SVN information used by the LaTeX3 Project.

```
⟨*package⟩
\protected\def\GetIdInfo
  {
    \begingroup
    \catcode 32 = 10 \relax
    \GetIdInfoAuxI
  }
\protected\def\GetIdInfoAuxI$#1$#2%
  {
    \def\tempa{#1}%
    \def\tempb{Id}%
    \ifx\tempa\tempb
      \def\tempa
        {%
          \endgroup
          \def\ExplFileDate{9999/99/99}%
          \def\ExplFileDescription{#2}%
          \def\ExplFileName{[unknown name]}%
          \def\ExplFileVersion{999}%
        }%
    \else
      \def\tempa
        {%
          \endgroup
          \GetIdInfoAuxII$#1${#2}%
        }%
    \fi
    \tempa
  }
\protected\def\GetIdInfoAuxII$#1 #2.#3 #4 #5 #6 #7 #8$#9%
  {%
    \def\ExplFileName{#2}%
    \def\ExplFileVersion{#4}%
    \def\ExplFileDescription{#9}%
    \GetIdInfoAuxIII#5\relax#3\relax#5\relax#6\relax
  }
\protected\def\GetIdInfoAuxIII#1#2#3#4#5#6\relax
  {%
    \ifx#5/%
      \expandafter\GetIdInfoAuxCVS
    \else
      \expandafter\GetIdInfoAuxSVN
    \fi
  }
\protected\def\GetIdInfoAuxCVS#1,v\relax#2\relax#3\relax
  {\def\ExplFileDate{#2}}
```

```
\protected\def\GetIdInfoAuxSVN#1\relax#2-#3-#4\relax#5Z\relax
  {\def\ExplFileDate{#2/#3/#4}}
⟨/package⟩
```

(*End definition for* `\GetIdInfo`*. This function is documented on page 6.*)

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

For other packages and classes building on this one it is convenient not to need `\ExplSyntaxOn` each time.

```
⟨*package⟩
\protected\def\ProvidesExplPackage#1#2#3#4%
  {%
    \ProvidesPackage{#1}[#2 v#3 #4]%
    \ExplSyntaxOn
  }
\protected\def\ProvidesExplClass#1#2#3#4%
  {%
    \ProvidesClass{#1}[#2 v#3 #4]%
    \ExplSyntaxOn
  }
\protected\def\ProvidesExplFile#1#2#3#4%
  {%
    \ProvidesFile{#1}[#2 v#3 #4]%
    \ExplSyntaxOn
  }
⟨/package⟩
```

(*End definition for* `\ProvidesExplPackage`*,* `\ProvidesExplClass`*, and* `\ProvidesExplFile`*. These functions are documented on page 6.*)

`\@pushfilename`
`\@popfilename`

The idea here is to use LaTeX 2$_\varepsilon$'s `\@pushfilename` and `\@popfilename` to track the current syntax status. This can be achieved by saving the current status flag at each push to a stack, then recovering it at the pop stage and checking if the code environment should still be active.

```
⟨*package⟩
\edef\@pushfilename
  {%
    \edef\expandafter\noexpand
      \csname\detokenize{l_expl_status_stack_tl}\endcsname
      {%
        \noexpand\ifodd\expandafter\noexpand
          \csname\detokenize{l_expl_status_bool}\endcsname
          1%
        \noexpand\else
          0%
        \noexpand\fi
        \expandafter\noexpand
          \csname\detokenize{l_expl_status_stack_tl}\endcsname
      }%
    \ExplSyntaxOff
    \unexpanded\expandafter{\@pushfilename}%
  }
```

```
\edef\@popfilename
  {%
    \unexpanded\expandafter{\@popfilename}%
    \noexpand\if a\expandafter\noexpand\csname
      \detokenize{l_expl_status_stack_tl}\endcsname a%
      \ExplSyntaxOff
    \noexpand\else
      \noexpand\expandafter
        \expandafter\noexpand\csname
          \detokenize{expl_status_pop:w}\endcsname
          \expandafter\noexpand\csname
            \detokenize{l_expl_status_stack_tl}\endcsname
          \noexpand\@nil
    \noexpand\fi
  }
⟨/package⟩
```

(*End definition for* `\@pushfilename` *and* `\@popfilename`*. These functions are documented on page* **??**.)

`\l_expl_status_stack_tl` As expl3 itself cannot be loaded with the code environment already active, at the end of the package `\ExplSyntaxOff` can safely be called.

```
⟨*package⟩
\@namedef{\detokenize{l_expl_status_stack_tl}}{0}
⟨/package⟩
```

(*End definition for* `\l_expl_status_stack_tl`*. This function is documented on page* **??**.)

`\expl_status_pop:w` The pop auxiliary function removes the first item from the stack, saves the rest of the stack and then does the test. As `\ExplSyntaxOff` is already defined as a protected macro, there is no need for `\noexpand` here.

```
⟨*package⟩
\expandafter\edef\csname\detokenize{expl_status_pop:w}\endcsname#1#2\@nil
  {%
    \def\expandafter\noexpand
      \csname\detokenize{l_expl_status_stack_tl}\endcsname{#2}%
    \noexpand\ifodd#1\space
      \noexpand\expandafter\noexpand\ExplSyntaxOn
    \noexpand\else
      \noexpand\expandafter\ExplSyntaxOff
    \noexpand\fi
  }
⟨/package⟩
```

(*End definition for* `\expl_status_pop:w`*.*)

We want the expl3 bundle to be loaded "as one"; this command is used to ensure that one of the `l3` packages isn't loaded on its own.

```
⟨*package⟩
\expandafter\protected\expandafter\def
  \csname\detokenize{package_check_loaded_expl:}\endcsname
  {%
    \@ifpackageloaded{expl3}
```

```
      {}
      {%
        \PackageError{expl3}
          {Cannot load the expl3 modules separately}
          {%
            The expl3 modules cannot be loaded separately;\MessageBreak
            please \string\usepackage\string{expl3\string} instead.
          }%
      }%
  }
⟨/package⟩
```

183.4 The \pdfstrcmp primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
  \expandafter\ifx\csname pdfstrcmp\endcsname\relax
  \let\pdfstrcmp\strcmp
\fi
```

183.5 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε-TeX. The former is therefore used as a test for a suitable engine.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname pdfstrcmp\endcsname\relax
⟨*package⟩
  \PackageError{l3names}{Required primitive not found: \protect\pdfstrcmp}
    {%
      LaTeX3 requires the e-TeX primitives and
        \string\pdfstrcmp.\MessageBreak
      These are available in engine versions: \MessageBreak
      - pdfTeX 1.30 \MessageBreak
      - XeTeX 0.9994 \MessageBreak
      - LuaTeX 0.60 \MessageBreak
      or later. \MessageBreak
      \MessageBreak
      Loading of expl3 will abort!
    }
⟨/package⟩
⟨*initex⟩
  \newlinechar`\^^J\relax
  \errhelp{%
      LaTeX3 requires the e-TeX primitives and
        \string\pdfstrcmp. ^^J
      These are available in engine versions:  ^^J
      - pdfTeX 1.30  ^^J
      - XeTeX 0.9994 ^^J
```

```
      - LuaTeX 0.60  ^^J
      or later. ^^J
      For pdfTeX and XeTeX the '-etex' command-line switch is also
      needed.^^J
       ^^J
      Format building will abort!
  }
⟨/initex⟩
  \expandafter\endinput
\fi
```

183.6 The LaTeX3 code environment

\ExplSyntaxNamesOn
\ExplSyntaxNamesOff

These can be set up early, as they are not used anywhere in the package or format itself. Using an `\edef` here makes the definitions that bit clearer later.

```
\protected\edef\ExplSyntaxNamesOn
  {%
    \expandafter\noexpand
      \csname\detokenize{char_set_catcode_letter:n}\endcsname{58}%
    \expandafter\noexpand
      \csname\detokenize{char_set_catcode_letter:n}\endcsname{95}%
  }
\protected\edef\ExplSyntaxNamesOff
  {%
    \expandafter\noexpand
      \csname\detokenize{char_set_catcode_other:n}\endcsname{58}%
    \expandafter\noexpand
      \csname\detokenize{char_set_catcode_math_subscript:n}\endcsname{95}%
  }
```

(*End definition for* `\ExplSyntaxNamesOn` *and* `\ExplSyntaxNamesOff`*. These functions are documented on page 6.*)

The code environment is now set up for the format: the package deals with this using `\ProvidesExplPackage`.

```
⟨*initex⟩
\catcode 9   = 9  \relax
\catcode 32  = 9  \relax
\catcode 34  = 12 \relax
\catcode 58  = 11 \relax
\catcode 94  = 7  \relax
\catcode 95  = 11 \relax
\catcode 124 = 12 \relax
\catcode 126 = 10 \relax
\endlinechar = 32 \relax
⟨/initex⟩
```

\ExplSyntaxOn
\ExplSyntaxOff

The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time.

```
⟨*initex⟩
```

```
\protected \def \ExplSyntaxOn
  {
    \bool_if:NF \l_expl_status_bool
      {
        \cs_set_protected_nopar:Npx \ExplSyntaxOff
          {
            \char_set_catcode:nn { 9 }   { \char_value_catcode:n { 9 } }
            \char_set_catcode:nn { 32 }  { \char_value_catcode:n { 32 } }
            \char_set_catcode:nn { 34 }  { \char_value_catcode:n { 34 } }
            \char_set_catcode:nn { 38 }  { \char_value_catcode:n { 38 } }
            \char_set_catcode:nn { 58 }  { \char_value_catcode:n { 58 } }
            \char_set_catcode:nn { 94 }  { \char_value_catcode:n { 94 } }
            \char_set_catcode:nn { 95 }  { \char_value_catcode:n { 95 } }
            \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
            \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
            \tex_endlinechar:D =
              \tex_the:D \tex_endlinechar:D \scan_stop:
            \bool_set_false:N \l_expl_status_bool
            \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
          }
      }
    \char_set_catcode_ignore:n           { 9 }   % tab
    \char_set_catcode_ignore:n           { 32 }  % space
    \char_set_catcode_other:n            { 34 }  % double quote
    \char_set_catcode_alignment:n        { 38 }  % ampersand
    \char_set_catcode_letter:n           { 58 }  % colon
    \char_set_catcode_math_superscript:n { 94 }  % circumflex
    \char_set_catcode_letter:n           { 95 }  % underscore
    \char_set_catcode_other:n            { 124 } % pipe
    \char_set_catcode_space:n            { 126 } % tilde
    \tex_endlinechar:D = 32 \scan_stop:
    \bool_set_true:N \l_expl_status_bool
  }
\protected \def \ExplSyntaxOff { }
⟨/initex⟩
```

(*End definition for* `\ExplSyntaxOn` *and* `\ExplSyntaxOff`*. These functions are documented on page 6.*)

`\l_expl_status_bool` A flag to show the current syntax status.

```
⟨*initex⟩
\chardef \l_expl_status_bool = 0 ~
⟨/initex⟩
```

(*End definition for* `\l_expl_status_bool`*. This variable is documented on page* **??***.*)

```
⟨/initex | package⟩
```

184 l3names implementation

```
⟨*initex | package⟩
⟨*package⟩
```

```
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
⟨/package⟩
```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some od come back later).

`\tex_undefined:D` This function does not exist at all, but is the name used by the plain TeX format for an undefined function. So it should be marked here as "taken".

(*End definition for `\tex_undefined:D`. This function is documented on page* ??.)

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
\let \tex_global:D \global
\let \tex_let:D    \let
```

Everything is inside a (rather long) group, which kees `\name_primitve:NN` trapped.

```
\begingroup
```

`\name_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
  \long \def \name_primitive:NN #1#2
    {
      \tex_global:D \tex_let:D #2 #1
⟨*initex⟩
      \tex_global:D \tex_let:D #1 \tex_undefined:D
⟨/initex⟩
    }
```

(*End definition for `\name_primitive:NN`.*)

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 requires 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
```

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective. In the case of the pdfTEX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTEX primitives that start `\pdf...` but are not related to PDF output. These ones related to PDF output.

```
\name_primitive:NN \pdfcreationdate       \pdftex_pdfcreationdate:D
\name_primitive:NN \pdfcolorstack         \pdftex_pdfcolorstack:D
\name_primitive:NN \pdfcompresslevel      \pdftex_pdfcompresslevel:D
\name_primitive:NN \pdfdecimaldigits      \pdftex_pdfdecimaldigits:D
\name_primitive:NN \pdfhorigin            \pdftex_pdfhorigin:D
\name_primitive:NN \pdfinfo               \pdftex_pdfinfo:D
\name_primitive:NN \pdflastxform          \pdftex_pdflastxform:D
\name_primitive:NN \pdfliteral            \pdftex_pdfliteral:D
\name_primitive:NN \pdfminorversion       \pdftex_pdfminorversion:D
\name_primitive:NN \pdfobjcompresslevel   \pdftex_pdfobjcompresslevel:D
\name_primitive:NN \pdfoutput             \pdftex_pdfoutput:D
\name_primitive:NN \pdfrefxform           \pdftex_pdfrefxform:D
\name_primitive:NN \pdfrestore            \pdftex_pdfrestore:D
\name_primitive:NN \pdfsave               \pdftex_pdfsave:D
\name_primitive:NN \pdfsetmatrix          \pdftex_pdfsetmatrix:D
```

```
\name_primitive:NN \pdfpkresolution      \pdftex_pdfpkresolution:D
\name_primitive:NN \pdftexrevision       \pdftex_pdftextrevision:D
\name_primitive:NN \pdfvorigin           \pdftex_pdfvorigin:D
\name_primitive:NN \pdfxform             \pdftex_pdfxform:D
```

While these are not.

```
\name_primitive:NN \pdfstrcmp            \pdftex_strcmp:D
```

XeTeX-specific primitives. Note that XeTeX's `\strcmp` is handled earlier and is "rolled up" into `\pdfstrcmp`.

```
\name_primitive:NN \XeTeXversion         \xetex_XeTeXversion:D
```

Primitives from LuaTeX.

```
\name_primitive:NN \catcodetable         \luatex_catcodetable:D
\name_primitive:NN \directlua            \luatex_directlua:D
\name_primitive:NN \initcatcodetable     \luatex_initcatcodetable:D
\name_primitive:NN \latelua              \luatex_latelua:D
\name_primitive:NN \luatexversion        \luatex_luatexversion:D
\name_primitive:NN \savecatcodetable     \luatex_savecatcodetable:D
```

The job is done: close the group (using the primitive renamed!).

```
\tex_endgroup:D
```

LaTeX 2_ε will have moved a few primitives, so these are sorted out.

```
⟨*package⟩
\tex_let:D \tex_end:D                \@@end
\tex_let:D \tex_everydisplay:D       \frozen@everydisplay
\tex_let:D \tex_everymath:D          \frozen@everymath
\tex_let:D \tex_hyphen:D             \@@hyph
\tex_let:D \tex_input:D              \@@input
\tex_let:D \tex_italic_correction:D \@@italiccorr
\tex_let:D \tex_underline:D          \@@underline
```

That is also true for the luatex package for LaTeX 2_ε.

```
\tex_let:D \luatex_catcodetable:D      \luatexcatcodetable
\tex_let:D \luatex_initcatcodetable:D  \luatexinitcatcodetable
\tex_let:D \luatex_latelua:D           \luatexlatelua
\tex_let:D \luatex_savecatcodetable:D  \luatexsavecatcodetable
⟨/package⟩
```

```
⟨/initex | package⟩
```

185 l3basics implementation

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

185.1 Renaming some TeX primitives (again)

Having given all the TeX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.[2]

\if_true:
\if_false:
\or:
\else:
\fi:
\reverse_if:N
\if:w
\if_charcode:w
\if_catcode:w
\if_meaning:w

Then some conditionals.

```
\tex_let:D \if_true:            \tex_iftrue:D
\tex_let:D \if_false:           \tex_iffalse:D
\tex_let:D \or:                 \tex_or:D
\tex_let:D \else:               \tex_else:D
\tex_let:D \fi:                 \tex_fi:D
\tex_let:D \reverse_if:N        \etex_unless:D
\tex_let:D \if:w                \tex_if:D
\tex_let:D \if_charcode:w       \tex_if:D
\tex_let:D \if_catcode:w        \tex_ifcat:D
\tex_let:D \if_meaning:w        \tex_ifx:D
```

(*End definition for* `\if_true:` *and others. These functions are documented on page 23.*)

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

TeX lets us detect some if its modes.

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

(*End definition for* `\if_mode_math:` *and others. These functions are documented on page* **??**.)

\if_cs_exist:N
\if_cs_exist:w
\cs:w
\cs_end:

Building csnames and testing if control sequences exist.

```
\tex_let:D \if_cs_exist:N       \etex_ifdefined:D
\tex_let:D \if_cs_exist:w       \etex_ifcsname:D
\tex_let:D \cs:w                \tex_csname:D
\tex_let:D \cs_end:             \tex_endcsname:D
```

(*End definition for* `\if_cs_exist:N` *and others. These functions are documented on page* **??**.)

\exp_after:wN
\exp_not:N
\exp_not:n

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

```
\tex_let:D \exp_after:wN        \tex_expandafter:D
\tex_let:D \exp_not:N           \tex_noexpand:D
\tex_let:D \exp_not:n           \etex_unexpanded:D
```

(*End definition for* `\exp_after:wN`*,* `\exp_not:N`*, and* `\exp_not:n`*. These functions are documented on page 31.*)

\token_to_meaning:N
\token_to_str:N
\cs_meaning:N
\cs_show:N

Examining a control sequence or token.

```
\tex_let:D \token_to_meaning:N \tex_meaning:D
\tex_let:D \token_to_str:N      \tex_string:D
\tex_let:D \cs_meaning:N        \tex_meaning:D
\tex_let:D \cs_show:N           \tex_show:D
```

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

(*End definition for* `\token_to_meaning:N`*,* `\token_to_str:N`*, and* `\cs_meaning:N`*. These functions are documented on page 16.*)

`\scan_stop:`
`\group_begin:`
`\group_end:`

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

```
\tex_let:D \scan_stop:          \tex_relax:D
\tex_let:D \group_begin:        \tex_begingroup:D
\tex_let:D \group_end:          \tex_endgroup:D
```

(*End definition for* `\scan_stop:`*,* `\group_begin:`*, and* `\group_end:`*. These functions are documented on page* **??**.)

`\if_int_compare:w`
`\int_to_roman:w`

For integers.

```
\tex_let:D \if_int_compare:w   \tex_ifnum:D
\tex_let:D \int_to_roman:w     \tex_romannumeral:D
```

(*End definition for* `\if_int_compare:w` *and* `\int_to_roman:w`*. These functions are documented on page 70.*)

`\group_insert_after:N`

Adding material after the end of a group.

```
\tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(*End definition for* `\group_insert_after:N`*. This function is documented on page 9.*)

`\tex_global:D`
`\tex_long:D`
`\etex_protected:D`

Three prefixes for assignments.

```
\tex_let:D \tex_global:D      \tex_global:D
\tex_let:D \tex_long:D        \tex_long:D
\tex_let:D \tex_protected:D   \etex_protected:D
```

(*End definition for* `\tex_global:D`*,* `\tex_long:D`*, and* `\etex_protected:D`*. These functions are documented on page* **??**.)

`\exp_args:Nc`

Discussed in l3expan, but needed much earlier.

```
\tex_long:D \tex_def:D \exp_args:Nc #1#2 { \exp_after:wN #1 \cs:w #2 \cs_end: }
```

(*End definition for* `\exp_args:Nc`*. This function is documented on page 27.*)

`\token_to_str:c`
`\cs_meaning:c`
`\cs_show:c`

A small number of variants by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:NNc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly. The `\cs_show:c` command is “protected” because its action is not expandable. Also, the conversion of its argument to a control sequence is done within a group to avoid converting it to `\relax`.

```
\tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
\tex_long:D \tex_def:D \cs_meaning:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \exp_args:Nc \cs_meaning:N {#1} }
    { \tl_to_str:n {undefined} }
```

```
  }
\tex_protected:D \tex_def:D \cs_show:c
  { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
```

(*End definition for* `\token_to_str:c` *and* `\cs_meaning:c`*. These functions are documented on page* ??.)

185.2 Defining some constants

`\c_minus_one` `\c_zero` `\c_sixteen` `\c_six` `\c_seven` `\c_twelve`

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

```
⟨*package⟩
\tex_let:D \c_minus_one \m@ne
⟨/package⟩
⟨*initex⟩
\tex_countdef:D \c_minus_one = 10 ~
\c_minus_one = -1 ~
⟨/initex⟩
\tex_chardef:D \c_sixteen = 16~
\tex_chardef:D \c_zero    = 0~
\tex_chardef:D \c_six     = 6~
\tex_chardef:D \c_seven   = 7~
\tex_chardef:D \c_twelve  = 12~
```

(*End definition for* `\c_minus_one`*,* `\c_zero`*, and* `\c_sixteen`*. These functions are documented on page 69.*)

`\c_max_register_int`

This is here as this particular integer is needed both in package mode and to bootstrap l3alloc.

```
\etex_ifdefined:D \luatex_luatexversion:D
  \tex_chardef:D \c_max_register_int = 65 535 ~
\tex_else:D
  \tex_mathchardef:D \c_max_register_int = 32 767 ~
\tex_fi:D
```

(*End definition for* `\c_max_register_int`*. This variable is documented on page 69.*)

185.3 Defining functions

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

`\cs_set_nopar:Npn` `\cs_set_nopar:Npx` `\cs_set:Npn` `\cs_set:Npx` `\cs_set_protected_nopar:Npn` `\cs_set_protected_nopar:Npx` `\cs_set_protected:Npn` `\cs_set_protected:Npx`

All assignment functions in LATEX3 should be naturally robust; after all, the TEX primitives for assignments are and it can be a cause of problems if others aren't.

```
\tex_let:D \cs_set_nopar:Npn             \tex_def:D
\tex_let:D \cs_set_nopar:Npx             \tex_edef:D
\tex_protected:D \cs_set_nopar:Npn \cs_set:Npn
  { \tex_long:D \cs_set_nopar:Npn }
```

```
\tex_protected:D \cs_set_nopar:Npn \cs_set:Npx
  { \tex_long:D \cs_set_nopar:Npx }
\tex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
  { \tex_protected:D \cs_set_nopar:Npn }
\tex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
  { \tex_protected:D \cs_set_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_set_protected:Npn
  { \tex_protected:D \tex_long:D \cs_set_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_set_protected:Npx
  { \tex_protected:D \tex_long:D \cs_set_nopar:Npx }
```

(*End definition for* `\cs_set_nopar:Npn` *and others. These functions are documented on page* **??**.)

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:Npx`
`\cs_gset:Npn`
`\cs_gset:Npx`
`\cs_gset_protected_nopar:Npn`
`\cs_gset_protected_nopar:Npx`
`\cs_gset_protected:Npn`
`\cs_gset_protected:Npx`

Global versions of the above functions.

```
\tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
\tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D
\cs_set_protected_nopar:Npn \cs_gset:Npn
  { \tex_long:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset:Npx
  { \tex_long:D \cs_gset_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
  { \tex_protected:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
  { \tex_protected:D \cs_gset_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_gset_protected:Npn
  { \tex_protected:D \tex_long:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset_protected:Npx
  { \tex_protected:D \tex_long:D \cs_gset_nopar:Npx }
```

(*End definition for* `\cs_gset_nopar:Npn` *and others. These functions are documented on page* **??**.)

185.4 Selecting tokens

`\use:c`

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

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

(*End definition for* `\use:c`*. This function is documented on page 16.*)

`\use:x`

Fully expands its argument and passes it to the input stream. Uses the reserved `\l_exp_internal_tl` which will be set up in l3expan.

```
\cs_set_protected:Npn \use:x #1
  {
    \cs_set_nopar:Npx \l_exp_internal_tl {#1}
    \l_exp_internal_tl
  }
```

(*End definition for* `\use:x`*. This function is documented on page 19.*)

`\use:n`
`\use:nn`
`\use:nnn`
`\use:nnnn`

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

```
\cs_set:Npn \use:n    #1      {#1}
\cs_set:Npn \use:nn   #1#2    {#1#2}
```

```
\cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
\cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(*End definition for* `\use:n` *and others. These functions are documented on page* **??**.)

`\use_i:nn`
`\use_ii:nn`

The equivalent to LaTeX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```
\cs_set:Npn \use_i:nn  #1#2 {#1}
\cs_set:Npn \use_ii:nn #1#2 {#2}
```

(*End definition for* `\use_i:nn` *and* `\use_ii:nn`*. These functions are documented on page 18.*)

`\use_i:nnn`
`\use_ii:nnn`
`\use_iii:nnn`
`\use_i_ii:nnn`
`\use_i:nnnn`
`\use_ii:nnnn`
`\use_iii:nnnn`
`\use_iv:nnnn`

We also need something for picking up arguments from a longer list.

```
\cs_set:Npn \use_i:nnn    #1#2#3 {#1}
\cs_set:Npn \use_ii:nnn   #1#2#3 {#2}
\cs_set:Npn \use_iii:nnn  #1#2#3 {#3}
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
\cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
\cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
\cs_set:Npn \use_iv:nnnn  #1#2#3#4 {#4}
```

(*End definition for* `\use_i:nnn` *and others. These functions are documented on page 18.*)

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

Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\cs_set:Npn \use_none_delimit_by_q_nil:w  #1 \q_nil  { }
\cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
\cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(*End definition for* `\use_none_delimit_by_q_nil:w`*,* `\use_none_delimit_by_q_stop:w`*, and* `\use_none_delimit_by_q_recurs`
These functions are documented on page 46.)

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

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

```
\cs_set:Npn \use_i_delimit_by_q_nil:nw  #1#2 \q_nil  {#1}
\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
\cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(*End definition for* `\use_i_delimit_by_q_nil:nw`*,* `\use_i_delimit_by_q_stop:nw`*, and* `\use_i_delimit_by_q_recursion_sto`
These functions are documented on page 46.)

185.5 Gobbling tokens from input

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

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```
\cs_set:Npn \use_none:n         #1              { }
\cs_set:Npn \use_none:nn        #1#2            { }
\cs_set:Npn \use_none:nnn       #1#2#3          { }
\cs_set:Npn \use_none:nnnn      #1#2#3#4        { }
```

```
\cs_set:Npn \use_none:nnnnn     #1#2#3#4#5         { }
\cs_set:Npn \use_none:nnnnnn    #1#2#3#4#5#6       { }
\cs_set:Npn \use_none:nnnnnnn   #1#2#3#4#5#6#7     { }
\cs_set:Npn \use_none:nnnnnnnn  #1#2#3#4#5#6#7#8   { }
\cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(*End definition for* `\use_none:n` *and others. These functions are documented on page* **??**.)

185.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (`TF`, etc.) is a built-in logic saying that it after all of the testing and processing must return the ⟨*state*⟩ this leaves TeX in. Therefore, a simple user interface could be something like

```
  \if_meaning:w #1#2   \prg_return_true:  \else:
     \if_meaning:w #1#3  \prg_return_true: \else:
     \prg_return_false:
 \fi: \fi:
```

Usually, a TeX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the TeX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:`
`\prg_return_false:`

The idea here is that `\int_to_roman:w` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\c_zero` which will leave the expansion null. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
\cs_set_nopar:Npn \prg_return_true:
  { \exp_after:wN \use_i:nn  \int_to_roman:w }
\cs_set_nopar:Npn \prg_return_false:
  { \exp_after:wN \use_ii:nn \int_to_roman:w}
```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn`/`\use_ii:nn`. Provided two arguments are absorbed then the code will work.
(*End definition for* `\prg_return_true:` *and* `\prg_return_false:`*. These functions are documented on page* **??**.)

`\prg_set_conditional:Npnn`
`\prg_new_conditional:Npnn`
`\prg_set_protected_conditional:Npnn`
`\prg_new_protected_conditional:Npnn`
`\prg_generate_conditional_parm_aux:NNpnn`

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed `{`⟨*name*⟩`}` `{`⟨*signature*⟩`}` ⟨*boolean*⟩ ⟨*defining function*⟩ `{parm}` `{`⟨*parameters*⟩`}` `{TF,...}` `{`⟨*code*⟩`}` to the auxiliary function responsible for defining all conditionals.

```
\cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
  { \prg_generate_conditional_parm_aux:NNpnn \cs_set:Npn }
\cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
  { \prg_generate_conditional_parm_aux:NNpnn \cs_new:Npn }
```

```
\cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
  { \prg_generate_conditional_parm_aux:NNpnn \cs_set_protected:Npn }
\cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
  { \prg_generate_conditional_parm_aux:NNpnn \cs_new_protected:Npn }
\cs_set_protected:Npn \prg_generate_conditional_parm_aux:NNpnn #1#2#3#
  {
    \cs_split_function:NN #2 \prg_generate_conditional_aux:nnNNnnnn
    #1 { parm } {#3}
  }
```

(*End definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page 33.*)

`\prg_set_conditional:Nnn`
`\prg_new_conditional:Nnn`
`\prg_set_protected_conditional:Nnn`
`\prg_new_protected_conditional:Nnn`
`\prg_generate_conditional_count_aux:NNnn`

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. For those `Nnn` type functions, we calculate the number of arguments. Then split the base function into name and signature, and feed `{`⟨*name*⟩`}` `{`⟨*signature*⟩`}` ⟨*boolean*⟩ ⟨*defining function*⟩ `{count}` `{`⟨*arg count*⟩`}` `{TF,...}` `{`⟨*code*⟩`}` to the auxiliary function responsible for defining all conditionals.

```
\cs_set_protected_nopar:Npn \prg_set_conditional:Nnn
  { \prg_generate_conditional_count_aux:NNnn \cs_set:Npn }
\cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
  { \prg_generate_conditional_count_aux:NNnn \cs_new:Npn }
\cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
  { \prg_generate_conditional_count_aux:NNnn \cs_set_protected:Npn }
\cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
  { \prg_generate_conditional_count_aux:NNnn \cs_new_protected:Npn }
\cs_set_protected:Npn \prg_generate_conditional_count_aux:NNnn #1#2
  {
    \exp_args:Nnf \use:n
      {
        \cs_split_function:NN #2 \prg_generate_conditional_aux:nnNNnnnn
        #1 { count }
      }
      { \cs_get_arg_count_from_signature:N #2 }
  }
```

(*End definition for* `\prg_set_conditional:Nnn` *and others. These functions are documented on page* ??.)

`\prg_set_eq_conditional:NNn`
`\prg_new_eq_conditional:NNn`

The obvious setting-equal functions.

```
\cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3
  { \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3} }
\cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3
  { \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3} }
```

(*End definition for* `\prg_set_eq_conditional:NNn` *and* `\prg_new_eq_conditional:NNn`*. These functions are documented on page 35.*)

`\prg_generate_conditional_aux:nnNNnnnn`
`\prg_generate_conditional_aux:nnw`

The workhorse here is going through a list of desired forms, *i.e.*, `p`, `TF`, `T` and `F`. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name.

For the time being, we do not use this piece of information but could well throw an error. The fourth argument is how to define this function, the fifth is the text `parm` or `count` for which version to use to define the functions, the sixth is the parameters to use (possibly empty) or number of arguments, the seventh is the list of forms to define, the eight is the replacement text which we will augment when defining the forms.

```
\cs_set_protected:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8
  {
    \prg_generate_conditional_aux:nnw {#5}
      {
        #4 {#1} {#2} {#6} {#8}
      }
      #7 , ? , \q_recursion_stop
  }
```

Looping through the list of desired forms. First is the text `parm` or `count`, second is five arguments packed together and third is the form. Use text and form to call the correct type.

```
\cs_set_protected:Npn \prg_generate_conditional_aux:nnw #1#2#3 ,
  {
    \if:w ?#3
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { prg_generate_#3_form_#1:Nnnnn } #2
    \prg_generate_conditional_aux:nnw {#1} {#2}
  }
```

(*End definition for* `\prg_generate_conditional_aux:nnNNnnnn` *and* `\prg_generate_conditional_aux:nnw`.)

\prg_generate_p_form_parm:Nnnnn
\prg_generate_TF_form_parm:Nnnnn
\prg_generate_T_form_parm:Nnnnn
\prg_generate_F_form_parm:Nnnnn

How to generate the various forms. The `parm` types here takes the following arguments: 1: how to define (an N-type), 2: name, 3: signature, 4: parameter text (or empty), 5: replacement. Remember that the logic-returning functions expect two arguments to be present after `\c_zero`: notice the construction of the different variants relies on this, and that the `TF` variant will be slightly faster than the `T` version.

```
\cs_set_protected:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5
  {
    \exp_args:Nc #1 { #2 _p: #3 } #4
      {
        #5 \c_zero
        \c_true_bool \c_false_bool
      }
  }
\cs_set_protected:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5
  {
    \exp_args:Nc #1 { #2 : #3 T } #4
      {
        #5 \c_zero
        \use:n \use_none:n
      }
  }
\cs_set_protected:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5
```

```
  {
    \exp_args:Nc #1 { #2 : #3 F } #4
      {
        #5 \c_zero
        { }
      }
  }
\cs_set_protected:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5
  {
    \exp_args:Nc #1 { #2 : #3 TF } #4
      { #5 \c_zero }
  }
```

(*End definition for* `\prg_generate_p_form_parm:Nnnnn` *and others.*)

\prg_generate_p_form_count:Nnnnn
\prg_generate_TF_form_count:Nnnnn
\prg_generate_T_form_count:Nnnnn
\prg_generate_F_form_count:Nnnnn

The `count` form is similar, but of course requires a number rather than a primitive argument specification.

```
\cs_set_protected:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5
  {
    \cs_generate_from_arg_count:cNnn { #2 _p: #3 } #1 {#4}
      {
        #5 \c_zero
        \c_true_bool \c_false_bool
      }
  }
\cs_set_protected:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5
  {
    \cs_generate_from_arg_count:cNnn { #2 : #3 T } #1 {#4}
      {
        #5 \c_zero
        \use:n \use_none:n
      }
  }
\cs_set_protected:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5
  {
    \cs_generate_from_arg_count:cNnn { #2 : #3 F } #1 {#4}
      {
        #5 \c_zero
        { }
      }
  }
\cs_set_protected:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5
  {
    \cs_generate_from_arg_count:cNnn { #2 : #3 TF } #1 {#4}
      { #5 \c_zero }
  }
```

(*End definition for* `\prg_generate_p_form_count:Nnnnn` *and others.*)

\prg_set_eq_conditional_aux:NNNn
\prg_set_eq_conditional_aux:NNNw
\prg_conditional_form_p:nnn
\prg_conditional_form_TF:nnn
\prg_conditional_form_T:nnn
\prg_conditional_form_F:nnn

Manual clist loop over argument `#4`.

```
\cs_set_protected:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4
```

```
  { \prg_set_eq_conditional_aux:NNNw #1#2#3#4 , ? , \q_recursion_stop }
\cs_set_protected:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4 ,
  {
    \if:w ? #4 \scan_stop:
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    #1
    { \exp_args:NNc \cs_split_function:NN #2 { prg_conditional_form_#4:nnn } }
    { \exp_args:NNc \cs_split_function:NN #3 { prg_conditional_form_#4:nnn } }
    \prg_set_eq_conditional_aux:NNNw #1 {#2} {#3}
  }
\cs_set:Npn \prg_conditional_form_p:nnn  #1#2#3 { #1 _p : #2 }
\cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 { #1    : #2 TF }
\cs_set:Npn \prg_conditional_form_T:nnn  #1#2#3 { #1    : #2 T }
\cs_set:Npn \prg_conditional_form_F:nnn  #1#2#3 { #1    : #2 F }
```

(*End definition for* `\prg_set_eq_conditional_aux:NNNn` *and* `\prg_set_eq_conditional_aux:NNNw`*. These functions are documented on page 35.*)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either `TT` or `TF` to be tested using `\if:w` but this was later changed to `00` and `01`, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool`
`\c_false_bool`

Here are the canonical boolean values.

```
\tex_chardef:D \c_true_bool  = 1~
\tex_chardef:D \c_false_bool = 0~
```

(*End definition for* `\c_true_bool` *and* `\c_false_bool`*. These variables are documented on page 21.*)

185.7 Dissecting a control sequence

`\cs_to_str:N`
`\cs_to_str_aux:N`
`\cs_to_str_aux:w`

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of `\tex_escapechar:D` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N␣\␣` yields the escape character itself and a space. The character codes are different, thus the \if:w test is false, and TEX reads \cs_to_str_-aux:N after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial \int_to_roman:w. The second case is that the escape character is not printable. Then the \if:w test is unfinished after reading a the space from \token_to_str:N␣\␣, and the auxiliary \cs_to_str_aux:w is expanded, feeding - as a second character for the test; the test is false, and TEX skips to \fi:, then performs \token_to_str:N, and stops the \int_to_roman:w with \c_zero. The last case is that the escape character is itself a space. In this case, the \if:w test is true, and the auxiliary \cs_to_str_aux:w comes into play, inserting -\int_value:w, which expands \c_zero to the character 0. The initial \int_to_roman:w then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the argument of \int_to_roman:w. In all three cases, \cs_to_-str:N takes two expansion steps to be fully expanded.

```
\cs_set_nopar:Npn \cs_to_str:N
  {
    \int_to_roman:w
      \if:w \token_to_str:N \ \cs_to_str_aux:w \fi:
      \exp_after:wN \cs_to_str_aux:N \token_to_str:N
  }
\cs_set:Npn \cs_to_str_aux:N #1 { \c_zero }
\cs_set:Npn \cs_to_str_aux:w #1 \cs_to_str_aux:N
  { - \int_value:w \fi: \exp_after:wN \c_zero }
```

(*End definition for \cs_to_str:N. This function is documented on page 17.*)

\cs_split_function:NN
\cs_split_function_aux:w
\cs_split_function_auxii:w

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean ⟨*true*⟩ or ⟨*false*⟩ is returned with ⟨*true*⟩ for when there is a colon in the function and ⟨*false*⟩ if there is not. Lastly, the second argument of \cs_split_function:NN is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, \cs_split_function:NN\foo_bar:cnx\use_i:nnn as input becomes \use_i:nnn {foo_bar}{cnx}\c_true_bool.

Can't use a literal : because it has the wrong catcode here, so it's transformed from @ with \tex_lowercase:D.

```
\group_begin:
  \tex_lccode:D  `\@ = `\: \scan_stop:
  \tex_catcode:D `\@ = 12~
\tex_lowercase:D
  {
    \group_end:
```

First ensure that we actually get a properly evaluated str by expanding \cs_to_str:N twice. Insert extra colon to catch the error cases.

```
    \cs_set:Npn \cs_split_function:NN #1#2
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \cs_split_function_aux:w
          \cs_to_str:N #1 @ a \q_stop #2
      }
```

If no colon in the name, #2 is a with catcode 11 and #3 is empty. If colon in the name, then either #2 is a colon or the first letter of the signature. The letters here have catcode 12. If a colon was given we need to a) split off the colon and quark at the end and b) ensure we return the name, signature and boolean true We can't use \quark_if_no_value:NTF yet but this is very safe anyway as all tokens have catcode 12.

```
    \cs_set:Npn \cs_split_function_aux:w #1 @ #2#3 \q_stop #4
      {
        \if_meaning:w a #2
          \exp_after:wN \use_i:nn
        \else:
          \exp_after:wN\use_ii:nn
        \fi:
        { #4 {#1} { } \c_false_bool }
        { \cs_split_function_auxii:w #2#3 \q_stop #4 {#1} }
      }
    \cs_set:Npn \cs_split_function_auxii:w #1 @a \q_stop #2#3
      { #2{#3}{#1}\c_true_bool }
```

End of lowercase

```
  }
```

(*End definition for* `\cs_split_function:NN`*. This function is documented on page* *20**.*)

\cs_get_function_name:N
\cs_get_function_signature:N

Now returning the name is trivial: just discard the last two arguments. Similar for signature.

```
\cs_set:Npn \cs_get_function_name:N #1
  { \cs_split_function:NN #1 \use_i:nnn }
\cs_set:Npn \cs_get_function_signature:N #1
  { \cs_split_function:NN #1 \use_ii:nnn }
```

(*End definition for* `\cs_get_function_name:N` *and* `\cs_get_function_signature:N`*. These functions are documented on page* *19**.*)

185.8 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive \tex_relax:D token. A control sequence is said to be *free* (to be defined) if it does not already exist.

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

Two versions for checking existence. For the N form we firstly check for \scan_stop: and then if it is in the hash table. There is no problem when inputting something like \else: or \fi: as TeX will only ever skip input in case the token tested against is \scan_stop:.

```
\prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
```

```
        \prg_return_false:
      \fi:
    \fi:
  }
```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```
\prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    {
      \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    }
    \prg_return_false:
  }
```

(*End definition for* `\cs_if_exist:N` *and* `\cs_if_exist:c`*. These functions are documented on page* ??.)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The logical reversal of the above.

```
\prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_true:
    \else:
      \if_cs_exist:N #1
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    \fi:
  }
\prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
```

```
      \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
    { \prg_return_true: }
}
```

(*End definition for* `\cs_if_free:N` *and* `\cs_if_free:c`*. These functions are documented on page* **??**.)

`\cs_if_exist_use:NTF` `\cs_if_exist_use:cTF` `\cs_if_exist_use:N` `\cs_if_exist_use:c` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist.

```
\cs_set:Npn \cs_if_exist_use:NTF #1#2
  { \cs_if_exist:NTF #1 { #1 #2 } }
\cs_set:Npn \cs_if_exist_use:NF #1
  { \cs_if_exist:NTF #1 { #1 } }
\cs_set:Npn \cs_if_exist_use:NT #1 #2
  { \cs_if_exist:NTF #1 { #1 #2 } { } }
\cs_set:Npn \cs_if_exist_use:N #1
  { \cs_if_exist:NTF #1 { #1 } { } }
\cs_set:Npn \cs_if_exist_use:cTF #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
\cs_set:Npn \cs_if_exist_use:cF #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } }
\cs_set:Npn \cs_if_exist_use:cT #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
\cs_set:Npn \cs_if_exist_use:c #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(*End definition for* `\cs_if_exist_use:N` *and* `\cs_if_exist_use:c`*. These functions are documented on page* **??**.)

185.9 Defining and checking (new) functions

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

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

`\iow_log:x` `\iow_term:x` We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by l3io.

```
\cs_set_protected_nopar:Npn \iow_log:x
  { \tex_immediate:D \tex_write:D \c_minus_one }
\cs_set_protected_nopar:Npn \iow_term:x
  { \tex_immediate:D \tex_write:D \c_sixteen }
```

(*End definition for* `\iow_log:x` *and* `\iow_term:x`*. These functions are documented on page* **??**.)

`\msg_kernel_error:nnxx`
`\msg_kernel_error:nnx`
`\msg_kernel_error:nn`

If an internal error occurs before LaTeX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response.

```
\cs_set_protected:Npn \msg_kernel_error:nnxx #1#2#3#4
  {
    \tex_errmessage:D
      {
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
        Argh,~internal~LaTeX3~error! ^^J ^^J
        Module ~ #1 , ~ message~name~"#2": ^^J
        Arguments~'#3'~and~'#4' ^^J ^^J
        This~is~one~for~The~LaTeX3~Project:~bailing~out
      }
    \tex_end:D
  }
\cs_set_protected:Npn \msg_kernel_error:nnx #1#2#3
  { \msg_kernel_error:nnxx {#1} {#2} {#3} { } }
\cs_set_protected:Npn \msg_kernel_error:nn #1#2
  { \msg_kernel_error:nnxx {#1} {#2} { } { } }
```

(*End definition for* `\msg_kernel_error:nnxx`*,* `\msg_kernel_error:nnx`*, and* `\msg_kernel_error:nn`*. These functions are documented on page* **??**.)

`\msg_line_context:`

Another one from l3msg which will be altered later.

```
\cs_set_nopar:Npn \msg_line_context:
  { on~line~\tex_the:D \tex_inputlineno:D }
```

(*End definition for* `\msg_line_context:`*. This function is documented on page* **??**.)

`\chk_if_free_cs:N`
`\chk_if_free_cs:c`

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

```
\cs_set_protected:Npn \chk_if_free_cs:N #1
  {
    \cs_if_free:NF #1
      {
        \msg_kernel_error:nnxx { kernel } { command-already-defined }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      }
}
⟨*package⟩
\tex_ifodd:D \l@expl@log@functions@bool
  \cs_set_protected:Npn \chk_if_free_cs:N #1
    {
      \cs_if_free:NF #1
        {
          \msg_kernel_error:nnxx { kernel } { command-already-defined }
```

```
            { \token_to_str:N #1 } { \token_to_meaning:N #1 }
        }
      \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
    }
\fi:
⟨/package⟩
\cs_set_protected_nopar:Npn \chk_if_free_cs:c
  { \exp_args:Nc \chk_if_free_cs:N }
```

(*End definition for* `\chk_if_free_cs:N` *and* `\chk_if_free_cs:c`*. These functions are documented on page* **??**.)

`\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.

```
\cs_set_protected:Npn \chk_if_exist_cs:N #1
  {
    \cs_if_exist:NF #1
      {
        \msg_kernel_error:nnxx { kernel } { command-not-defined }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      }
  }
\cs_set_protected_nopar:Npn \chk_if_exist_cs:c
  { \exp_args:Nc \chk_if_exist_cs:N }
```

(*End definition for* `\chk_if_exist_cs:N` *and* `\chk_if_exist_cs:c`*. These functions are documented on page* **??**.)

185.10 More new definitions

`\cs_new_nopar:Npn`
`\cs_new_nopar:Npx`
`\cs_new:Npn`
`\cs_new:Npx`
`\cs_new_protected_nopar:Npn`
`\cs_new_protected_nopar:Npx`
`\cs_new_protected:Npn`
`\cs_new_protected:Npx`

Function which check that the control sequence is free before defining it.

```
\cs_set:Npn \cs_tmp:w #1#2
  {
    \cs_set_protected:Npn #1 ##1
       {
         \chk_if_free_cs:N ##1
         #2 ##1
      }
  }
\cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
\cs_tmp:w \cs_new_nopar:Npx           \cs_gset_nopar:Npx
\cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
\cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
\cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
\cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
\cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
\cs_tmp:w \cs_new_protected:Npx       \cs_gset_protected:Npx
```

(*End definition for* `\cs_new_nopar:Npn` *and others. These functions are documented on page* **??**.)

`\cs_set_nopar:cpn`
`\cs_set_nopar:cpx`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:cpx`
`\cs_new_nopar:cpn`
`\cs_new_nopar:cpx`

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

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

```
\cs_set:Npn \cs_tmp:w #1#2
  { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
\cs_tmp:w \cs_set_nopar:cpn  \cs_set_nopar:Npn
\cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
\cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
\cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
\cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
\cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
```

(*End definition for* `\cs_set_nopar:cpn` *and others. These functions are documented on page* **??**.)

`\cs_set:cpn`
`\cs_set:cpx`
`\cs_gset:cpn`
`\cs_gset:cpx`
`\cs_new:cpn`
`\cs_new:cpx`

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

```
\cs_tmp:w \cs_set:cpn  \cs_set:Npn
\cs_tmp:w \cs_set:cpx  \cs_set:Npx
\cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\cs_tmp:w \cs_gset:cpx \cs_gset:Npx
\cs_tmp:w \cs_new:cpn  \cs_new:Npn
\cs_tmp:w \cs_new:cpx  \cs_new:Npx
```

(*End definition for* `\cs_set:cpn` *and others. These functions are documented on page* **??**.)

`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:cpx`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpx`

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

```
\cs_tmp:w \cs_set_protected_nopar:cpn  \cs_set_protected_nopar:Npn
\cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
\cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
\cs_tmp:w \cs_new_protected_nopar:cpn  \cs_new_protected_nopar:Npn
\cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
```

(*End definition for* `\cs_set_protected_nopar:cpn` *and others. These functions are documented on page* **??**.)

`\cs_set_protected:cpn`
`\cs_set_protected:cpx`
`\cs_gset_protected:cpn`
`\cs_gset_protected:cpx`
`\cs_new_protected:cpn`
`\cs_new_protected:cpx`

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

```
\cs_tmp:w \cs_set_protected:cpn  \cs_set_protected:Npn
\cs_tmp:w \cs_set_protected:cpx  \cs_set_protected:Npx
\cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
\cs_tmp:w \cs_new_protected:cpn  \cs_new_protected:Npn
\cs_tmp:w \cs_new_protected:cpx  \cs_new_protected:Npx
```

(*End definition for* `\cs_set_protected:cpn` *and others. These functions are documented on page* **??**.)

185.11 Copying definitions

\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

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

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

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an "already defined" error rather than "runaway argument".

```
\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
\cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc  \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D  \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_new_eq:NN #1
  {
    \chk_if_free_cs:N #1
    \tex_global:D \cs_set_eq:NN #1
  }
\cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(*End definition for* `\cs_set_eq:NN` *and others. These functions are documented on page* **??**.)

185.12 Undefining functions

\cs_undefine:N
\cs_undefine:c

The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting TeX conditionals in case `#1` is unbalanced in this matter.

```
\cs_new_protected:Npn \cs_undefine:N #1
  { \cs_gset_eq:NN #1 \c_undefined:D }
\cs_new_protected:Npn \cs_undefine:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
    { \cs_gset_eq:cN {#1} \c_undefined:D }
  }
```

(*End definition for* `\cs_undefine:N` *and* `\cs_undefine:c`*. These functions are documented on page* **??**.)

185.13 Defining functions from a given number of arguments

\cs_get_arg_count_from_signature:N
\cs_get_arg_count_from_signature_aux:nnN
\cs_get_arg_count_from_signature_auxii:w

Counting the number of tokens in the signature, i.e., the number of arguments the function should take. If there is no signature, we return that there is −1 arguments to signal an error. Otherwise we insert the string `9876543210` after the signature. If the signature is empty, the number we want is 0 so we remove the first nine tokens and return the tenth. Similarly, if the signature is `nnn` we want to remove the nine tokens `nnn987654` and return 3. Therefore, we simply remove the first nine tokens and then return the tenth.

```
\cs_new:Npn \cs_get_arg_count_from_signature:N #1
  { \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN }
\cs_new:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3
  {
    \if_meaning:w \c_true_bool #3
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN\use_ii:nn
    \fi:
    {
      \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
        \use_none:nnnnnnnnn  #2 9876543210 \q_stop
    }
    { -1 }
}
\cs_new:Npn \cs_get_arg_count_from_signature_auxii:w #1#2 \q_stop {#1}
```

A variant form we need right away.

```
\cs_new_nopar:Npn \cs_get_arg_count_from_signature:c
  { \exp_args:Nc \cs_get_arg_count_from_signature:N }
```

(*End definition for* `\cs_get_arg_count_from_signature:N`*. This function is documented on page* *19*.)

\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count:cNnn
\cs_generate_from_arg_count:Ncnn
\cs_generate_from_arg_count_error_msg:Nn
\cs_generate_from_arg_count_aux:nwn

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
  {
    \if_case:w \int_eval:w #3 \int_eval_end:
         \cs_generate_from_arg_count_aux:nwn {}
    \or: \cs_generate_from_arg_count_aux:nwn {##1}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4##5}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4##5##6}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4##5##6##7}
```

```
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4##5##6##7##8}
    \or: \cs_generate_from_arg_count_aux:nwn {##1##2##3##4##5##6##7##8##9}
    \else:
      \cs_generate_from_arg_count_error_msg:Nn #1 {#3}
      \use_i:nnn
    \fi:
    {#2#1}
    {#4}
  }
\cs_new_protected:Npn
  \cs_generate_from_arg_count_aux:nwn #1 #2 \fi: #3
  { \fi: #3 #1 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
\cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
  { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
\cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
  { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

The error message. Elsewhere we use the value of -1 to signal a missing colon in a function, so provide a hint for help on this.

```
\cs_new_protected:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2
  {
    \msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
      { \token_to_str:N #1 } { \int_eval:n {#2} }
  }
```

(*End definition for* `\cs_generate_from_arg_count:NNnn`*,* `\cs_generate_from_arg_count:cNnn`*, and* `\cs_generate_from_arg_count:Ncnn`*. These functions are documented on page 19.*)

185.14 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, *i.e.*, the number of arguments is read from the signature.

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn

We want to define `\cs_set:Nn` as

```
\cs_set_protected:Npn \cs_set:Nn #1#2
  {
    \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
      { \cs_get_arg_count_from_signature:N #1 } {#2}
  }
```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
\cs_set:Npn \cs_tmp:w #1#2#3
  {
    \cs_new_protected:cpx { cs_ #1 : #2 } ##1##2
```

```
      {
        \exp_not:N \cs_generate_from_arg_count:NNnn ##1
        \exp_after:wN \exp_not:N \cs:w cs_#1 : #3 \cs_end:
          { \exp_not:N \cs_get_arg_count_from_signature:N ##1 }{##2}
      }
  }
```

Then we define the 24 variants beginning with N.

```
\cs_tmp:w { set }                  { Nn } { Npn }
\cs_tmp:w { set }                  { Nx } { Npx }
\cs_tmp:w { set_nopar }            { Nn } { Npn }
\cs_tmp:w { set_nopar }            { Nx } { Npx }
\cs_tmp:w { set_protected }        { Nn } { Npn }
\cs_tmp:w { set_protected }        { Nx } { Npx }
\cs_tmp:w { set_protected_nopar }  { Nn } { Npn }
\cs_tmp:w { set_protected_nopar }  { Nx } { Npx }
\cs_tmp:w { gset }                 { Nn } { Npn }
\cs_tmp:w { gset }                 { Nx } { Npx }
\cs_tmp:w { gset_nopar }           { Nn } { Npn }
\cs_tmp:w { gset_nopar }           { Nx } { Npx }
\cs_tmp:w { gset_protected }       { Nn } { Npn }
\cs_tmp:w { gset_protected }       { Nx } { Npx }
\cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
\cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
\cs_tmp:w { new }                  { Nn } { Npn }
\cs_tmp:w { new }                  { Nx } { Npx }
\cs_tmp:w { new_nopar }            { Nn } { Npn }
\cs_tmp:w { new_nopar }            { Nx } { Npx }
\cs_tmp:w { new_protected }        { Nn } { Npn }
\cs_tmp:w { new_protected }        { Nx } { Npx }
\cs_tmp:w { new_protected_nopar }  { Nn } { Npn }
\cs_tmp:w { new_protected_nopar }  { Nx } { Npx }
```

(*End definition for* `\cs_set:Nn` *and others. These functions are documented on page* **??**.)

\cs_set:cn
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

Then something similar for the c variants.

```
\cs_set_protected:Npn \cs_set:cn #1#2
  {
    \cs_generate_from_arg_count:cNnn {#1} \cs_set:Npn
      { \cs_get_arg_count_from_signature:c {#1} } {#2}
  }
```

The 24 c variants.

```
\cs_set:Npn \cs_tmp:w #1#2#3
  {
    \cs_new_protected:cpx {cs_#1:#2} ##1##2
      {
        \exp_not:N \cs_generate_from_arg_count:cNnn {##1}
        \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
          { \exp_not:N \cs_get_arg_count_from_signature:c {##1} } {##2}
```

```
      }
  }
\cs_tmp:w { set }                   { cn } { Npn }
\cs_tmp:w { set }                   { cx } { Npx }
\cs_tmp:w { set_nopar }             { cn } { Npn }
\cs_tmp:w { set_nopar }             { cx } { Npx }
\cs_tmp:w { set_protected }         { cn } { Npn }
\cs_tmp:w { set_protected }         { cx } { Npx }
\cs_tmp:w { set_protected_nopar }   { cn } { Npn }
\cs_tmp:w { set_protected_nopar }   { cx } { Npx }
\cs_tmp:w { gset }                  { cn } { Npn }
\cs_tmp:w { gset }                  { cx } { Npx }
\cs_tmp:w { gset_nopar }            { cn } { Npn }
\cs_tmp:w { gset_nopar }            { cx } { Npx }
\cs_tmp:w { gset_protected }        { cn } { Npn }
\cs_tmp:w { gset_protected }        { cx } { Npx }
\cs_tmp:w { gset_protected_nopar } { cn } { Npn }
\cs_tmp:w { gset_protected_nopar } { cx } { Npx }
\cs_tmp:w { new }                   { cn } { Npn }
\cs_tmp:w { new }                   { cx } { Npx }
\cs_tmp:w { new_nopar }             { cn } { Npn }
\cs_tmp:w { new_nopar }             { cx } { Npx }
\cs_tmp:w { new_protected }         { cn } { Npn }
\cs_tmp:w { new_protected }         { cx } { Npx }
\cs_tmp:w { new_protected_nopar }   { cn } { Npn }
\cs_tmp:w { new_protected_nopar }   { cx } { Npx }
```

(*End definition for* `\cs_set:cn` *and others. These functions are documented on page* **??**.)

185.15 Checking control sequence equality

`\cs_if_eq_p:NN`
`\cs_if_eq_p:cN`
`\cs_if_eq_p:Nc`
`\cs_if_eq_p:cc`
`\cs_if_eq:NN`*TF*
`\cs_if_eq:cN`*TF*
`\cs_if_eq:Nc`*TF*
`\cs_if_eq:cc`*TF*

Check if two control sequences are identical.

```
\prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1#2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc  \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc  \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:cNT  { \exp_args:Nc  \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:cNF  { \exp_args:Nc  \cs_if_eq:NNF }
\cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:NcT  { \exp_args:NNc \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:NcF  { \exp_args:NNc \cs_if_eq:NNF }
\cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:ccT  { \exp_args:Ncc \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:ccF  { \exp_args:Ncc \cs_if_eq:NNF }
```

(*End definition for* `\cs_if_eq:NN` *and others. These functions are documented on page* **??**.)

185.16 Diagnostic wrapper functions

\kernel_register_show:N
\kernel_register_show:c

Check that the variable, then apply the `\showthe` primitive to the variable. The odd-looking `\use:n` gives a nicer output.

```
\cs_new:Npn \kernel_register_show:N #1
  {
    \cs_if_exist:NTF #1
      { \tex_showthe:D \use:n {#1} }
      {
        \msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_nopar:Npn \kernel_register_show:c
  { \exp_args:Nc \kernel_register_show:N }
```

(*End definition for* `\kernel_register_show:N` *and* `\kernel_register_show:c`*. These functions are documented on page* **??**.)

185.17 Engine specific definitions

\xetex_if_engine_p:
\luatex_if_engine_p:
\pdftex_if_engine_p:
\xetex_if_engine:*TF*
\luatex_if_engine:*TF*
\pdftex_if_engine:*TF*

In some cases it will be useful to know which engine we're running. This can all be hard-coded for speed.

```
\cs_new_eq:NN \luatex_if_engine:T  \use_none:n
\cs_new_eq:NN \luatex_if_engine:F  \use:n
\cs_new_eq:NN \luatex_if_engine:TF \use_ii:nn
\cs_new_eq:NN \pdftex_if_engine:T  \use:n
\cs_new_eq:NN \pdftex_if_engine:F  \use_none:n
\cs_new_eq:NN \pdftex_if_engine:TF \use_i:nn
\cs_new_eq:NN \xetex_if_engine:T   \use_none:n
\cs_new_eq:NN \xetex_if_engine:F   \use:n
\cs_new_eq:NN \xetex_if_engine:TF  \use_ii:nn
\cs_new_eq:NN \luatex_if_engine_p: \c_false_bool
\cs_new_eq:NN \pdftex_if_engine_p: \c_true_bool
\cs_new_eq:NN \xetex_if_engine_p:  \c_false_bool
\cs_if_exist:NT \xetex_XeTeXversion:D
  {
    \cs_gset_eq:NN \pdftex_if_engine:T  \use_none:n
    \cs_gset_eq:NN \pdftex_if_engine:F  \use:n
    \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
    \cs_gset_eq:NN \xetex_if_engine:T   \use:n
    \cs_gset_eq:NN \xetex_if_engine:F   \use_none:n
    \cs_gset_eq:NN \xetex_if_engine:TF  \use_i:nn
    \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
    \cs_gset_eq:NN \xetex_if_engine_p:  \c_true_bool
  }
\cs_if_exist:NT \luatex_directlua:D
  {
    \cs_gset_eq:NN \luatex_if_engine:T  \use:n
    \cs_gset_eq:NN \luatex_if_engine:F  \use_none:n
```

```
    \cs_gset_eq:NN \luatex_if_engine:TF \use_i:nn
    \cs_gset_eq:NN \pdftex_if_engine:T  \use_none:n
    \cs_gset_eq:NN \pdftex_if_engine:F  \use:n
    \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
    \cs_gset_eq:NN \luatex_if_engine_p: \c_true_bool
    \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
  }
```

(*End definition for* `\xetex_if_engine:`*,* `\luatex_if_engine:`*, and* `\pdftex_if_engine:`*. These functions are documented on page* ??.)

185.18 Doing nothing functions

`\prg_do_nothing:`

This does not fit anywhere else!

```
\cs_new_nopar:Npn \prg_do_nothing: { }
```

(*End definition for* `\prg_do_nothing:`*. This function is documented on page* ??.)

185.19 String comparisons

`\str_if_eq_p:nn`
`\str_if_eq_p:xx`
`\str_if_eq:nnTF`
`\str_if_eq:xxTF`

Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient. These should eventually move somewhere else.

```
\prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D { \exp_not:n {#1} } { \exp_not:n {#2} }
      = \c_zero
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\prg_new_conditional:Npnn \str_if_eq:xx #1#2 { p , T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D {#1} {#2} = \c_zero
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\str_if_eq:nn` *and* `\str_if_eq:xx`*. These functions are documented on page* ??.)

185.20 Breaking out of mapping functions

`\prg_break_point:n`
`\prg_map_break:`
`\prg_map_break:n`

In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `\prg_break_point:n`. The breaking functions are then defined to jump to that point and perform the argument of `\prg_break_point:n`, before the user's code (if any).

```
\cs_new_eq:NN \prg_break_point:n \use:n
\cs_new:Npn \prg_map_break: #1 \prg_break_point:n #2 { #2 }
\cs_new:Npn \prg_map_break:n #1 #2 \prg_break_point:n #3 { #3 #1 }
```

(*End definition for* `\prg_break_point:n`*,* `\prg_map_break:`*, and* `\prg_map_break:n`*. These functions are documented on page* ??.)

185.21 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

```
⟨*deprecated⟩
\cs_new_eq:NN           \cs_gnew_nopar:Npn           \cs_new_nopar:Npn
\cs_new_eq:NN                 \cs_gnew:Npn                 \cs_new:Npn
\cs_new_eq:NN \cs_gnew_protected_nopar:Npn  \cs_new_protected_nopar:Npn
\cs_new_eq:NN       \cs_gnew_protected:Npn       \cs_new_protected:Npn
\cs_new_eq:NN           \cs_gnew_nopar:Npx           \cs_new_nopar:Npx
\cs_new_eq:NN                 \cs_gnew:Npx                 \cs_new:Npx
\cs_new_eq:NN \cs_gnew_protected_nopar:Npx  \cs_new_protected_nopar:Npx
\cs_new_eq:NN       \cs_gnew_protected:Npx       \cs_new_protected:Npx
\cs_new_eq:NN           \cs_gnew_nopar:cpn           \cs_new_nopar:cpn
\cs_new_eq:NN                 \cs_gnew:cpn                 \cs_new:cpn
\cs_new_eq:NN \cs_gnew_protected_nopar:cpn  \cs_new_protected_nopar:cpn
\cs_new_eq:NN       \cs_gnew_protected:cpn       \cs_new_protected:cpn
\cs_new_eq:NN           \cs_gnew_nopar:cpx           \cs_new_nopar:cpx
\cs_new_eq:NN                 \cs_gnew:cpx                 \cs_new:cpx
\cs_new_eq:NN \cs_gnew_protected_nopar:cpx  \cs_new_protected_nopar:cpx
\cs_new_eq:NN       \cs_gnew_protected:cpx       \cs_new_protected:cpx
⟨/deprecated⟩
```

```
⟨*deprecated⟩
\cs_new_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
\cs_new_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
\cs_new_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
\cs_new_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc
⟨/deprecated⟩
```

```
⟨*deprecated⟩
\cs_new_eq:NN \cs_gundefine:N  \cs_undefine:N
\cs_new_eq:NN \cs_gundefine:c  \cs_undefine:c
⟨/deprecated⟩
```

```
⟨*deprecated⟩
\cs_new_eq:NN \group_execute_after:N \group_insert_after:N
⟨/deprecated⟩
```

Deprecated 2011-09-06, for removal by 2011-12-31.

`\c_pdftex_is_engine_bool`
`\c_luatex_is_engine_bool`
`\c_xetex_is_engine_bool`

Predicates are better

```
⟨*deprecated⟩
\cs_new_eq:NN \c_luatex_is_engine_bool \luatex_if_engine_p:
\cs_new_eq:NN \c_pdftex_is_engine_bool \pdftex_if_engine_p:
\cs_new_eq:NN \c_xetex_is_engine_bool  \xetex_if_engine_p:
⟨/deprecated⟩
```

(*End definition for* `\c_pdftex_is_engine_bool`, `\c_luatex_is_engine_bool`, *and* `\c_xetex_is_engine_bool`. *These variables are documented on page* **??**.)

`\use_i_after_fi:nw`
`\use_i_after_else:nw`
`\use_i_after_or:nw`
`\use_i_after_orelse:nw`

These functions return the first argument after ending the conditional. This is rather specialized, and we want to de-emphasize the use of primitive TeX conditionals.

```
⟨*deprecated⟩
\cs_set:Npn \use_i_after_fi:nw #1 \fi: { \fi: #1 }
\cs_set:Npn \use_i_after_else:nw #1 \else: #2 \fi: { \fi: #1 }
\cs_set:Npn \use_i_after_or:nw #1 \or: #2 \fi: { \fi: #1 }
\cs_set:Npn \use_i_after_orelse:nw #1#2#3 \fi: { \fi: #1 }
⟨/deprecated⟩
```

(*End definition for* `\use_i_after_fi:nw` *and others. These functions are documented on page* **??**.)

Deprecated 2011-09-07, for removal by 2011-12-31.

`\cs_set_eq:NwN`

```
⟨*deprecated⟩
\tex_let:D \cs_set_eq:NwN \tex_let:D
⟨/deprecated⟩
```

(*End definition for* `\cs_set_eq:NwN`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

186 l3expan implementation

```
⟨*initex | package⟩
```

We start by ensuring that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

`\exp_after:wN`
`\exp_not:N`
`\exp_not:n`

These are defined in l3basics.

(*End definition for* `\exp_after:wN`*. This function is documented on page* *31*.)

186.1 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the LaTeX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.[3])

The definition of expansion functions with this technique happens in section 186.3. In section 186.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l_exp_internal_tl`

We need a scratch token list variable. We don't use `tl` methods so that l3expan can be loaded earlier.

```
\cs_new_nopar:Npn \l_exp_internal_tl { }
```

[3]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.

(*End definition for* `\l_exp_internal_tl`*. This variable is documented on page* 32*.*)

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

An argument manipulator `\::`⟨*Z*⟩ always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed.

`\exp_arg_next:nnn`
`\exp_arg_next:Nnn`

`#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we will require to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

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

(*End definition for* `\exp_arg_next:nnn`*. This function is documented on page* 32*.*)

`\:::` The end marker is just another name for the identity function.

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

(*End definition for* `\:::`*. This function is documented on page* 32*.*)

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

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

(*End definition for* `\::n`*. This function is documented on page* 32*.*)

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

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

(*End definition for* `\::N`*. This function is documented on page* 32*.*)

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

```
\cs_new:Npn \::c #1 \::: #2#3
  { \exp_after:wN \exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for* `\::c`*. This function is documented on page* 32*.*)

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

```
\cs_new:Npn \::o #1 \::: #2#3
  { \exp_after:wN \exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for* `\::o`*. This function is documented on page* 32*.*)

\::f
\exp_stop_f:

This function is used to expand a token list until the first unexpandable token is found. The underlying `\romannumeral -'0` expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once TEX had fully expanded `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` into `\cs_set_eq:NN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NN`. Since the expansion of `\romannumeral -'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.

```
\cs_new:Npn \::f #1 \::: #2#3
  {
    \exp_after:wN \exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D -'0 #3 }
      {#1} {#2}
  }
\use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End definition for* `\::f`. *This function is documented on page* **??**.)

\::x

This function is used to expand an argument fully.

```
\cs_new_protected:Npn \::x #1 \::: #2#3
  {
    \cs_set_nopar:Npx \l_exp_internal_tl { {#3} }
    \exp_after:wN \exp_arg_next:nnn \l_exp_internal_tl {#1} {#2}
  }
```

(*End definition for* `\::x`. *This function is documented on page 32*.)

\::v
\::V

These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim` and `muskip`. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The primitive `\romannumeral` sets off an expansion similar to an `f` type expansion, which we will terminate using `\c_zero`. The argument is returned in braces.

```
\cs_new:Npn \::V #1 \::: #2#3
  {
    \exp_after:wN \exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
      {#1} {#2}
}
\cs_new:Npn \::v # 1\::: #2#3
  {
    \exp_after:wN \exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#3} }
      {#1} {#2}
  }
```

(*End definition for* `\::v`. *This function is documented on page 32*.)

`\exp_eval_register:N`
`\exp_eval_register:c`
`\exp_eval_error_msg:w`

This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```
\cs_new:Npn \exp_eval_register:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

```
! You can't use `\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
      \if_meaning:w \scan_stop: #1
        \exp_eval_error_msg:w
      \fi:
```

The next bit requires some explanation. The function must be initiated by the primitive `\romannumeral` and we want to terminate this expansion chain by inserting the `\c_zero` integer constant. However, we have to expand the register `#1` before we do that. If it is a TeX register, we need to execute the sequence `\exp_after:wN \c_zero \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \c_zero #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
    \else:
      \exp_after:wN \use_i_ii:nnn
    \fi:
    \exp_after:wN \c_zero \tex_the:D #1
  }
\cs_new:Npn \exp_eval_register:c #1
  { \exp_after:wN \exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                           Erroneous variable used!
l.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
\cs_new:Npn \exp_eval_error_msg:w #1 \tex_the:D #2
  {
      \fi:
    \fi:
```

```
    \msg_expandable_kernel_error:nnn { kernel } { bad-var } {#2}
    \c_zero
  }
```

(*End definition for* `\exp_eval_register:N` *and* `\exp_eval_register:c`*. These functions are documented on page* **??**.)

186.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

`\exp_args:No`
`\exp_args:NNo`
`\exp_args:NNNo`

Those lovely runs of expansion!

```
\cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
\cs_new:Npn \exp_args:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
\cs_new:Npn \exp_args:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(*End definition for* `\exp_args:No`*. This function is documented on page 29.*)

`\exp_args:Nc`

In l3basics

(*End definition for* `\exp_args:Nc`*. This function is documented on page 27.*)

`\exp_args:cc`
`\exp_args:NNc`
`\exp_args:Ncc`
`\exp_args:Nccc`

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

```
\cs_new:Npn \exp_args:cc #1#2
  { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
\cs_new:Npn \exp_args:NNc #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
\cs_new:Npn \exp_args:Ncc #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
\cs_new:Npn \exp_args:Nccc #1#2#3#4
  {
    \exp_after:wN #1
      \cs:w #2 \exp_after:wN \cs_end:
      \cs:w #3 \exp_after:wN \cs_end:
      \cs:w #4 \cs_end:
  }
```

(*End definition for* `\exp_args:cc` *and others. These functions are documented on page* **??**.)

`\exp_args:Nf`
`\exp_args:NV`
`\exp_args:Nv`

```
\cs_new:Npn \exp_args:Nf #1#2
  { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -`0 #2 } }
\cs_new:Npn \exp_args:Nv #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \tex_romannumeral:D \exp_eval_register:c {#2} }
  }
\cs_new:Npn \exp_args:NV #1#2
  {
    \exp_after:wN #1 \exp_after:wN
```

```
    { \tex_romannumeral:D \exp_eval_register:N #2 }
  }
```

(*End definition for* `\exp_args:Nf`*,* `\exp_args:NV`*, and* `\exp_args:Nv`*. These functions are documented on page 28.*)

`\exp_args:NNV`
`\exp_args:NNv`
`\exp_args:NNf`
`\exp_args:NVV`
`\exp_args:Ncf`
`\exp_args:Nco`

Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```
\cs_new:Npn \exp_args:NNf #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D -`0 #3 }
  }
\cs_new:Npn \exp_args:NNv #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#3} }
  }
\cs_new:Npn \exp_args:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
  }
\cs_new:Npn \exp_args:Nco #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN {#3}
  }
\cs_new:Npn \exp_args:Ncf #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \tex_romannumeral:D -`0 #3 }
  }
\cs_new:Npn \exp_args:NVV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN { \tex_romannumeral:D \exp_after:wN
      \exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
    \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #3 }
  }
```

(*End definition for* `\exp_args:NNV` *and others. These functions are documented on page* **??**.)

`\exp_args:Ncco`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:NNNV`

A few more that we can hand-tune.

```
\cs_new:Npn \exp_args:NNNV #1#2#3#4
```

```
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #4 }
  }
\cs_new:Npn \exp_args:NcNc #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \cs:w #4 \cs_end:
  }
\cs_new:Npn \exp_args:NcNo #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \exp_after:wN {#4}
  }
\cs_new:Npn \exp_args:Ncco #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \cs:w #3 \exp_after:wN \cs_end:
    \exp_after:wN {#4}
  }
```

(*End definition for* `\exp_args:Ncco` *and others. These functions are documented on page* **??**.)

186.3 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions are all not long: they don't actually take any arguments themselves.

`\exp_args:Nx`

```
\cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }
```

(*End definition for* `\exp_args:Nx`*. This function is documented on page 28.*)

`\exp_args:Nnc`
`\exp_args:Nfo`
`\exp_args:Nff`
`\exp_args:Nnf`
`\exp_args:Nno`
`\exp_args:NnV`
`\exp_args:Noo`
`\exp_args:Nof`
`\exp_args:Noc`
`\exp_args:NNx`
`\exp_args:Ncx`
`\exp_args:Nnx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

Here are the actual function definitions, using the helper functions above.

```
\cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
\cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
\cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
\cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
\cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
\cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
\cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
\cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
```

```
\cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
\cs_new_protected_nopar:Npn \exp_args:Nxx { \::x \::x \::: }
```

(*End definition for* `\exp_args:Nnc` *and others. These functions are documented on page* **??**.)

`\exp_args:NNno`
`\exp_args:NNoo`
`\exp_args:Nnnc`
`\exp_args:Nnno`
`\exp_args:Nooo`
`\exp_args:NNnx`
`\exp_args:NNox`
`\exp_args:Nnnx`
`\exp_args:Nnox`
`\exp_args:Nccx`
`\exp_args:Ncnx`
`\exp_args:Noox`

```
\cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
\cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
\cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
\cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(*End definition for* `\exp_args:NNno` *and others. These functions are documented on page* **??**.)

186.4 Last-unbraced versions

`\exp_arg_last_unbraced:nn`
`\::f_unbraced`
`\::o_unbraced`
`\::V_unbraced`
`\::v_unbraced`
`\::x_unbraced`

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
\cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
\cs_new:Npn \::f_unbraced \::: #1#2
  {
    \exp_after:wN \exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D -`0 #2 } {#1}
  }
\cs_new:Npn \::o_unbraced \::: #1#2
  { \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
\cs_new:Npn \::V_unbraced \::: #1#2
  {
    \exp_after:wN \exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D \exp_eval_register:N #2 } {#1}
  }
\cs_new:Npn \::v_unbraced \::: #1#2
  {
    \exp_after:wN \exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D \exp_eval_register:c {#2} } {#1}
  }
\cs_new_protected:Npn \::x_unbraced \::: #1#2
  {
    \cs_set_nopar:Npx \l_exp_internal_tl { \exp_not:n {#1} #2 }
    \l_exp_internal_tl
```

```
  }
```

(*End definition for* `\exp_arg_last_unbraced:nn`*. This function is documented on page* **??**.)

`\exp_last_unbraced:NV`
`\exp_last_unbraced:Nv`
`\exp_last_unbraced:Nf`
`\exp_last_unbraced:No`
`\exp_last_unbraced:Nco`
`\exp_last_unbraced:NcV`
`\exp_last_unbraced:NNV`
`\exp_last_unbraced:NNo`
`\exp_last_unbraced:NNNV`
`\exp_last_unbraced:NNNo`
`\exp_last_unbraced:Nno`
`\exp_last_unbraced:Noo`
`\exp_last_unbraced:Nfo`
`\exp_last_unbraced:NnNo`
`\exp_last_unbraced:Nx`

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```
\cs_new:Npn \exp_last_unbraced:NV #1#2
  { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:N #2 }
\cs_new:Npn \exp_last_unbraced:Nv #1#2
  { \exp_after:wN #1 \tex_romannumeral:D \exp_eval_register:c {#2} }
\cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
\cs_new:Npn \exp_last_unbraced:Nf #1#2
  { \exp_after:wN #1 \tex_romannumeral:D -`0 #2 }
\cs_new:Npn \exp_last_unbraced:Nco #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\cs_new:Npn \exp_last_unbraced:NcV #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \tex_romannumeral:D \exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \tex_romannumeral:D \exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 #3 }
\cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \tex_romannumeral:D \exp_eval_register:N #4
  }
\cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
\cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
\cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }
```

(*End definition for* `\exp_last_unbraced:NV`*. This function is documented on page 30.*)

`\exp_last_two_unbraced:Noo`

If `#2` is a single token then this can be implemented as

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
 { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
  { \exp_after:wN \exp_last_two_unbraced_aux:noN \exp_after:wN {#3} {#2} #1 }
\cs_new:Npn \exp_last_two_unbraced_aux:noN #1#2#3
   { \exp_after:wN #3 #2 #1 }
```

(*End definition for* `\exp_last_two_unbraced:Noo`. *This function is documented on page 30.*)

186.5 Preventing expansion

`\exp_not:o`
`\exp_not:c`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

```
\cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
\cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\cs_new:Npn \exp_not:f #1
  { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -`0 #1 } }
\cs_new:Npn \exp_not:V #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \tex_romannumeral:D \exp_eval_register:N #1 }
  }
\cs_new:Npn \exp_not:v #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \tex_romannumeral:D \exp_eval_register:c {#1} }
  }
```

(*End definition for* `\exp_not:o`. *This function is documented on page 31.*)

186.6 Defining function variants

`\cs_generate_variant:Nn`
`\cs_generate_variant_aux:nnNNn`
`\cs_generate_variant_aux:Nnnw`
`\cs_generate_variant_aux:NNn`

#1 : Base form of a function; *e.g.*, `\tl_set:Nn`
#2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

Test whether the base function is protected or not and define `\cs_tmp:w` as either `\cs_new_nopar:Npx` or `\cs_new_protected_nopar:Npx`, then used to define all the variants. Split up the original base function to grab its name and signature consisting of k letters. Then we wish to iterate through the list of variant argument specifiers, and for each one construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature. For example, for a base function `\tl_set:Nn` which needs a `c` variant form, we want the new signature to be `cn`.

```
\cs_new_protected:Npn \cs_generate_variant:Nn #1
  {
    \chk_if_exist_cs:N #1
    \cs_generate_variant_aux:N #1
    \cs_split_function:NN #1 \cs_generate_variant_aux:nnNNn
    #1
  }
```

We discard the boolean #3 and then set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```
\cs_new_protected:Npn \cs_generate_variant_aux:nnNNn #1#2#3#4#5
  { \cs_generate_variant_aux:Nnnw #4 {#1}{#2} #5 , ? , \q_recursion_stop }
```

Next is the real work to be done. We now have 1: original function, 2: base name, 3: base signature, 4: beginning of variant signature. To construct the new csname and the `\exp_args:Ncc` form, we need the variant signature. In our example, we wanted to discard the first two letters of the base signature because the variant form started with `cc`. This is the same as putting first `cc` in the signature and then `\use_none:nn` followed by the base signature `NNn`. Depending on the number of characters in #4, the relevant `\use_none:n...n` is called.

Firstly though, we check whether to terminate the loop. Then build the variant function once, to avoid repeating this relatively expensive operation. Then recurse.

```
\cs_new_protected:Npn \cs_generate_variant_aux:Nnnw #1#2#3#4 ,
  {
    \if:w ? #4
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \exp_args:NNc \cs_generate_variant_aux:NNn
      #1
      {
        #2 : #4
        \exp_after:wN \use_i_delimit_by_q_stop:nw
        \use_none:nnnnnnnnn #4
        \use_none:nnnnnnnnn
        \use_none:nnnnnnnn
        \use_none:nnnnnnn
        \use_none:nnnnnn
        \use_none:nnnnn
        \use_none:nnnn
        \use_none:nnn
        \use_none:nn
        \use_none:n
        { }
        \q_stop
        #3
      }
      {#4}
    \cs_generate_variant_aux:Nnnw #1 {#2} {#3}
  }
```

Check if the variant form has already been defined. If not, then define it and then additionally check if the `\exp_args:N` form needed is defined. Otherwise tell that it was already defined.

```
\cs_new_protected:Npn \cs_generate_variant_aux:NNn #1 #2 #3
  {
    \cs_if_free:NTF #2
      {
```

```
      \cs_tmp:w #2 { \exp_not:c { exp_args:N #3 } \exp_not:N #1 }
      \cs_generate_internal_variant:n {#3}
    }
    {
      \iow_log:x
        {
          Variant~\token_to_str:N #2~%
          already~defined;~ not~ changing~ it~on~line~%
          \tex_the:D \tex_inputlineno:D
        }
    }
  }
```

(*End definition for* `\cs_generate_variant:Nn`*. This function is documented on page 26.*)

`\cs_generate_variant_aux:N`
`\cs_generate_variant_aux:w`

The idea here is to pick up protected parent functions, using the nature of the meaning string that they generate. The test here is almost the same as `\tl_if_empty:nTF`, but has to be hard-coded as that function is not yet available and because it has to match both long and short macros.

```
\group_begin:
  \tex_lccode:D `\Z = `\d \scan_stop:
  \tex_lccode:D `\? =`\\ \scan_stop:
  \tex_catcode:D `\P = 12 \scan_stop:
  \tex_catcode:D `\R = 12 \scan_stop:
  \tex_catcode:D `\O = 12 \scan_stop:
  \tex_catcode:D `\T = 12 \scan_stop:
  \tex_catcode:D `\E = 12 \scan_stop:
  \tex_catcode:D `\C = 12 \scan_stop:
  \tex_catcode:D `\Z = 12 \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \cs_generate_variant_aux:N #1
      {
        \exp_after:wN \cs_generate_variant_aux:w
          \token_to_meaning:N #1
          \q_mark \cs_new_protected_nopar:Npx
          ? PROTECTEZ
          \q_mark \cs_new_nopar:Npx
          \q_stop
      }
    \cs_new_protected:Npn \cs_generate_variant_aux:w
        #1 ? PROTECTEZ #2 \q_mark #3 #4 \q_stop
      {
        \cs_set_eq:NN \cs_tmp:w #3
      }
  }
```

(*End definition for* `\cs_generate_variant_aux:N`*. This function is documented on page 26.*)

`\cs_generate_internal_variant:n`
`\cs_generate_internal_variant_aux:N`

Test if `exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in `#1`

```
\cs_new_protected:Npn \cs_generate_internal_variant:n #1
  {
    \cs_if_free:cT { exp_args:N #1 }
      {
        \cs_new:cpx { exp_args:N #1 }
          { \cs_generate_internal_variant_aux:N #1 : }
      }
  }
```

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.

```
\cs_new:Npn \cs_generate_internal_variant_aux:N #1
  {
    \exp_not:c { :: #1 }
    \if_meaning:w : #1
      \exp_after:wN \use_none:n
    \fi:
    \cs_generate_internal_variant_aux:N
  }
```

(*End definition for* `\cs_generate_internal_variant:n`*. This function is documented on page 32.*)

186.7 Variants which cannot be created earlier

`\str_if_eq_p:Vn`
`\str_if_eq_p:on`
`\str_if_eq_p:nV`
`\str_if_eq_p:no`
`\str_if_eq_p:VV`
`\str_if_eq:VnTF`
`\str_if_eq:onTF`
`\str_if_eq:nVTF`
`\str_if_eq:noTF`
`\str_if_eq:VVTF`

These cannot come earlier as they need `\cs_generate_variant:Nn`.

```
\cs_generate_variant:Nn \str_if_eq_p:nn {  V ,  o }
\cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnT  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnT  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnF  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnF  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnTF {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
```

(*End definition for* `\str_if_eq:Vn` *and others. These functions are documented on page* ??.)

```
⟨/initex | package⟩
```

187 l3prg implementation

The following test files are used for this code: m3prg001.lvt,m3prg002.lvt,m3prg003.lvt.

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

187.1 Primitive conditionals

\if_bool:N
\if_predicate:w

Those two primitive TeX conditionals are synonyms. They should not be used outside the kernel code.

```
\tex_let:D \if_bool:N         \tex_ifodd:D
\tex_let:D \if_predicate:w    \tex_ifodd:D
```

(*End definition for* `\if_bool:N`*. This function is documented on page* 42*.*)

187.2 Defining a set of conditional functions

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\prg_return_true:
\prg_return_false:

These are all defined in l3basics, as they are needed "early". This is just a reminder that that is the case!

(*End definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page* **??***.*)

187.3 The boolean data type

\bool_new:N
\bool_new:c

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_new:N { c }
```

(*End definition for* `\bool_new:N` *and* `\bool_new:c`*. These functions are documented on page* **??***.*)

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

Setting is already pretty easy.

```
\cs_new_protected:Npn \bool_set_true:N #1
  { \cs_set_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_set_false:N #1
  { \cs_set_eq:NN #1 \c_false_bool }
\cs_new_protected:Npn \bool_gset_true:N #1
  { \cs_gset_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_gset_false:N #1
  { \cs_gset_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_set_true:N   { c }
\cs_generate_variant:Nn \bool_set_false:N  { c }
\cs_generate_variant:Nn \bool_gset_true:N  { c }
\cs_generate_variant:Nn \bool_gset_false:N { c }
```

(*End definition for* `\bool_set_true:N` *and others. These functions are documented on page* **??***.*)

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

The usual copy code.

```
\cs_new_eq:NN \bool_set_eq:NN  \cs_set_eq:NN
\cs_new_eq:NN \bool_set_eq:Nc  \cs_set_eq:Nc
\cs_new_eq:NN \bool_set_eq:cN  \cs_set_eq:cN
\cs_new_eq:NN \bool_set_eq:cc  \cs_set_eq:cc
\cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc
```

(*End definition for* `\bool_set_eq:NN` *and others. These functions are documented on page* **??***.*)

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`.

```
\cs_new_protected:Npn \bool_set:Nn #1#2
  { \tex_chardef:D #1 = \bool_if_p:n {#2} }
\cs_new_protected:Npn \bool_gset:Nn #1#2
  { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
\cs_generate_variant:Nn \bool_set:Nn  { c }
\cs_generate_variant:Nn \bool_gset:Nn { c }
```

(*End definition for* `\bool_set:Nn` *and* `\bool_set:cn`*. These functions are documented on page* **??**.)

\bool_if_p:N
\bool_if_p:c
\bool_if:N*TF*
\bool_if:c*TF*

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
  {
    \if_meaning:w \c_true_bool #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \bool_if_p:N { c }
\cs_generate_variant:Nn \bool_if:NT  { c }
\cs_generate_variant:Nn \bool_if:NF  { c }
\cs_generate_variant:Nn \bool_if:NTF { c }
```

(*End definition for* `\bool_set:Nn` *and* `\bool_set:cn`*. These functions are documented on page* **??**.)

\bool_show:N
\bool_show:c
\bool_show:n

Show the truth value of the boolean, as `true` or `false`. We use `\msg_aux_show:x` to get a better output; this function requires its argument to start with `>`.

```
\cs_new_protected:Npn \bool_show:N #1
  {
    \bool_if_exist:NTF #1
      { \bool_show:n {#1} }
      {
        \msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \bool_show:n #1
  {
    \bool_if:nTF {#1}
      { \msg_aux_show:x { > true } }
      { \msg_aux_show:x { > false } }
  }
\cs_generate_variant:Nn \bool_show:N { c }
```

(*End definition for* `\bool_show:N`*,* `\bool_show:c`*, and* `\bool_show:n`*. These functions are documented on page 36.*)

\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
```

(*End definition for* `\l_tmpa_bool` *and* `\g_tmpa_bool`*. These variables are documented on page 36.*)

`\bool_if_exist_p:N`
`\bool_if_exist_p:c`
`\bool_if_exist:NTF`
`\bool_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \bool_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \bool_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \bool_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \bool_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \bool_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \bool_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \bool_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \bool_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\bool_if_exist:N` *and* `\bool_if_exist:c`*. These functions are documented on page* ??.)

187.4 Boolean expressions

`\bool_if_p:n`
`\bool_if:nTF`
`\bool_get_next:N`
`\bool_cleanup:N`
`\bool_choose:NN`
`bool_!:w`
`\bool_Not:w`
`\bool_Not:w`
`\bool_(:w`
`\bool_p:w`
`\bool_8_1:w`
`\bool_I_1:w`
`\bool_8_0:w`
`\bool_I_0:w`
`\bool_)_0:w`
`\bool_)_1:w`
`\bool_S_0:w`
`\bool_S_1:w`
`\bool_eval_skip_to_end:Nw`
`\bool_eval_skip_to_end_aux:Nw`
`\bool_eval_skip_to_end_aux_ii:Nw`

Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical "Not", `&&` for logical "And" and `||` for logical "Or". We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, insert a negating function (if-even in this case) and call GetNext.
- If none of the above, start evaluating a new expression by reinserting the token found (this is supposed to be a predicate function) in front of Eval.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

⟨*true*⟩And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*false*⟩And Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return ⟨*false*⟩.

⟨*true*⟩Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return ⟨*true*⟩.

⟨*false*⟩Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*true*⟩Close Current truth value is true, Close seen, return ⟨*true*⟩.

⟨*false*⟩Close Current truth value is false, Close seen, return ⟨*false*⟩.

We introduce an additional Stop operation with the following semantics:

⟨***true***⟩**Stop** Current truth value is true, return ⟨*true*⟩.

⟨***false***⟩**Stop** Current truth value is false, return ⟨*false*⟩.

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\int_value:w:D` operation. First we issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for TEX. We also need to finish this special group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a `S` following the last Close operation.

```
\prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_if_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \bool_if_p:n #1
  {
    \group_align_safe_begin:
    \bool_get_next:N ( #1 ) S
  }
```

The GetNext operation. We make it a switch: If not a `!` or `(`, we assume it is a predicate.

```
\cs_new:Npn \bool_get_next:N #1
  {
    \use:c
      {
        bool_
        \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
        :w
      }
      #1
  }
```

This variant gets called when a Not has just been entered. It (eventually) results in a reversal of the logic of the directly following material.

```
\cs_new:Npn \bool_get_not_next:N #1
  {
    \use:c
      {
      bool_not_
      \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
      :w
      }
```

```
        #1
  }
```

We need these later on to nullify the unity operation !!.

```
\cs_new:Npn \bool_get_next:NN #1#2 { \bool_get_next:N #2 }
\cs_new:Npn \bool_get_not_next:NN #1#2 { \bool_get_not_next:N #2 }
```

The Not operation. Discard the token read and reverse the truth value of the next expression if there are brackets; otherwise if we're coming up to a ! then we don't need to reverse anything (but we then want to continue scanning ahead in case some fool has written !!(...)); otherwise we have a boolean that we can reverse here and now.

```
\cs_new:cpn { bool_!:w } #1#2
  {
    \if_meaning:w ( #2
      \exp_after:wN \bool_Not:w
    \else:
      \if_meaning:w ! #2
        \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
      \fi:
    \fi:
    #2
  }
```

Variant called when already inside a Not. Essentially the opposite of the above.

```
\cs_new:cpn { bool_not_!:w } #1#2
  {
    \if_meaning:w ( #2
      \exp_after:wN \bool_not_Not:w
    \else:
      \if_meaning:w ! #2
        \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
      \fi:
    \fi:
    #2
  }
```

These occur when processing !(...). The idea is to use a variant of \bool_get_next:N that finishes its parsing with a logic reversal. Of course, the double logic reversal gets us back to where we started.

```
\cs_new:Npn \bool_Not:w { \exp_after:wN \int_value:w \bool_get_not_next:N }
\cs_new:Npn \bool_not_Not:w { \exp_after:wN \int_value:w \bool_get_next:N }
```

These occur when processing !<bool> and can be evaluated directly.

```
\cs_new:Npn \bool_Not:N #1
  {
    \exp_after:wN \bool_p:w
    \if_meaning:w #1 \c_true_bool
```

```
      \c_false_bool
    \else:
      \c_true_bool
    \fi:
  }
\cs_new:Npn \bool_not_Not:N #1
  {
    \exp_after:wN \bool_p:w
    \if_meaning:w #1 \c_true_bool
      \c_true_bool
    \else:
      \c_false_bool
    \fi:
  }
```

The Open operation. Discard the token read and start a sub-expression. `\bool_get_next:N` continues building up the logical expressions as usual; `\bool_not_cleanup:N` is what reverses the logic if we're inside `!(...)`.

```
\cs_new:cpn { bool_(:w } #1
  { \exp_after:wN  \bool_cleanup:N \int_value:w \bool_get_next:N }
\cs_new:cpn { bool_not_(:w } #1
  { \exp_after:wN  \bool_not_cleanup:N \int_value:w \bool_get_next:N }
```

Otherwise just evaluate the predicate and look for And, Or or Close afterwards.

```
\cs_new:cpn { bool_p:w } { \exp_after:wN \bool_cleanup:N \int_value:w  }
\cs_new:cpn { bool_not_p:w } {\exp_after:wN \bool_not_cleanup:N \int_value:w }
```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```
\cs_new:Npn \bool_cleanup:N #1
  {
    \exp_after:wN \bool_choose:NN \exp_after:wN #1
    \int_to_roman:w - `\q
  }
\cs_new:Npn \bool_not_cleanup:N #1
  {
    \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
    \int_to_roman:w - `\q
  }
```

Branching the six way switch. Reversals should be reasonably straightforward.

```
\cs_new:Npn \bool_choose:NN #1#2 { \use:c { bool_ #2 _ #1 :w } }
\cs_new:Npn \bool_not_choose:NN #1#2 { \use:c { bool_not_ #2 _ #1 :w } }
```

Continues scanning. Must remove the second `&` or `|`.

```
\cs_new_nopar:cpn { bool_&_1:w } & { \bool_get_next:N }
\cs_new_nopar:cpn { bool_|_0:w } | { \bool_get_next:N }
\cs_new_nopar:cpn { bool_not_&_0:w } & { \bool_get_next:N }
\cs_new_nopar:cpn { bool_not_|_1:w } | { \bool_get_next:N }
```

Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```
\cs_new_nopar:cpn { bool_)_0:w } { \c_false_bool }
\cs_new_nopar:cpn { bool_)_1:w } { \c_true_bool }
\cs_new_nopar:cpn { bool_not_)_0:w } { \c_true_bool }
\cs_new_nopar:cpn { bool_not_)_1:w } { \c_false_bool }
\cs_new_nopar:cpn { bool_S_0:w } { \group_align_safe_end: \c_false_bool }
\cs_new_nopar:cpn { bool_S_1:w } { \group_align_safe_end: \c_true_bool }
```

When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the `()` manually.

```
\cs_new_nopar:cpn { bool_&_0:w } & { \bool_eval_skip_to_end:Nw \c_false_bool }
\cs_new_nopar:cpn { bool_|_1:w } | { \bool_eval_skip_to_end:Nw \c_true_bool }
\cs_new_nopar:cpn { bool_not_&_1:w } &
  { \bool_eval_skip_to_end:Nw \c_false_bool }
\cs_new_nopar:cpn { bool_not_|_0:w } |
  { \bool_eval_skip_to_end:Nw \c_true_bool }
```

There is always at least one `)` waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool  && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a `()` pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an Open so we remove another `()` pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a Close and again find Open tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

```
%% (
\cs_new:Npn \bool_eval_skip_to_end:Nw #1#2 )
  {
    \bool_eval_skip_to_end_aux:Nw #1#2 ( % )
    \q_no_value \q_stop
    {#2}
  }
```

If no right parenthesis, then `#3` is no_value and we are done, return the boolean `#1`. If there is, we need to grab a `()` pair and then recurse

```
\cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2 ( #3#4 \q_stop #5 % )
  {
    \quark_if_no_value:NTF #3
    {#1}
    { \bool_eval_skip_to_end_aux_ii:Nw #1 #5 }
  }
```

Keep the boolean, throw away anything up to the `(` as it is irrelevant, remove a `()` pair but remember to reinsert `#3` as it may contain `(` tokens!

```
\cs_new:Npn \bool_eval_skip_to_end_aux_ii:Nw #1#2 ( #3 )
  { % (
    \bool_eval_skip_to_end:Nw #1#3 )
  }
```

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
\cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
\cs_new:Npn \bool_xor_p:nn #1#2
  {
    \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
      \c_false_bool
      \c_true_bool
  }
```

187.5 Logical loops

\bool_while_do:Nn
\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn

A `while` loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

```
\cs_new:Npn \bool_while_do:Nn #1#2
  { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
\cs_new:Npn \bool_until_do:Nn #1#2
  { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_while_do:Nn { c }
\cs_generate_variant:Nn \bool_until_do:Nn { c }
```

\bool_do_while:Nn
\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn

A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```
\cs_new:Npn \bool_do_while:Nn #1#2
  { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
\cs_new:Npn \bool_do_until:Nn #1#2
  { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_do_while:Nn { c }
\cs_generate_variant:Nn \bool_do_until:Nn { c }
```

\bool_while_do:nn
\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn

Loop functions with the test either before or after the first body expansion.

```
\cs_new:Npn \bool_while_do:nn #1#2
  {
    \bool_if:nT {#1}
      {
        #2
        \bool_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_while:nn #1#2
  {
    #2
    \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \bool_until_do:nn #1#2
  {
    \bool_if:nF {#1}
      {
        #2
        \bool_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_until:nn #1#2
  {
    #2
    \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2}  }
  }
```

187.6 Switching by case

A family of functions to select one case of a number: the same ideas are used for a number of different situations.

\prg_case_end:nw

In all cases the end statement is the same. Here, #1 will be the code needed, #2 the other cases to throw away, including the "else" case. The \c_zero marker stops the expansion of \romannumeral which begins each \prg_case_... function.

```
\cs_new:Npn \prg_case_end:nw #1 #2 \q_recursion_stop { \c_zero #1 }
```

\prg_case_int:nnn
\prg_case_int_aux:nnn
\prg_case_int_aux:nw

For integer cases, the first task to fully expand the check condition. After that, a loop is started to compare each possible value and stop if the test is true. The tested value is put at the end to ensure that there is necessarily a match, which will fire the "else" pathway. The leading \romannumeral triggers an expansion which is then stopped in \prg_case_end:nw.

```
\cs_new:Npn \prg_case_int:nnn #1
  {
    \tex_romannumeral:D
    \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n {#1} }
  }
\cs_new:Npn \prg_case_int_aux:nnn #1 #2 #3
  { \prg_case_int_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
\cs_new:Npn \prg_case_int_aux:nw #1#2#3
  {
    \int_compare:nNnTF {#1} = {#2}
      { \prg_case_end:nw {#3} }
      { \prg_case_int_aux:nw {#1} }
  }
```

\prg_case_dim:nnn
\prg_case_dim_aux:nnn
\prg_case_dim_aux:nw

The dimension function is the same, just a change of calculation method.

```
\cs_new:Npn \prg_case_dim:nnn #1
  {
    \tex_romannumeral:D
    \exp_args:Nf \prg_case_dim_aux:nnn { \dim_eval:n {#1} }
  }
\cs_new:Npn \prg_case_dim_aux:nnn #1 #2 #3
  { \prg_case_dim_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop }
\cs_new:Npn \prg_case_dim_aux:nw #1#2#3
  {
    \dim_compare:nNnTF {#1} = {#2}
      { \prg_case_end:nw {#3} }
      { \prg_case_dim_aux:nw {#1} }
  }
```

\prg_case_str:nnn
\prg_case_str:onn
\prg_case_str:xxn
\prg_case_str_aux:nw
\prg_case_str_x_aux:nw

No calculations for strings, otherwise no surprises.

```
\cs_new:Npn \prg_case_str:nnn #1#2#3
  {
    \tex_romannumeral:D
    \prg_case_str_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop
```

```
  }
\cs_new:Npn \prg_case_str_aux:nw #1#2#3
  {
    \str_if_eq:nnTF {#1} {#2}
      { \prg_case_end:nw {#3} }
      { \prg_case_str_aux:nw {#1} }
  }
\cs_generate_variant:Nn \prg_case_str:nnn { o }
\cs_new:Npn \prg_case_str:xxn #1#2#3
  {
    \tex_romannumeral:D
    \prg_case_str_x_aux:nw {#1} #2 {#1} {#3} \q_recursion_stop
  }
\cs_new:Npn \prg_case_str_x_aux:nw #1#2#3
  {
    \str_if_eq:xxTF {#1} {#2}
      { \prg_case_end:nw {#3} }
      { \prg_case_str_x_aux:nw {#1} }
  }
```

\prg_case_tl:Nnn
\prg_case_tl:cnn
\prg_case_tl_aux:Nw

Similar again, but this time with some variants.

```
\cs_new:Npn \prg_case_tl:Nnn #1#2#3
  {
    \tex_romannumeral:D
    \prg_case_tl_aux:Nw #1 #2 #1 {#3} \q_recursion_stop
  }
\cs_new:Npn \prg_case_tl_aux:Nw #1#2#3
  {
    \tl_if_eq:NNTF #1 #2
      { \prg_case_end:nw {#3} }
      { \prg_case_tl_aux:Nw #1 }
  }
\cs_generate_variant:Nn \prg_case_tl:Nnn { c }
```

187.7 Producing n copies

\prg_replicate:nn
\prg_replicate_aux:N
\prg_replicate_first_aux:N
\prg_replicate_
\prg_replicate_0:n
\prg_replicate_1:n
\prg_replicate_2:n
\prg_replicate_3:n
\prg_replicate_4:n
\prg_replicate_5:n
\prg_replicate_6:n
\prg_replicate_7:n
\prg_replicate_8:n
\prg_replicate_9:n
\prg_replicate_first_-:n
\prg_replicate_first_0:n
\prg_replicate_first_1:n
\prg_replicate_first_2:n
\prg_replicate_first_3:n
\prg_replicate_first_4:n
\prg_replicate_first_5:n
\prg_replicate_first_6:n

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input `25` result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\int_to_roman:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it 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, and if necessary, it is possible to write `\prg_replicate:nn{1000}{\prg_replicate:nn{1000}{`⟨*code*⟩ An alternative approach is to create a string of `m`'s with `\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.

```
\cs_new:Npn \prg_replicate:nn #1
  {
    \int_to_roman:w
      \exp_after:wN \prg_replicate_first_aux:N
        \int_value:w \int_eval:w #1 \int_eval_end:
      \cs_end:
  }
\cs_new:Npn \prg_replicate_aux:N #1
  { \cs:w prg_replicate_#1 :n \prg_replicate_aux:N }
\cs_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.

```
\cs_new:Npn \prg_replicate_ :n #1 { \cs_end: }
\cs_new:cpn { prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
\cs_new:cpn { prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
\cs_new:cpn { prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
\cs_new:cpn { prg_replicate_3:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
\cs_new:cpn { prg_replicate_4:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
\cs_new:cpn { prg_replicate_5:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
\cs_new:cpn { prg_replicate_6:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
\cs_new:cpn { prg_replicate_7:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
\cs_new:cpn { prg_replicate_8:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
\cs_new:cpn { prg_replicate_9:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
```

Users shouldn't ask for something to be replicated once or even not at all but…

```
\cs_new:cpn { prg_replicate_first_-:n } #1
  { \c_zero \msg_expandable_kernel_error:nn { prg } { replicate-neg } }
\cs_new:cpn { prg_replicate_first_0:n } #1 { \c_zero }
\cs_new:cpn { prg_replicate_first_1:n } #1 { \c_zero #1 }
\cs_new:cpn { prg_replicate_first_2:n } #1 { \c_zero #1#1 }
\cs_new:cpn { prg_replicate_first_3:n } #1 { \c_zero #1#1#1 }
\cs_new:cpn { prg_replicate_first_4:n } #1 { \c_zero #1#1#1#1 }
\cs_new:cpn { prg_replicate_first_5:n } #1 { \c_zero #1#1#1#1#1 }
\cs_new:cpn { prg_replicate_first_6:n } #1 { \c_zero #1#1#1#1#1#1 }
```

```
\cs_new:cpn { prg_replicate_first_7:n } #1 { \c_zero #1#1#1#1#1#1#1 }
\cs_new:cpn { prg_replicate_first_8:n } #1 { \c_zero #1#1#1#1#1#1#1#1 }
\cs_new:cpn { prg_replicate_first_9:n } #1 { \c_zero #1#1#1#1#1#1#1#1#1 }
```

(*End definition for* `\bool_if:n`*. These functions are documented on page 40.*)

`\prg_stepwise_function:nnnN`
`\prg_stepwise_aux:nnnN`
`\prg_stepwise_aux:NnnnN`

Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
\cs_new:Npn \prg_stepwise_function:nnnN #1#2#3#4
  {
    \prg_stepwise_aux:nnnN {#1} {#2} {#3} #4
    \prg_break_point:n { }
  }
\cs_new:Npn \prg_stepwise_aux:nnnN #1#2#3#4
  {
    \int_compare:nNnTF {#2} > \c_zero
      { \exp_args:NNf \prg_stepwise_aux:NnnnN > }
      {
        \int_compare:nNnTF {#2} = \c_zero
          {
            \msg_expandable_kernel_error:nnn { prg } { zero-step } {#4}
            \prg_map_break:
          }
          { \exp_args:NNf \prg_stepwise_aux:NnnnN < }
      }
      { \int_eval:n {#1} } {#2} {#3} #4
  }
\cs_new:Npn \prg_stepwise_aux:NnnnN #1#2#3#4#5
  {
    \int_compare:nNnF {#2} #1 {#4}
      {
        #5 {#2}
        \exp_args:NNf \prg_stepwise_aux:NnnnN
          #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
      }
  }
```

(*End definition for* `\prg_stepwise_function:nnnN`*. This function is documented on page 40.*)

`\prg_stepwise_inline:nnnn`
`\prg_stepwise_variable:nnnNn`
`\prg_stepwise_aux:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\prg_stepwise_function:nnnN`.

```
\cs_new_protected:Npn \prg_stepwise_inline:nnnn
  {
    \exp_args:NNc \prg_stepwise_aux:NNnnnn
      \cs_gset_nopar:Npn
      { g_prg_stepwise_ \int_use:N \g_prg_map_int :n }
  }
\cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2#3#4#5
```

```
  {
    \exp_args:NNc \prg_stepwise_aux:NNnnnn
      \cs_gset_nopar:Npx
      { g_prg_stepwise_ \int_use:N \g_prg_map_int :n }
      {#1}{#2}{#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \prg_stepwise_aux:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \int_gincr:N \g_prg_map_int
    \prg_stepwise_aux:nnnN {#3} {#4} {#5} #2
    \prg_break_point:n { \int_gdecr:N \g_prg_map_int }
  }
```

(*End definition for* `\prg_stepwise_inline:nnnn`. *This function is documented on page 41.*)

187.8 Detecting TEX's mode

`\mode_if_vertical_p:`
`\mode_if_vertical:TF`

For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
\prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
  { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_vertical:`. *These functions are documented on page* ??.)

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF`

For testing horizontal mode.

```
\prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
  { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_horizontal:`. *These functions are documented on page* ??.)

`\mode_if_inner_p:`
`\mode_if_inner:TF`

For testing inner mode.

```
\prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
  { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_inner:`. *These functions are documented on page* ??.)

`\mode_if_math_p:`
`\mode_if_math:TF`

For testing math mode. At the beginning of an alignment cell, the programmer should insert `\scan_align_safe_stop:` before the test.

```
\prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
  { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_math:`. *These functions are documented on page* ??.)

187.9 Internal programming functions

`\group_align_safe_begin:`
`\group_align_safe_end:`

TEX's alignment structures present many problems. As Knuth says himself in *TEX: The Program*: "It's sort of a miracle whenever `\halign` or `\valign` work, [...]" One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we will get some sort of weird error message because the underlying `\futurelet` 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*... We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
\cs_new_nopar:Npn \group_align_safe_begin:
  { \if_int_compare:w \if_false: { \fi: `} = \c_zero \fi: }
\cs_new_nopar:Npn \group_align_safe_end:
  { \if_int_compare:w `{ = \c_zero } \fi: }
```

(*End definition for* `\group_align_safe_begin:` *and* `\group_align_safe_end:`*. These functions are documented on page* **??***.*)

`\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 `\omit` or `\noalign` and hasn't looked at the preamble yet. Thus an `\ifmmode` 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[4] 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 if an only if 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. Thus an older definition here was

```
\cs_new_nopar:Npn \scan_align_safe_stop:
  {
    \int_compare:nNnT \etex_currentgrouptype:D = \c_six
      {
        \int_compare:nNnF \etex_lastnodetype:D = \c_zero
          {
            \int_compare:nNnF \etex_lastnodetype:D = \c_seven
              { \scan_stop: }
          }
      }
  }
```

[4] Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

However, this is not truly expandable, as there are places where the `\scan_stop:` ends up in the result. A simpler alternative, which can be used selectively, is therefore defined.

```
\cs_new_protected_nopar:Npn \scan_align_safe_stop: { }
```

(*End definition for* `\scan_align_safe_stop:`. *This function is documented on page* **??**.)

`\prg_variable_get_scope:N`
`\prg_variable_get_scope_aux:w`
`\prg_variable_get_type:N`
`\prg_variable_get_type:w`

Expandable functions to find the type of a variable, and to return `g` if the variable is global. The trick for `\prg_variable_get_scope:N` is the same as that in `\cs_split_function:NN`, but it can be simplified as the requirements here are less complex.

```
\group_begin:
  \tex_lccode:D `\& = `\g \scan_stop:
  \tex_catcode:D `\& = \c_twelve
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \prg_variable_get_scope:N #1
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \prg_variable_get_scope_aux:w
          \cs_to_str:N #1 \exp_stop_f: \q_stop
      }
    \cs_new:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop
      { \token_if_eq_meaning:NNT & #1 { g } }
  }
\group_begin:
  \tex_lccode:D `\& = `\_ \scan_stop:
  \tex_catcode:D `\& = \c_twelve
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \prg_variable_get_type:N #1
      {
        \exp_after:wN \prg_variable_get_type_aux:w
          \token_to_str:N #1 & a \q_stop
      }
    \cs_new:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop
      {
        \token_if_eq_meaning:NNTF a #2
          {#1}
          { \prg_variable_get_type_aux:w #2#3 \q_stop }
      }
  }
```

(*End definition for* `\prg_variable_get_scope:N`. *This function is documented on page 42.*)

`\g_prg_map_int`

A nesting counter for mapping.

```
\int_new:N \g_prg_map_int
```

(*End definition for* `\g_prg_map_int`. *This variable is documented on page* **??**.)

`\prg_break_point:n`
`\prg_map_break:`
`\prg_map_break:n`

These are all defined in l3basics, as they are needed "early". This is just a reminder that that is the case!

(*End definition for* `\prg_break_point:n`. *This function is documented on page* **??**.)

187.10 Deprecated functions

These were deprecated on 2012-02-08, and will be removed entirely by 2012-05-31.

`\prg_define_quicksort:nnn`

`#1` is the name, `#2` and `#3` are the tokens enclosing the argument. For the somewhat strange ⟨*clist*⟩ type which doesn't enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```
\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}
```

defines the user function `\seq_quicksort:n` and furthermore expects to use the two functions `\seq_quicksort_compare:nnTF` which compares the items and `\seq_quicksort_function:n` which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the `seq` type a sequence is a token list variable, so one additionally has to define

```
\cs_set_nopar:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}
```

For details on the implementation see "Sorting in TEX's Mouth" by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```
\cs_new_protected:Npn \prg_define_quicksort:nnn #1#2#3 {
  \cs_set:cpx{#1_quicksort:n}##1{
    \exp_not:c{#1_quicksort_start_partition:w} ##1
    \exp_not:n{#2\q_nil#3\q_stop}
  }
  \cs_set:cpx{#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
  }
  \cs_set: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}{}{}
  }
  \cs_set:cpx {#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.

```
  \cs_set: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}
```

```
  }
  \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##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}
  }
  \cs_set: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}
  }
  \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##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.

```
  \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{{##4}##3}}
  \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3{##4}}}
  \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{{##4}##2}{##3}}
  \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2{##4}}{##3}}
  \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{{##4}##3}}
  \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3{##4}}}
  \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
    \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{{##4}##2}{##3}}
  \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##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.

```
  \cs_set: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}
  }
}
```

(*End definition for* `\prg_define_quicksort:nnn`.)

`\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}{}{}
```

(*End definition for* `\prg_quicksort:n`. *This function is documented on page* ??.)

`\prg_quicksort_function:n`
`\prg_quicksort_compare:nnTF`

```
\cs_set:Npn \prg_quicksort_function:n {\ERROR}
\cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}
```

(*End definition for* `\prg_quicksort_function:n`. *This function is documented on page* ??.)

These were deprecated on 2011-05-27 and will be removed entirely by 2011-08-31.

`\prg_new_map_functions:Nn`
`\prg_set_map_functions:Nn`

As we have restructured the structured variables, these are no longer needed.

```
⟨*deprecated⟩
\cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 { \deprecated }
\cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 { \deprecated }
⟨/deprecated⟩
```

(*End definition for* `\prg_new_map_functions:Nn`. *This function is documented on page* ??.)

```
⟨/initex | package⟩
```

188 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

188.1 Quarks

`\quark_new:N`

Allocate a new quark.

```
\cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }
```

(*End definition for* `\quark_new:N`. *This function is documented on page* 44.)

\q_nil
\q_mark
\q_no_value
\q_stop

Some "public" quarks. \q_stop is an "end of argument" marker, \q_nil is a empty value and \q_no_value marks an empty argument.

```
\quark_new:N \q_nil
\quark_new:N \q_mark
\quark_new:N \q_no_value
\quark_new:N \q_stop
```

(*End definition for* \q_nil *and others. These variables are documented on page 44.*)

\q_recursion_tail
\q_recursion_stop

Quarks for ending recursions. Only ever used there! \q_recursion_tail is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. \q_recursion_stop is placed directly after the list.

```
\quark_new:N \q_recursion_tail
\quark_new:N \q_recursion_stop
```

(*End definition for* \q_recursion_tail *and* \q_recursion_stop*. These variables are documented on page 45.*)

\quark_if_recursion_tail_stop:N
\quark_if_recursion_tail_stop_do:Nn

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
\cs_new:Npn \quark_if_recursion_tail_stop:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
```

(*End definition for* \quark_if_recursion_tail_stop:N*. This function is documented on page 45.*)

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on

The same idea applies when testing multiple tokens, but here we just compare the token list to \q_recursion_tail as a string.

```
\cs_new:Npn \quark_if_recursion_tail_stop:n #1
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
```

```
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(*End definition for* `\quark_if_recursion_tail_stop:n` *and* `\quark_if_recursion_tail_stop:o`*. These functions are documented on page* **??**.)

`\quark_if_recursion_tail_break:N`
`\quark_if_recursion_tail_break:n`

Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using `\prg_map_break:`.

```
\cs_new:Npn \quark_if_recursion_tail_break:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \prg_map_break:
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_break:n #1
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN \prg_map_break:
    \fi:
  }
```

(*End definition for* `\quark_if_recursion_tail_break:N`*. This function is documented on page* **??**.)

`\quark_if_nil_p:N`
`\quark_if_nil:NTF`
`\quark_if_no_value_p:N`
`\quark_if_no_value_p:c`
`\quark_if_no_value:NTF`
`\quark_if_no_value:cTF`

Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if `#1` is wrongly given a string like `aabc` instead of a single token.[5]

```
\prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
  {
    \if_meaning:w \q_nil #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
  {
    \if_meaning:w \q_no_value #1
      \prg_return_true:
    \else:
```

[5] It may still loop in special circumstances however!

```
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \quark_if_no_value_p:N { c }
\cs_generate_variant:Nn \quark_if_no_value:NT  { c }
\cs_generate_variant:Nn \quark_if_no_value:NF  { c }
\cs_generate_variant:Nn \quark_if_no_value:NTF { c }
```

(*End definition for* `\quark_if_nil:N`*. These functions are documented on page* **??**.)

`\quark_if_nil_p:n`
`\quark_if_nil_p:V`
`\quark_if_nil_p:o`
`\quark_if_nil:nTF`
`\quark_if_nil:VTF`
`\quark_if_nil:oTF`
`\quark_if_no_value_p:n`
`\quark_if_no_value:nTF`

These are essentially `\str_if_eq:nn` tests but done directly.

```
\prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
\cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
\cs_generate_variant:Nn \quark_if_nil:nT  { V , o }
\cs_generate_variant:Nn \quark_if_nil:nF  { V , o }
```

(*End definition for* `\quark_if_nil:n`*,* `\quark_if_nil:V`*, and* `\quark_if_nil:o`*. These functions are documented on page 45.*)

`\q_tl_act_mark`
`\q_tl_act_stop`

These private quarks are needed by l3tl, but that is loaded before the quark module, hence their definition is deferred.

```
\quark_new:N \q_tl_act_mark
\quark_new:N \q_tl_act_stop
```

(*End definition for* `\q_tl_act_mark` *and* `\q_tl_act_stop`*. These variables are documented on page 97.*)

188.2 Scan marks

`\g_scan_marks_tl`

The list of all scan marks currently declared.

```
\tl_new:N \g_scan_marks_tl
```

(*End definition for* `\g_scan_marks_tl`*. This variable is documented on page* **??**.)

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to \scan_stop: globally.

```
\cs_new_protected:Npn \scan_new:N #1
  {
    \tl_if_in:NnTF \g_scan_marks_tl { #1 }
      {
        \msg_kernel_error:nnx { scan } { already-defined }
          { \token_to_str:N #1 }
      }
      {
        \tl_gput_right:Nn \g_scan_marks_tl {#1}
        \cs_new_eq:NN #1 \scan_stop:
      }
  }
```

(*End definition for* \scan_new:N. *This function is documented on page 46.*)

\s_stop We only declare one scan mark here, more can be defined by specific modules.

```
\scan_new:N \s_stop
```

(*End definition for* \s_stop. *This variable is documented on page 46.*)

\use_none_delimit_by_s_stop:w Similar to \use_none_delimit_by_q_stop:w.

```
\cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }
```

(*End definition for* \use_none_delimit_by_s_stop:w. *This function is documented on page 46.*)

```
⟨/initex | package⟩
```

189 l3token implementation

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

189.1 Character tokens

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

Category code changes.

```
\cs_new_protected:Npn \char_set_catcode:nn #1#2
  { \tex_catcode:D #1 = \int_eval:w #2 \int_eval_end: }
\cs_new:Npn \char_value_catcode:n #1
  { \tex_the:D \tex_catcode:D \int_eval:w #1\int_eval_end: }
\cs_new_protected:Npn \char_show_value_catcode:n #1
  { \tex_showthe:D \tex_catcode:D \int_eval:w #1 \int_eval_end: }
```

(*End definition for* \char_set_catcode:nn. *This function is documented on page 49.*)

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

```
\cs_new_protected:Npn \char_set_catcode_escape:N #1
  { \char_set_catcode:nn { `#1 } \c_zero }
\cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  { \char_set_catcode:nn { `#1 } \c_one }
\cs_new_protected:Npn \char_set_catcode_group_end:N #1
  { \char_set_catcode:nn { `#1 } \c_two }
\cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  { \char_set_catcode:nn { `#1 } \c_three }
\cs_new_protected:Npn \char_set_catcode_alignment:N #1
  { \char_set_catcode:nn { `#1 } \c_four }
\cs_new_protected:Npn \char_set_catcode_end_line:N #1
  { \char_set_catcode:nn { `#1 } \c_five }
\cs_new_protected:Npn \char_set_catcode_parameter:N #1
  { \char_set_catcode:nn { `#1 } \c_six }
\cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  { \char_set_catcode:nn { `#1 } \c_seven }
\cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  { \char_set_catcode:nn { `#1 } \c_eight }
\cs_new_protected:Npn \char_set_catcode_ignore:N #1
  { \char_set_catcode:nn { `#1 } \c_nine }
\cs_new_protected:Npn \char_set_catcode_space:N #1
  { \char_set_catcode:nn { `#1 } \c_ten }
\cs_new_protected:Npn \char_set_catcode_letter:N #1
  { \char_set_catcode:nn { `#1 } \c_eleven }
\cs_new_protected:Npn \char_set_catcode_other:N #1
  { \char_set_catcode:nn { `#1 } \c_twelve }
\cs_new_protected:Npn \char_set_catcode_active:N #1
  { \char_set_catcode:nn { `#1 } \c_thirteen }
\cs_new_protected:Npn \char_set_catcode_comment:N #1
  { \char_set_catcode:nn { `#1 } \c_fourteen }
\cs_new_protected:Npn \char_set_catcode_invalid:N #1
  { \char_set_catcode:nn { `#1 } \c_fifteen }
```

(*End definition for* \char_set_catcode_escape:N *and others. These functions are documented on page* *48*.)

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

```
\cs_new_protected:Npn \char_set_catcode_escape:n #1
  { \char_set_catcode:nn {#1} \c_zero }
\cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  { \char_set_catcode:nn {#1} \c_one }
\cs_new_protected:Npn \char_set_catcode_group_end:n #1
  { \char_set_catcode:nn {#1} \c_two }
\cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  { \char_set_catcode:nn {#1} \c_three }
\cs_new_protected:Npn \char_set_catcode_alignment:n #1
  { \char_set_catcode:nn {#1} \c_four }
\cs_new_protected:Npn \char_set_catcode_end_line:n #1
  { \char_set_catcode:nn {#1} \c_five }
```

```
\cs_new_protected:Npn \char_set_catcode_parameter:n #1
  { \char_set_catcode:nn {#1} \c_six }
\cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  { \char_set_catcode:nn {#1} \c_seven }
\cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
  { \char_set_catcode:nn {#1} \c_eight }
\cs_new_protected:Npn \char_set_catcode_ignore:n #1
  { \char_set_catcode:nn {#1} \c_nine }
\cs_new_protected:Npn \char_set_catcode_space:n #1
  { \char_set_catcode:nn {#1} \c_ten }
\cs_new_protected:Npn \char_set_catcode_letter:n #1
  { \char_set_catcode:nn {#1} \c_eleven }
\cs_new_protected:Npn \char_set_catcode_other:n #1
  { \char_set_catcode:nn {#1} \c_twelve }
\cs_new_protected:Npn \char_set_catcode_active:n #1
  { \char_set_catcode:nn {#1} \c_thirteen }
\cs_new_protected:Npn \char_set_catcode_comment:n #1
  { \char_set_catcode:nn {#1} \c_fourteen }
\cs_new_protected:Npn \char_set_catcode_invalid:n #1
  { \char_set_catcode:nn {#1} \c_fifteen }
```

(*End definition for* `\char_set_catcode_escape:n` *and others. These functions are documented on page 48.*)

`\char_set_mathcode:nn`
`\char_value_mathcode:n`
`\char_show_value_mathcode:n`
`\char_set_lccode:nn`
`\char_value_lccode:n`
`\char_show_value_lccode:n`
`\char_set_uccode:nn`
`\char_value_uccode:n`
`\char_show_value_uccode:n`
`\char_set_sfcode:nn`
`\char_value_sfcode:n`
`\char_show_value_sfcode:n`

Pretty repetitive, but necessary!

```
\cs_new_protected:Npn \char_set_mathcode:nn #1#2
  { \tex_mathcode:D #1 = \int_eval:w #2 \int_eval_end: }
\cs_new:Npn \char_value_mathcode:n #1
  { \tex_the:D \tex_mathcode:D \int_eval:w #1\int_eval_end: }
\cs_new_protected:Npn \char_show_value_mathcode:n #1
  { \tex_showthe:D \tex_mathcode:D \int_eval:w #1 \int_eval_end: }
\cs_new_protected:Npn \char_set_lccode:nn #1#2
  { \tex_lccode:D #1 = \int_eval:w #2 \int_eval_end: }
\cs_new:Npn \char_value_lccode:n #1
  { \tex_the:D \tex_lccode:D \int_eval:w #1\int_eval_end: }
\cs_new_protected:Npn \char_show_value_lccode:n #1
  { \tex_showthe:D \tex_lccode:D \int_eval:w #1 \int_eval_end: }
\cs_new_protected:Npn \char_set_uccode:nn #1#2
  { \tex_uccode:D #1 = \int_eval:w #2 \int_eval_end: }
\cs_new:Npn \char_value_uccode:n #1
  { \tex_the:D \tex_uccode:D \int_eval:w #1\int_eval_end: }
\cs_new_protected:Npn \char_show_value_uccode:n #1
  { \tex_showthe:D \tex_uccode:D \int_eval:w #1 \int_eval_end: }
\cs_new_protected:Npn \char_set_sfcode:nn #1#2
  { \tex_sfcode:D #1 = \int_eval:w #2 \int_eval_end: }
\cs_new:Npn \char_value_sfcode:n #1
  { \tex_the:D \tex_sfcode:D \int_eval:w #1\int_eval_end: }
\cs_new_protected:Npn \char_show_value_sfcode:n #1
  { \tex_showthe:D \tex_sfcode:D \int_eval:w #1 \int_eval_end: }
```

(*End definition for* `\char_set_mathcode:nn`*. This function is documented on page 51.*)

189.2 Generic tokens

\token_new:Nn

Creates a new token.

```
\cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }
```

(*End definition for* `\token_new:Nn`*. This function is documented on page 51.*)

\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_catcode_letter_token
\c_catcode_other_token

We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```
\cs_new_eq:NN \c_group_begin_token {
\cs_new_eq:NN \c_group_end_token }
\group_begin:
  \char_set_catcode_math_toggle:N \*
  \token_new:Nn \c_math_toggle_token { * }
  \char_set_catcode_alignment:N \*
  \token_new:Nn \c_alignment_token { * }
  \token_new:Nn \c_parameter_token { # }
  \token_new:Nn \c_math_superscript_token { ^ }
  \char_set_catcode_math_subscript:N \*
  \token_new:Nn \c_math_subscript_token { * }
  \token_new:Nn \c_space_token { ~ }
  \token_new:Nn \c_catcode_letter_token { a }
  \token_new:Nn \c_catcode_other_token { 1 }
\group_end:
```

(*End definition for* `\c_group_begin_token` *and others. These functions are documented on page 51.*)

\c_catcode_active_tl

Not an implicit token!

```
\group_begin:
  \char_set_catcode_active:N \*
  \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
\group_end:
```

(*End definition for* `\c_catcode_active_tl`*. This variable is documented on page 51.*)

\l_char_active_seq
\l_char_special_seq

Two sequences for dealing with special characters. The first is characters which may be active, and contains the active characters themselves to allow easy redefinition. The second longer list is for "special" characters more generally, and these are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`). The only complication is dealing with `_`, which requires the use of `\use:n` *and* `\use:nn`.

```
\seq_new:N \l_char_active_seq
\use:n
  {
    \group_begin:
    \char_set_catcode_active:N \"
    \char_set_catcode_active:N \$
    \char_set_catcode_active:N \&
    \char_set_catcode_active:N \^
    \char_set_catcode_active:N \_
    \char_set_catcode_active:N \~
```

```
    \use:nn
      {
        \group_end:
        \seq_set_from_clist:Nn \l_char_active_seq
      }
  }
    { { " , $ , & , ^ , _ , ~ } } %$
\seq_new:N \l_char_special_seq
\seq_set_from_clist:Nn \l_char_special_seq
  { \  , \" , \# , \$ , \% , \& , \\ , \^ , \_ , \{ , \} , \~ }
```

(*End definition for* `\l_char_active_seq` *and* `\l_char_special_seq`*. These variables are documented on page 51.*)

189.3 Token conditionals

`\token_if_group_begin_p:N`
`\token_if_group_begin:NTF`

Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
\prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_begin_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_group_begin:N`*. These functions are documented on page 52.*)

`\token_if_group_end_p:N`
`\token_if_group_end:NTF`

Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
\prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_end_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_group_end:N`*. These functions are documented on page 52.*)

`\token_if_math_toggle_p:N`
`\token_if_math_toggle:NTF`

Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_toggle_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_math_toggle:N`*. These functions are documented on page 52.*)

`\token_if_alignment_p:N`
`\token_if_alignment:NTF`

Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token` for this.

```
\prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_alignment_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_alignment:N`. *These functions are documented on page 52.*)

`\token_if_parameter_p:N`
`\token_if_parameter:NTF`

Check if token is a parameter token. We use the constant `\c_parameter_token` for this. We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they will remain after the group.

```
\group_begin:
\cs_set_eq:NN \c_parameter_token \scan_stop:
\prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_parameter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\group_end:
```

(*End definition for* `\token_if_parameter:N`. *These functions are documented on page 53.*)

`\token_if_math_superscript_p:N`
`\token_if_math_superscript:NTF`

Check if token is a math superscript token. We use the constant `\c_superscript_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_superscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_math_superscript:N`. *These functions are documented on page 53.*)

`\token_if_math_subscript_p:N`
`\token_if_math_subscript:NTF`

Check if token is a math subscript token. We use the constant `\c_subscript_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_subscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_math_subscript:N`. *These functions are documented on page 53.*)

`\token_if_space_p:N`
`\token_if_space:NTF`

Check if token is a space token. We use the constant `\c_space_token` for this.

```
\prg_new_conditional:Npnn \token_if_space:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_space_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_space:N`. *These functions are documented on page 53.*)

`\token_if_letter_p:N`
`\token_if_letter:NTF`

Check if token is a letter token. We use the constant `\c_letter_token` for this.

```
\prg_new_conditional:Npnn \token_if_letter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_letter:N`. *These functions are documented on page 53.*)

`\token_if_other_p:N`
`\token_if_other:NTF`

Check if token is an other char token. We use the constant `\c_other_char_token` for this.

```
\prg_new_conditional:Npnn \token_if_other:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_other_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_other:N`. *These functions are documented on page 53.*)

`\token_if_active_p:N`
`\token_if_active:NTF`

Check if token is an active char token. We use the constant `\c_active_char_tl` for this. A technical point is that `\c_active_char_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.

```
\prg_new_conditional:Npnn \token_if_active:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_active:N`. *These functions are documented on page 53.*)

`\token_if_eq_meaning_p:NN`
`\token_if_eq_meaning:NNTF`

Check if the tokens `#1` and `#2` have same meaning.

```
\prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T ,  F , TF }
  {
    \if_meaning:w  #1  #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_meaning:NN`. *These functions are documented on page 54.*)

`\token_if_eq_catcode_p:NN`
`\token_if_eq_catcode:NNTF`

Check if the tokens `#1` and `#2` have same category code.

```
\prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_catcode:NN`. *These functions are documented on page 53.*)

`\token_if_eq_charcode_p:NN`
`\token_if_eq_charcode:NNTF`

Check if the tokens `#1` and `#2` have same character code.

```
\prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_charcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_eq_charcode:NN`. *These functions are documented on page 53.*)

`\token_if_macro_p:N` `\token_if_macro:NTF` `\token_if_macro_p_aux:w`

When a token is a macro, `\token_to_meaning:N` will always output something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. However, this can fail the five `\...mark` primitives, whose meaning has the form `...mark:`⟨*user material*⟩. The problem is that the ⟨*user material*⟩ can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in LaTeX3) before the string `macro:`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m`...

Both `ma` and `:` must be of category code 12 (other), and we achieve using the standard lowercasing technique.

```
\group_begin:
\char_set_catcode_other:N \M
\char_set_catcode_other:N \A
\char_set_lccode:nn { `\; } { `\: }
\char_set_lccode:nn { `\T } { `\T }
\char_set_lccode:nn { `\F } { `\F }
\tl_to_lowercase:n
  {
    \group_end:
    \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T ,  F , TF }
      {
        \exp_after:wN \token_if_macro_p_aux:w
        \token_to_meaning:N #1 MA; \q_stop
      }
    \cs_new:Npn \token_if_macro_p_aux:w #1 MA #2 ; #3 \q_stop
      {
        \if_int_compare:w \pdftex_strcmp:D { #2 } { cro } = \c_zero
            \prg_return_true:
        \else:
            \prg_return_false:
        \fi:
      }
  }
```

(*End definition for* `\token_if_macro:N`*. These functions are documented on page* *54*.)

`\token_if_cs_p:N` `\token_if_cs:NTF`

Check if token has same catcode as a control sequence. This follows the same pattern as for `\token_if_letter:N` *etc.* We use `\scan_stop:` for this.

```
\prg_new_conditional:Npnn \token_if_cs:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \scan_stop:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_cs:N`. *These functions are documented on page 54.*)

`\token_if_expandable_p:N`
`\token_if_expandable:NTF`

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.

```
\prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T ,  F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \exp_after:wN \if_meaning:w \exp_not:N #1 #1
          \prg_return_false: \else: \prg_return_true: \fi:
      }
      { \prg_return_false: }
  }
```

(*End definition for* `\token_if_expandable:N`. *These functions are documented on page 54.*)

`\token_if_chardef_p:N`
`\token_if_mathchardef_p:N`
`\token_if_dim_register_p:N`
`\token_if_int_register_p:N`
`\token_if_muskip_register_p:N`
`\token_if_skip_register_p:N`
`\token_if_toks_register_p:N`
`\token_if_long_macro_p:N`
`\token_if_protected_macro_p:N`
`\token_if_protected_long_macro_p:N`
`\token_if_chardef:NTF`
`\token_if_mathchardef:NTF`
`\token_if_dim_register:NTF`
`\token_if_int_register:NTF`
`\token_if_muskip_register:NTF`
`\token_if_skip_register:NTF`
`\token_if_toks_register:NTF`
`\token_if_long_macro:NTF`
`\token_if_protected_macro:NTF`
`\token_if_protected_long_macro:NTF`
`\token_if_chardef_aux:w`
`\token_if_dim_register_aux:w`
`\token_if_int_register_aux:w`
`\token_if_muskip_register_aux:w`
`\token_if_skip_register_aux:w`
`\token_if_toks_register_aux:w`
`\token_if_protected_macro_aux:w`
`\token_if_long_macro_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 { `T } { `T }
  \char_set_lccode:nn { `F } { `F }
  \char_set_lccode:nn { `X } { `n }
  \char_set_lccode:nn { `Y } { `t }
  \char_set_lccode:nn { `Z } { `d }
  \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
    { \char_set_catcode:nn { `#1 } \c_twelve }
```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```
\tl_to_lowercase:n
  {
    \group_end:
```

First up is checking if something has been defined with `\chardef` or `\mathchardef`. This is easy since TEX thinks of such tokens as hexadecimal so it stores them as `\char"`⟨*hex number*⟩ or `\mathchar"`⟨*hex number*⟩. Grab until the first occurrence of `char"`, and compare what preceeds with `\` or `\math`. In fact, the escape character may not be a backslash, so we compare with the result of converting some other control sequence to a string, namely `\char` or `\mathchar` (the auxiliary adds the `char` back).

```
    \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T ,  F , TF }
      {
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_chardef_aux:w
              \token_to_meaning:N #1 CHAR" \q_stop
          }
          { \token_to_str:N \char }
      }
```

```
\prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T ,  F , TF }
  {
    \str_if_eq_return:xx
      {
        \exp_after:wN \token_if_chardef_aux:w
          \token_to_meaning:N #1 CHAR" \q_stop
      }
      { \token_to_str:N \mathchar }
  }
\cs_new:Npn \token_if_chardef_aux:w #1 CHAR" #2 \q_stop { #1 CHAR }
```

Dim registers are a little more difficult since their `\meaning` has the form `\dimen`⟨*number*⟩, and we must take care of the two primitives `\dimen` and `\dimendef`.

```
\prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_dimen:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_dimendef:D #1
        \prg_return_false:
      \else:
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_dim_register_aux:w
              \token_to_meaning:N #1 ZIMEX \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \token_if_dim_register_aux:w #1 ZIMEX #2 \q_stop { #1 ~ }
```

Integer registers are one step harder since constants are implemented differently from variables, and we also have to take care of the primitives `\count` and `\countdef`.

```
\prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T ,  F , TF }
  {
    % \token_if_chardef:NTF #1 { \prg_return_true: }
    %   {
    %     \token_if_mathchardef:NTF #1 { \prg_return_true: }
    %       {
    \if_meaning:w \tex_count:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_countdef:D #1
        \prg_return_false:
      \else:
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_int_register_aux:w
              \token_to_meaning:N #1 COUXY \q_stop
```

```
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
    %       }
    %   }
  }
\cs_new:Npn \token_if_int_register_aux:w #1 COUXY #2 \q_stop { #1 ~ }
```

Muskip registers are done the same way as the dimension registers.

```
\prg_new_conditional:Npnn \token_if_muskip_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_muskip:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_muskipdef:D #1
        \prg_return_false:
      \else:
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_muskip_register_aux:w
              \token_to_meaning:N #1 MUSKIP \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \token_if_muskip_register_aux:w #1 MUSKIP #2 \q_stop { #1 ~ }
```

Skip registers.

```
\prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_skip:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_skipdef:D #1
        \prg_return_false:
      \else:
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_skip_register_aux:w
              \token_to_meaning:N #1 SKIP \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \token_if_skip_register_aux:w #1 SKIP #2 \q_stop { #1 ~ }
```

Toks registers.

```
\prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T ,  F , TF }
```

```
  {
    \if_meaning:w \tex_toks:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_toksdef:D #1
        \prg_return_false:
      \else:
        \str_if_eq_return:xx
          {
            \exp_after:wN \token_if_toks_register_aux:w
              \token_to_meaning:N #1 YOKS \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \token_if_toks_register_aux:w #1 YOKS #2 \q_stop { #1 ~ }
```

Protected macros.

```
\prg_new_conditional:Npnn \token_if_protected_macro:N #1
  { p , T ,  F , TF }
  {
    \str_if_eq_return:xx
      {
        \exp_after:wN \token_if_protected_macro_aux:w
          \token_to_meaning:N #1 PROYECYEZ~MACRO \q_stop
      }
      { \token_to_str:N \  }
  }
\cs_new:Npn \token_if_protected_macro_aux:w
  #1 PROYECYEZ~MACRO #2 \q_stop { #1 ~ }
```

Long macros and protected long macros share an auxiliary.

```
\prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T ,  F , TF }
  {
    \str_if_eq_return:xx
      {
        \exp_after:wN \token_if_long_macro_aux:w
          \token_to_meaning:N #1 LOXG~MACRO \q_stop
      }
      { \token_to_str:N \  }
  }
\prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
  { p , T ,  F , TF }
  {
    \str_if_eq_return:xx
      {
        \exp_after:wN \token_if_long_macro_aux:w
          \token_to_meaning:N #1 LOXG~MACRO \q_stop
      }
      { \token_to_str:N \protected \token_to_str:N \  }
```

```
      }
    \cs_new:Npn \token_if_long_macro_aux:w #1 LOXG~MACRO #2 \q_stop { #1 ~ }
```

Finally the `\tl_to_lowercase:n` ends!

```
  }
```

(*End definition for* `\token_if_chardef:N` *and others. These functions are documented on page* *54*.)

`\token_if_primitive_p:N`
`\token_if_primitive:NTF`
`\token_if_primitive_aux:NNw`
`\token_if_primitive_aux_space:w`
`\token_if_primitive_aux_nullfont:N`
`\token_if_primitive_aux_loop:N`
`\token_if_primitive_auxii:Nw`
`\token_if_primitive_aux_undefined:N`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., `the letter A`), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\c_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be inexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form ⟨*letters*⟩`:`⟨*user material*⟩. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\c_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than `` `A `` (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\c_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

```
\tex_chardef:D \c_token_A_int = `A ~ %
\group_begin:
\char_set_catcode_other:N \;
\char_set_lccode:nn { `\; } { `\: }
\char_set_lccode:nn { `\T } { `\T }
\char_set_lccode:nn { `\F } { `\F }
\tl_to_lowercase:n {
  \group_end:
  \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
    {
      \token_if_macro:NTF #1
        \prg_return_false:
        {
          \exp_after:wN \token_if_primitive_aux:NNw
          \token_to_meaning:N #1 ; ; ; \q_stop #1
        }
    }
  \cs_new:Npn \token_if_primitive_aux:NNw #1#2 #3 ; #4 \q_stop
```

```
    {
      \tl_if_empty:oTF { \token_if_primitive_aux_space:w #3 ~ }
        { \token_if_primitive_aux_loop:N #3 ; \q_stop }
        { \token_if_primitive_aux_nullfont:N }
    }
}
\cs_new:Npn \token_if_primitive_aux_space:w #1 ~ { }
\cs_new:Npn \token_if_primitive_aux_nullfont:N #1
  {
    \if_meaning:w \tex_nullfont:D #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \token_if_primitive_aux_loop:N #1
  {
    \if_num:w `#1 < \c_token_A_int %
      \exp_after:wN \token_if_primitive_auxii:Nw
      \exp_after:wN #1
    \else:
      \exp_after:wN \token_if_primitive_aux_loop:N
    \fi:
  }
\cs_new:Npn \token_if_primitive_auxii:Nw #1 #2 \q_stop
  {
    \if:w : #1
      \exp_after:wN \token_if_primitive_aux_undefined:N
    \else:
      \prg_return_false:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new:Npn \token_if_primitive_aux_undefined:N #1
  {
    \if_cs_exist:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\token_if_primitive:N`. *These functions are documented on page* *55*.)

189.4 Peeking ahead at the next token

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;

2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token
\g_peek_token

Storage tokens which are publicly documented: the token peeked.

```
\cs_new_eq:NN \l_peek_token ?
\cs_new_eq:NN \g_peek_token ?
```

(*End definition for* `\l_peek_token`. *This function is documented on page 56.*)

\l_peek_search_token

The token to search for as an implicit token: *cf.* `\l_peek_search_tl`.

```
\cs_new_eq:NN \l_peek_search_token ?
```

(*End definition for* `\l_peek_search_token`. *This variable is documented on page* **??**.)

\l_peek_search_tl

The token to search for as an explicit token: *cf.* `\l_peek_search_token`.

```
\tl_new:N \l_peek_search_tl
```

(*End definition for* `\l_peek_search_tl`. *This variable is documented on page* **??**.)

\peek_true:w
\peek_true_aux:w
\peek_false:w
\peek_tmp:w

Functions used by the branching and space-stripping code.

```
\cs_new_nopar:Npn \peek_true:w  { }
\cs_new_nopar:Npn \peek_true_aux:w  { }
\cs_new_nopar:Npn \peek_false:w { }
\cs_new:Npn \peek_tmp:w { }
```

(*End definition for* `\peek_true:w` *and others.*)

\peek_after:Nw
\peek_after:Nw

Simple wrappers for `\futurelet`: no arguments absorbed here.

```
\cs_new_protected_nopar:Npn \peek_after:Nw
  { \tex_futurelet:D \l_peek_token }
\cs_new_protected_nopar:Npn \peek_gafter:Nw
  { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(*End definition for* `\peek_after:Nw`. *This function is documented on page 56.*)

\peek_true_remove:w

A function to remove the next token and then regain control.

```
\cs_new_protected:Npn \peek_true_remove:w
  {
    \group_align_safe_end:
    \tex_afterassignment:D \peek_true_aux:w
    \cs_set_eq:NN \peek_tmp:w
  }
```

(*End definition for* `\peek_true_remove:w`.)

\peek_token_generic:NNTF The generic function stores the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself.

```
\cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4
  {
    \cs_set_eq:NN \l_peek_search_token #2
    \tl_set:Nn \l_peek_search_tl {#2}
    \cs_set_nopar:Npx \peek_true:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#3}
      }
    \cs_set_nopar:Npx \peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#4}
      }
    \group_align_safe_begin:
      \peek_after:Nw #1
  }
\cs_new_protected:Npn \peek_token_generic:NNT #1#2#3
  { \peek_token_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \peek_token_generic:NNF #1#2#3
  { \peek_token_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* `\peek_token_generic:NNTF`*. This function is documented on page* **??**.)

\peek_token_remove_generic:NNTF For token removal there needs to be a call to the auxiliary function which does the work.

```
\cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4
  {
    \cs_set_eq:NN \l_peek_search_token #2
    \tl_set:Nn \l_peek_search_tl {#2}
    \cs_set_eq:NN \peek_true:w \peek_true_remove:w
    \cs_set_nopar:Npx \peek_true_aux:w { \exp_not:n {#3} }
    \cs_set_nopar:Npx \peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#4}
      }
    \group_align_safe_begin:
      \peek_after:Nw #1
  }
\cs_new_protected:Npn \peek_token_remove_generic:NNT #1#2#3
  { \peek_token_remove_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \peek_token_remove_generic:NNF #1#2#3
  { \peek_token_remove_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* `\peek_token_remove_generic:NNTF`*. This function is documented on page* **??**.)

\peek_execute_branches_catcode:
\peek_execute_branches_meaning:

The category code and meaning tests are straight forward.

```
\cs_new_nopar:Npn \peek_execute_branches_catcode:
```

```
  {
    \if_catcode:w
      \exp_not:N \l_peek_token \exp_not:N \l_peek_search_token
      \exp_after:wN \peek_true:w
    \else:
      \exp_after:wN \peek_false:w
    \fi:
  }
\cs_new_nopar:Npn \peek_execute_branches_meaning:
  {
    \if_meaning:w \l_peek_token \l_peek_search_token
      \exp_after:wN \peek_true:w
    \else:
      \exp_after:wN \peek_false:w
    \fi:
  }
```

(*End definition for* `\peek_execute_branches_catcode:` *and* `\peek_execute_branches_meaning:`*. These functions are documented on page* **??**.)

\peek_execute_branches_charcode:
\peek_execute_branches_charcode:NN

First the character code test there is a need to worry about TEX grabbing brace group or skipping spaces. These are all tested for using a category code check before grabbing what must be a real single token and doing the comparison.

```
\cs_new_nopar:Npn \peek_execute_branches_charcode:
  {
    \bool_if:nTF
      {
           \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token
        || \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token
        || \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
      }
      { \peek_false:w }
      {
        \exp_after:wN \peek_execute_branches_charcode_aux:NN
          \l_peek_search_tl
      }
  }
\cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2
  {
    \if:w \exp_not:N #1 \exp_not:N #2
      \exp_after:wN \peek_true:w
    \else:
      \exp_after:wN \peek_false:w
    \fi:
    #2
  }
```

(*End definition for* `\peek_execute_branches_charcode:`*. This function is documented on page* **??**.)

\peek_ignore_spaces_execute_branches:
\peek_ignore_spaces_execute_branches_aux:

This function removes one token at a time with a mechanism that can be applied to things other than spaces.

```
\cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches:
  {
    \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
      {
        \tex_afterassignment:D \peek_ignore_spaces_execute_branches_aux:
        \cs_set_eq:NN \peek_tmp:w
      }
      { \peek_execute_branches: }
  }
\cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches_aux:
  { \peek_after:Nw \peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_ignore_spaces_execute_branches:`. *This function is documented on page* **??**.)

`\peek_def:nnnn`
`\peek_def_aux:nnnnn`

The public functions themselves cannot be defined using `\prg_new_conditional:Npnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```
\group_begin:
  \cs_set:Npn \peek_def:nnnn #1#2#3#4
    {
      \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { TF }
      \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { T }
      \peek_def_aux:nnnnn {#1} {#2} {#3} {#4} { F }
    }
  \cs_set:Npn \peek_def_aux:nnnnn #1#2#3#4#5
    {
      \cs_new_nopar:cpx { #1 #5 }
        {
          \tl_if_empty:nF {#2}
            { \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 } }
          \exp_not:c { #3 #5 }
          \exp_not:n {#4}
        }
    }
```

(*End definition for* `\peek_def:nnnn`. *This function is documented on page* **??**.)

`\peek_catcode:NTF`
`\peek_catcode_ignore_spaces:NTF`
`\peek_catcode_remove:NTF`
`\peek_catcode_remove_ignore_spaces:NTF`

With everything in place the definitions can take place. First for category codes.

```
  \peek_def:nnnn { peek_catcode:N }
    { }
    { peek_token_generic:NN }
    { \peek_execute_branches_catcode: }
  \peek_def:nnnn { peek_catcode_ignore_spaces:N }
    { \peek_execute_branches_catcode: }
    { peek_token_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
  \peek_def:nnnn { peek_catcode_remove:N }
    { }
    { peek_token_remove_generic:NN }
    { \peek_execute_branches_catcode: }
  \peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
```

```
    { \peek_execute_branches_catcode: }
    { peek_token_remove_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_catcode:NTF` *and others. These functions are documented on page 57.*)

`\peek_charcode:NTF`
`\peek_charcode_ignore_spaces:NTF`
`\peek_charcode_remove:NTF`
`\peek_charcode_remove_ignore_spaces:NTF`

Then for character codes.

```
  \peek_def:nnnn { peek_charcode:N }
    { }
    { peek_token_generic:NN }
    { \peek_execute_branches_charcode: }
  \peek_def:nnnn { peek_charcode_ignore_spaces:N }
    { \peek_execute_branches_charcode: }
    { peek_token_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
  \peek_def:nnnn { peek_charcode_remove:N }
    { }
    { peek_token_remove_generic:NN }
    { \peek_execute_branches_charcode: }
  \peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
    { \peek_execute_branches_charcode: }
    { peek_token_remove_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_charcode:NTF` *and others. These functions are documented on page 57.*)

`\peek_meaning:NTF`
`\peek_meaning_ignore_spaces:NTF`
`\peek_meaning_remove:NTF`
`\peek_meaning_remove_ignore_spaces:NTF`

Finally for meaning, with the group closed to remove the temporary definition functions.

```
  \peek_def:nnnn { peek_meaning:N }
    { }
    { peek_token_generic:NN }
    { \peek_execute_branches_meaning: }
  \peek_def:nnnn { peek_meaning_ignore_spaces:N }
    { \peek_execute_branches_meaning: }
    { peek_token_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
  \peek_def:nnnn { peek_meaning_remove:N }
    { }
    { peek_token_remove_generic:NN }
    { \peek_execute_branches_meaning: }
  \peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
    { \peek_execute_branches_meaning: }
    { peek_token_remove_generic:NN }
    { \peek_ignore_spaces_execute_branches: }
\group_end:
```

(*End definition for* `\peek_meaning:NTF` *and others. These functions are documented on page 58.*)

189.5 Decomposing a macro definition

`\token_get_prefix_spec:N`
`\token_get_arg_spec:N`
`\token_get_replacement_spec:N`
`\token_get_prefix_arg_replacement_aux:wN`

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```
\exp_args:Nno \use:nn
  { \cs_new:Npn \token_get_prefix_arg_replacement_aux:wN #1 }
  { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
  { #4 {#1} {#2} {#3} }
\cs_new:Npn \token_get_prefix_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
          \token_to_meaning:N #1 \q_stop \use_i:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_arg_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
          \token_to_meaning:N #1 \q_stop \use_ii:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_replacement_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \token_get_prefix_arg_replacement_aux:wN
          \token_to_meaning:N #1 \q_stop \use_iii:nnn
      }
      { \scan_stop: }
  }
```

(*End definition for* `\token_get_prefix_spec:N`. *This function is documented on page 59.*)

189.6 Experimental token functions

`\char_set_active:Npn`
`\char_set_active:Npx`
`\char_set_active:Npn`
`\char_set_active:Npx`
`\char_set_active_eq:NN`
`\char_gset_active_eq:NN`

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_set:Npn \char_tmp:NN #1#2
    {
      \cs_new:Npn #1 ##1
        {
          \char_set_catcode_active:n { `##1 }
          \group_begin:
          \char_set_lccode:nn { `\^^@ } { `##1 }
          \tl_to_lowercase:n { \group_end: #2 ^^@ }
```

```
        }
    }
  \char_tmp:NN \char_set_active:Npn    \cs_set:Npn
  \char_tmp:NN \char_set_active:Npx    \cs_set:Npx
  \char_tmp:NN \char_gset_active:Npn   \cs_gset:Npn
  \char_tmp:NN \char_gset_active:Npx   \cs_gset:Npx
  \char_tmp:NN \char_set_active_eq:NN  \cs_set_eq:NN
  \char_tmp:NN \char_gset_active_eq:NN \cs_gset_eq:NN
\group_end:
```

(*End definition for* `\char_set_active:Npn` *and* `\char_set_active:Npx`*. These functions are documented on page* 60.)

`\peek_N_type:TF`
`\peek_execute_branches_N_type:`

The next token is normal if it is neither a begin-group token, nor an end-group token, nor a charcode-32 space token. Note that implicit begin-group tokens, end-group tokens, and spaces are also recognized as non-N-type. Here, there is no ⟨*search token*⟩, so we feed a dummy `\scan_stop:` to the `\peek_token_generic::NN` functions.

```
\cs_new_protected_nopar:Npn \peek_execute_branches_N_type:
  {
    \bool_if:nTF
      {
        \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
        \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token   ||
        \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
      }
      { \peek_false:w }
      { \peek_true:w }
  }
\cs_new_protected_nopar:Npn \peek_N_type:TF
  { \peek_token_generic:NNTF \peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected_nopar:Npn \peek_N_type:T
  { \peek_token_generic:NNT  \peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected_nopar:Npn \peek_N_type:F
  { \peek_token_generic:NNF  \peek_execute_branches_N_type: \scan_stop: }
```

(*End definition for* `\peek_N_type:`*. This function is documented on page* **??**.)

189.7 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\char_set_catcode:w`
`\char_set_mathcode:w`
`\char_set_lccode:w`
`\char_set_uccode:w`
`\char_set_sfcode:w`

Primitives renamed.

```
⟨*deprecated⟩
\cs_new_eq:NN \char_set_catcode:w  \tex_catcode:D
\cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
\cs_new_eq:NN \char_set_lccode:w   \tex_lccode:D
\cs_new_eq:NN \char_set_uccode:w   \tex_uccode:D
\cs_new_eq:NN \char_set_sfcode:w   \tex_sfcode:D
⟨/deprecated⟩
```

(*End definition for* `\char_set_catcode:w`*. This function is documented on page* **??**.)

\char_value_catcode:w
\char_show_value_catcode:w
\char_value_mathcode:w
\char_show_value_mathcode:w
\char_value_lccode:w
\char_show_value_lccode:w
\char_value_uccode:w
\char_show_value_uccode:w
\char_value_sfcode:w
\char_show_value_sfcode:w

More `w` functions we should not have.

```
⟨*deprecated⟩
\cs_new_nopar:Npn \char_value_catcode:w { \tex_the:D \char_set_catcode:w }
\cs_new_nopar:Npn \char_show_value_catcode:w
  { \tex_showthe:D \char_set_catcode:w }
\cs_new_nopar:Npn \char_value_mathcode:w { \tex_the:D \char_set_mathcode:w }
\cs_new_nopar:Npn \char_show_value_mathcode:w
  { \tex_showthe:D \char_set_mathcode:w }
\cs_new_nopar:Npn \char_value_lccode:w { \tex_the:D \char_set_lccode:w }
\cs_new_nopar:Npn \char_show_value_lccode:w
  { \tex_showthe:D \char_set_lccode:w }
\cs_new_nopar:Npn \char_value_uccode:w { \tex_the:D \char_set_uccode:w }
\cs_new_nopar:Npn \char_show_value_uccode:w
  { \tex_showthe:D \char_set_uccode:w }
\cs_new_nopar:Npn \char_value_sfcode:w { \tex_the:D \char_set_sfcode:w }
\cs_new_nopar:Npn \char_show_value_sfcode:w
  { \tex_showthe:D \char_set_sfcode:w }
⟨/deprecated⟩
```

(*End definition for* `\char_value_catcode:w`*. This function is documented on page* **??**.)

\peek_after:NN
\peek_gafter:NN

The second argument here must be `w`.

```
⟨*deprecated⟩
\cs_new_eq:NN \peek_after:NN  \peek_after:Nw
\cs_new_eq:NN \peek_gafter:NN \peek_gafter:Nw
⟨/deprecated⟩
```

(*End definition for* `\peek_after:NN`*. This function is documented on page* **??**.)

Functions deprecated 2011-05-28 for removal by 2011-08-31.

\c_alignment_tab_token
\c_math_shift_token
\c_letter_token
\c_other_char_token

```
⟨*deprecated⟩
\cs_new_eq:NN \c_alignment_tab_token \c_alignment_token
\cs_new_eq:NN \c_math_shift_token    \c_math_toggle_token
\cs_new_eq:NN \c_letter_token        \c_catcode_letter_token
\cs_new_eq:NN \c_other_char_token    \c_catcode_other_token
⟨/deprecated⟩
```

(*End definition for* `\c_alignment_tab_token`*. This function is documented on page* **??**.)

\c_active_char_token

An odd one: this was never a `token`!

```
⟨*deprecated⟩
\cs_new_eq:NN \c_active_char_token \c_catcode_active_tl
⟨/deprecated⟩
```

(*End definition for* `\c_active_char_token`*. This function is documented on page* **??**.)

\char_make_escape:N
\char_make_group_begin:N
\char_make_group_end:N
\char_make_math_toggle:N
\char_make_alignment:N
\char_make_end_line:N
\char_make_parameter:N
\char_make_math_superscript:N
\char_make_math_subscript:N
\char_make_ignore:N
\char_make_space:N
\char_make_letter:N
\char_make_other:N
\char_make_active:N
\char_make_comment:N

Two renames in one block!

```
⟨*deprecated⟩
\cs_new_eq:NN \char_make_escape:N         \char_set_catcode_escape:N
\cs_new_eq:NN \char_make_begin_group:N    \char_set_catcode_group_begin:N
\cs_new_eq:NN \char_make_end_group:N      \char_set_catcode_group_end:N
```

```
\cs_new_eq:NN \char_make_math_shift:N       \char_set_catcode_math_toggle:N
\cs_new_eq:NN \char_make_alignment_tab:N    \char_set_catcode_alignment:N
\cs_new_eq:NN \char_make_end_line:N         \char_set_catcode_end_line:N
\cs_new_eq:NN \char_make_parameter:N        \char_set_catcode_parameter:N
\cs_new_eq:NN \char_make_math_superscript:N
  \char_set_catcode_math_superscript:N
\cs_new_eq:NN \char_make_math_subscript:N
  \char_set_catcode_math_subscript:N
\cs_new_eq:NN \char_make_ignore:N           \char_set_catcode_ignore:N
\cs_new_eq:NN \char_make_space:N            \char_set_catcode_space:N
\cs_new_eq:NN \char_make_letter:N           \char_set_catcode_letter:N
\cs_new_eq:NN \char_make_other:N            \char_set_catcode_other:N
\cs_new_eq:NN \char_make_active:N           \char_set_catcode_active:N
\cs_new_eq:NN \char_make_comment:N          \char_set_catcode_comment:N
\cs_new_eq:NN \char_make_invalid:N          \char_set_catcode_invalid:N
\cs_new_eq:NN \char_make_escape:n           \char_set_catcode_escape:n
\cs_new_eq:NN \char_make_begin_group:n      \char_set_catcode_group_begin:n
\cs_new_eq:NN \char_make_end_group:n        \char_set_catcode_group_end:n
\cs_new_eq:NN \char_make_math_shift:n       \char_set_catcode_math_toggle:n
\cs_new_eq:NN \char_make_alignment_tab:n    \char_set_catcode_alignment:n
\cs_new_eq:NN \char_make_end_line:n         \char_set_catcode_end_line:n
\cs_new_eq:NN \char_make_parameter:n        \char_set_catcode_parameter:n
\cs_new_eq:NN \char_make_math_superscript:n
  \char_set_catcode_math_superscript:n
\cs_new_eq:NN \char_make_math_subscript:n
  \char_set_catcode_math_subscript:n
\cs_new_eq:NN \char_make_ignore:n           \char_set_catcode_ignore:n
\cs_new_eq:NN \char_make_space:n            \char_set_catcode_space:n
\cs_new_eq:NN \char_make_letter:n           \char_set_catcode_letter:n
\cs_new_eq:NN \char_make_other:n            \char_set_catcode_other:n
\cs_new_eq:NN \char_make_active:n           \char_set_catcode_active:n
\cs_new_eq:NN \char_make_comment:n          \char_set_catcode_comment:n
\cs_new_eq:NN \char_make_invalid:n          \char_set_catcode_invalid:n
⟨/deprecated⟩
```

(*End definition for* `\char_make_escape:N` *and others. These functions are documented on page* **??**.)

`\token_if_alignment_tab_p:N`
`\token_if_alignment_tab:NTF`
`\token_if_math_shift_p:N`
`\token_if_math_shift:NTF`
`\token_if_other_char_p:N`
`\token_if_other_char:NTF`
`\token_if_active_char_p:N`
`\token_if_active_char:NTF`

```
⟨*deprecated⟩
\cs_new_eq:NN \token_if_alignment_tab_p:N \token_if_alignment_p:N
\cs_new_eq:NN \token_if_alignment_tab:NT  \token_if_alignment:NT
\cs_new_eq:NN \token_if_alignment_tab:NF  \token_if_alignment:NF
\cs_new_eq:NN \token_if_alignment_tab:NTF \token_if_alignment:NTF
\cs_new_eq:NN \token_if_math_shift_p:N \token_if_math_toggle_p:N
\cs_new_eq:NN \token_if_math_shift:NT  \token_if_math_toggle:NT
\cs_new_eq:NN \token_if_math_shift:NF  \token_if_math_toggle:NF
\cs_new_eq:NN \token_if_math_shift:NTF \token_if_math_toggle:NTF
\cs_new_eq:NN \token_if_other_char_p:N \token_if_other_p:N
\cs_new_eq:NN \token_if_other_char:NT  \token_if_other:NT
\cs_new_eq:NN \token_if_other_char:NF  \token_if_other:NF
\cs_new_eq:NN \token_if_other_char:NTF \token_if_other:NTF
```

```
\cs_new_eq:NN \token_if_active_char_p:N \token_if_active_p:N
\cs_new_eq:NN \token_if_active_char:NT  \token_if_active:NT
\cs_new_eq:NN \token_if_active_char:NF  \token_if_active:NF
\cs_new_eq:NN \token_if_active_char:NTF \token_if_active:NTF
⟨/deprecated⟩
```

(*End definition for* `\token_if_alignment_tab:N`. *These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

190 l3int implementation

```
⟨*initex | package⟩
```

The following test files are used for this code: m3int001,m3int002,m3int03.

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

`\int_to_roman:w`
`\if_int_compare:w`

Done in l3basics.

(*End definition for* `\int_to_roman:w`. *This function is documented on page 71.*)

`\int_value:w`
`\int_eval:w`
`\int_eval_end:`
`\if_num:w`
`\if_int_odd:w`
`\if_case:w`

Here are the remaining primitives for number comparisons and expressions.

```
\cs_new_eq:NN \int_value:w   \tex_number:D
\cs_new_eq:NN \int_eval:w    \etex_numexpr:D
\cs_new_eq:NN \int_eval_end: \tex_relax:D
\cs_new_eq:NN \if_num:w      \tex_ifnum:D
\cs_new_eq:NN \if_int_odd:w  \tex_ifodd:D
\cs_new_eq:NN \if_case:w     \tex_ifcase:D
```

(*End definition for* `\int_value:w`. *This function is documented on page 71.*)

190.1 Integer expressions

`\int_eval:n`

Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream. In format mode, there is already a definition in l3alloc for bookstrapping, which is therefore corrected to the "real" version here.

```
⟨*initex⟩
\cs_set:Npn \int_eval:n #1 { \int_value:w \int_eval:w #1 \int_eval_end: }
⟨/initex⟩
⟨*package⟩
\cs_new:Npn \int_eval:n #1 { \int_value:w \int_eval:w #1 \int_eval_end: }
⟨/package⟩
```

(*End definition for* `\int_eval:n`. *This function is documented on page 61.*)

`\int_max:nn`
`\int_min:nn`
`\int_abs:n`

Functions for min, max, and absolute value.

```
\cs_new:Npn \int_abs:n #1
  {
    \int_value:w
```

```
      \if_int_compare:w \int_eval:w #1 < \c_zero
        -
      \fi:
    \int_eval:w #1 \int_eval_end:
  }
\cs_new:Npn \int_max:nn #1#2
  {
    \int_value:w \int_eval:w
      \if_int_compare:w
        \int_eval:w #1 > \int_eval:w #2 \int_eval_end:
        #1
      \else:
        #2
      \fi:
    \int_eval_end:
  }
\cs_new:Npn \int_min:nn #1#2
  {
    \int_value:w \int_eval:w
      \if_int_compare:w
        \int_eval:w #1 < \int_eval:w #2 \int_eval_end:
        #1
      \else:
        #2
      \fi:
    \int_eval_end:
  }
```

(*End definition for* `\int_max:nn`*. This function is documented on page* *61*.)

`\int_div_truncate:nn`
`\int_div_round:nn`
`\int_mod:nn`
`\int_div_truncate_aux:NwNw`

As `\int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (*e.g.*, `\tl_length:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|\texttt{#3#4}|-1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε-TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```
\cs_new:Npn \int_div_truncate:nn #1#2
  {
    \int_use:N \int_eval:w
      \exp_after:wN \int_div_truncate_aux:NwNw
      \int_use:N \int_eval:w #1 \exp_after:wN ;
      \int_use:N \int_eval:w #2 ;
    \int_eval_end:
  }
\cs_new:Npn \int_div_truncate_aux:NwNw #1#2; #3#4;
  {
    \if_meaning:w 0 #1
```

```
      \c_zero
    \else:
      (
        #1#2
        \if_meaning:w - #1 + \else: - \fi:
        ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
      )
    \fi:
    / #3#4
  }
```

For the sake of completeness:

```
\cs_new:Npn \int_div_round:nn #1#2 { \int_eval:n { ( #1 ) / ( #2 ) } }
```

Finally there's the modulus operation.

```
\cs_new:Npn \int_mod:nn #1#2
  {
    \int_value:w \int_eval:w
      #1 - \int_div_truncate:nn {#1} {#2} * ( #2 )
    \int_eval_end:
  }
```

(*End definition for* `\int_div_truncate:nn`*. This function is documented on page* *62*.)

190.2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

Two ways to do this: one for the format and one for the LaTeX 2_ε package.

```
⟨*package⟩
\cs_new_protected:Npn \int_new:N #1
  {
    \chk_if_free_cs:N #1
    \newcount #1
  }
⟨/package⟩
\cs_generate_variant:Nn \int_new:N { c }
```

(*End definition for* `\int_new:N` *and* `\int_new:c`*. These functions are documented on page* **??**.)

`\int_const:Nn`
`\int_const:cn`
`\int_constdef:Nw`
`\c_max_const_int`

As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done.

```
\cs_new_protected:Npn \int_const:Nn #1#2
  {
    \int_compare:nNnTF {#2} > \c_minus_one
      {
        \int_compare:nNnTF {#2} > \c_max_const_int
          {
            \int_new:N #1
            \int_gset:Nn #1 {#2}
          }
          {
            \chk_if_free_cs:N #1
```

```
          \tex_global:D \int_constdef:Nw #1 =
            \int_eval:w #2 \int_eval_end:
        }
    }
    {
      \int_new:N #1
      \int_gset:Nn #1 {#2}
    }
  }
\cs_generate_variant:Nn \int_const:Nn { c }
\pdftex_if_engine:TF
  {
    \cs_new_eq:NN \int_constdef:Nw \tex_mathchardef:D
    \tex_mathchardef:D \c_max_const_int 32 767 ~
  }
  {
    \cs_new_eq:NN \int_constdef:Nw \tex_chardef:D
    \tex_chardef:D \c_max_const_int 1 114 111 ~
  }
```

(*End definition for* `\int_const:Nn` *and* `\int_const:cn`*. These functions are documented on page* **??**.)

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

Functions that reset an ⟨*integer*⟩ register to zero.

```
\cs_new_protected:Npn \int_zero:N  #1 { #1 = \c_zero }
\cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\cs_generate_variant:Nn \int_zero:N  { c }
\cs_generate_variant:Nn \int_gzero:N { c }
```

(*End definition for* `\int_zero:N` *and* `\int_zero:c`*. These functions are documented on page* **??**.)

`\int_zero_new:N`
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \int_zero_new:N  #1
  { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\cs_new_protected:Npn \int_gzero_new:N #1
  { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
\cs_generate_variant:Nn \int_zero_new:N  { c }
\cs_generate_variant:Nn \int_gzero_new:N { c }
```

(*End definition for* `\int_zero_new:N` *and others. These functions are documented on page* **??**.)

`\int_set_eq:NN`
`\int_set_eq:cN`
`\int_set_eq:Nc`
`\int_set_eq:cc`
`\int_gset_eq:NN`
`\int_gset_eq:cN`
`\int_gset_eq:Nc`
`\int_gset_eq:cc`

Setting equal means using one integer inside the set function of another.

```
\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \int_set_eq:NN {       c }
\cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \int_gset_eq:NN {       c }
\cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
```

(*End definition for* `\int_set_eq:NN` *and others. These functions are documented on page* **??**.)

\int_if_exist_p:N
\int_if_exist_p:c
\int_if_exist:NTF
\int_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\cs_new_eq:NN \int_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \int_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \int_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \int_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \int_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \int_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \int_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \int_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\int_if_exist:N` *and* `\int_if_exist:c`*. These functions are documented on page* **??**.)

190.3 Setting and incrementing integers

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Adding and subtracting to and from a counter …

```
\cs_new_protected:Npn \int_add:Nn #1#2
  { \tex_advance:D #1 by \int_eval:w #2 \int_eval_end: }
\cs_new_protected:Npn \int_sub:Nn #1#2
  { \tex_advance:D #1 by - \int_eval:w #2 \int_eval_end: }
\cs_new_protected_nopar:Npn \int_gadd:Nn
  { \tex_global:D \int_add:Nn }
\cs_new_protected_nopar:Npn \int_gsub:Nn
  { \tex_global:D \int_sub:Nn }
\cs_generate_variant:Nn \int_add:Nn  { c }
\cs_generate_variant:Nn \int_gadd:Nn { c }
\cs_generate_variant:Nn \int_sub:Nn  { c }
\cs_generate_variant:Nn \int_gsub:Nn { c }
```

(*End definition for* `\int_add:Nn` *and* `\int_add:cn`*. These functions are documented on page* **??**.)

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

```
\cs_new_protected:Npn \int_incr:N #1
  { \tex_advance:D #1 \c_one }
\cs_new_protected:Npn \int_decr:N #1
  { \tex_advance:D #1 \c_minus_one }
\cs_new_protected_nopar:Npn \int_gincr:N
  { \tex_global:D \int_incr:N }
\cs_new_protected_nopar:Npn \int_gdecr:N
  { \tex_global:D \int_decr:N }
\cs_generate_variant:Nn \int_incr:N  { c }
\cs_generate_variant:Nn \int_decr:N  { c }
\cs_generate_variant:Nn \int_gincr:N { c }
\cs_generate_variant:Nn \int_gdecr:N { c }
```

(*End definition for* `\int_incr:N` *and* `\int_incr:c`*. These functions are documented on page* **??**.)

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

As integers are register-based TeX will issue an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```
\cs_new_protected:Npn \int_set:Nn #1#2
  { #1 ~ \int_eval:w #2\int_eval_end: }
```

```
\cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
\cs_generate_variant:Nn \int_set:Nn  { c }
\cs_generate_variant:Nn \int_gset:Nn { c }
```

(*End definition for* `\int_set:Nn` *and* `\int_set:cn`*. These functions are documented on page* **??**.)

190.4 Using integers

`\int_use:N`
`\int_use:c`

Here is how counters are accessed:

```
\cs_new_eq:NN \int_use:N \tex_the:D
\cs_new:Npn \int_use:c #1 { \int_use:N \cs:w #1 \cs_end: }
```

(*End definition for* `\int_use:N` *and* `\int_use:c`*. These functions are documented on page* **??**.)

190.5 Integer expression conditionals

`\kernel_compare_error:`
`\kernel_compare_error:NNw`

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `\kernel_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `\kernel_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
\cs_new_protected_nopar:Npn \kernel_compare_error:
  {
    \if_num:w \c_zero \c_zero \fi:
    =
    \kernel_compare_error:
  }
\cs_new:Npn \kernel_compare_error:Nw
    #1#2 \prg_return_true: \else: \prg_return_false: \fi:
  {
    \msg_expandable_kernel_error:nnn
      { kernel } { unknown-comparison } {#1}
    \prg_return_false:
  }
```

(*End definition for* `\kernel_compare_error:` *and* `\kernel_compare_error:NNw`*.*)

`\int_compare_p:n`
`\int_compare:nTF`
`\int_compare_aux:Nw`
`\int_compare_aux:NNw`
`\int_compare_=:NNw`
`\int_compare_<:NNw`
`\int_compare_>:NNw`
`\int_compare_==:NNw`
`\int_compare_!=:NNw`
`\int_compare_<=:NNw`
`\int_compare_>=:NNw`

Comparison tests using a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. We can start evaluating from the left using `\int_eval:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let TeX evaluate this left hand side of the (in)equality. We also insert at that stage the end of the test: `\int_eval_end:` will end the evaluation of the right-hand side.

```
\prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \int_compare_aux:Nw \int_use:N \int_eval:w #1
        \kernel_compare_error: \int_eval_end:
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

We have just evaluated the left-hand side. To access the relation symbol, we remove the number by applying `\int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. The `\int_compare_aux:NNw` auxiliary then probes the first two tokens to determine the relation symbol, building a control sequence from it. All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by TEX into `\scan_stop:`, and `\kernel_compare_error:Nw` raises an error.

```
\cs_new:Npn \int_compare_aux:Nw #1#2 \kernel_compare_error:
  {
    \exp_after:wN \int_compare_aux:NNw
      \int_to_roman:w - 0 #2 ?? \q_mark
    #1#2
  }
\cs_new:Npn \int_compare_aux:NNw #1#2#3 \q_mark
  {
    \use:c { int_compare_  #1  \if_meaning:w = #2 =  \fi: :NNw }
      \kernel_compare_error:Nw #1
  }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\kernel_compare_error:Nw` ⟨*token*⟩ responsible for error detection.

```
\cs_new:cpn { int_compare_=:NNw } #1#2#3 =
  { \if_int_compare:w #3 = \int_eval:w }
\cs_new:cpn { int_compare_<:NNw } #1#2#3 <
  { \if_int_compare:w #3 < \int_eval:w }
\cs_new:cpn { int_compare_>:NNw } #1#2#3 >
  { \if_int_compare:w #3 > \int_eval:w }
\cs_new:cpn { int_compare_==:NNw } #1#2#3 ==
  { \if_int_compare:w #3 = \int_eval:w }
\cs_new:cpn { int_compare_!=:NNw } #1#2#3 !=
  { \reverse_if:N \if_int_compare:w #3 = \int_eval:w }
\cs_new:cpn { int_compare_<=:NNw } #1#2#3 <=
  { \reverse_if:N \if_int_compare:w #3 > \int_eval:w }
\cs_new:cpn { int_compare_>=:NNw } #1#2#3 >=
  { \reverse_if:N \if_int_compare:w #3 < \int_eval:w }
```

(*End definition for* `\int_compare:n`. *These functions are documented on page 64.*)

`\int_compare_p:nNn`
`\int_compare:nNnTF`

More efficient but less natural in typing.

```
\prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF}
  {
    \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3 \int_eval_end:
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\int_compare:nNn`. *These functions are documented on page 64.*)

`\int_if_odd_p:n`
`\int_if_odd:nTF`
`\int_if_even_p:n`
`\int_if_even:nTF`

A predicate function.

```
\prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \int_eval:w #1 \int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \int_eval:w #1 \int_eval_end:
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\int_if_odd:n`. *These functions are documented on page 64.*)

190.6 Integer expression loops

`\int_while_do:nn`
`\int_until_do:nn`
`\int_do_while:nn`
`\int_do_until:nn`

These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```
\cs_new:Npn \int_while_do:nn #1#2
  {
    \int_compare:nT {#1}
      {
        #2
        \int_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_until_do:nn #1#2
  {
    \int_compare:nF {#1}
      {
        #2
        \int_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_do_while:nn #1#2
  {
    #2
```

```
    \int_compare:nT {#1}
      { \int_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \int_do_until:nn #1#2
  {
    #2
    \int_compare:nF {#1}
      { \int_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\int_while_do:nn`. *This function is documented on page 65.*)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```
\cs_new:Npn \int_while_do:nNnn #1#2#3#4
  {
    \int_compare:nNnT {#1} #2 {#3}
      {
        #4
        \int_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_until_do:nNnn #1#2#3#4
  {
  \int_compare:nNnF {#1} #2 {#3}
    {
      #4
      \int_until_do:nNnn {#1} #2 {#3} {#4}
    }
  }
\cs_new:Npn \int_do_while:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnT {#1} #2 {#3}
      { \int_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \int_do_until:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnF {#1} #2 {#3}
      { \int_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\int_while_do:nNnn`. *This function is documented on page 65.*)

190.7 Formatting integers

`\int_to_arabic:n`

Nothing exciting here.

```
\cs_new:Npn \int_to_arabic:n #1 { \int_eval:n {#1} }
```

(*End definition for* `\int_to_arabic:n`. *This function is documented on page 66.*)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
\cs_new:Npn \int_to_symbols:nnn #1#2#3
  {
    \int_compare:nNnTF {#1} > {#2}
      {
        \exp_args:NNo \exp_args:No \int_to_symbols_aux:nnnn
          {
            \prg_case_int:nnn
              { 1 + \int_mod:nn { #1 - 1 } {#2} }
              {#3} { }
          }
          {#1} {#2} {#3}
      }
      { \prg_case_int:nnn {#1} {#3} { } }
  }
\cs_new:Npn \int_to_symbols_aux:nnnn #1#2#3#4
  {
    \exp_args:Nf \int_to_symbols:nnn
      { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
    #1
  }
```

(*End definition for* `\int_to_symbols:nnn`. *This function is documented on page 67.*)

`\int_to_alph:n`
`\int_to_Alph:n` These both use the above function with input functions that make sense for the alphabet in English.

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        {  3 } { c }
        {  4 } { d }
        {  5 } { e }
        {  6 } { f }
        {  7 } { g }
        {  8 } { h }
        {  9 } { i }
        { 10 } { j }
        { 11 } { k }
        { 12 } { l }
        { 13 } { m }
        { 14 } { n }
```

```
        { 15 } { o }
        { 16 } { p }
        { 17 } { q }
        { 18 } { r }
        { 19 } { s }
        { 20 } { t }
        { 21 } { u }
        { 22 } { v }
        { 23 } { w }
        { 24 } { x }
        { 25 } { y }
        { 26 } { z }
      }
  }
\cs_new:Npn \int_to_Alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { A }
        {  2 } { B }
        {  3 } { C }
        {  4 } { D }
        {  5 } { E }
        {  6 } { F }
        {  7 } { G }
        {  8 } { H }
        {  9 } { I }
        { 10 } { J }
        { 11 } { K }
        { 12 } { L }
        { 13 } { M }
        { 14 } { N }
        { 15 } { O }
        { 16 } { P }
        { 17 } { Q }
        { 18 } { R }
        { 19 } { S }
        { 20 } { T }
        { 21 } { U }
        { 22 } { V }
        { 23 } { W }
        { 24 } { X }
        { 25 } { Y }
        { 26 } { Z }
      }
  }
```

(*End definition for* `\int_to_alph:n` *and* `\int_to_Alph:n`*. These functions are documented on page* *66*.)

`\int_to_base:nn`
`\int_to_base_aux_i:nn`
`\int_to_base_aux_ii:nnN`
`\int_to_base_aux_iii:nnnN`
`\int_to_letter:n`

Converting from base ten (`#1`) to a second base (`#2`) starts with computing `#1`: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.

```
\cs_new:Npn \int_to_base:nn #1
  { \exp_args:Nf \int_to_base_aux_i:nn { \int_eval:n {#1} } }
\cs_new:Npn \int_to_base_aux_i:nn #1#2
  {
    \int_compare:nNnTF {#1} < \c_zero
      { \exp_args:No \int_to_base_aux_ii:nnN { \use_none:n #1 } {#2} - }
      { \int_to_base_aux_ii:nnN {#1} {#2} \c_empty_tl }
  }
```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```
\cs_new:Npn \int_to_base_aux_ii:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \int_to_letter:n {#1} } }
      {
        \exp_args:Nf \int_to_base_aux_iii:nnnN
          { \int_to_letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \int_to_base_aux_iii:nnnN #1#2#3#4
  {
    \exp_args:Nf \int_to_base_aux_ii:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use \prg_case_int:nnn, but in our case, the cases are contiguous, so it is forty times faster to use the \if_case:w primitive. The first \exp_after:wN expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since #1 might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing \fi:.

```
\cs_new:Npn \int_to_letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \int_eval:w #1 - \c_ten \int_eval_end:
         A
    \or: B
```

```
    \or: C
    \or: D
    \or: E
    \or: F
    \or: G
    \or: H
    \or: I
    \or: J
    \or: K
    \or: L
    \or: M
    \or: N
    \or: O
    \or: P
    \or: Q
    \or: R
    \or: S
    \or: T
    \or: U
    \or: V
    \or: W
    \or: X
    \or: Y
    \or: Z
    \else: \int_value:w \int_eval:w #1 \exp_after:wN \int_eval_end:
    \fi:
  }
```

(*End definition for* `\int_to_base:nn`. *This function is documented on page 70.*)

\int_to_binary:n
\int_to_hexadecimal:n
\int_to_octal:n

Wrappers around the generic function.

```
\cs_new:Npn \int_to_binary:n #1
  { \int_to_base:nn {#1} { 2 } }
\cs_new:Npn \int_to_hexadecimal:n #1
  { \int_to_base:nn {#1} { 16 } }
\cs_new:Npn \int_to_octal:n #1
  { \int_to_base:nn {#1} { 8 } }
```

(*End definition for* `\int_to_binary:n`, `\int_to_hexadecimal:n`, *and* `\int_to_octal:n`. *These functions are documented on page 67.*)

\int_to_roman:n
\int_to_Roman:n
\int_to_roman_aux:N
\int_to_roman_aux:N
\int_to_roman_i:w
\int_to_roman_v:w
\int_to_roman_x:w
\int_to_roman_l:w
\int_to_roman_c:w
\int_to_roman_d:w
\int_to_roman_m:w
\int_to_roman_Q:w
\int_to_Roman_i:w
\int_to_Roman_v:w
\int_to_Roman_x:w
\int_to_Roman_l:w
\int_to_Roman_c:w
\int_to_Roman_d:w
\int_to_Roman_m:w
\int_to_Roman_Q:w

The `\int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the `Q`.

```
\cs_new:Npn \int_to_roman:n #1
  {
    \exp_after:wN \int_to_roman_aux:N
      \int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \int_to_roman_aux:N #1
```

```
  {
    \use:c { int_to_roman_ #1 :w }
    \int_to_roman_aux:N
  }
\cs_new:Npn \int_to_Roman:n #1
  {
    \exp_after:wN \int_to_Roman_aux:N
      \int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \int_to_Roman_aux:N #1
  {
    \use:c { int_to_Roman_ #1 :w }
    \int_to_Roman_aux:N
  }
\cs_new_nopar:Npn \int_to_roman_i:w { i }
\cs_new_nopar:Npn \int_to_roman_v:w { v }
\cs_new_nopar:Npn \int_to_roman_x:w { x }
\cs_new_nopar:Npn \int_to_roman_l:w { l }
\cs_new_nopar:Npn \int_to_roman_c:w { c }
\cs_new_nopar:Npn \int_to_roman_d:w { d }
\cs_new_nopar:Npn \int_to_roman_m:w { m }
\cs_new_nopar:Npn \int_to_roman_Q:w #1 { }
\cs_new_nopar:Npn \int_to_Roman_i:w { I }
\cs_new_nopar:Npn \int_to_Roman_v:w { V }
\cs_new_nopar:Npn \int_to_Roman_x:w { X }
\cs_new_nopar:Npn \int_to_Roman_l:w { L }
\cs_new_nopar:Npn \int_to_Roman_c:w { C }
\cs_new_nopar:Npn \int_to_Roman_d:w { D }
\cs_new_nopar:Npn \int_to_Roman_m:w { M }
\cs_new:Npn \int_to_Roman_Q:w #1 { }
```

(*End definition for* `\int_to_roman:n` *and* `\int_to_Roman:n`*. These functions are documented on page 68.*)

190.8 Converting from other formats to integers

`\int_get_sign:n`
`\int_get_digits:n`
`\int_get_sign_and_digits_aux:nNNN`
`\int_get_sign_and_digits_aux:oNNN`

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```
\cs_new:Npn \int_get_sign:n #1
  {
    \int_get_sign_and_digits_aux:nNNN {#1}
      \c_true_bool \c_true_bool \c_false_bool
  }
\cs_new:Npn \int_get_digits:n #1
  {
    \int_get_sign_and_digits_aux:nNNN {#1}
      \c_true_bool \c_false_bool \c_true_bool
  }
```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```
\cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4
  {
    \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
      {
        \bool_if:NTF #2
          {
            \int_get_sign_and_digits_aux:oNNN
              { \use_none:n #1 } \c_false_bool #3#4
          }
          {
            \int_get_sign_and_digits_aux:oNNN
              { \use_none:n #1 } \c_true_bool #3#4
          }
      }
      {
        \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +
          { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
          {
            \bool_if:NT #3 { \bool_if:NF #2 - }
            \bool_if:NT #4 {#1}
          }
      }
  }
\cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN { o }
```

(*End definition for* `\int_get_sign:n`*. This function is documented on page 70.*)

`\int_from_alph:n`
`\int_from_alph_aux:n`
`\int_from_alph_aux:nN`
`\int_from_alph_aux:N`

The aim here is to iterate through the input, converting one letter at a time to a number. The same approach is also used for base conversion, but this needs a different final auxiliary.

```
\cs_new:Npn \int_from_alph:n #1
  {
    \int_eval:n
      {
        \int_get_sign:n {#1}
        \exp_args:Nf \int_from_alph_aux:n { \int_get_digits:n {#1} }
      }
  }
\cs_new:Npn \int_from_alph_aux:n #1
  { \int_from_alph_aux:nN { 0 } #1 \q_nil }
\cs_new:Npn \int_from_alph_aux:nN #1#2
  {
    \quark_if_nil:NTF #2
      {#1}
      {
        \exp_args:Nf \int_from_alph_aux:nN
          { \int_eval:n { #1 * 26 + \int_from_alph_aux:N #2 } }
      }
```

```
  }
\cs_new:Npn \int_from_alph_aux:N #1
  { \int_eval:n { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 64 } { 96 } } }
```

(*End definition for* `\int_from_alph:n`*. This function is documented on page 68.*)

`\int_from_base:nn`
`\int_from_base_aux:nn`
`\int_from_base_aux:nnN`
`\int_from_base_aux:N`

Conversion to base ten means stripping off the sign then iterating through the input one token at a time. The total number is then added up as the code loops.

```
\cs_new:Npn \int_from_base:nn #1#2
  {
    \int_eval:n
      {
        \int_get_sign:n {#1}
        \exp_args:Nf \int_from_base_aux:nn
          { \int_get_digits:n {#1} } {#2}
      }
  }
\cs_new:Npn \int_from_base_aux:nn #1#2
  { \int_from_base_aux:nnN { 0 } { #2 } #1 \q_nil }
\cs_new:Npn \int_from_base_aux:nnN #1#2#3
  {
    \quark_if_nil:NTF #3
      {#1}
      {
        \exp_args:Nf \int_from_base_aux:nnN
          { \int_eval:n { #1 * #2 + \int_from_base_aux:N #3 } }
          {#2}
      }
  }
```

The conversion here will take lower or upper case letters and turn them into the appropriate number, hence the two-part nature of the function.

```
\cs_new:Npn \int_from_base_aux:N #1
  {
    \int_compare:nNnTF { `#1 } < { 58 }
      {#1}
      {
        \int_eval:n
          { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 55 } { 87 } }
      }
  }
```

(*End definition for* `\int_from_base:nn`*. This function is documented on page 68.*)

`\int_from_binary:n`
`\int_from_hexadecimal:n`
`\int_from_octal:n`

Wrappers around the generic function.

```
\cs_new:Npn \int_from_binary:n #1
  { \int_from_base:nn {#1} \c_two }
\cs_new:Npn \int_from_hexadecimal:n #1
  { \int_from_base:nn {#1} \c_sixteen }
\cs_new:Npn \int_from_octal:n #1
  { \int_from_base:nn {#1} \c_eight }
```

(*End definition for* `\int_from_binary:n`*,* `\int_from_hexadecimal:n`*, and* `\int_from_octal:n`*. These functions are documented on page 68.*)

`\c_int_from_roman_i_int`
`\c_int_from_roman_v_int`
`\c_int_from_roman_x_int`
`\c_int_from_roman_l_int`
`\c_int_from_roman_c_int`
`\c_int_from_roman_d_int`
`\c_int_from_roman_m_int`
`\c_int_from_roman_I_int`
`\c_int_from_roman_V_int`
`\c_int_from_roman_X_int`
`\c_int_from_roman_L_int`
`\c_int_from_roman_C_int`
`\c_int_from_roman_D_int`
`\c_int_from_roman_M_int`

Constants used to convert from Roman numerals to integers.

```
\int_const:cn { c_int_from_roman_i_int } { 1 }
\int_const:cn { c_int_from_roman_v_int } { 5 }
\int_const:cn { c_int_from_roman_x_int } { 10 }
\int_const:cn { c_int_from_roman_l_int } { 50 }
\int_const:cn { c_int_from_roman_c_int } { 100 }
\int_const:cn { c_int_from_roman_d_int } { 500 }
\int_const:cn { c_int_from_roman_m_int } { 1000 }
\int_const:cn { c_int_from_roman_I_int } { 1 }
\int_const:cn { c_int_from_roman_V_int } { 5 }
\int_const:cn { c_int_from_roman_X_int } { 10 }
\int_const:cn { c_int_from_roman_L_int } { 50 }
\int_const:cn { c_int_from_roman_C_int } { 100 }
\int_const:cn { c_int_from_roman_D_int } { 500 }
\int_const:cn { c_int_from_roman_M_int } { 1000 }
```

(*End definition for* `\c_int_from_roman_i_int` *and others. These variables are documented on page* **??**.)

`\int_from_roman:n`
`\int_from_roman_aux:NN`
`\int_from_roman_end:w`
`\int_from_roman_clean_up:w`

The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by TeX.

```
\cs_new:Npn \int_from_roman:n #1
  {
    \tl_if_blank:nF {#1}
      {
        \exp_after:wN \int_from_roman_end:w
          \int_value:w \int_eval:w
            \int_from_roman_aux:NN #1 Q \q_stop
      }
  }
\cs_new:Npn \int_from_roman_aux:NN #1#2
  {
    \str_if_eq:nnTF {#1} { Q }
      {#1#2}
      {
        \str_if_eq:nnTF {#2} { Q }
          {
            \int_if_exist:cF { c_int_from_roman_ #1 _int }
              { \int_from_roman_clean_up:w }
            +
            \use:c { c_int_from_roman_ #1 _int }
            #2
          }
          {
            \int_if_exist:cF { c_int_from_roman_ #1 _int }
              { \int_from_roman_clean_up:w }
            \int_if_exist:cF { c_int_from_roman_ #2 _int }
              { \int_from_roman_clean_up:w }
```

```
            \int_compare:nNnTF
              { \use:c { c_int_from_roman_ #1 _int } }
              <
              { \use:c { c_int_from_roman_ #2 _int  } }
              {
                + \use:c { c_int_from_roman_ #2 _int }
                - \use:c { c_int_from_roman_ #1 _int }
                \int_from_roman_aux:NN
              }
              {
                + \use:c { c_int_from_roman_ #1 _int }
                \int_from_roman_aux:NN #2
              }
          }
      }
  }
\cs_new:Npn \int_from_roman_end:w #1 Q #2 \q_stop
  { \tl_if_empty:nTF {#2} {#1} {#2} }
\cs_new:Npn \int_from_roman_clean_up:w #1 Q { + 0 Q -1 }
```

(*End definition for* `\int_from_roman:n`. *This function is documented on page 68.*)

190.9 Viewing integer

`\int_show:N`
`\int_show:c`

```
\cs_new_eq:NN \int_show:N \kernel_register_show:N
\cs_new_eq:NN \int_show:c \kernel_register_show:c
```

(*End definition for* `\int_show:N` *and* `\int_show:c`*. These functions are documented on page* **??**.)

`\int_show:n`

```
\cs_new_protected:Npn \int_show:n #1
  { \tex_showthe:D \int_eval:w #1 \int_eval_end: }
```

(*End definition for* `\int_show:n`. *This function is documented on page 69.*)

190.10 Constant integers

`\c_minus_one` This is needed early, and so is in l3basics

(*End definition for* `\c_minus_one`. *This variable is documented on page 69.*)

`\c_zero` Again, one in l3basics for obvious reasons.

(*End definition for* `\c_zero`. *This variable is documented on page 69.*)

`\c_six`
`\c_seven`
`\c_twelve`
`\c_sixteen`

Once again, in l3basics.

(*End definition for* `\c_six` *and* `\c_seven`*. These functions are documented on page 69.*)

`\c_one`
`\c_two`
`\c_three`
`\c_four`
`\c_five`
`\c_eight`
`\c_nine`
`\c_ten`
`\c_eleven`
`\c_thirteen`
`\c_fourteen`
`\c_fifteen`

Low-number values not previously defined.

```
\int_const:Nn \c_one       {  1 }
\int_const:Nn \c_two       {  2 }
\int_const:Nn \c_three     {  3 }
\int_const:Nn \c_four      {  4 }
```

```
\int_const:Nn \c_five     {  5 }
\int_const:Nn \c_eight    {  8 }
\int_const:Nn \c_nine     {  9 }
\int_const:Nn \c_ten      { 10 }
\int_const:Nn \c_eleven   { 11 }
\int_const:Nn \c_thirteen { 13 }
\int_const:Nn \c_fourteen { 14 }
\int_const:Nn \c_fifteen  { 15 }
```

(*End definition for* `\c_one` *and others. These variables are documented on page* *69*.)

`\c_thirty_two`

One middling value.

```
\int_const:Nn \c_thirty_two { 32 }
```

(*End definition for* `\c_thirty_two`*. This variable is documented on page* *69*.)

`\c_two_hundred_fifty_five`
`\c_two_hundred_fifty_six`

Two classic mid-range integer constants.

```
\int_const:Nn \c_two_hundred_fifty_five { 255 }
\int_const:Nn \c_two_hundred_fifty_six  { 256 }
```

(*End definition for* `\c_two_hundred_fifty_five` *and* `\c_two_hundred_fifty_six`*. These variables are documented on page* *69*.)

`\c_one_hundred`
`\c_one_thousand`
`\c_ten_thousand`

Simple runs of powers of ten.

```
\int_const:Nn \c_one_hundred  {   100 }
\int_const:Nn \c_one_thousand {  1000 }
\int_const:Nn \c_ten_thousand { 10000 }
```

(*End definition for* `\c_one_hundred`*,* `\c_one_thousand`*, and* `\c_ten_thousand`*. These variables are documented on page* *69*.)

`\c_max_int`

The largest number allowed is $2^{31} - 1$

```
\int_const:Nn \c_max_int { 2 147 483 647 }
```

(*End definition for* `\c_max_int`*. This variable is documented on page* *69*.)

190.11 Scratch integers

`\l_tmpa_int`
`\l_tmpb_int`
`\l_tmpc_int`
`\g_tmpa_int`
`\g_tmpb_int`

We provide three 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
```

(*End definition for* `\l_tmpa_int`*,* `\l_tmpb_int`*, and* `\l_tmpc_int`*. These functions are documented on page* *70*.)

190.12 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

\int_convert_from_base_ten:nn
\int_convert_to_symbols:nnn
\int_convert_to_base_ten:nn

Some simple renames.

```
⟨*deprecated⟩
\cs_new_eq:NN \int_convert_from_base_ten:nn \int_to_base:nn
\cs_new_eq:NN \int_convert_to_symbols:nnn   \int_to_symbols:nnn
\cs_new_eq:NN \int_convert_to_base_ten:nn   \int_from_base:nn
⟨/deprecated⟩
```

(*End definition for* `\int_convert_from_base_ten:nn`. *This function is documented on page* **??**.)

\int_to_symbol:n
\int_to_symbol_math:n
\int_to_symbol_text:n

This is rather too tied to LaTeX 2_ε.

```
⟨*deprecated⟩
\cs_new_nopar:Npn \int_to_symbol:n
  {
    \scan_align_safe_stop:
    \mode_if_math:TF
      { \int_to_symbol_math:n }
      { \int_to_symbol_text:n }
  }
\cs_new:Npn \int_to_symbol_math:n #1
  {
    \int_to_symbols:nnn {#1} { 9 }
      {
        { 1 } {                 * }
        { 2 } {           \dagger }
        { 3 } {          \ddagger }
        { 4 } {      \mathsection }
        { 5 } {    \mathparagraph }
        { 6 } {                \| }
        { 7 } {                ** }
        { 8 } {   \dagger \dagger }
        { 9 } { \ddagger \ddagger }
      }
  }
\cs_new:Npn \int_to_symbol_text:n #1
  {
    \int_to_symbols:nnn {#1} { 9 }
      {
        { 1 } {                       \textasteriskcentered }
        { 2 } {                                 \textdagger }
        { 3 } {                              \textdaggerdbl }
        { 4 } {                                \textsection }
        { 5 } {                              \textparagraph }
        { 6 } {                                \textbardbl }
        { 7 } { \textasteriskcentered \textasteriskcentered }
        { 8 } {                     \textdagger \textdagger }
        { 9 } {               \textdaggerdbl \textdaggerdbl }
      }
```

```
  }
⟨/deprecated⟩
```

(*End definition for* `\int_to_symbol:n`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

191 l3skip implementation

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

191.1 Length primitives renamed

`\if_dim:w` `\dim_eval:w` `\dim_eval_end:`

Primitives renamed.

```
\cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\cs_new_eq:NN \dim_eval:w    \etex_dimexpr:D
\cs_new_eq:NN \dim_eval_end: \tex_relax:D
```

(*End definition for* `\if_dim:w`*. This function is documented on page* **??**.)

191.2 Creating and initialising dim variables

`\dim_new:N` `\dim_new:c`

Allocating ⟨*dim*⟩ registers ...

```
⟨*package⟩
\cs_new_protected:Npn \dim_new:N #1
  {
    \chk_if_free_cs:N #1
    \newdimen #1
  }
⟨/package⟩
\cs_generate_variant:Nn \dim_new:N { c }
```

(*End definition for* `\dim_new:N` *and* `\dim_new:c`*. These functions are documented on page* **??**.)

`\dim_const:Nn` `\dim_const:cn`

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \dim_const:Nn #1
  {
    \dim_new:N #1
    \dim_gset:Nn #1
  }
\cs_generate_variant:Nn \dim_const:Nn { c }
```

(*End definition for* `\dim_const:Nn` *and* `\dim_const:cn`*. These functions are documented on page* **??**.)

\dim_zero:N
\dim_zero:c
\dim_gzero:N
\dim_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\cs_generate_variant:Nn \dim_zero:N  { c }
\cs_generate_variant:Nn \dim_gzero:N { c }
```

(*End definition for* `\dim_zero:N` *and* `\dim_zero:c`*. These functions are documented on page* **??**.)

\dim_zero_new:N
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \dim_zero_new:N  #1
  { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\cs_new_protected:Npn \dim_gzero_new:N #1
  { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
\cs_generate_variant:Nn \dim_zero_new:N  { c }
\cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(*End definition for* `\dim_zero_new:N` *and others. These functions are documented on page* **??**.)

\dim_if_exist_p:N
\dim_if_exist_p:c
\dim_if_exist:N*TF*
\dim_if_exist:c*TF*

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \dim_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \dim_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \dim_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \dim_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \dim_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \dim_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \dim_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \dim_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\dim_if_exist:N` *and* `\dim_if_exist:c`*. These functions are documented on page* **??**.)

191.3 Setting dim variables

\dim_set:Nn
\dim_set:cn
\dim_gset:Nn
\dim_gset:cn

Setting dimensions is easy enough.

```
\cs_new_protected:Npn \dim_set:Nn #1#2
  { #1 ~ \dim_eval:w #2 \dim_eval_end: }
\cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
\cs_generate_variant:Nn \dim_set:Nn  { c }
\cs_generate_variant:Nn \dim_gset:Nn { c }
```

(*End definition for* `\dim_set:Nn` *and* `\dim_set:cn`*. These functions are documented on page* **??**.)

\dim_set_eq:NN
\dim_set_eq:cN
\dim_set_eq:Nc
\dim_set_eq:cc
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \dim_set_eq:NN {       c }
\cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \dim_gset_eq:NN {       c }
\cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
```

(*End definition for* `\dim_set_eq:NN` *and others. These functions are documented on page* **??**.)

\dim_set_max:Nn \dim_set_max:cn \dim_set_min:Nn \dim_set_min:cn \dim_gset_max:Nn \dim_gset_max:cn \dim_gset_min:Nn \dim_gset_min:cn \dim_set_max_aux:NNNn

Setting maximum and minimum values is simply a case of so build-in comparison. This only applies to dimensions as skips are not ordered.

```
\cs_new_protected_nopar:Npn \dim_set_max:Nn
  { \dim_set_max_aux:NNNn < \dim_set:Nn }
\cs_new_protected_nopar:Npn \dim_gset_max:Nn
  { \dim_set_max_aux:NNNn < \dim_gset:Nn }
\cs_new_protected_nopar:Npn \dim_set_min:Nn
  { \dim_set_max_aux:NNNn > \dim_set:Nn }
\cs_new_protected_nopar:Npn \dim_gset_min:Nn
  { \dim_set_max_aux:NNNn > \dim_gset:Nn }
\cs_new_protected:Npn \dim_set_max_aux:NNNn #1#2#3#4
  { \dim_compare:nNnT {#3} #1 {#4} { #2 #3 {#4} } }
\cs_generate_variant:Nn \dim_set_max:Nn  { c }
\cs_generate_variant:Nn \dim_gset_max:Nn { c }
\cs_generate_variant:Nn \dim_set_min:Nn  { c }
\cs_generate_variant:Nn \dim_gset_min:Nn { c }
```

(*End definition for* `\dim_set_max:Nn` *and* `\dim_set_max:cn`*. These functions are documented on page* **??**.)

\dim_add:Nn \dim_add:cn \dim_gadd:Nn \dim_gadd:cn \dim_sub:Nn \dim_sub:cn \dim_gsub:Nn \dim_gsub:cn

Using `by` here deals with the (incorrect) case `\dimen123`.

```
\cs_new_protected:Npn \dim_add:Nn #1#2
  { \tex_advance:D #1 by \dim_eval:w #2 \dim_eval_end: }
\cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\cs_generate_variant:Nn \dim_add:Nn  { c }
\cs_generate_variant:Nn \dim_gadd:Nn { c }
\cs_new_protected:Npn \dim_sub:Nn #1#2
  { \tex_advance:D #1 by - \dim_eval:w #2 \dim_eval_end: }
\cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
\cs_generate_variant:Nn \dim_sub:Nn  { c }
\cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(*End definition for* `\dim_add:Nn` *and* `\dim_add:cn`*. These functions are documented on page* **??**.)

191.4 Utilities for dimension calculations

\dim_abs:n

Similar to the `\int_abs:n` function, but here an additional ⟨*dimexpr*⟩ is needed as TeX won't simply tidy up an additional `-` for us.

```
\cs_new:Npn \dim_abs:n #1
  {
    \dim_use:N
      \dim_eval:w
        \if_dim:w \dim_eval:w #1 < \c_zero_dim
          -
        \fi:
      \dim_eval:w #1 \dim_eval_end:
    \dim_eval_end:
  }
```

(*End definition for* `\dim_abs:n`*. This function is documented on page 73.*)

\dim_ratio:nn
\dim_ratio_aux:n

With dimension expressions, something like `10 pt * ( 5 pt / 10 pt )` will not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```
\cs_new:Npn \dim_ratio:nn #1#2
  { \dim_ratio_aux:n {#1} / \dim_ratio_aux:n {#2} }
\cs_new:Npn \dim_ratio_aux:n #1
  { \int_value:w \dim_eval:w #1 \dim_eval_end: }
```

(*End definition for* `\dim_ratio:nn`*. This function is documented on page* *74*.)

191.5 Dimension expression conditionals

\dim_compare_p:nNn
\dim_compare:nNn*TF*

Simple comparison.

```
\prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_dim:w \dim_eval:w #1 #2 \dim_eval:w #3 \dim_eval_end:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\dim_compare_p:nNn`*. This function is documented on page* *74*.)

\dim_compare_p:n
\dim_compare:n*TF*
\dim_compare_aux:w
\dim_compare_aux:NNw
\dim_compare_=:NNw
\dim_compare_<:NNw
\dim_compare_>:NNw
\dim_compare_==:NNw
\dim_compare_!=:NNw
\dim_compare_<=:NNw
\dim_compare_>=:NNw

This code is adapted from the `\int_compare:nTF` function. First evaluate the left-hand side. Then access the relation symbol by grabbing until `pt` (with category other), and pursue otherwise just as for `\int_compare:nTF`.

```
\prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \dim_compare_aux:w \dim_use:N \dim_eval:w #1
        \kernel_compare_error: \dim_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \dim_compare_aux:w #1 }
  { \tl_to_str:n { pt } }
  #2 \kernel_compare_error:
  {
    \exp_after:wN \dim_compare_aux:NNw #2 ?? \q_mark
    #1 pt #2
  }
\cs_new:Npn \dim_compare_aux:NNw #1#2#3 \q_mark
  {
    \use:c { dim_compare_  #1  \if_meaning:w = #2 =  \fi: :NNw }
      \kernel_compare_error:Nw #1
  }
\cs_new:cpn { dim_compare_=:NNw } #1#2#3 =
  { \if_dim:w #3 = \dim_eval:w }
\cs_new:cpn { dim_compare_<:NNw } #1#2#3 <
  { \if_dim:w #3 < \dim_eval:w }
```

```
\cs_new:cpn { dim_compare_>:NNw } #1#2#3 >
  { \if_dim:w #3 > \dim_eval:w }
\cs_new:cpn { dim_compare_==:NNw } #1#2#3 ==
  { \if_dim:w #3 = \dim_eval:w }
\cs_new:cpn { dim_compare_!=:NNw } #1#2#3 !=
  { \reverse_if:N \if_dim:w #3 = \dim_eval:w }
\cs_new:cpn { dim_compare_<=:NNw } #1#2#3 <=
  { \reverse_if:N \if_dim:w #3 > \dim_eval:w }
\cs_new:cpn { dim_compare_>=:NNw } #1#2#3 >=
  { \reverse_if:N \if_dim:w #3 < \dim_eval:w }
```

(*End definition for* `\dim_compare:n`. *These functions are documented on page* 74.)

191.6 Dimension expression loops

`\dim_while_do:nn`
`\dim_until_do:nn`
`\dim_do_while:nn`
`\dim_do_until:nn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_set:Npn \dim_while_do:nn #1#2
  {
    \dim_compare:nT {#1}
      {
        #2
        \dim_while_do:nn {#1} {#2}
      }
  }
\cs_set:Npn \dim_until_do:nn #1#2
  {
    \dim_compare:nF {#1}
      {
        #2
        \dim_until_do:nn {#1} {#2}
      }
  }
\cs_set:Npn \dim_do_while:nn #1#2
  {
    #2
    \dim_compare:nT {#1}
      { \dim_do_while:nn {#1} {#2} }
  }
\cs_set:Npn \dim_do_until:nn #1#2
  {
    #2
    \dim_compare:nF {#1}
      { \dim_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\dim_while_do:nn`. *This function is documented on page* 75.)

`\dim_while_do:nNnn`
`\dim_until_do:nNnn`
`\dim_do_while:nNnn`
`\dim_do_until:nNnn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_set:Npn \dim_while_do:nNnn #1#2#3#4
```

```
  {
    \dim_compare:nNnT {#1} #2 {#3}
      {
        #4
        \dim_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_set:Npn \dim_until_do:nNnn #1#2#3#4
  {
  \dim_compare:nNnF {#1} #2 {#3}
    {
      #4
      \dim_until_do:nNnn {#1} #2 {#3} {#4}
    }
  }
\cs_set:Npn \dim_do_while:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnT {#1} #2 {#3}
      { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_set:Npn \dim_do_until:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnF {#1} #2 {#3}
      { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\dim_while_do:nNnn`. *This function is documented on page* 75.)

191.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```
\cs_new:Npn \dim_eval:n #1
  { \dim_use:N \dim_eval:w #1 \dim_eval_end: }
```

(*End definition for* `\dim_eval:n`. *This function is documented on page* 76.)

`\dim_strip_bp:n`

```
\cs_new:Npn \dim_strip_bp:n #1
  { \dim_strip_pt:n { 0.996 26 \dim_eval:w #1 \dim_eval_end: } }
```

(*End definition for* `\dim_strip_bp:n`. *This function is documented on page* 83.)

`\dim_strip_pt:n`
`\dim_strip_pt:w`

A function which comes up often enough to deserve a place in the kernel. The idea here is that the input is assumed to be in `pt`, but can be given in other units, while the output is the value of the dimension in `pt` but with no units given. This is used a lot by low-level manipulations.

```
\cs_new:Npn \dim_strip_pt:n #1
  {
    \exp_after:wN
```

```
      \dim_strip_pt:w \dim_use:N \dim_eval:w #1 \dim_eval_end: \q_stop
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \dim_strip_pt:w
      ##1 . ##2 \tl_to_str:n { pt } ##3 \exp_not:N \q_stop
      {
        ##1
        \exp_not:N \int_compare:nNnT {##2} > \c_zero
          { . ##2 }
      }
  }
```

(*End definition for* `\dim_strip_pt:n`*. This function is documented on page 83.*)

`\dim_use:N`
`\dim_use:c`

Accessing a ⟨*dim*⟩.

```
\cs_new_eq:NN \dim_use:N \tex_the:D
\cs_generate_variant:Nn \dim_use:N { c }
```

(*End definition for* `\dim_use:N` *and* `\dim_use:c`*. These functions are documented on page* **??**.)

191.8 Viewing dim variables

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

Diagnostics.

```
\cs_new_eq:NN  \dim_show:N \kernel_register_show:N
\cs_generate_variant:Nn \dim_show:N { c }
```

(*End definition for* `\dim_show:N` *and* `\dim_show:c`*. These functions are documented on page* **??**.)

`\dim_show:n`

Diagnostics.

```
\cs_new_protected:Npn \dim_show:n #1
  { \tex_showthe:D \dim_eval:w #1 \dim_eval_end: }
```

(*End definition for* `\dim_show:n`*. This function is documented on page 76.*)

191.9 Constant dimensions

`\c_zero_dim`
`\c_max_dim`

The source for these depends on whether we are in package mode.

```
⟨*initex⟩
\dim_new:N \c_zero_dim
\dim_new:N \c_max_dim
\dim_set:Nn \c_max_dim { 16383.99999 pt }
⟨/initex⟩
⟨*package⟩
\cs_new_eq:NN \c_zero_dim \z@
\cs_new_eq:NN \c_max_dim  \maxdimen
⟨/package⟩
```

(*End definition for* `\c_zero_dim`*. This function is documented on page 76.*)

191.10 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
\l_tmpc_dim
\g_tmpa_dim
\g_tmpb_dim

We provide three local and two global scratch registers, maybe we need more or less.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \l_tmpc_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

(*End definition for* `\l_tmpa_dim`*,* `\l_tmpb_dim`*, and* `\l_tmpc_dim`*. These functions are documented on page 77.*)

191.11 Creating and initialising skip variables

\skip_new:N
\skip_new:c

Allocation of a new internal registers.

```
⟨*package⟩
\cs_new_protected:Npn \skip_new:N #1
  {
    \chk_if_free_cs:N #1
    \newskip #1
  }
⟨/package⟩
\cs_generate_variant:Nn \skip_new:N { c }
```

(*End definition for* `\skip_new:N` *and* `\skip_new:c`*. These functions are documented on page* **??**.)

\skip_const:Nn
\skip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \skip_const:Nn #1
  {
    \skip_new:N #1
    \skip_gset:Nn #1
  }
\cs_generate_variant:Nn \skip_const:Nn { c }
```

(*End definition for* `\skip_const:Nn` *and* `\skip_const:cn`*. These functions are documented on page* **??**.)

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\cs_generate_variant:Nn \skip_zero:N  { c }
\cs_generate_variant:Nn \skip_gzero:N { c }
```

(*End definition for* `\skip_zero:N` *and* `\skip_zero:c`*. These functions are documented on page* **??**.)

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \skip_zero_new:N  #1
  { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\cs_new_protected:Npn \skip_gzero_new:N #1
  { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
\cs_generate_variant:Nn \skip_zero_new:N  { c }
\cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(*End definition for* `\skip_zero_new:N` *and others. These functions are documented on page* **??**.)

\skip_if_exist_p:N
\skip_if_exist_p:c
\skip_if_exist:NTF
\skip_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\cs_new_eq:NN \skip_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \skip_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \skip_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \skip_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \skip_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \skip_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \skip_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \skip_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\skip_if_exist:N` *and* `\skip_if_exist:c`*. These functions are documented on page* **??**.)

191.12 Setting skip variables

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Much the same as for dimensions.

```
\cs_new_protected:Npn \skip_set:Nn #1#2
  { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
\cs_generate_variant:Nn \skip_set:Nn  { c }
\cs_generate_variant:Nn \skip_gset:Nn { c }
```

(*End definition for* `\skip_set:Nn` *and* `\skip_set:cn`*. These functions are documented on page* **??**.)

\skip_set_eq:NN
\skip_set_eq:cN
\skip_set_eq:Nc
\skip_set_eq:cc
\skip_gset_eq:NN
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \skip_set_eq:NN {      c }
\cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \skip_gset_eq:NN {      c }
\cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
```

(*End definition for* `\skip_set_eq:NN` *and others. These functions are documented on page* **??**.)

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Using by here deals with the (incorrect) case \skip123.

```
\cs_new_protected:Npn \skip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\cs_generate_variant:Nn \skip_add:Nn  { c }
\cs_generate_variant:Nn \skip_gadd:Nn { c }
\cs_new_protected:Npn \skip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
\cs_generate_variant:Nn \skip_sub:Nn  { c }
\cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(*End definition for* `\skip_add:Nn` *and* `\skip_add:cn`*. These functions are documented on page* **??**.)

191.13 Skip expression conditionals

\skip_if_eq_p:nn
\skip_if_eq:nnTF

Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
\prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if_int_compare:w
      \pdftex_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
      = \c_zero
        \prg_return_true:
    \else:
        \prg_return_false:
    \fi:
  }
```

(*End definition for* \skip_if_eq:nn*. These functions are documented on page 78.*)

\skip_if_finite_p:n
\skip_if_finite:nTF
\skip_if_finite_aux:wwNw

With ε-TEX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```
\cs_set_protected:Npn \cs_tmp:w #1
  {
    \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
      {
        \exp_after:wN \skip_if_finite_aux:wwNw
        \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
        #1 ; \prg_return_true: \q_stop
      }
    \cs_new:Npn \skip_if_finite_aux:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
  }
\exp_args:No \cs_tmp:w { \tl_to_str:n { fil } }
```

(*End definition for* \skip_if_finite:n*. These functions are documented on page 79.*)

\skip_if_infinite_glue_p:n
\skip_if_infinite_glue:nTF

Reverse of \skip_if_finite:nTF.

```
\prg_new_conditional:Npnn \skip_if_infinite_glue:n #1 { p , T , F , TF }
  { \skip_if_finite:nTF {#1} \prg_return_false: \prg_return_true: }
```

(*End definition for* \skip_if_infinite_glue:n*. These functions are documented on page 78.*)

191.14 Using skip expressions and variables

\skip_eval:n

Evaluating a skip expression expandably.

```
\cs_new:Npn \skip_eval:n #1
  { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }
```

(*End definition for* \skip_eval:n*. This function is documented on page 79.*)

\skip_use:N
\skip_use:c

Accessing a ⟨*skip*⟩.

```
\cs_new_eq:NN \skip_use:N \tex_the:D
\cs_generate_variant:Nn \skip_use:N { c }
```

(*End definition for* `\skip_use:N` *and* `\skip_use:c`*. These functions are documented on page* **??**.)

191.15 Inserting skips into the output

`\skip_horizontal:N`
`\skip_horizontal:c`
`\skip_horizontal:n`
`\skip_vertical:N`
`\skip_vertical:c`
`\skip_vertical:n`

Inserting skips.

```
\cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
\cs_new:Npn \skip_horizontal:n #1
  { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
\cs_new:Npn \skip_vertical:n #1
  { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
\cs_generate_variant:Nn \skip_horizontal:N { c }
\cs_generate_variant:Nn \skip_vertical:N { c }
```

(*End definition for* `\skip_horizontal:N`*,* `\skip_horizontal:c`*, and* `\skip_horizontal:n`*. These functions are documented on page* **??**.)

191.16 Viewing skip variables

`\skip_show:N`
`\skip_show:c`

Diagnostics.

```
\cs_new_eq:NN  \skip_show:N \kernel_register_show:N
\cs_generate_variant:Nn \skip_show:N { c }
```

(*End definition for* `\skip_show:N` *and* `\skip_show:c`*. These functions are documented on page* **??**.)

`\skip_show:n`

Diagnostics.

```
\cs_new_protected:Npn \skip_show:n #1
  { \tex_showthe:D \etex_glueexpr:D #1 \scan_stop: }
```

(*End definition for* `\skip_show:n`*. This function is documented on page 79.*)

191.17 Constant skips

`\c_zero_skip`
`\c_max_skip`

Skips with no rubber component are just dimensions

```
\cs_new_eq:NN \c_zero_skip \c_zero_dim
\cs_new_eq:NN \c_max_skip  \c_max_dim
```

(*End definition for* `\c_zero_skip`*. This function is documented on page 79.*)

191.18 Scratch skips

`\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.

```
\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
```

(*End definition for* `\l_tmpa_skip`*,* `\l_tmpb_skip`*, and* `\l_tmpc_skip`*. These functions are documented on page 80.*)

191.19 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

And then we add muskips.

```
⟨*package⟩
\cs_new_protected:Npn \muskip_new:N #1
  {
    \chk_if_free_cs:N #1
    \newmuskip #1
  }
⟨/package⟩
\cs_generate_variant:Nn \muskip_new:N { c }
```

(*End definition for* \muskip_new:N *and* \muskip_new:c*. These functions are documented on page* **??**.)

\muskip_const:Nn
\muskip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \muskip_const:Nn #1
  {
    \muskip_new:N #1
    \muskip_gset:Nn #1
  }
\cs_generate_variant:Nn \muskip_const:Nn { c }
```

(*End definition for* \muskip_const:Nn *and* \muskip_const:cn*. These functions are documented on page* **??**.)

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \muskip_zero:N #1
  { #1 \c_zero_muskip }
\cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
\cs_generate_variant:Nn \muskip_zero:N  { c }
\cs_generate_variant:Nn \muskip_gzero:N { c }
```

(*End definition for* \muskip_zero:N *and* \muskip_zero:c*. These functions are documented on page* **??**.)

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \muskip_zero_new:N  #1
  { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\cs_new_protected:Npn \muskip_gzero_new:N #1
  { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
\cs_generate_variant:Nn \muskip_zero_new:N  { c }
\cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(*End definition for* \muskip_zero_new:N *and others. These functions are documented on page* **??**.)

\muskip_if_exist_p:N
\muskip_if_exist_p:c
\muskip_if_exist:N*TF*
\muskip_if_exist:c*TF*

Copies of the cs functions defined in l3basics.

```
\cs_new_eq:NN \muskip_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \muskip_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \muskip_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \muskip_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \muskip_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \muskip_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \muskip_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \muskip_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* \muskip_if_exist:N *and* \muskip_if_exist:c*. These functions are documented on page* **??**.)

191.20 Setting muskip variables

\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn

This should be pretty familiar.

```
\cs_new_protected:Npn \muskip_set:Nn #1#2
  { #1 ~ \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
\cs_generate_variant:Nn \muskip_set:Nn  { c }
\cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(*End definition for* \muskip_set:Nn *and* \muskip_set:cn*. These functions are documented on page* **??**.)

\muskip_set_eq:NN
\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \muskip_set_eq:NN {      c }
\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \muskip_gset_eq:NN {      c }
\cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
```

(*End definition for* \muskip_set_eq:NN *and others. These functions are documented on page* **??**.)

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Using by here deals with the (incorrect) case \muskip123.

```
\cs_new_protected:Npn \muskip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
\cs_generate_variant:Nn \muskip_add:Nn  { c }
\cs_generate_variant:Nn \muskip_gadd:Nn { c }
\cs_new_protected:Npn \muskip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
\cs_generate_variant:Nn \muskip_sub:Nn  { c }
\cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(*End definition for* \muskip_add:Nn *and* \muskip_add:cn*. These functions are documented on page* **??**.)

191.21 Using muskip expressions and variables

\muskip_eval:n

Evaluating a muskip expression expandably.

```
\cs_new:Npn \muskip_eval:n #1
  { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(*End definition for* \muskip_eval:n*. This function is documented on page 81.*)

\muskip_use:N
\muskip_use:c

Accessing a ⟨*muskip*⟩.

```
\cs_new_eq:NN \muskip_use:N \tex_the:D
\cs_generate_variant:Nn \muskip_use:N { c }
```

(*End definition for* \muskip_use:N *and* \muskip_use:c*. These functions are documented on page* **??**.)

191.22 Viewing muskip variables

\muskip_show:N
\muskip_show:c

Diagnostics.

```
\cs_new_eq:NN  \muskip_show:N \kernel_register_show:N
\cs_generate_variant:Nn \muskip_show:N { c }
```

(*End definition for* `\muskip_show:N` *and* `\muskip_show:c`*. These functions are documented on page* **??**.)

\muskip_show:n

Diagnostics.

```
\cs_new_protected:Npn \muskip_show:n #1
  { \tex_showthe:D \etex_muexpr:D #1 \scan_stop: }
```

(*End definition for* `\muskip_show:n`*. This function is documented on page 82.*)

191.23 Experimental skip functions

\skip_split_finite_else_action:nnNN

This macro is useful when performing error checking in certain circumstances. If the ⟨*skip*⟩ register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```
\cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
  {
    \skip_if_finite:nTF {#1}
      {
        #3 = \etex_gluestretch:D #1 \scan_stop:
        #4 = \etex_glueshrink:D  #1 \scan_stop:
      }
      {
        #3 = \c_zero_skip
        #4 = \c_zero_skip
        #2
      }
  }
```

(*End definition for* `\skip_split_finite_else_action:nnNN`*. This function is documented on page 83.*)

```
⟨/initex | package⟩
```

192 l3tl implementation

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

A token list variable is a TeX macro that holds tokens. By using the ε-TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including `#`, in this way.

192.1 Functions

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

Creating new token list variables is a case of checking for an existing definition and if free doing the definition.

```
\cs_new_protected:Npn \tl_new:N #1
  {
    \chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_tl
  }
\cs_generate_variant:Nn \tl_new:N { c }
```

(*End definition for* `\tl_new:N` *and* `\tl_new:c`*. These functions are documented on page* **??**.)

`\tl_const:Nn`
`\tl_const:Nx`
`\tl_const:cn`
`\tl_const:cx`

Constants are also easy to generate.

```
\cs_new_protected:Npn \tl_const:Nn #1#2
  {
    \chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
  }
\cs_new_protected:Npn \tl_const:Nx #1#2
  {
    \chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 {#2}
  }
\cs_generate_variant:Nn \tl_const:Nn { c }
\cs_generate_variant:Nn \tl_const:Nx { c }
```

(*End definition for* `\tl_const:Nn` *and others. These functions are documented on page* **??**.)

`\c_empty_tl`

Never full. We need to define that constant early for `\tl_new:N` to work properly.

```
\tl_const:Nn \c_empty_tl { }
```

(*End definition for* `\c_empty_tl`*. This variable is documented on page 95.*)

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

Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear:N  #1
  { \tl_set_eq:NN #1 \c_empty_tl }
\cs_new_protected:Npn \tl_gclear:N #1
  { \tl_gset_eq:NN #1 \c_empty_tl }
\cs_generate_variant:Nn \tl_clear:N  { c }
\cs_generate_variant:Nn \tl_gclear:N { c }
```

(*End definition for* `\tl_clear:N` *and* `\tl_clear:c`*. These functions are documented on page* **??**.)

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

Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear_new:N  #1
  { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
```

```
\cs_new_protected:Npn \tl_gclear_new:N #1
  { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
\cs_generate_variant:Nn \tl_clear_new:N  { c }
\cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(*End definition for* `\tl_clear_new:N` *and* `\tl_clear_new:c`*. These functions are documented on page* **??**.)

`\tl_set_eq:NN`
`\tl_set_eq:Nc`
`\tl_set_eq:cN`
`\tl_set_eq:cc`
`\tl_gset_eq:NN`
`\tl_gset_eq:Nc`
`\tl_gset_eq:cN`
`\tl_gset_eq:cc`

For setting token list variables equal to each other.

```
\cs_new_eq:NN \tl_set_eq:NN  \cs_set_eq:NN
\cs_new_eq:NN \tl_set_eq:cN  \cs_set_eq:cN
\cs_new_eq:NN \tl_set_eq:Nc  \cs_set_eq:Nc
\cs_new_eq:NN \tl_set_eq:cc  \cs_set_eq:cc
\cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc
```

(*End definition for* `\tl_set_eq:NN` *and others. These functions are documented on page* **??**.)

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

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \tl_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \tl_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \tl_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \tl_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \tl_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \tl_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \tl_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \tl_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\tl_if_exist:N` *and* `\tl_if_exist:c`*. These functions are documented on page* **??**.)

192.2 Adding to token list variables

`\tl_set:Nn`
`\tl_set:NV`
`\tl_set:Nv`
`\tl_set:No`
`\tl_set:Nf`
`\tl_set:Nx`
`\tl_set:cn`
`\tl_set:NV`
`\tl_set:Nv`
`\tl_set:co`
`\tl_set:cf`
`\tl_set:cx`
`\tl_gset:Nn`
`\tl_gset:NV`
`\tl_gset:Nv`
`\tl_gset:No`
`\tl_gset:Nf`
`\tl_gset:Nx`
`\tl_gset:cn`
`\tl_gset:NV`
`\tl_gset:Nv`
`\tl_gset:co`
`\tl_gset:cf`
`\tl_gset:cx`

By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by TEX more or less redundant. The `\tl_set:No` version is done "by hand" as it is used quite a lot.

```
\cs_new_protected:Npn \tl_set:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_set:Nx #1#2
  { \cs_set_nopar:Npx #1 {#2} }
\cs_new_protected:Npn \tl_gset:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_gset:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_gset:Nx #1#2
  { \cs_gset_nopar:Npx #1 {#2} }
\cs_generate_variant:Nn \tl_set:Nn  {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_set:Nx  { c }
```

```
\cs_generate_variant:Nn \tl_set:Nn  { c, co , cV , cv , cf }
\cs_generate_variant:Nn \tl_gset:Nn {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_gset:Nx { c }
\cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }
```

(*End definition for* `\tl_set:Nn` *and others. These functions are documented on page* **??**.)

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

Adding to the left is done directly to gain a little performance.

```
\cs_new_protected:Npn \tl_put_left:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:Nx #1#2
  { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nx #1#2
  { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
\cs_generate_variant:Nn \tl_put_left:Nn  { c }
\cs_generate_variant:Nn \tl_put_left:NV  { c }
\cs_generate_variant:Nn \tl_put_left:No  { c }
\cs_generate_variant:Nn \tl_put_left:Nx  { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:NV { c }
\cs_generate_variant:Nn \tl_gput_left:No { c }
\cs_generate_variant:Nn \tl_gput_left:Nx { c }
```

(*End definition for* `\tl_put_left:Nn` *and others. These functions are documented on page* **??**.)

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

The same on the right.

```
\cs_new_protected:Npn \tl_put_right:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_put_right:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\cs_new_protected:Npn \tl_put_right:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_put_right:Nx #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\cs_new_protected:Npn \tl_gput_right:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_gput_right:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\cs_new_protected:Npn \tl_gput_right:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_gput_right:Nx #1#2
```

```
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
\cs_generate_variant:Nn \tl_put_right:Nn  { c }
\cs_generate_variant:Nn \tl_put_right:NV  { c }
\cs_generate_variant:Nn \tl_put_right:No  { c }
\cs_generate_variant:Nn \tl_put_right:Nx  { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\cs_generate_variant:Nn \tl_gput_right:Nx { c }
```

(*End definition for* `\tl_put_right:Nn` *and others. These functions are documented on page* **??**.)

192.3 Reassigning token list category codes

`\c_tl_rescan_marker_tl`

The rescanning code needs a special token list containing the same character with two different category codes. This is set up here, while the detail is described below.

```
\group_begin:
  \tex_lccode:D ‘\A = ‘\@ \scan_stop:
  \tex_lccode:D ‘\B = ‘\@ \scan_stop:
  \tex_catcode:D ‘\A = 8 \scan_stop:
  \tex_catcode:D ‘\B = 3 \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \tl_const:Nn \c_tl_rescan_marker_tl { A B }
  }
```

(*End definition for* `\c_tl_rescan_marker_tl`. *This variable is documented on page* **??**.)

`\tl_set_rescan:Nnn`
`\tl_set_rescan:Nno`
`\tl_set_rescan:Nnx`
`\tl_set_rescan:cnn`
`\tl_set_rescan:cno`
`\tl_set_rescan:cnx`
`\tl_gset_rescan:Nnn`
`\tl_gset_rescan:Nno`
`\tl_gset_rescan:Nnx`
`\tl_gset_rescan:cnn`
`\tl_gset_rescan:cno`
`\tl_gset_rescan:cnx`
`\tl_rescan:nn`
`\tl_set_rescan_aux:NNnn`
`\tl_rescan_aux:w`

The idea here is to deal cleanly with the problem that `\scantokens` treats the argument as a file, and without the correct settings a TeX error occurs:

```
! File ended while scanning definition of ...
```

When expanding a token list this can be handled using `\exp_not:N` but this fails if the token list is not being expanded. So instead a delimited argument is used with an end marker which cannot appear within the token list which is scanned: two `@` symbols with different category codes. The rescanned token list cannot contain the end marker, because all `@` present in the token list are read with the same category code. As every character with charcode `\newlinechar` is replaced by the `\endlinechar`, and an extra `\endlinechar` is added at the end, we need to set both of those to -1, “unprintable”.

```
\cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
  { \tl_set_rescan_aux:NNnn \tl_set:Nn }
\cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
  { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
\cs_new_protected_nopar:Npn \tl_rescan:nn
  { \tl_set_rescan_aux:NNnn \prg_do_nothing: \use:n }
\cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4
  {
    \group_begin:
      \exp_args:No \etex_everyeof:D { \c_tl_rescan_marker_tl \exp_not:N }
```

```
      \tex_endlinechar:D \c_minus_one
      \tex_newlinechar:D \c_minus_one
      #3
      \use:x
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \exp_after:wN \tl_rescan_aux:w
              \exp_after:wN \prg_do_nothing:
              \etex_scantokens:D {#4}
            }
        }
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \tl_rescan_aux:w ##1
      \c_tl_rescan_marker_tl
      { \exp_not:N \exp_not:o { ##1 } }
  }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  {     Nno , Nnx }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  { c , cno , cnx }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn {     Nno , Nnx }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }
```

(*End definition for* `\tl_set_rescan:Nnn` *and others. These functions are documented on page* *87*.)

192.4 Reassigning token list character codes

`\tl_to_lowercase:n`
`\tl_to_uppercase:n`

Just some names for a few primitives.

```
\cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
\cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D
```

(*End definition for* `\tl_to_lowercase:n`. *This function is documented on page* *87*.)

192.5 Modifying token list variables

`\tl_replace_all:Nnn`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`
`\tl_replace_aux:NNNnn`
`\tl_replace_aux_ii:w`
`\tl_replace_all_aux:`
`\tl_replace_once_aux:`
`\tl_replace_once_aux_end:w`

All of the replace functions are based on `\tl_replace_aux:NNNnn`, whose arguments are: ⟨*function*⟩, `\tl_(g)set:Nx`, ⟨*tl var*⟩, ⟨*search tokens*⟩, ⟨*replacement tokens*⟩.

```
\cs_new_protected_nopar:Npn \tl_replace_once:Nnn
  { \tl_replace_aux:NNNnn \tl_replace_once_aux: \tl_set:Nx }
\cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
  { \tl_replace_aux:NNNnn \tl_replace_once_aux: \tl_gset:Nx }
\cs_new_protected_nopar:Npn \tl_replace_all:Nnn
  { \tl_replace_aux:NNNnn \tl_replace_all_aux: \tl_set:Nx }
\cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
  { \tl_replace_aux:NNNnn \tl_replace_all_aux: \tl_gset:Nx }
\cs_generate_variant:Nn \tl_replace_once:Nnn  { c }
\cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
\cs_generate_variant:Nn \tl_replace_all:Nnn   { c }
\cs_generate_variant:Nn \tl_greplace_all:Nnn  { c }
```

The idea is easier to understand by considering the case of `\tl_replace_all:Nnn`. The replacement happens within an `x`-type expansion. We use an auxiliary function `\tl_tmp:w`, which essentially replaces the next ⟨*search tokens*⟩ by ⟨*replacement tokens*⟩. To avoid runaway arguments, we expand something like `\tl_tmp:w` ⟨*token list*⟩ `\q_mark` ⟨*search tokens*⟩ `\q_stop`, repeating until the end. How do we detect that we have reached the last occurrence of ⟨*search tokens*⟩? The last replacement is characterized by the fact that the argument of `\tl_tmp:w` contains `\q_mark`. In the code below, `\tl_replace_aux_ii:w` takes an argument delimited by `\q_mark`, and removes the following token. Before we reach the end, this gobbles `\q_mark \use_none_delimit_by_q_stop:w` which appear in the definition of `\tl_tmp:w`, and leaves the ⟨*replacement tokens*⟩, passed to `\exp_not:n`, to be included in the `x`-expanding definition. At the end, the first `\q_mark` is within the argument of `\tl_tmp:w`, and `\tl_replace_aux_ii:w` gobbles the second `\q_mark` as well, leaving `\use_none_delimit_by_q_stop:w`, which ends the recursion cleanly.

```
\cs_new_protected:Npn \tl_replace_aux:NNNnn #1#2#3#4#5
  {
    \tl_if_empty:nTF {#4}
      {
        \msg_kernel_error:nnx { tl } { empty-search-pattern }
          { \tl_to_str:n {#5} }
      }
      {
        \group_align_safe_begin:
        \cs_set:Npx \tl_tmp:w ##1##2 #4
          {
            ##2
            \exp_not:N \q_mark
            \exp_not:N \use_none_delimit_by_q_stop:w
            \exp_not:n { \exp_not:n {#5} }
            ##1
          }
        \group_align_safe_end:
        #2 #3
          {
            \exp_after:wN #1
            #3 \q_mark #4 \q_stop
          }
      }
  }
\cs_new:Npn \tl_replace_aux_ii:w #1 \q_mark #2 { \exp_not:o {#1} }
```

The first argument of `\tl_tmp:w` is responsible for repeating the replacement in the case of `replace_all`, and stopping it early for `replace_once`. Note also that we build `\tl_tmp:w` within an `x`-expansion so that the ⟨*replacement tokens*⟩ can contain `#`. The second `\exp_not:n` ensures that the ⟨*replacement tokens*⟩ are not expanded by `\tl_(g)set:Nx`.

Now on to the difference between "once" and "all". The `\prg_do_nothing:` and accompanying `o`-expansion ensure that we don't lose braces in case the tokens between two occurrences of the ⟨*search tokens*⟩ form a brace group.

```
\cs_new:Npn \tl_replace_all_aux:
  {
    \exp_after:wN \tl_replace_aux_ii:w
    \tl_tmp:w \tl_replace_all_aux: \prg_do_nothing:
  }
\cs_new_nopar:Npn \tl_replace_once_aux:
  {
    \exp_after:wN \tl_replace_aux_ii:w
    \tl_tmp:w { \tl_replace_once_aux_end:w \prg_do_nothing: } \prg_do_nothing:
  }
\cs_new:Npn \tl_replace_once_aux_end:w #1 \q_mark #2 \q_stop
  { \exp_not:o {#1} }
```

(*End definition for* `\tl_replace_all:Nnn` *and* `\tl_replace_all:cnn`*. These functions are documented on page* **??**.)

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

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_once:Nn #1#2
  { \tl_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_once:Nn #1#2
  { \tl_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_once:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(*End definition for* `\tl_remove_once:Nn` *and* `\tl_remove_once:cn`*. These functions are documented on page* **??**.)

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

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_all:Nn #1#2
  { \tl_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_all:Nn #1#2
  { \tl_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_all:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

192.6 Token list conditionals

`\tl_if_blank_p:n`
`\tl_if_blank_p:V`
`\tl_if_blank_p:o`
`\tl_if_blank:nTF`
`\tl_if_blank:VTF`
`\tl_if_blank:oTF`
`\tl_if_blank_p_aux:NNw`

TeX skips spaces when reading a non-delimited arguments. Thus, a ⟨*token list*⟩ is blank if and only if `\use_none:n` ⟨*token list*⟩ `?` is empty. For performance reasons, we hard-code the emptyness test done in `\tl_if_empty:n(TF)`: convert to harmless characters with `\tl_to_str:n`, and then use `\if_meaning:w \q_nil ... \q_nil`. Note that converting to a string is done after reading the delimited argument for `\use_none:n`. The similar construction `\exp_after:wN \use_none:n \tl_to_str:n {`⟨*token list*⟩`} ?` would fail if the token list contains the control sequence `\`, while `\escapechar` is a space or is unprintable.

```
\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
  { \tl_if_empty_return:o { \use_none:n #1 ? } }
\cs_generate_variant:Nn \tl_if_blank_p:n { V }
\cs_generate_variant:Nn \tl_if_blank:nT  { V }
\cs_generate_variant:Nn \tl_if_blank:nF  { V }
```

```
\cs_generate_variant:Nn \tl_if_blank:nTF { V }
\cs_generate_variant:Nn \tl_if_blank_p:n { o }
\cs_generate_variant:Nn \tl_if_blank:nT  { o }
\cs_generate_variant:Nn \tl_if_blank:nF  { o }
\cs_generate_variant:Nn \tl_if_blank:nTF { o }
```

(*End definition for* `\tl_remove_all:Nn` *and* `\tl_remove_all:cn`*. These functions are documented on page* **??**.)

`\tl_if_empty_p:N`
`\tl_if_empty_p:c`
`\tl_if_empty:N`*TF*
`\tl_if_empty:c`*TF*

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

```
\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:N { c }
\cs_generate_variant:Nn \tl_if_empty:NT  { c }
\cs_generate_variant:Nn \tl_if_empty:NF  { c }
\cs_generate_variant:Nn \tl_if_empty:NTF { c }
```

(*End definition for* `\tl_if_empty:N` *and* `\tl_if_empty:c`*. These functions are documented on page* **??**.)

`\tl_if_empty_p:n`
`\tl_if_empty_p:V`
`\tl_if_empty:n`*TF*
`\tl_if_empty:V`*TF*

It would be tempting to just use `\if_meaning:w \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:w`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the end starts executing… 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.

```
\prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:n { V }
\cs_generate_variant:Nn \tl_if_empty:nTF { V }
\cs_generate_variant:Nn \tl_if_empty:nT  { V }
\cs_generate_variant:Nn \tl_if_empty:nF  { V }
```

(*End definition for* `\tl_if_empty:n` *and* `\tl_if_empty:V`*. These functions are documented on page* **??**.)

`\tl_if_empty_p:o`
`\tl_if_empty:o`*TF*
`\tl_if_empty_return:o`

The auxiliary function `\tl_if_empty_return:o` is for use in conditionals on token lists, which mostly reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:n(TF)`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because `\tl_to_str:n` expands tokens that follow until reading a catcode 1 (begin-group) token.

```
\cs_new:Npn \tl_if_empty_return:o #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \tl_to_str:n \exp_after:wN {#1} \q_nil
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
  { \tl_if_empty_return:o {#1} }
```

(*End definition for* `\tl_if_empty:o`. *These functions are documented on page* **??**.)

`\tl_if_eq_p:NN`
`\tl_if_eq_p:Nc`
`\tl_if_eq_p:cN`
`\tl_if_eq_p:cc`
`\tl_if_eq:NNTF`
`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

Returns `\c_true_bool` if and only if the two token list variables are equal.

```
\prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1 #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNT  { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNF  { Nc , c , cc }
```

(*End definition for* `\tl_if_eq:NN` *and others. These functions are documented on page* **??**.)

`\tl_if_eq:nnTF`
`\l_tl_internal_a_tl`
`\l_tl_internal_b_tl`

A simple store and compare routine.

```
\prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F ,  TF }
  {
    \group_begin:
      \tl_set:Nn \l_tl_internal_a_tl {#1}
      \tl_set:Nn \l_tl_internal_b_tl {#2}
      \if_meaning:w \l_tl_internal_a_tl \l_tl_internal_b_tl
        \group_end:
        \prg_return_true:
      \else:
        \group_end:
        \prg_return_false:
      \fi:
  }
\tl_new:N \l_tl_internal_a_tl
\tl_new:N \l_tl_internal_b_tl
```

(*End definition for* `\tl_if_eq:nn`. *This function is documented on page* **??**.)

\tl_if_in:NnTF
\tl_if_in:cnTF

See \tl_if_in:nn(TF) for further comments. Here we simply expand the token list variable and pass it to \tl_if_in:nn(TF).

```
\cs_new_protected_nopar:Npn \tl_if_in:NnT  { \exp_args:No \tl_if_in:nnT  }
\cs_new_protected_nopar:Npn \tl_if_in:NnF  { \exp_args:No \tl_if_in:nnF  }
\cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
\cs_generate_variant:Nn \tl_if_in:NnT { c }
\cs_generate_variant:Nn \tl_if_in:NnF  { c }
\cs_generate_variant:Nn \tl_if_in:NnTF { c }
```

(*End definition for* \tl_if_in:NnTF *and* \tl_if_in:cnTF*. These functions are documented on page* ??.)

\tl_if_in:nnTF
\tl_if_in:VnTF
\tl_if_in:onTF
\tl_if_in:noTF

Once more, the test relies on \tl_to_str:n for robustness. The function \tl_tmp:w removes tokens until the first occurrence of #2. If this does not appear in #1, then the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See \tl_if_empty:n(TF) for details on the emptyness test.

Special care is needed to treat correctly cases like \tl_if_in:nnTF {a state}{states}, where #1#2 contains #2 before the end. To cater for this case, we insert {}{} between the two token lists. This marker may not appear in #2 because of TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments.

```
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T  , F , TF }
  {
    \cs_set:Npn \tl_tmp:w ##1 #2 { }
    \tl_if_empty:oTF { \tl_tmp:w #1 {} {} #2 }
      { \prg_return_false: } { \prg_return_true: }
  }
\cs_generate_variant:Nn \tl_if_in:nnT  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnF  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }
```

(*End definition for* \tl_if_in:nnTF *and others. These functions are documented on page* ??.)

192.7 Mapping to token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
\tl_map_function_aux:Nn

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

```
\cs_new:Npn \tl_map_function:nN #1#2
  {
    \tl_map_function_aux:Nn #2 #1
      \q_recursion_tail
    \prg_break_point:n { }
  }
\cs_new_nopar:Npn \tl_map_function:NN
  { \exp_args:No \tl_map_function:nN }
\cs_new:Npn \tl_map_function_aux:Nn #1#2
  {
    \quark_if_recursion_tail_break:n {#2}
    #1 {#2} \tl_map_function_aux:Nn #1
  }
```

```
\cs_generate_variant:Nn \tl_map_function:NN { c }
```

(*End definition for* `\tl_map_function:nN`*. This function is documented on page* **??**.)

`\tl_map_inline:nn`
`\tl_map_inline:Nn`
`\tl_map_inline:cn`

The inline functions are straight forward by now. We use a little trick with the counter `\g_prg_map_int` to make them nestable. We can also make use of `\tl_map_function_aux:Nn` from before.

```
\cs_new_protected:Npn \tl_map_inline:nn #1#2
  {
    \int_gincr:N \g_prg_map_int
    \cs_gset:cpn { tl_map_inline_ \int_use:N \g_prg_map_int :n }
      ##1 {#2}
    \exp_args:Nc \tl_map_function_aux:Nn
      { tl_map_inline_ \int_use:N \g_prg_map_int :n }
      #1 \q_recursion_tail
    \prg_break_point:n { \int_gdecr:N \g_prg_map_int }
  }
\cs_new_protected:Npn \tl_map_inline:Nn
  { \exp_args:No \tl_map_inline:nn }
\cs_generate_variant:Nn \tl_map_inline:Nn { c }
```

(*End definition for* `\tl_map_inline:nn`*. This function is documented on page* **??**.)

`\tl_map_variable:nNn`
`\tl_map_variable:NNn`
`\tl_map_variable:cNn`
`\tl_map_variable_aux:Nnn`

`\tl_map_variable:nNn` ⟨*token list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
\cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
  {
    \tl_map_variable_aux:Nnn #2 {#3} #1
      \q_recursion_tail
    \prg_break_point:n { }
  }
\cs_new_protected_nopar:Npn \tl_map_variable:NNn
  { \exp_args:No \tl_map_variable:nNn }
\cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3
  {
    \tl_set:Nn #1 {#3}
    \quark_if_recursion_tail_break:N #1
    \use:n {#2}
    \tl_map_variable_aux:Nnn #1 {#2}
  }
\cs_generate_variant:Nn \tl_map_variable:NNn { c }
```

(*End definition for* `\tl_map_variable:nNn`*. This function is documented on page* **??**.)

`\tl_map_break:`
`\tl_map_break:n`

The break statements are simply copies.

```
\cs_new_eq:NN \tl_map_break:  \prg_map_break:
\cs_new_eq:NN \tl_map_break:n \prg_map_break:n
```

(*End definition for* `\tl_map_break:`*. This function is documented on page* **??**.)

192.8 Using token lists

\tl_to_str:n

Another name for a primitive.

```
\cs_new_eq:NN \tl_to_str:n \etex_detokenize:D
```

(*End definition for* `\tl_to_str:n`*. This function is documented on page* *90**.*)

\tl_to_str:N
\tl_to_str:c

These functions return the replacement text of a token list as a string.

```
\cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
\cs_generate_variant:Nn \tl_to_str:N { c }
```

(*End definition for* `\tl_to_str:N` *and* `\tl_to_str:c`*. These functions are documented on page* **??**.)

\tl_use:N
\tl_use:c

Token lists which are simply not defined will give a clear TEX error here. No such luck for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```
\cs_new:Npn \tl_use:N #1
  {
    \tl_if_exist:NTF #1 {#1}
      { \msg_expandable_kernel_error:nnn { kernel } { bad-var } {#1} }
  }
\cs_generate_variant:Nn \tl_use:N { c }
```

(*End definition for* `\tl_use:N` *and* `\tl_use:c`*. These functions are documented on page* **??**.)

192.9 Working with the contents of token lists

\tl_length:n
\tl_length:V
\tl_length:o
\tl_length:N
\tl_length:c
\tl_length_aux:n

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

```
\cs_new:Npn \tl_length:n #1
  {
    \int_eval:n
      { 0 \tl_map_function:nN {#1} \tl_length_aux:n }
  }
\cs_new:Npn \tl_length:N #1
  {
    \int_eval:n
      { 0 \tl_map_function:NN #1 \tl_length_aux:n }
  }
\cs_new:Npn \tl_length_aux:n #1 { + \c_one }
\cs_generate_variant:Nn \tl_length:n { V , o }
\cs_generate_variant:Nn \tl_length:N { c }
```

(*End definition for* `\tl_length:n`*,* `\tl_length:V`*, and* `\tl_length:o`*. These functions are documented on page* **??**.)

\tl_reverse_items:n
\tl_reverse_items_aux:nwNwn
\tl_reverse_items_aux:wn

Reversal of a token list is done by taking one item at a time and putting it after `\q_stop`.

```
\cs_new:Npn \tl_reverse_items:n #1
  {
    \tl_reverse_items_aux:nwNwn #1 ?
      \q_mark \tl_reverse_items_aux:nwNwn
```

```
      \q_mark \tl_reverse_items_aux:wn
      \q_stop { }
  }
\cs_new:Npn \tl_reverse_items_aux:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
  {
    #3 #2
      \q_mark \tl_reverse_items_aux:nwNwn
      \q_mark \tl_reverse_items_aux:wn
      \q_stop { {#1} #5 }
  }
\cs_new:Npn \tl_reverse_items_aux:wn #1 \q_stop #2
  { \exp_not:o { \use_none:nn #2 } }
```

(*End definition for* `\tl_reverse_items:n`*. This function is documented on page 91.*)

`\tl_trim_spaces:n`
`\tl_trim_spaces:N`
`\tl_trim_spaces:c`
`\tl_gtrim_spaces:N`
`\tl_gtrim_spaces:c`
`\tl_trim_spaces_aux_i:w`
`_trim_spaces_aux_ii:w\tl_trim_spaces_aux_iii:w`
`\tl_trim_spaces_aux_iv:w`

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `\tl_tmp:w`, which then receives a single space as its argument: `#1` is ␣. Removing leading spaces is done with `\tl_trim_spaces_aux_i:w`, which loops until `\q_mark`␣ matches the end of the token list: then `##1` is the token list and `##3` is `\tl_trim_spaces_aux_ii:w`. This hands the relevant tokens to the loop `\tl_trim_spaces_aux_iii:w`, responsible for trimming trailing spaces. The end is reached when ␣ `\q_nil` matches the one present in the definition of `\tl_trim_spacs:n`. Then `\tl_trim_spaces_aux_iv:w` puts the token list into a group, as the argument of the initial `\unexpanded`. The `\unexpanded` here is used so that space trimming will behave correctly within an `x`-type expansion.

Some of the auxiliaries used in this code are also used in the l3clist module. Change with care.

```
\cs_set:Npn \tl_tmp:w #1
  {
    \cs_new:Npn \tl_trim_spaces:n ##1
      {
        \etex_unexpanded:D
          \tl_trim_spaces_aux_i:w
          \q_mark
          ##1
          \q_nil
          \q_mark #1 { }
          \q_mark \tl_trim_spaces_aux_ii:w
          \tl_trim_spaces_aux_iii:w
          #1 \q_nil
          \tl_trim_spaces_aux_iv:w
          \q_stop
      }
    \cs_new:Npn \tl_trim_spaces_aux_i:w ##1 \q_mark #1 ##2 \q_mark ##3
      {
        ##3
        \tl_trim_spaces_aux_i:w
        \q_mark
        ##2
```

```
          \q_mark #1 {##1}
        }
      \cs_new:Npn \tl_trim_spaces_aux_ii:w ##1 \q_mark \q_mark ##2
        {
          \tl_trim_spaces_aux_iii:w
          ##2
        }
      \cs_new:Npn \tl_trim_spaces_aux_iii:w ##1 #1 \q_nil ##2
        {
          ##2
          ##1 \q_nil
          \tl_trim_spaces_aux_iii:w
        }
      \cs_new:Npn \tl_trim_spaces_aux_iv:w ##1 \q_nil ##2 \q_stop
        { \exp_after:wN { \use_none:n ##1 } }
  }
\tl_tmp:w { ~ }
\cs_new_protected:Npn \tl_trim_spaces:N #1
  { \tl_set:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
\cs_new_protected:Npn \tl_gtrim_spaces:N #1
  { \tl_gset:Nx #1 { \exp_after:wN \tl_trim_spaces:n \exp_after:wN {#1} } }
\cs_generate_variant:Nn \tl_trim_spaces:N  { c }
\cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

(*End definition for* `\tl_trim_spaces:n`*. This function is documented on page* **??**.)

192.10 The first token from a token list

`\tl_head:N` `\tl_head:n` `\tl_head:V` `\tl_head:v` `\tl_head:f` `\tl_head:w` `\tl_tail:N` `\tl_tail:n` `\tl_tail:V` `\tl_tail:v` `\tl_tail:f` `\tl_tail:w`

These functions pick up either the head or the tail of a list. The empty brace groups in `\tl_head:n` and `\tl_tail:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive.

```
\cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
\cs_new:Npn \tl_tail:w #1#2 \q_stop {#2}
\cs_new:Npn \tl_head:n #1
  { \etex_unexpanded:D \exp_after:wN { \tl_head:w #1 { } \q_stop } }
\cs_new:Npn \tl_tail:n #1
  { \etex_unexpanded:D \tl_tail_aux:w #1 \q_mark { } \q_mark \q_stop }
\cs_new:Npn \tl_tail_aux:w #1 #2 \q_mark #3 \q_stop { {#2} }
\cs_new_nopar:Npn \tl_head:N { \exp_args:No \tl_head:n }
\cs_generate_variant:Nn \tl_head:n { V , v , f }
\cs_new_nopar:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
\cs_generate_variant:Nn \tl_tail:n { V , v , f }
```

(*End definition for* `\tl_head:N` *and others. These functions are documented on page 93.*)

`\str_head:n` `\str_tail:n` `\str_head_aux:w` `\str_tail_aux:w`

After `\tl_to_str:n`, we have a list of character tokens, all with category code 12, except the space, which has category code 10. Directly using `\tl_head:w` would thus lose leading spaces. Instead, we take an argument delimited by an explicit space, and then only use `\tl_head:w`. If the string started with a space, then the argument of `\str_head_aux:w` is empty, and the function correctly returns a space character. Otherwise, it returns the first token of #1, which is the first token of the string. If the string is empty, we return an empty result.

To remove the first character of `\tl_to_str:n {#1}`, we test it using `\if_charcode:w \scan_stop:`, always false for characters. If the argument was non-empty, then `\str_tail_aux:w` returns everything until the first X (with category code letter, no risk of confusing with the user input). If the argument was empty, the first X is taken by `\if_charcode:w`, and nothing is returned. We use X as a ⟨*marker*⟩, rather than a quark because the test `\if_charcode:w \scan_stop:` ⟨*marker*⟩ has to be false.

```
\cs_new:Npn \str_head:n #1
  {
    \exp_after:wN \str_head_aux:w
    \tl_to_str:n {#1}
    { { } } ~ \q_stop
  }
\cs_new:Npn \str_head_aux:w #1 ~ %
  { \tl_head:w #1 { ~ } }
\cs_new:Npn \str_tail:n #1
  {
    \exp_after:wN \str_tail_aux:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \q_stop
  }
\cs_new:Npn \str_tail_aux:w #1 X #2 \q_stop { \fi: #1 }
```

(*End definition for `\str_head:n` and `\str_tail:n`. These functions are documented on page 93.*)

`\tl_if_head_eq_meaning_p:nN`
`\tl_if_head_eq_meaning:nNTF`
`\tl_if_head_eq_charcode_p:nN`
`\tl_if_head_eq_charcode:nNTF`
`\tl_if_head_eq_charcode_p:fN`
`\tl_if_head_eq_charcode:fNTF`
`\tl_if_head_eq_catcode_p:nN`
`\tl_if_head_eq_catcode:nNTF`

Accessing the first token of a token list is tricky in two cases: when it has category code 1 (begin-group token), or when it is an explicit space, with category code 10 and character code 32.

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, an empty #1 argument yields `\q_nil`, otherwise the first token of the token list.

```
\if_charcode:w
    \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
    \exp_not:N #2
```

The special cases are detected using `\tl_if_head_N_type:n` (the extra ? takes care of empty arguments). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character).

```
\prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
  {
    \if_charcode:w
        \exp_not:N #2
        \tl_if_head_N_type:nTF { #1 ? }
          { \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop }
          { \str_head:n {#1} }
      \prg_return_true:
```

```
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT  { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF  { f }
```

For \tl_if_head_eq_catcode:nN, again we detect special cases with a \tl_if_head_-N_type. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either \c_group_begin_token or \c_-space_token.

```
\prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
  {
    \if_catcode:w
        \exp_not:N #2
        \tl_if_head_N_type:nTF { #1 ? }
          { \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop }
          {
            \tl_if_head_group:nTF {#1}
              { \c_group_begin_token }
              { \c_space_token }
          }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

For \tl_if_head_eq_meaning:nN, again, detect special cases. In the normal case, use \tl_head:w, with no \exp_not:N this time, since \if_meaning:w causes no expansion. In the special cases, we know that the first token is a character, hence \if_charcode:w and \if_catcode:w together are enough. We combine them in some order, hopefully faster than the reverse.

```
\prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
  {
    \tl_if_head_N_type:nTF { #1 ? }
      { \tl_if_head_eq_meaning_aux_normal:nN }
      { \tl_if_head_eq_meaning_aux_special:nN }
    {#1} #2
  }
\cs_new:Npn \tl_if_head_eq_meaning_aux_normal:nN #1 #2
  {
    \exp_after:wN \if_meaning:w \tl_head:w #1 \q_nil \q_stop #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

```
\cs_new:Npn \tl_if_head_eq_meaning_aux_special:nN #1 #2
  {
    \if_charcode:w \str_head:n {#1} \exp_not:N #2
      \exp_after:wN \use:n
    \else:
      \prg_return_false:
      \exp_after:wN \use_none:n
    \fi:
    {
      \if_catcode:w \exp_not:N #2
                    \tl_if_head_group:nTF {#1}
                      { \c_group_begin_token }
                      { \c_space_token }
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
  }
```

(*End definition for* `\tl_if_head_eq_meaning:nN`. *These functions are documented on page 93.*)

`\tl_if_head_N_type_p:n`
`\tl_if_head_N_type:nTF`

The first token of a token list can be either an N-type argument, a begin-group token (catcode 1), or an explicit space token (catcode 10 and charcode 32). These two cases are characterized by the fact that `\use:n` removes some tokens from `#1`, hence changing its string representation (no token can have an empty string representation). The extra brace group covers the case of an empty argument, whose head is not "normal".

```
\prg_new_conditional:Npnn \tl_if_head_N_type:n #1 { p , T , F , TF }
  {
    \str_if_eq_return:xx
      { \exp_not:o { \use:n #1 { } } }
      { \exp_not:n { #1 { } } }
  }
```

(*End definition for* `\tl_if_head_N_type:n`. *These functions are documented on page 94.*)

`\tl_if_head_group_p:n`
`\tl_if_head_group:nTF`

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument.[6]

```
\prg_new_conditional:Npnn \tl_if_head_group:n #1 { p , T , F , TF }
  {
    \if_catcode:w *
        \exp_after:wN \use_none:n
          \exp_after:wN {
            \exp_after:wN {
              \token_to_str:N #1 ?
            }
          }
```

[6]Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we need all brace "tricks" to happen in one step of expansion, keeping the token list brace balanced at all times.

```
          *
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
  }
```

(*End definition for* `\tl_if_head_group:n`. *These functions are documented on page 94.*)

`\tl_if_head_space_p:n`
`\tl_if_head_space:nTF`
`\tl_if_head_space_aux:w`

If the first token of the token list is an explicit space, i.e., a character token with character code 32 and category code 10, then this test will be ⟨*true*⟩. It is ⟨*false*⟩ if the token list is empty, if the first token is an implicit space token, such as `\c_space_token`, or any token other than an explicit space. The slightly convoluted approach with `\romannumeral` ensures that each expansion step gives a balanced token list.

```
\prg_new_conditional:Npnn \tl_if_head_space:n #1 { p , T , F , TF }
  {
    \tex_romannumeral:D \if_false: { \fi:
      \tl_if_head_space_aux:w ? #1 ? ~ }
  }
\cs_new:Npn \tl_if_head_space_aux:w #1 ~
  {
    \tl_if_empty:oTF { \use_none:n #1 }
      { \exp_after:wN \c_zero \exp_after:wN \prg_return_true: }
      { \exp_after:wN \c_zero \exp_after:wN \prg_return_false: }
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
```

(*End definition for* `\tl_if_head_space:n`. *These functions are documented on page 94.*)

192.11 Viewing token lists

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

Showing token list variables is done directly: at the moment do not worry if they are defined.

```
\cs_new_protected:Npn \tl_show:N #1 { \cs_show:N #1 }
\cs_generate_variant:Nn \tl_show:N { c }
```

(*End definition for* `\tl_show:N` *and* `\tl_show:c`. *These functions are documented on page* ??.)

`\tl_show:n`

For literal token lists, life is easy.

```
\cs_new_eq:NN \tl_show:n \etex_showtokens:D
```

(*End definition for* `\tl_show:n`. *This function is documented on page 95.*)

192.12 Constant token lists

`\c_job_name_tl`

Inherited from the LaTeX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course. LuaTeX does not quote file names containing spaces, whereas pdfTeX and XeTeX do. So there may be a correction to make in the LuaTeX case.

```
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
```

```
  {
    \tex_the:D \tex_everyjob:D
    \luatex_if_engine:T
      {
        \lua_now:x
          { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
      }
  }
⟨/initex⟩
\tl_const:Nx \c_job_name_tl { \tex_jobname:D }
```

(*End definition for* `\c_job_name_tl`. *This variable is documented on page 95.*)

`\c_space_tl` A space as a token list (as opposed to as a character).

```
\tl_const:Nn \c_space_tl { ~ }
```

(*End definition for* `\c_space_tl`. *This variable is documented on page 95.*)

192.13 Scratch token lists

`\g_tmpa_tl`
`\g_tmpb_tl`

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

```
\tl_new:N \g_tmpa_tl
\tl_new:N \g_tmpb_tl
```

(*End definition for* `\g_tmpa_tl` *and* `\g_tmpb_tl`. *These variables are documented on page 95.*)

`\l_tmpa_tl`
`\l_tmpb_tl`

These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
\tl_new:N \l_tmpa_tl
\tl_new:N \l_tmpb_tl
```

(*End definition for* `\l_tmpa_tl` *and* `\l_tmpb_tl`. *These variables are documented on page 95.*)

192.14 Experimental functions

`\str_if_eq_return:xx` It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF`, but hard-coded for speed.

```
\cs_new:Npn \str_if_eq_return:xx #1 #2
  {
    \if_int_compare:w \pdftex_strcmp:D {#1} {#2} = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\str_if_eq_return:xx`. *This function is documented on page* **??**.)

\tl_if_single_p:N
\tl_if_single:NTF

Expand the token list and feed it to \tl_if_single:n.

```
\cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
\cs_new:Npn \tl_if_single:NT  { \exp_args:No \tl_if_single:nT  }
\cs_new:Npn \tl_if_single:NF  { \exp_args:No \tl_if_single:nF  }
\cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
```

(*End definition for \tl_if_single:N. These functions are documented on page 88.*)

\tl_if_single_p:n
\tl_if_single:nTF

A token list has exactly one item if it is either a single token surrounded by optional explicit spaces, or a single brace group surrounded by optional explicit spaces. The naive version of this test would do \use_none:n #1, and test if the result is empty. However, this will fail when the token list is empty. Furthermore, it does not allow optional trailing spaces.

```
\prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
  { \str_if_eq_return:xx { \exp_not:o { \use_none:nn #1 ?? } } {?} }
```

(*End definition for \tl_if_single:n. These functions are documented on page 88.*)

\tl_if_single_token_p:n
\tl_if_single_token:nTF

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptyness. Otherwise, compare with a single space, only case where we have a single token.

```
\prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
  {
    \tl_if_head_N_type:nTF {#1}
      { \str_if_eq_return:xx { \exp_not:o { \use_none:n #1 } } { } }
      { \str_if_eq_return:xx { \exp_not:n {#1} } { ~ } }
  }
```

(*End definition for \tl_if_single_token:n. These functions are documented on page 88.*)

\q_tl_act_mark
\q_tl_act_stop

The \tl_act functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list.Only \q_tl_act_mark and \q_tl_act_stop may not appear in the token lists manipulated by \tl_act functions. The quarks are effectively defined in l3quark.

(*End definition for \q_tl_act_mark and \q_tl_act_stop. These variables are documented on page 97.*)

\tl_act:NNNnn
\tl_act_aux:NNNnn
\tl_act_output:n
\tl_act_reverse_output:n
\tl_act_group_recurse:Nnn
\tl_act_loop:w
\tl_act_normal:NwnNNN
\tl_act_group:nwnNNN
\tl_act_space:wwnNNN
\tl_act_end:w

To help control the expansion, \tl_act:NNNnn starts with \romannumeral and ends by producing \c_zero once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker \q_tl_act_mark is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function \tl_act_result:n.

```
\cs_new:Npn \tl_act:NNNnn { \tex_romannumeral:D \tl_act_aux:NNNnn }
\cs_new:Npn \tl_act_aux:NNNnn #1 #2 #3 #4 #5
  {
    \group_align_safe_begin:
    \tl_act_loop:w #5 \q_tl_act_mark \q_tl_act_stop
    {#4} #1 #2 #3
    \tl_act_result:n { }
  }
```

In the loop, we check how the token list begins and act accordingly. In the "normal" case, we may have reached `\q_tl_act_mark`, the end of the list. Then leave `\c_zero` and the result in the input stream, to terminate the expansion of `\romannumeral`. Otherwise, apply the relevant function to the "arguments", `#3` and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `\tl_act_space:wwnNNN` gobble the space.

```
\cs_new:Npn \tl_act_loop:w #1 \q_tl_act_stop
  {
    \tl_if_head_N_type:nTF {#1}
      { \tl_act_normal:NwnNNN }
      {
        \tl_if_head_group:nTF {#1}
          { \tl_act_group:nwnNNN }
          { \tl_act_space:wwnNNN }
      }
    #1 \q_tl_act_stop
  }
\cs_new:Npn \tl_act_normal:NwnNNN #1 #2 \q_tl_act_stop #3#4
  {
    \if_meaning:w \q_tl_act_mark #1
      \exp_after:wN \tl_act_end:wn
    \fi:
    #4 {#3} #1
    \tl_act_loop:w #2 \q_tl_act_stop
    {#3} #4
  }
\cs_new:Npn \tl_act_end:wn #1 \tl_act_result:n #2
  { \group_align_safe_end: \c_zero #2 }
\cs_new:Npn \tl_act_group:nwnNNN #1 #2 \q_tl_act_stop #3#4#5
  {
    #5 {#3} {#1}
    \tl_act_loop:w #2 \q_tl_act_stop
    {#3} #4 #5
  }
\exp_last_unbraced:NNo
  \cs_new:Npn \tl_act_space:wwnNNN \c_space_tl #1 \q_tl_act_stop #2#3#4#5
  {
    #5 {#2}
    \tl_act_loop:w #1 \q_tl_act_stop
    {#2} #3 #4 #5
  }
```

Typically, the output is done to the right of what was already output, using `\tl_act_output:n`, but for the `\tl_act_reverse` functions, it should be done to the left.

```
\cs_new:Npn \tl_act_output:n #1 #2 \tl_act_result:n #3
  { #2 \tl_act_result:n { #3 #1 } }
\cs_new:Npn \tl_act_reverse_output:n #1 #2 \tl_act_result:n #3
  { #2 \tl_act_result:n { #1 #3 } }
```

In many applications of `\tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, `#1` is the output function, `#2` is the transformation, which should expand in two steps, and `#3` is the group.

```
\cs_new:Npn \tl_act_group_recurse:Nnn #1#2#3
  {
    \exp_args:Nf #1
      { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
  }
```

(*End definition for* `\tl_act:NNNnn` *and* `\tl_act_aux:NNNnn`*. These functions are documented on page* **??**.)

`\tl_reverse_tokens:n`
`\tl_act_reverse_normal:nN`
`\tl_act_reverse_group:nn`
`\tl_act_reverse_space:n`

The goal is to reverse a token list. This is done by feeding `\tl_act_aux:NNNnn` three functions, an empty fourth argument (we don't use it for `\tl_act_reverse_tokens:n`), and as a fifth argument the token list to be reversed. Spaces and normal tokens are output to the left of the current output. For groups, we must recursively apply `\tl_act_reverse_tokens:n` to the group, and output, still on the left. Note that in all three cases, we throw one argument away: this ⟨*parameter*⟩ is where for instance the upper/lowercasing action stores the information of whether it is uppercasing or lowercasing.

```
\cs_new:Npn \tl_reverse_tokens:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
        \tl_act_aux:NNNnn
          \tl_act_reverse_normal:nN
          \tl_act_reverse_group:nn
          \tl_act_reverse_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \tl_act_reverse_space:n #1
  { \tl_act_reverse_output:n {~} }
\cs_new:Npn \tl_act_reverse_normal:nN #1 #2
  { \tl_act_reverse_output:n {#2} }
\cs_new:Npn \tl_act_reverse_group:nn #1
  {
    \tl_act_group_recurse:Nnn
      \tl_act_reverse_output:n
      { \tl_reverse_tokens:n }
  }
```

(*End definition for* `\tl_reverse_tokens:n`*. This function is documented on page 96.*)

`\tl_reverse:n`
`\tl_reverse:o`
`\tl_reverse:V`
`\tl_reverse_group_preserve:nn`

The goal here is to reverse without losing spaces nor braces. The only difference with `\tl_reverse_tokens:n` is that we now simply output groups without entering them.

```
\cs_new:Npn \tl_reverse:n #1
  {
    \etex_unexpanded:D \exp_after:wN
```

```
      {
        \tex_romannumeral:D
        \tl_act_aux:NNNnn
          \tl_act_reverse_normal:nN
          \tl_act_reverse_group_preserve:nn
          \tl_act_reverse_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \tl_act_reverse_group_preserve:nn #1 #2
  { \tl_act_reverse_output:n { {#2} } }
\cs_generate_variant:Nn \tl_reverse:n { o , V }
```

(*End definition for* `\tl_reverse:n`*,* `\tl_reverse:o`*, and* `\tl_reverse:V`*. These functions are documented on page* **??***.*)

`\tl_reverse:N`
`\tl_reverse:c`
`\tl_greverse:N`
`\tl_greverse:c`

This reverses the list, leaving `\exp_stop_f:` in front, which stops the `f`-expansion.

```
\cs_new_protected:Npn \tl_reverse:N #1
  { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_new_protected:Npn \tl_greverse:N #1
  { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_generate_variant:Nn \tl_reverse:N  { c }
\cs_generate_variant:Nn \tl_greverse:N { c }
```

(*End definition for* `\tl_reverse:N` *and others. These functions are documented on page* **??***.*)

`\tl_length_tokens:n`
`\tl_act_length_normal:nN`
`\tl_act_length_group:nn`
`\tl_act_length_space:n`

The length is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\c_zero` inserted by `\tl_act_end:wn`. Somewhat a hack.

```
\cs_new:Npn \tl_length_tokens:n #1
  {
    \int_eval:n
      {
        \tl_act_aux:NNNnn
          \tl_act_length_normal:nN
          \tl_act_length_group:nn
          \tl_act_length_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \tl_act_length_normal:nN #1 #2 { 1 + }
\cs_new:Npn \tl_act_length_space:n #1 { 1 + }
\cs_new:Npn \tl_act_length_group:nn #1 #2
  { 2 + \tl_length_tokens:n {#2} + }
```

(*End definition for* `\tl_length_tokens:n`*. This function is documented on page 96.*)

`\c_tl_act_uppercase_tl`
`\c_tl_act_lowercase_tl`

These constants contain the correspondance between lowercase and uppercase letters, in the form `aAbBcC...` and `AaBbCc...` respectively.

```
\tl_const:Nn \c_tl_act_uppercase_tl
```

```
  {
    aA bB cC dD eE fF gG hH iI jJ kK lL mM
    nN oO pP qQ rR sS tT uU vV wW xX yY zZ
  }
\tl_const:Nn \c_tl_act_lowercase_tl
  {
    Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
    Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
  }
```

(*End definition for* `\c_tl_act_uppercase_tl` *and* `\c_tl_act_lowercase_tl`*. These variables are documented on page* **??**.)

`\tl_expandable_uppercase:n`
`\tl_expandable_lowercase:n`
`\tl_act_case_normal:nN`
`\tl_act_case_group:nn`
`\tl_act_case_space:n`

The only difference between uppercasing and lowercasing is the table of correspondance that is used. As for other token list actions, we feed `\tl_act_aux:NNNnn` three functions, and this time, we use the ⟨*parameters*⟩ argument to carry which case-changing we are applying. A space is simply output. A normal token is compared to each letter in the alphabet using `\str_if_eq:nn` tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group (the `\exp_after:wN` trigger `\romannumeral`, which expands fully to give the converted group), then output.

```
\cs_new:Npn \tl_expandable_uppercase:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
          \tl_act_case_aux:nn { \c_tl_act_uppercase_tl } {#1}
      }
  }
\cs_new:Npn \tl_expandable_lowercase:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
          \tl_act_case_aux:nn { \c_tl_act_lowercase_tl } {#1}
      }
  }
\cs_new:Npn \tl_act_case_aux:nn
  {
    \tl_act_aux:NNNnn
      \tl_act_case_normal:nN
      \tl_act_case_group:nn
      \tl_act_case_space:n
  }
\cs_new:Npn \tl_act_case_space:n #1 { \tl_act_output:n {~} }
\cs_new:Npn \tl_act_case_normal:nN #1 #2
  {
    \exp_args:Nf \tl_act_output:n
      {
        \exp_args:NNo \prg_case_str:nnn #2 {#1}
```

```
          { \exp_stop_f: #2 }
      }
  }
\cs_new:Npn \tl_act_case_group:nn #1 #2
  {
    \exp_after:wN \tl_act_output:n \exp_after:wN
      { \exp_after:wN { \tex_romannumeral:D \tl_act_case_aux:nn {#1} {#2} } }
  }
```

(*End definition for* `\tl_expandable_uppercase:n` *and* `\tl_expandable_lowercase:n`. *These functions are documented on page 96.*)

`\tl_item:nn`
`\tl_item:Nn`
`\tl_item:cn`
`\tl_item_aux:nn`

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\quark_if_recursion_tail_stop:n` terminates the loop, and returns nothing at all.

```
\cs_new:Npn \tl_item:nn #1#2
  {
    \exp_args:Nf \tl_item_aux:nn
      {
        \int_eval:n
          {
            \int_compare:nNnT {#2} < \c_zero
              { \tl_length:n {#1} + }
            #2
          }
      }
    #1
    \q_recursion_tail
    \prg_break_point:n { }
  }
\cs_new:Npn \tl_item_aux:nn #1#2
  {
    \quark_if_recursion_tail_break:n {#2}
    \int_compare:nNnTF {#1} = \c_zero
      { \tl_map_break:n { \exp_not:n {#2} } }
      { \exp_args:Nf \tl_item_aux:nn { \int_eval:n { #1 - 1 } } }
  }
\cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
\cs_generate_variant:Nn \tl_item:Nn { c }
```

(*End definition for* `\tl_item:nn`, `\tl_item:Nn`, *and* `\tl_item:cn`. *These functions are documented on page* **??**.)

`\tl_if_empty_p:x`
`\tl_if_empty:xTF`

We can test expandably the emptyness of an expanded token list thanks to the primitive `\pdfstrcmp` which expands its argument: a token list is empty if and only if its string representation is empty.

```
\prg_new_conditional:Npnn \tl_if_empty:x #1 { p , T , F , TF }
  { \str_if_eq_return:xx { } {#1} }
```

(*End definition for* `\tl_if_empty:x`. *These functions are documented on page* **??**.)

192.15 Deprecated functions

\tl_new:Nn
\tl_new:cn
\tl_new:Nx

Use either \tl_const:Nn or \tl_new:N.

```
⟨*deprecated⟩
\cs_new_protected:Npn \tl_new:Nn #1#2
  {
    \tl_new:N #1
    \tl_gset:Nn #1 {#2}
  }
\cs_generate_variant:Nn \tl_new:Nn { c }
\cs_generate_variant:Nn \tl_new:Nn { Nx }
⟨/deprecated⟩
```

(*End definition for* \tl_new:Nn, \tl_new:cn, *and* \tl_new:Nx. *These functions are documented on page* **??**.)

\tl_gset:Nc
\tl_set:Nc

This was useful once, but nowadays does not make much sense.

```
⟨*deprecated⟩
\cs_new_protected_nopar:Npn \tl_gset:Nc
  { \tex_global:D \tl_set:Nc }
\cs_new_protected:Npn \tl_set:Nc #1#2
  { \tl_set:No #1 { \cs:w #2 \cs_end: } }
⟨/deprecated⟩
```

(*End definition for* \tl_gset:Nc. *This function is documented on page* **??**.)

\tl_replace_in:Nnn
\tl_replace_in:cnn
\tl_greplace_in:Nnn
\tl_greplace_in:cnn
\tl_replace_all_in:Nnn
\tl_replace_all_in:cnn
\tl_greplace_all_in:Nnn
\tl_greplace_all_in:cnn

These are renamed.

```
⟨*deprecated⟩
\cs_new_eq:NN \tl_replace_in:Nnn  \tl_replace_once:Nnn
\cs_new_eq:NN \tl_replace_in:cnn  \tl_replace_once:cnn
\cs_new_eq:NN \tl_greplace_in:Nnn \tl_greplace_once:Nnn
\cs_new_eq:NN \tl_greplace_in:cnn \tl_greplace_once:cnn
\cs_new_eq:NN \tl_replace_all_in:Nnn  \tl_replace_all:Nnn
\cs_new_eq:NN \tl_replace_all_in:cnn  \tl_replace_all:cnn
\cs_new_eq:NN \tl_greplace_all_in:Nnn \tl_greplace_all:Nnn
\cs_new_eq:NN \tl_greplace_all_in:cnn \tl_greplace_all:cnn
⟨/deprecated⟩
```

(*End definition for* \tl_replace_in:Nnn *and* \tl_replace_in:cnn. *These functions are documented on page* **??**.)

\tl_remove_in:Nn
\tl_remove_in:cn
\tl_gremove_in:Nn
\tl_gremove_in:cn
\tl_remove_all_in:Nn
\tl_remove_all_in:cn
\tl_gremove_all_in:Nn
\tl_gremove_all_in:cn

Also renamed.

```
⟨*deprecated⟩
\cs_new_eq:NN \tl_remove_in:Nn  \tl_remove_once:Nn
\cs_new_eq:NN \tl_remove_in:cn  \tl_remove_once:cn
\cs_new_eq:NN \tl_gremove_in:Nn \tl_gremove_once:Nn
\cs_new_eq:NN \tl_gremove_in:cn \tl_gremove_once:cn
\cs_new_eq:NN \tl_remove_all_in:Nn  \tl_remove_all:Nn
\cs_new_eq:NN \tl_remove_all_in:cn  \tl_remove_all:cn
\cs_new_eq:NN \tl_gremove_all_in:Nn \tl_gremove_all:Nn
\cs_new_eq:NN \tl_gremove_all_in:cn \tl_gremove_all:cn
⟨/deprecated⟩
```

(*End definition for* `\tl_remove_in:Nn` *and* `\tl_remove_in:cn`*. These functions are documented on page* **??**.)

`\tl_elt_count:n`
`\tl_elt_count:V`
`\tl_elt_count:o`
`\tl_elt_count:N`
`\tl_elt_count:c`

Another renaming job.

```
⟨*deprecated⟩
\cs_new_eq:NN \tl_elt_count:n \tl_length:n
\cs_new_eq:NN \tl_elt_count:V \tl_length:V
\cs_new_eq:NN \tl_elt_count:o \tl_length:o
\cs_new_eq:NN \tl_elt_count:N \tl_length:N
\cs_new_eq:NN \tl_elt_count:c \tl_length:c
⟨/deprecated⟩
```

(*End definition for* `\tl_elt_count:n`*,* `\tl_elt_count:V`*, and* `\tl_elt_count:o`*. These functions are documented on page* **??**.)

`\tl_head_i:n`
`\tl_head_i:w`
`\tl_head_iii:n`
`\tl_head_iii:f`
`\tl_head_iii:w`

Two renames, and a few that are rather too specialised.

```
⟨*deprecated⟩
\cs_new_eq:NN \tl_head_i:n \tl_head:n
\cs_new_eq:NN \tl_head_i:w \tl_head:w
\cs_new:Npn \tl_head_iii:n #1 { \tl_head_iii:w #1 \q_stop }
\cs_generate_variant:Nn \tl_head_iii:n { f }
\cs_new:Npn \tl_head_iii:w #1#2#3#4 \q_stop {#1#2#3}
⟨/deprecated⟩
```

(*End definition for* `\tl_head_i:n`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

193 l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

A sequence is a control sequence whose top-level expansion is of the form "`\seq_item:n` {⟨$item_0$⟩} ...`\seq_item:n` {⟨$item_{n-1}$⟩}". An earlier implementation used the structure "`\seq_elt:w` ⟨$item_1$⟩ `\seq_elt_end:` ...`\seq_elt:w` ⟨$item_n$⟩ `\seq_elt_end:`". This allows rapid searching using a delimited function, but is not suitable for items containing `{`, `}` and `#` tokens, and also leads to the loss of surrounding braces around items.

`\seq_item:n`

The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
\cs_new:Npn \seq_item:n
  {
    \msg_expandable_kernel_error:nn { seq } { misused }
```

```
    \use_none:n
  }
```

(*End definition for* `\seq_item:n`*. This function is documented on page 106.*)

`\l_seq_internal_a_tl`
`\l_seq_internal_b_tl`

Scratch space for various internal uses.

```
\tl_new:N \l_seq_internal_a_tl
\tl_new:N \l_seq_internal_b_tl
```

(*End definition for* `\l_seq_internal_a_tl` *and* `\l_seq_internal_b_tl`*. These variables are documented on page* **??***.*)

193.1 Allocation and initialisation

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

Internally, sequences are just token lists.

```
\cs_new_eq:NN \seq_new:N \tl_new:N
\cs_new_eq:NN \seq_new:c \tl_new:c
```

(*End definition for* `\seq_new:N` *and* `\seq_new:c`*. These functions are documented on page* **??***.*)

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

Clearing sequences is just the same as clearing token lists.

```
\cs_new_eq:NN \seq_clear:N  \tl_clear:N
\cs_new_eq:NN \seq_clear:c  \tl_clear:c
\cs_new_eq:NN \seq_gclear:N \tl_gclear:N
\cs_new_eq:NN \seq_gclear:c \tl_gclear:c
```

(*End definition for* `\seq_clear:N` *and* `\seq_clear:c`*. These functions are documented on page* **??***.*)

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

Once again a copy from the token list functions.

```
\cs_new_eq:NN \seq_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \seq_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c
```

(*End definition for* `\seq_clear_new:N` *and* `\seq_clear_new:c`*. These functions are documented on page* **??***.*)

`\seq_set_eq:NN`
`\seq_set_eq:cN`
`\seq_set_eq:Nc`
`\seq_set_eq:cc`
`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

Once again, these are simple copies from the token list functions.

```
\cs_new_eq:NN \seq_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \seq_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \seq_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \seq_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\seq_set_eq:NN` *and others. These functions are documented on page* **??***.*)

\seq_set_split:Nnn
\seq_gset_split:Nnn
\seq_set_split_aux:NNnn
\seq_set_split_aux_i:w
\seq_set_split_aux_ii:w
\seq_set_split_aux_end:

The goal is to split a given token list at a marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l_seq_internal_a_tl` is a repetition of the pattern `\seq_set_split_aux_i:w \prg_do_nothing:` ⟨*item with spaces*⟩ `\seq_set_split_aux_end:`. Then, x-expansion causes `\seq_set_split_aux_i:w` to trim spaces, and leaves its result as `\seq_set_split_aux_ii:w` ⟨*trimmed item*⟩ `\seq_set_split_aux_end:`. This is then converted to the l3seq internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```
\cs_new_protected_nopar:Npn \seq_set_split:Nnn
  { \seq_set_split_aux:NNnn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_split:Nnn
  { \seq_set_split_aux:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \seq_set_split_aux:NNnn #1 #2 #3 #4
  {
    \tl_if_empty:nTF {#3}
      { #1 #2 { \tl_map_function:nN {#4} \seq_wrap_item:n } }
      {
        \tl_set:Nn \l_seq_internal_a_tl
          {
            \seq_set_split_aux_i:w \prg_do_nothing:
            #4
            \seq_set_split_aux_end:
          }
        \tl_replace_all:Nnn \l_seq_internal_a_tl { #3 }
          {
            \seq_set_split_aux_end:
            \seq_set_split_aux_i:w \prg_do_nothing:
          }
        \tl_set:Nx \l_seq_internal_a_tl { \l_seq_internal_a_tl }
        #1 #2 { \l_seq_internal_a_tl }
      }
  }
\cs_new:Npn \seq_set_split_aux_i:w #1 \seq_set_split_aux_end:
  {
    \exp_not:N \seq_set_split_aux_ii:w
    \exp_args:No \tl_trim_spaces:n {#1}
    \exp_not:N \seq_set_split_aux_end:
  }
\cs_new:Npn \seq_set_split_aux_ii:w #1 \seq_set_split_aux_end:
  { \seq_wrap_item:n {#1} }
```

(*End definition for* `\seq_set_split:Nnn` *and* `\seq_gset_split:Nnn`*. These functions are documented on page 98.*)

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

Concatenating sequences is easy.

```
\cs_new_protected:Npn \seq_concat:NNN #1#2#3
  { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
```

```
  { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_generate_variant:Nn \seq_concat:NNN  { ccc }
\cs_generate_variant:Nn \seq_gconcat:NNN { ccc }
```

(*End definition for* `\seq_concat:NNN` *and* `\seq_concat:ccc`*. These functions are documented on page* **??**.)

\seq_if_exist_p:N
\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \seq_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \seq_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \seq_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \seq_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \seq_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \seq_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \seq_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \seq_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\seq_if_exist:N` *and* `\seq_if_exist:c`*. These functions are documented on page* **??**.)

193.2 Appending data to either end

\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:Nv
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:cv
\seq_put_left:co
\seq_put_left:cx
\seq_put_right:Nn
\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx

The code here is just a wrapper for adding to token lists.

```
\cs_new_protected:Npn \seq_put_left:Nn #1#2
  { \tl_put_left:Nn #1 { \seq_item:n {#2} } }
\cs_new_protected:Npn \seq_put_right:Nn #1#2
  { \tl_put_right:Nn #1 { \seq_item:n {#2} } }
\cs_generate_variant:Nn \seq_put_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_left:Nn  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_put_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

(*End definition for* `\seq_put_left:Nn` *and others. These functions are documented on page* **??**.)

\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:Nv
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:cv
\seq_gput_left:co
\seq_gput_left:cx
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\seq_gput_right:cx

The same for global addition.

```
\cs_new_protected:Npn \seq_gput_left:Nn #1#2
  { \tl_gput_left:Nn #1 { \seq_item:n {#2} } }
\cs_new_protected:Npn \seq_gput_right:Nn #1#2
  { \tl_gput_right:Nn #1 { \seq_item:n {#2} } }
\cs_generate_variant:Nn \seq_gput_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_left:Nn  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_gput_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
```

(*End definition for* `\seq_gput_left:Nn` *and others. These functions are documented on page* **??**.)

193.3 Modifying sequences

\seq_wrap_item:n

This function converts its argument to a proper sequence item in an `x`-expansion context.

```
\cs_new:Npn \seq_wrap_item:n #1 { \exp_not:n { \seq_item:n {#1} } }
```

(*End definition for* `\seq_wrap_item:n`.)

\l_seq_internal_remove_seq An internal sequence for the removal routines.

```
\seq_new:N \l_seq_internal_remove_seq
```

(*End definition for* `\l_seq_internal_remove_seq`*. This variable is documented on page* ??.)

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\seq_remove_duplicates_aux:NN

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \seq_remove_duplicates:N
  { \seq_remove_duplicates_aux:NN \seq_set_eq:NN }
\cs_new_protected:Npn \seq_gremove_duplicates:N
  { \seq_remove_duplicates_aux:NN \seq_gset_eq:NN }
\cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2
  {
    \seq_clear:N \l_seq_internal_remove_seq
    \seq_map_inline:Nn #2
      {
        \seq_if_in:NnF \l_seq_internal_remove_seq {##1}
          { \seq_put_right:Nn \l_seq_internal_remove_seq {##1} }
      }
    #1 #2 \l_seq_internal_remove_seq
  }
\cs_generate_variant:Nn \seq_remove_duplicates:N  { c }
\cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(*End definition for* `\seq_remove_duplicates:N` *and* `\seq_remove_duplicates:c`*. These functions are documented on page* ??.)

\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
\seq_remove_all_aux:NNn

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\seq_pop_right_aux_ii:NNN`, using a "flexible" `x`-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the `x`-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The `x`-type is started again, including all of the items copied already. This will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off `x`-type expansion (`#1 #2 {#2}`) will ensure that nothing is lost.

```
\cs_new_protected:Npn \seq_remove_all:Nn
  { \seq_remove_all_aux:NNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gremove_all:Nn
  { \seq_remove_all_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \seq_remove_all_aux:NNn #1#2#3
  {
    \seq_push_item_def:n
      {
        \str_if_eq:nnT {##1} {#3}
          {
            \if_false: { \fi: }
            \tl_set:Nn \l_seq_internal_b_tl {##1}
            #1 #2
               { \if_false: } \fi:
```

```
                  \exp_not:o {#2}
                  \tl_if_eq:NNT \l_seq_internal_a_tl \l_seq_internal_b_tl
                    { \use_none:nn }
              }
            \seq_wrap_item:n {##1}
          }
        \tl_set:Nn \l_seq_internal_a_tl {#3}
        #1 #2 {#2}
        \seq_pop_item_def:
      }
    \cs_generate_variant:Nn \seq_remove_all:Nn  { c }
    \cs_generate_variant:Nn \seq_gremove_all:Nn { c }
```

(*End definition for* `\seq_remove_all:Nn` *and* `\seq_remove_all:cn`*. These functions are documented on page* **??**.)

193.4 Sequence conditionals

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

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \seq_if_empty:N \tl_if_empty:N
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \seq_if_empty:c \tl_if_empty:c
  { p , T , F , TF }
```

(*End definition for* `\seq_if_empty:N` *and* `\seq_if_empty:c`*. These functions are documented on page* **??**.)

`\seq_if_in:NnTF`
`\seq_if_in:NVTF`
`\seq_if_in:NvTF`
`\seq_if_in:NoTF`
`\seq_if_in:NxTF`
`\seq_if_in:cnTF`
`\seq_if_in:cVTF`
`\seq_if_in:cvTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`
`\seq_if_in_aux:`

The approach here is to define `\seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\prg_return_true:` is inserted. On the other hand, if there is no match then the loop will break returning `\prg_return_false:`. In either case, `\prg_break_point:n` ensures that the group ends before the logical value is returned. Everything is inside a group so that `\seq_item:n` is preserved in nested situations.

```
\prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
  { T , F , TF }
  {
    \group_begin:
      \tl_set:Nn \l_seq_internal_a_tl {#2}
      \cs_set_protected:Npn \seq_item:n ##1
        {
          \tl_set:Nn \l_seq_internal_b_tl {##1}
          \if_meaning:w \l_seq_internal_a_tl \l_seq_internal_b_tl
            \exp_after:wN \seq_if_in_aux:
          \fi:
        }
      #1
      \seq_break:n { \prg_return_false: }
    \prg_break_point:n { \group_end: }
  }
\cs_new_nopar:Npn \seq_if_in_aux: { \seq_break:n { \prg_return_true: } }
\cs_generate_variant:Nn \seq_if_in:NnT  {     NV , Nv , No , Nx }
```

```
\cs_generate_variant:Nn \seq_if_in:NnT  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnF  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnF  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnTF {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
```

(*End definition for* `\seq_if_in:Nn` *and others. These functions are documented on page* **??**.)

193.5 Recovering data from sequences

`\seq_get_left:NN`
`\seq_get_left:cN`
`\seq_get_left_aux:NnwN`

Getting an item from the left of a sequence is pretty easy: just trim off the first item after removing the `\seq_item:n` at the start.

```
\cs_new_protected:Npn \seq_get_left:NN #1#2
  {
    \seq_if_empty_err_break:N #1
    \exp_after:wN \seq_get_left_aux:NnwN #1 \q_stop #2
    \prg_break_point:n { }
  }
\cs_new_protected:Npn \seq_get_left_aux:NnwN \seq_item:n #1#2 \q_stop #3
  { \tl_set:Nn #3 {#1} }
\cs_generate_variant:Nn \seq_get_left:NN { c }
```

(*End definition for* `\seq_get_left:NN` *and* `\seq_get_left:cN`*. These functions are documented on page* **??**.)

`\seq_pop_left:NN`
`\seq_pop_left:cN`
`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
`\seq_pop_left_aux:NNN`
`\seq_pop_left_aux:NnwNNN`

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```
\cs_new_protected_nopar:Npn \seq_pop_left:NN
  { \seq_pop_left_aux:NNN \tl_set:Nn }
\cs_new_protected_nopar:Npn \seq_gpop_left:NN
  { \seq_pop_left_aux:NNN \tl_gset:Nn }
\cs_new_protected:Npn \seq_pop_left_aux:NNN #1#2#3
  {
    \seq_if_empty_err_break:N #2
    \exp_after:wN \seq_pop_left_aux:NnwNNN #2 \q_stop #1#2#3
    \prg_break_point:n { }
  }
\cs_new_protected:Npn \seq_pop_left_aux:NnwNNN \seq_item:n #1#2 \q_stop #3#4#5
  {
    #3 #4 {#2}
    \tl_set:Nn #5 {#1}
  }
\cs_generate_variant:Nn \seq_pop_left:NN  { c }
\cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(*End definition for* `\seq_pop_left:NN` *and* `\seq_pop_left:cN`*. These functions are documented on page* **??**.)

`\seq_get_right:NN`
`\seq_get_right:cN`
`\seq_get_right_aux:NN`
`\seq_get_right_loop:nn`

The idea here is to remove the very first `\seq_item:n` from the sequence, leaving a token list starting with the first braced entry. Two arguments at a time are then grabbed: apart from the right-hand end of the sequence, this will be a brace group followed by `\seq_-item:n`. The set up code means that these all disappear. At the end of the sequence, the assignment is placed in front of the very last entry in the sequence, before a tidying-up step takes place to remove the loop and reset the meaning of `\seq_item:n`.

```
\cs_new_protected:Npn \seq_get_right:NN #1#2
  {
    \seq_if_empty_err_break:N #1
    \seq_get_right_aux:NN #1#2
    \prg_break_point:n { }
  }
\cs_new_protected:Npn \seq_get_right_aux:NN #1#2
  {
    \seq_push_item_def:n { }
    \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
      \exp_after:wN \use_none:n #1
      { \tl_set:Nn #2 }
      { }
      {
        \seq_pop_item_def:
        \seq_break:
      }
}
\cs_new:Npn \seq_get_right_loop:nn #1#2
  {
    #2 {#1}
    \seq_get_right_loop:nn
  }
\cs_generate_variant:Nn \seq_get_right:NN { c }
```

(*End definition for `\seq_get_right:NN` and `\seq_get_right:cN`. These functions are documented on page* **??**.)

`\seq_pop_right:NN`
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
`\seq_pop_right_aux:NNN`
`\seq_pop_right_aux_ii:NNN`

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a "flexible length" way to set a token list variable. This is supplied by the `{ \if_false:} \fi:` ...`\if_false: { \fi: }` construct. Using an `x`-type expansion and a "non-expanding" definition for `\seq_item:n`, the left-most $n-1$ entries in a sequence of n items will be stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including brackets. When the last item of the sequence is reached, the closing bracket for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment, then a final loop clears up the code.

```
\cs_new_protected_nopar:Npn \seq_pop_right:NN
  { \seq_pop_right_aux:NNN \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gpop_right:NN
  { \seq_pop_right_aux:NNN \tl_gset:Nx }
\cs_new_protected:Npn \seq_pop_right_aux:NNN #1#2#3
  {
    \seq_if_empty_err_break:N #2
    \seq_pop_right_aux_ii:NNN #1 #2 #3
```

```
    \prg_break_point:n { }
  }
\cs_new_protected:Npn \seq_pop_right_aux_ii:NNN #1#2#3
  {
    \seq_push_item_def:n { \seq_wrap_item:n {##1} }
    #1 #2 { \if_false: } \fi:
      \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
         \exp_after:wN \use_none:n #2
        {
          \if_false: { \fi: }
          \tl_set:Nn #3
        }
        { }
        {
          \seq_pop_item_def:
          \seq_break:
        }
  }
\cs_generate_variant:Nn \seq_pop_right:NN  { c }
\cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(*End definition for* `\seq_pop_right:NN` *and* `\seq_pop_right:cN`*. These functions are documented on page* **??***.*)

193.6 Mapping to sequences

`\seq_map_break:`
`\seq_map_break:n`
`\seq_break:`
`\seq_break:n`

To break a function, the special token `\prg_break_point:n` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted. Semantically-logical copies of the break functions for use inside mappings.

```
\cs_new_eq:NN \seq_break:      \prg_map_break:
\cs_new_eq:NN \seq_break:n     \prg_map_break:n
\cs_new_eq:NN \seq_map_break:  \prg_map_break:
\cs_new_eq:NN \seq_map_break:n \prg_map_break:n
```

(*End definition for* `\seq_map_break:`*. This function is documented on page 107.*)

`\seq_if_empty_err_break:N`

A function to check that sequences really have some content. This is optimised for speed, hence the direct primitive use.

```
\cs_new_protected:Npn \seq_if_empty_err_break:N #1
  {
    \if_meaning:w #1 \c_empty_tl
      \msg_kernel_error:nnx { seq } { empty-sequence } { \token_to_str:N #1 }
      \exp_after:wN \seq_break:
    \fi:
  }
```

(*End definition for* `\seq_if_empty_err_break:N`*. This function is documented on page 106.*)

`\seq_map_function:NN`
`\seq_map_function:cN`
`\seq_map_function_aux:NNn`

The idea here is to apply the code of `#2` to each item in the sequence without altering the definition of `\seq_item:n`. This is done as by noting that every odd token in the sequence must be `\seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, #2 is instead ? \seq_map_break:, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```
\cs_new:Npn \seq_map_function:NN #1#2
  {
    \exp_after:wN \seq_map_function_aux:NNn \exp_after:wN #2 #1
      { ? \seq_map_break: } { }
    \prg_break_point:n { }
  }
\cs_new:Npn \seq_map_function_aux:NNn #1#2#3
  {
    \use_none:n #2
    #1 {#3}
    \seq_map_function_aux:NNn #1
  }
\cs_generate_variant:Nn \seq_map_function:NN { c }
```

(*End definition for* \seq_map_function:NN *and* \seq_map_function:cN*. These functions are documented on page* ??.)

\seq_push_item_def:n
\seq_push_item_def:x
\seq_push_item_def_aux:
\seq_pop_item_def:

The definition of \seq_item:n needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```
\cs_new_protected:Npn \seq_push_item_def:n
  {
    \seq_push_item_def_aux:
    \cs_gset:Npn \seq_item:n ##1
  }
\cs_new_protected:Npn \seq_push_item_def:x
  {
    \seq_push_item_def_aux:
    \cs_gset:Npx \seq_item:n ##1
  }
\cs_new_protected:Npn \seq_push_item_def_aux:
  {
    \cs_gset_eq:cN { seq_item_ \int_use:N \g_prg_map_int :n }
      \seq_item:n
    \int_gincr:N \g_prg_map_int
  }
\cs_new_protected_nopar:Npn \seq_pop_item_def:
  {
    \int_gdecr:N \g_prg_map_int
    \cs_gset_eq:Nc \seq_item:n
      { seq_item_ \int_use:N \g_prg_map_int :n }
  }
```

(*End definition for* \seq_push_item_def:n *and* \seq_push_item_def:x*. These functions are documented on page* ??.)

\seq_map_inline:Nn
\seq_map_inline:cn

The idea here is that \seq_item:n is already "applied" to each item in a sequence, and so an in-line mapping is just a case of redefining \seq_item:n.

```
\cs_new_protected:Npn \seq_map_inline:Nn #1#2
```

```
  {
    \seq_push_item_def:n {#2}
    #1
    \prg_break_point:n { \seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(*End definition for* `\seq_map_inline:Nn` *and* `\seq_map_inline:cn`*. These functions are documented on page* **??**.)

`\seq_map_variable:NNn`
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

This is just a specialised version of the in-line mapping function, using an `x`-type expansion for the code set up so that the number of `#` tokens required is as expected.

```
\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
  {
    \seq_push_item_def:x
      {
        \tl_set:Nn \exp_not:N #2 {##1}
        \exp_not:n {#3}
      }
    #1
    \prg_break_point:n { \seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_variable:NNn {     Nc }
\cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(*End definition for* `\seq_map_variable:NNn` *and others. These functions are documented on page* **??**.)

193.7 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn`
`\seq_push:NV`
`\seq_push:Nv`
`\seq_push:No`
`\seq_push:Nx`
`\seq_push:cn`
`\seq_push:cV`
`\seq_push:cV`
`\seq_push:co`
`\seq_push:cx`
`\seq_gpush:Nn`
`\seq_gpush:NV`
`\seq_gpush:Nv`
`\seq_gpush:No`
`\seq_gpush:Nx`
`\seq_gpush:cn`
`\seq_gpush:cV`
`\seq_gpush:cv`
`\seq_gpush:co`
`\seq_gpush:cx`

Pushing to a sequence is the same as adding on the left.

```
\cs_new_eq:NN \seq_push:Nn  \seq_put_left:Nn
\cs_new_eq:NN \seq_push:NV  \seq_put_left:NV
\cs_new_eq:NN \seq_push:Nv  \seq_put_left:Nv
\cs_new_eq:NN \seq_push:No  \seq_put_left:No
\cs_new_eq:NN \seq_push:Nx  \seq_put_left:Nx
\cs_new_eq:NN \seq_push:cn  \seq_put_left:cn
\cs_new_eq:NN \seq_push:cV  \seq_put_left:cV
\cs_new_eq:NN \seq_push:cv  \seq_put_left:cv
\cs_new_eq:NN \seq_push:co  \seq_put_left:co
\cs_new_eq:NN \seq_push:cx  \seq_put_left:cx
\cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
\cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx
```

(*End definition for* `\seq_push:Nn` *and others. These functions are documented on page* **??**.)

`\seq_get:NN`
`\seq_get:cN`
`\seq_pop:NN`
`\seq_pop:cN`
`\seq_gpop:NN`
`\seq_gpop:cN`

In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```
\cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
```

(*End definition for* `\seq_get:NN` *and* `\seq_get:cN`*. These functions are documented on page* **??**.)

193.8 Viewing sequences

`\seq_show:N`
`\seq_show:c`

Apply the general `\msg_aux_show:Nnx`.

```
\cs_new_protected:Npn \seq_show:N #1
  {
    \msg_aux_show:Nnx
      #1
      { seq }
      { \seq_map_function:NN #1 \msg_aux_show:n }
  }
\cs_generate_variant:Nn \seq_show:N { c }
```

(*End definition for* `\seq_show:N` *and* `\seq_show:c`*. These functions are documented on page* **??**.)

193.9 Experimental functions

`\seq_if_empty_break_return_false:N`

The name says it all: of the sequence is empty, returns logical `false`.

```
\cs_new:Npn \seq_if_empty_break_return_false:N #1
  {
    \if_meaning:w #1 \c_empty_tl
      \prg_return_false:
      \exp_after:wN \seq_break:
    \fi:
  }
```

(*End definition for* `\seq_if_empty_break_return_false:N`*.*)

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

Getting from the left or right with a check on the results.

```
\prg_new_protected_conditional:Npnn \seq_get_left:NN #1 #2 { T , F , TF }
  {
    \seq_if_empty_break_return_false:N #1
    \exp_after:wN \seq_get_left_aux:Nw #1 \q_stop #2
    \prg_return_true:
    \seq_break:
    \prg_break_point:n { }
  }
\prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
  {
```

```
    \seq_if_empty_break_return_false:N #1
    \seq_get_right_aux:NN #1#2
    \prg_return_true: \seq_break:
    \prg_break_point:n { }
  }
\cs_generate_variant:Nn \seq_get_left:NNT   { c }
\cs_generate_variant:Nn \seq_get_left:NNF   { c }
\cs_generate_variant:Nn \seq_get_left:NNTF  { c }
\cs_generate_variant:Nn \seq_get_right:NNT  { c }
\cs_generate_variant:Nn \seq_get_right:NNF  { c }
\cs_generate_variant:Nn \seq_get_right:NNTF { c }
```

(*End definition for* `\seq_get_left:NN` *and* `\seq_get_left:cN`*. These functions are documented on page* **??**.)

`\seq_pop_left:NNTF`
`\seq_pop_left:cNTF`
`\seq_gpop_left:NNTF`
`\seq_gpop_left:cNTF`
`\seq_pop_right:NNTF`
`\seq_pop_right:cNTF`
`\seq_gpop_right:NNTF`
`\seq_gpop_right:cNTF`

More or less the same for popping.

```
\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
  {
    \seq_if_empty_break_return_false:N #1
    \exp_after:wN \seq_pop_left_aux:NnwNNN #1 \q_stop \tl_set:Nn #1#2
    \prg_return_true: \seq_break:
    \prg_break_point:n { }
  }
\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
  {
    \seq_if_empty_break_return_false:N #1
    \exp_after:wN \seq_pop_left_aux:NnwNNN #1 \q_stop \tl_gset:Nn #1#2
    \prg_return_true: \seq_break:
    \prg_break_point:n { }
  }
\prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
  {
    \seq_if_empty_break_return_false:N #1
    \seq_pop_right_aux_ii:NNN \tl_set:Nx #1 #2
    \prg_return_true: \seq_break:
    \prg_break_point:n { }
  }
\prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
  {
    \seq_if_empty_break_return_false:N #1
    \seq_pop_right_aux_ii:NNN \tl_gset:Nx #1 #2
    \prg_return_true: \seq_break:
    \prg_break_point:n { }
  }
\cs_generate_variant:Nn \seq_pop_left:NNT    { c }
\cs_generate_variant:Nn \seq_pop_left:NNF    { c }
\cs_generate_variant:Nn \seq_pop_left:NNTF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNT   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNTF  { c }
\cs_generate_variant:Nn \seq_pop_right:NNT   { c }
```

```
\cs_generate_variant:Nn \seq_pop_right:NNF   { c }
\cs_generate_variant:Nn \seq_pop_right:NNTF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNT  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNTF { c }
```

(*End definition for* `\seq_pop_left:NN` *and* `\seq_pop_left:cN`. *These functions are documented on page* ??.)

`\seq_length:N`
`\seq_length:c`
`\seq_length_aux:n`

Counting the items in a sequence is done using the same approach as for other length functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics.

```
\cs_new:Npn \seq_length:N #1
  {
    \int_eval:n
      {
        0
        \seq_map_function:NN #1 \seq_length_aux:n
      }
  }
\cs_new:Npn \seq_length_aux:n #1 { +1 }
\cs_generate_variant:Nn \seq_length:N { c }
```

(*End definition for* `\seq_length:N` *and* `\seq_length:c`. *These functions are documented on page* ??.)

`\seq_item:Nn`
`\seq_item:cn`
`\seq_item_aux:nnn`

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the stop code `{ ? \seq_break: } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.

```
\cs_new:Npn \seq_item:Nn #1#2
  {
    \exp_last_unbraced:Nfo \seq_item_aux:nnn
      {
        \int_eval:n
          {
            \int_compare:nNnT {#2} < \c_zero
              { \seq_length:N #1 + }
            #2
          }
      }
    #1
    { ? \seq_break: }
    { }
    \prg_break_point:n { }
  }
\cs_new:Npn \seq_item_aux:nnn #1#2#3
  {
    \use_none:n #2
    \int_compare:nNnTF {#1} = \c_zero
      { \seq_break:n { \exp_not:n {#3} } }
      { \exp_args:Nf \seq_item_aux:nnn { \int_eval:n { #1 - 1 } } }
  }
```

```
\cs_generate_variant:Nn \seq_item:Nn { c }
```

(*End definition for* `\seq_item:Nn` *and* `\seq_item:cn`*. These functions are documented on page* **??**.)

`\seq_use:N`
`\seq_use:c`

A simple short cut for a mapping.

```
\cs_new:Npn \seq_use:N #1 { \seq_map_function:NN #1 \use:n }
\cs_generate_variant:Nn \seq_use:N { c }
```

(*End definition for* `\seq_use:N` *and* `\seq_use:c`*. These functions are documented on page* **??**.)

`\seq_mapthread_function:NNN`
`\seq_mapthread_function:NcN`
`\seq_mapthread_function:cNN`
`\seq_mapthread_function:ccN`
`\seq_mapthread_function_aux:NN`
`\seq_mapthread_function_aux:Nnnwnn`

The idea here is to first expand both of the sequences, adding the usual `{ ? \seq_break: } { }` to the end of each on. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first token of `#2` and `#5`, which for items in the sequences will both be `\seq_item:n`. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the length of the two sequences, or worrying about which is longer.

```
\cs_new:Npn \seq_mapthread_function:NNN #1#2#3
  {
    \exp_after:wN \seq_mapthread_function_aux:NN
      \exp_after:wN #3
      \exp_after:wN #1
      #2
      { ? \seq_break: } { }
    \prg_break_point:n { }
  }
\cs_new:Npn \seq_mapthread_function_aux:NN #1#2
  {
    \exp_after:wN \seq_mapthread_function_aux:Nnnwnn
      \exp_after:wN #1
      #2
      { ? \seq_break: } { }
      \q_stop
  }
\cs_new:Npn \seq_mapthread_function_aux:Nnnwnn #1#2#3#4 \q_stop #5#6
  {
    \use_none:n #2
    \use_none:n #5
    #1 {#3} {#6}
    \seq_mapthread_function_aux:Nnnwnn #1 #4 \q_stop
  }
\cs_generate_variant:Nn \seq_mapthread_function:NNN {     Nc }
\cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }
```

(*End definition for* `\seq_mapthread_function:NNN` *and others. These functions are documented on page* **??**.)

`\seq_set_from_clist:NN`
`\seq_set_from_clist:cN`
`\seq_set_from_clist:Nc`
`\seq_set_from_clist:cc`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:cN`
`\seq_gset_from_clist:Nc`
`\seq_gset_from_clist:cc`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

Setting a sequence from a comma-separated list is done using a simple mapping.

```
\cs_new_protected:Npn \seq_set_from_clist:NN #1#2
  {
    \tl_set:Nx #1
```

```
    { \clist_map_function:NN #2 \seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
  {
    \tl_set:Nx #1
      { \clist_map_function:nN {#2} \seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
  {
    \tl_gset:Nx #1
      { \clist_map_function:NN #2 \seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
  {
    \tl_gset:Nx #1
      { \clist_map_function:nN {#2} \seq_wrap_item:n }
  }
\cs_generate_variant:Nn \seq_set_from_clist:NN  {     Nc }
\cs_generate_variant:Nn \seq_set_from_clist:NN  { c , cc }
\cs_generate_variant:Nn \seq_set_from_clist:Nn  { c      }
\cs_generate_variant:Nn \seq_gset_from_clist:NN {     Nc }
\cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
\cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(*End definition for* `\seq_set_from_clist:NN` *and others. These functions are documented on page* **??**.)

`\seq_reverse:N`
`\seq_reverse:c`
`\seq_greverse:N`
`\seq_greverse:c`
`\seq_reverse_aux:NN`
`\seq_reverse_aux_item:nwn`

Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\exp_stop_f:` marker.

```
\cs_new_protected:Npn \seq_reverse:N #1
  {
    \cs_set_eq:NN \seq_item:n \seq_reverse_aux_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
  }
\cs_new:Npn \seq_reverse_aux_item:nw #1 #2 \exp_stop_f:
  {
    #2 \exp_stop_f:
    \seq_item:n {#1}
  }
```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, TEX's usual tail recursion does not take place in this case: since the following `\seq_reverse_aux_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\seq_item:n {#1}` left by the previous call, TEX cannot remove that previous call from the stack, and in particular must retain the various macro paramters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `\seq_reverse_aux_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TEX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of \exp_not:n. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```
\cs_new_protected_nopar:Npn \seq_tmp:w { }
\cs_new_protected_nopar:Npn \seq_reverse:N
  { \seq_reverse_aux:NN \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_greverse:N
  { \seq_reverse_aux:NN \tl_gset:Nx }
\cs_new_protected:Npn \seq_reverse_aux:NN #1 #2
  {
    \cs_set_eq:NN \seq_tmp:w \seq_item:n
    \cs_set_eq:NN \seq_item:n \seq_reverse_aux_item:nwn
    #1 #2 { #2 \exp_not:n { } }
    \cs_set_eq:NN \seq_item:n \seq_tmp:w
  }
\cs_new:Npn \seq_reverse_aux_item:nwn #1 #2 \exp_not:n #3
  {
    #2
    \exp_not:n { \seq_item:n {#1} #3 }
  }
\cs_generate_variant:Nn \seq_reverse:N  { c }
\cs_generate_variant:Nn \seq_greverse:N { c }
```

(*End definition for* \seq_reverse:N *and others. These functions are documented on page* ??.)

\seq_set_filter:NNn
\seq_gset_filter:NNn
\seq_set_filter_aux:NNNn

Similar to \seq_map_inline:Nn, without a \prg_break_point:n because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The \seq_wrap_item:n function inserts the relevant \seq_item:n without expansion in the input stream, hence in the x-expanding assignment.

```
\cs_new_protected_nopar:Npn \seq_set_filter:NNn
  { \seq_set_filter_aux:NNNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_filter:NNn
  { \seq_set_filter_aux:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \seq_set_filter_aux:NNNn #1#2#3#4
  {
    \seq_push_item_def:n { \bool_if:nT {#4} { \seq_wrap_item:n {##1} } }
    #1 #2 { #3 \prg_break_point:n { } }
    \seq_pop_item_def:
  }
```

(*End definition for* \seq_set_filter:NNn *and* \seq_gset_filter:NNn*. These functions are documented on page 106.*)

\seq_set_map:NNn
\seq_gset_map:NNn
\seq_set_map_aux:NNNn

Very similar to \seq_set_filter:NNn. We could actually merge the two within a single function, but it would have weird semantics.

```
\cs_new_protected_nopar:Npn \seq_set_map:NNn
  { \seq_set_map_aux:NNNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_map:NNn
  { \seq_set_map_aux:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \seq_set_map_aux:NNNn #1#2#3#4
  {
    \seq_push_item_def:n { \exp_not:N \seq_item:n {#4} }
```

```
    #1 #2 { #3 }
    \seq_pop_item_def:
  }
```

(*End definition for* `\seq_set_map:NNn` *and* `\seq_gset_map:NNn`*. These functions are documented on page* *106**.*)

193.10 Deprecated interfaces

A few functions which are no longer documented: these were moved here on or before 2011-04-20, and will be removed entirely by 2011-07-20.

`\seq_top:NN`
`\seq_top:cN`

These are old stack functions.

```
⟨*deprecated⟩
\cs_new_eq:NN \seq_top:NN \seq_get_left:NN
\cs_new_eq:NN \seq_top:cN \seq_get_left:cN
⟨/deprecated⟩
```

(*End definition for* `\seq_top:NN` *and* `\seq_top:cN`*. These functions are documented on page* **??***.*)

`\seq_display:N`
`\seq_display:c`

An older name for `\seq_show:N`.

```
⟨*deprecated⟩
\cs_new_eq:NN \seq_display:N \seq_show:N
\cs_new_eq:NN \seq_display:c \seq_show:c
⟨/deprecated⟩
```

(*End definition for* `\seq_display:N` *and* `\seq_display:c`*. These functions are documented on page* **??***.*)

```
⟨/initex | package⟩
```

194 l3clist implementation

The following test files are used for this code: m3clist002.

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

`\l_clist_internal_clist`

Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

```
\tl_new:N \l_clist_internal_clist
```

(*End definition for* `\l_clist_internal_clist`*. This variable is documented on page* **??***.*)

`\clist_tmp:w`

A temporary function for various purposes.

```
\cs_new_protected:Npn \clist_tmp:w { }
```

(*End definition for* `\clist_tmp:w`*.*)

194.1 Allocation and initialisation

\clist_new:N
\clist_new:c

Internally, comma lists are just token lists.

```
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:c \tl_new:c
```

(*End definition for* \clist_new:N *and* \clist_new:c *. These functions are documented on page* **??**.)

\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

Clearing comma lists is just the same as clearing token lists.

```
\cs_new_eq:NN \clist_clear:N  \tl_clear:N
\cs_new_eq:NN \clist_clear:c  \tl_clear:c
\cs_new_eq:NN \clist_gclear:N \tl_gclear:N
\cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(*End definition for* \clist_clear:N *and* \clist_clear:c *. These functions are documented on page* **??**.)

\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c

Once again a copy from the token list functions.

```
\cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \clist_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(*End definition for* \clist_clear_new:N *and* \clist_clear_new:c *. These functions are documented on page* **??**.)

\clist_set_eq:NN
\clist_set_eq:cN
\clist_set_eq:Nc
\clist_set_eq:cc
\clist_gset_eq:NN
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc

Once again, these are simple copies from the token list functions.

```
\cs_new_eq:NN \clist_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \clist_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \clist_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \clist_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* \clist_set_eq:NN *and others. These functions are documented on page* **??**.)

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
\clist_concat_aux:NNNN

Concatenating sequences is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
\cs_new_protected_nopar:Npn \clist_concat:NNN
  { \clist_concat_aux:NNNN \tl_set:Nx }
\cs_new_protected_nopar:Npn \clist_gconcat:NNN
  { \clist_concat_aux:NNNN \tl_gset:Nx }
\cs_new_protected:Npn \clist_concat_aux:NNNN #1#2#3#4
  {
    #1 #2
      {
        \exp_not:o #3
        \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
        \exp_not:o #4
      }
  }
\cs_generate_variant:Nn \clist_concat:NNN  { ccc }
\cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(*End definition for* `\clist_concat:NNN` *and* `\clist_concat:ccc`*. These functions are documented on page* ??.)

`\clist_if_exist_p:N`
`\clist_if_exist_p:c`
`\clist_if_exist:NTF`
`\clist_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \clist_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \clist_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \clist_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \clist_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \clist_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \clist_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \clist_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \clist_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\clist_if_exist:N` *and* `\clist_if_exist:c`*. These functions are documented on page* ??.)

194.2 Removing spaces around items

`\clist_trim_spaces_generic:nw`
`\clist_trim_spaces_generic_aux:w`
`\clist_trim_spaces_generic_aux_ii:nn`

Used as '`\clist_trim_spaces_generic:nw {`⟨*code*⟩`} \q_mark` ⟨*item*⟩ `,`' (including the comma). This expands to the ⟨*code*⟩, followed by a brace group containing the ⟨*item*⟩, with leading and trailing spaces removed. The calling function is responsible for inserting `\q_mark` in front of the ⟨*item*⟩, as well as testing for the end of the list. See `\tl_trim_spaces:n` for a partial explanation of what is happening here. We changed `\tl_trim_spaces_aux_iv:w` into `\clist_trim_spaces_generic_aux:w` compared to `\tl_trim_spaces:n`, and dropped a `\q_mark`, which is already included in the argument `##2`.

```
\cs_set:Npn \clist_tmp:w #1
  {
    \cs_new:Npn \clist_trim_spaces_generic:nw ##1 ##2 ,
      {
        \tl_trim_spaces_aux_i:w
        ##2
        \q_nil
        \q_mark #1 { }
        \q_mark \tl_trim_spaces_aux_ii:w
        \tl_trim_spaces_aux_iii:w
        #1 \q_nil
        \clist_trim_spaces_generic_aux:w
        \q_stop
        {##1}
      }
  }
\clist_tmp:w {~}
\cs_new:Npn \clist_trim_spaces_generic_aux:w #1 \q_nil #2 \q_stop
  { \exp_args:No \clist_trim_spaces_generic_aux_ii:nn { \use_none:n #1 } }
\cs_new:Npn \clist_trim_spaces_generic_aux_ii:nn #1 #2 { #2 {#1} }
```

(*End definition for* `\clist_trim_spaces_generic:nw`*. This function is documented on page* ??.)

\clist_trim_spaces:n
\clist_trim_spaces_aux:nn

The first argument of \clist_trim_spaces_aux:nn is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```
\cs_new:Npn \clist_trim_spaces:n #1
  {
    \clist_trim_spaces_generic:nw
      { \clist_trim_spaces_aux:nn { } }
      \q_mark #1 ,
    \q_recursion_tail, \q_recursion_stop
  }
\cs_new:Npn \clist_trim_spaces_aux:nn #1 #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \tl_if_empty:nTF {#2}
      {
        \clist_trim_spaces_generic:nw
          { \clist_trim_spaces_aux:nn {#1} } \q_mark
      }
      {
        #1 \exp_not:n {#2}
        \clist_trim_spaces_generic:nw
          { \clist_trim_spaces_aux:nn { , } } \q_mark
      }
  }
```

(*End definition for* \clist_trim_spaces:n*. This function is documented on page 115.*)

194.3 Adding data to comma lists

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:No
\clist_gset:Nx
\clist_gset:cn
\clist_gset:cV
\clist_gset:co
\clist_gset:cx

```
\cs_new_protected:Npn \clist_set:Nn #1#2
  { \tl_set:Nx #1 { \clist_trim_spaces:n {#2} } }
\cs_new_protected:Npn \clist_gset:Nn #1#2
  { \tl_gset:Nx #1 { \clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_set:Nn  { NV , No , Nx , c , cV , co , cx }
\cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
```

(*End definition for* \clist_set:Nn *and others. These functions are documented on page* ??.)

\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
\clist_put_left_aux:NNNn

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```
\cs_new_protected_nopar:Npn \clist_put_left:Nn
  { \clist_put_left_aux:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected_nopar:Npn \clist_gput_left:Nn
  { \clist_put_left_aux:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \clist_put_left_aux:NNNn #1#2#3#4
  {
    #2 \l_clist_internal_clist {#4}
    #1 #3 \l_clist_internal_clist #3
```

```
  }
\cs_generate_variant:Nn \clist_put_left:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_left:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_left:Nn {     NV , No , Nx }
\cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_left:Nn` *and others. These functions are documented on page* **??**.)

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\clist_put_right_aux:NNNn

```
\cs_new_protected_nopar:Npn \clist_put_right:Nn
  { \clist_put_right_aux:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected_nopar:Npn \clist_gput_right:Nn
  { \clist_put_right_aux:NNNn \clist_gconcat:NNN \clist_gset:Nn }
\cs_new_protected:Npn \clist_put_right_aux:NNNn #1#2#3#4
  {
    #2 \l_clist_internal_clist {#4}
    #1 #3 #3 \l_clist_internal_clist
  }
\cs_generate_variant:Nn \clist_put_right:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_right:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_right:Nn {     NV , No , Nx }
\cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_right:Nn` *and others. These functions are documented on page* **??**.)

194.4 Comma lists as stacks

\clist_get:NN
\clist_get:cN
\clist_get_aux:wN

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```
\cs_new_protected:Npn \clist_get:NN #1#2
  { \exp_after:wN \clist_get_aux:wN #1 , \q_stop #2 }
\cs_new_protected:Npn \clist_get_aux:wN #1 , #2 \q_stop #3
  { \tl_set:Nn #3 {#1} }
\cs_generate_variant:Nn \clist_get:NN { c }
```

(*End definition for* `\clist_get:NN` *and* `\clist_get:cN`*. These functions are documented on page* **??**.)

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
\clist_pop_aux:NNN
\clist_pop_aux:NwNNN
\clist_pop_aux:wNNN

The aim here is to get the popped item as `#1` in the auxiliary, with `#2` containing either the remainder of the list *or* `\q_nil` if there were insufficient items. That keeps the number of auxiliary functions down.

```
\cs_new_protected_nopar:Npn \clist_pop:NN
  { \clist_pop_aux:NNN \tl_set:Nf }
\cs_new_protected_nopar:Npn \clist_gpop:NN
  { \clist_pop_aux:NNN \tl_gset:Nf }
\cs_new_protected:Npn \clist_pop_aux:NNN #1#2#3
  {
    \exp_after:wN \clist_pop_aux:wNNN #2 , \q_nil \q_stop #1#2#3
  }
\cs_new_protected:Npn \clist_pop_aux:wNNN #1 , #2 \q_stop #3#4#5
  {
    \tl_set:Nn #5 {#1}
```

```
    \quark_if_nil:nTF {#2}
      { #3 #4 { } }
      { #3 #4 { \clist_pop_aux:w \exp_stop_f: #2 } }
  }
\cs_new_protected:Npn \clist_pop_aux:w #1 , \q_nil {#1}
\cs_generate_variant:Nn \clist_pop:NN  { c }
\cs_generate_variant:Nn \clist_gpop:NN { c }
```

(*End definition for* `\clist_pop:NN` *and* `\clist_pop:cN`*. These functions are documented on page* **??**.)

`\clist_push:Nn` `\clist_push:NV` `\clist_push:No` `\clist_push:Nx` `\clist_push:cn` `\clist_push:cV` `\clist_push:co` `\clist_push:cx` `\clist_gpush:Nn` `\clist_gpush:NV` `\clist_gpush:No` `\clist_gpush:Nx` `\clist_gpush:cn` `\clist_gpush:cV` `\clist_gpush:co` `\clist_gpush:cx`

Pushing to a sequence is the same as adding on the left.

```
\cs_new_eq:NN \clist_push:Nn  \clist_put_left:Nn
\cs_new_eq:NN \clist_push:NV  \clist_put_left:NV
\cs_new_eq:NN \clist_push:No  \clist_put_left:No
\cs_new_eq:NN \clist_push:Nx  \clist_put_left:Nx
\cs_new_eq:NN \clist_push:cn  \clist_put_left:cn
\cs_new_eq:NN \clist_push:cV  \clist_put_left:cV
\cs_new_eq:NN \clist_push:co  \clist_put_left:co
\cs_new_eq:NN \clist_push:cx  \clist_put_left:cx
\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
```

(*End definition for* `\clist_push:Nn` *and others. These functions are documented on page* **??**.)

194.5 Using comma lists

`\clist_use:N` `\clist_use:c`

The approach is the same as for `\tl_use:N`.

```
\cs_new_eq:NN \clist_use:N \tl_use:N
\cs_new_eq:NN \clist_use:c \tl_use:c
```

(*End definition for* `\clist_use:N` *and* `\clist_use:c`*. These functions are documented on page* **??**.)

194.6 Modifying comma lists

`\l_clist_internal_remove_clist`

An internal comma list for the removal routines.

```
\clist_new:N \l_clist_internal_remove_clist
```

(*End definition for* `\l_clist_internal_remove_clist`*. This variable is documented on page* **??**.)

`\clist_remove_duplicates:N` `\clist_remove_duplicates:c` `\clist_gremove_duplicates:N` `\clist_gremove_duplicates:c` `\clist_remove_duplicates_aux:NN`

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \clist_remove_duplicates:N
  { \clist_remove_duplicates_aux:NN \clist_set_eq:NN }
\cs_new_protected:Npn \clist_gremove_duplicates:N
  { \clist_remove_duplicates_aux:NN \clist_gset_eq:NN }
\cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2
  {
```

```
    \clist_clear:N \l_clist_internal_remove_clist
    \clist_map_inline:Nn #2
      {
        \clist_if_in:NnF \l_clist_internal_remove_clist {##1}
          { \clist_put_right:Nn \l_clist_internal_remove_clist {##1} }
      }
    #1 #2 \l_clist_internal_remove_clist
  }
\cs_generate_variant:Nn \clist_remove_duplicates:N  { c }
\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(*End definition for* `\clist_remove_duplicates:N` *and* `\clist_remove_duplicates:c`*. These functions are documented on page* **??**.)

`\clist_remove_all:Nn`
`\clist_remove_all:cn`
`\clist_gremove_all:Nn`
`\clist_gremove_all:cn`
`\clist_remove_all_aux:NNn`
`\clist_remove_all_aux:w`
`\clist_remove_all_aux:`

The method used here is very similar to `\tl_replace_all:Nnn`. Build a function delimited by the ⟨*item*⟩ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the ⟨*item*⟩. The loop is controlled by the argument grabbed by `\clist_remove_all_aux:w`: when the item was found, the `\q_mark` delimiter used is the one inserted by `\clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final ⟨*item*⟩ is grabbed, and the argument of `\clist_tmp:w` contains `\q_mark`: in that case, `\clist_remove_all_aux:w` removes the second `\q_mark` (inserted by `\clist_tmp:w`), and lets `\use_none_delimit_by_q_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn't remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```
\cs_new_protected:Npn \clist_remove_all:Nn
  { \clist_remove_all_aux:NNn \tl_set:Nx }
\cs_new_protected:Npn \clist_gremove_all:Nn
  { \clist_remove_all_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \clist_remove_all_aux:NNn #1#2#3
  {
    \cs_set:Npn \clist_tmp:w ##1 , #3 ,
      {
        ##1
        , \q_mark , \use_none_delimit_by_q_stop:w ,
        \clist_remove_all_aux:
      }
    #1 #2
      {
        \exp_after:wN \clist_remove_all_aux:
        #2 , \q_mark , #3 , \q_stop
      }
    \clist_if_empty:NF #2
      {
        #1 #2
          {
```

```
            \exp_args:No \exp_not:o
              { \exp_after:wN \use_none:n #2 }
          }
      }
  }
\cs_new:Npn \clist_remove_all_aux:
  { \exp_after:wN \clist_remove_all_aux:w \clist_tmp:w , }
\cs_new:Npn \clist_remove_all_aux:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
\cs_generate_variant:Nn \clist_remove_all:Nn  { c }
\cs_generate_variant:Nn \clist_gremove_all:Nn { c }
```

(*End definition for* `\clist_remove_all:Nn` *and* `\clist_remove_all:cn`*. These functions are documented on page* **??**.)

194.7 Comma list conditionals

`\clist_if_empty_p:N`
`\clist_if_empty_p:c`
`\clist_if_empty:NTF`
`\clist_if_empty:cTF`

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }
```

(*End definition for* `\clist_if_empty:N` *and* `\clist_if_empty:c`*. These functions are documented on page* **??**.)

`\clist_if_eq_p:NN`
`\clist_if_eq_p:Nc`
`\clist_if_eq_p:cN`
`\clist_if_eq_p:cc`
`\clist_if_eq:NNTF`
`\clist_if_eq:NcTF`
`\clist_if_eq:cNTF`
`\clist_if_eq:ccTF`

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc { p , T , F , TF }
```

(*End definition for* `\clist_if_eq:NN` *and others. These functions are documented on page* **??**.)

`\clist_if_in:NnTF`
`\clist_if_in:NVTF`
`\clist_if_in:NoTF`
`\clist_if_in:cnTF`
`\clist_if_in:cVTF`
`\clist_if_in:coTF`
`\clist_if_in:nnTF`
`\clist_if_in:nVTF`
`\clist_if_in:noTF`
`\clist_if_in_return:nn`

See description of the `\tl_if_in:Nn` function for details. We simply surround the comma list, and the item, with commas.

```
\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T  , F , TF }
  {
    \exp_args:No \clist_if_in_return:nn #1 {#2}
  }
\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T  , F , TF }
  {
    \clist_set:Nn \l_clist_internal_clist {#1}
    \exp_args:No \clist_if_in_return:nn \l_clist_internal_clist {#2}
  }
\cs_new_protected:Npn \clist_if_in_return:nn #1#2
  {
    \cs_set:Npn \clist_tmp:w ##1 ,#2, { }
    \tl_if_empty:oTF
      { \clist_tmp:w ,#1, {} {} ,#2, }
      { \prg_return_false: } { \prg_return_true: }
  }
\cs_generate_variant:Nn \clist_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnT  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnF  {     NV , No }
```

```
\cs_generate_variant:Nn \clist_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:nnT  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnF  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnTF {     nV , no }
```

(*End definition for `\clist_if_in:Nn` and others. These functions are documented on page* **??**.)

194.8 Mapping to comma lists

`\clist_map_function:NN`
`\clist_map_function:cN`
`\clist_map_function_aux:Nw`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `\clist_map_function_aux:Nw` is used directly in `\clist_map_inline:Nn`. Change with care.

```
\cs_new:Npn \clist_map_function:NN #1#2
  {
    \clist_if_empty:NF #1
      {
        \exp_last_unbraced:NNo \clist_map_function_aux:Nw #2 #1
          , \q_recursion_tail ,
        \prg_break_point:n { }
      }
  }
\cs_new:Npn \clist_map_function_aux:Nw #1#2 ,
  {
    \quark_if_recursion_tail_break:n {#2}
    #1 {#2}
    \clist_map_function_aux:Nw #1
  }
\cs_generate_variant:Nn \clist_map_function:NN { c }
```

(*End definition for `\clist_map_function:NN` and `\clist_map_function:cN`. These functions are documented on page* **??**.)

`\clist_map_function:nN`
`\clist_map_function_n_aux:Nn`
`\clist_map_aux_unbrace:Nw`

The `n`-type mapping function is a bit more awkward, since spaces mustbe trimmed from each item. Space trimming is again based on `\clist_trim_spaces_generic:nw`. The auxiliary `\clist_map_function_n_aux:Nn` receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by `\clist_map_aux_unbrace:Nw`.

```
\cs_new:Npn \clist_map_function:nN #1#2
  {
    \clist_trim_spaces_generic:nw { \clist_map_function_n_aux:Nn #2 }
    \q_mark #1, \q_recursion_tail,
    \prg_break_point:n { }
  }
\cs_new:Npn \clist_map_function_n_aux:Nn #1 #2
  {
```

```
    \quark_if_recursion_tail_break:n {#2}
    \tl_if_empty:nF {#2} { \clist_map_aux_unbrace:Nw #1 #2, }
    \clist_trim_spaces_generic:nw { \clist_map_function_n_aux:Nn #1 }
    \q_mark
  }
\cs_new:Npn \clist_map_aux_unbrace:Nw #1 #2, { #1 {#2} }
```

(*End definition for* `\clist_map_function:nN`*. This function is documented on page* **??**.)

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

Inline mapping is done by creating a suitable function "on the fly": this is done globally to avoid any issues with TEX's groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don't need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```
\cs_new_protected:Npn \clist_map_inline:Nn #1#2
  {
    \clist_if_empty:NF #1
      {
        \int_gincr:N \g_prg_map_int
        \cs_gset:cpn { clist_map_ \int_use:N \g_prg_map_int :n } ##1 {#2}
        \exp_last_unbraced:Nco \clist_map_function_aux:Nw
          { clist_map_ \int_use:N \g_prg_map_int :n }
          #1 , \q_recursion_tail ,
        \prg_break_point:n { \int_gdecr:N \g_prg_map_int }
      }
  }
\cs_new_protected:Npn \clist_map_inline:nn #1
  {
    \clist_set:Nn \l_clist_internal_clist {#1}
    \clist_map_inline:Nn \l_clist_internal_clist
  }
\cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(*End definition for* `\clist_map_inline:Nn` *and* `\clist_map_inline:cn`*. These functions are documented on page* **??**.)

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`
`\clist_map_variable_aux:Nnw`

As for other comma-list mappings, filter out the case of an empty list. Same approach as `\clist_map_function:Nn`, additionally we store each item in the given variable. As for inline mappings, space trimming for the `n` variant is done by storing the comma list in a variable.

```
\cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
  {
    \clist_if_empty:NF #1
      {
        \exp_args:Nno \use:nn
          { \clist_map_variable_aux:Nnw #2 {#3} }
          #1
          , \q_recursion_tail , \q_recursion_stop
        \prg_break_point:n { }
      }
```

```
  }
\cs_new_protected:Npn \clist_map_variable:nNn #1
  {
    \clist_set:Nn \l_clist_internal_clist {#1}
    \clist_map_variable:NNn \l_clist_internal_clist
  }
\cs_new_protected:Npn \clist_map_variable_aux:Nnw #1#2#3,
  {
    \tl_set:Nn #1 {#3}
    \quark_if_recursion_tail_stop:N #1
    \use:n {#2}
    \clist_map_variable_aux:Nnw #1 {#2}
  }
\cs_generate_variant:Nn \clist_map_variable:NNn { c }
```

(*End definition for `\clist_map_variable:NNn` and `\clist_map_variable:cNn`. These functions are documented on page* **??**.)

`\clist_map_break:`
`\clist_map_break:n`

The break statements are simply copies.

```
\cs_new_eq:NN \clist_map_break:  \prg_map_break:
\cs_new_eq:NN \clist_map_break:n \prg_map_break:n
```

(*End definition for `\clist_map_break:`. This function is documented on page 113.*)

194.9 Viewing comma lists

`\clist_show:N`
`\clist_show:c`
`\clist_show:n`

Apply the general `\msg_aux_show:Nnx`. In the case of an `n`-type comma-list, first store it in a scratch variable, then show that variable, omitting its name from the 4-th argument.

```
\cs_new_protected:Npn \clist_show:N #1
  {
    \msg_aux_show:Nnx
      #1
      { clist }
      { \clist_map_function:NN #1 \msg_aux_show:n }
  }
\cs_new_protected:Npn \clist_show:n #1
  {
    \clist_set:Nn \l_clist_internal_clist {#1}
    \msg_aux_show:Nnx
      \l_clist_internal_clist
      { clist }
      { \clist_map_function:NN \l_clist_internal_clist \msg_aux_show:n }
  }
\cs_generate_variant:Nn \clist_show:N { c }
```

(*End definition for `\clist_show:N` and `\clist_show:c`. These functions are documented on page 114.*)

194.10 Scratch comma lists

`\l_tmpa_clist`
`\l_tmpb_clist`
`\g_tmpa_clist`
`\g_tmpb_clist`

Temporary comma list variables.

```
\clist_new:N \l_tmpa_clist
```

```
\clist_new:N \l_tmpb_clist
\clist_new:N \g_tmpa_clist
\clist_new:N \g_tmpb_clist
```

(*End definition for* `\l_tmpa_clist` *and* `\l_tmpb_clist`*. These functions are documented on page 114.*)

194.11 Experimental functions

`\clist_length:N`
`\clist_length:c`
`\clist_length:n`
`\clist_length_aux:w`

Counting the items in a comma list is done using the same approach as for other length functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics. In the case of an `n`-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not `{}`, hence the extra spaces).

```
\cs_new:Npn \clist_length:N #1
  {
    \int_eval:n
      {
        0
        \clist_map_function:NN #1 \clist_length_aux:n
      }
  }
\cs_new:Npn \clist_length_aux:n #1 { +1 }
\cs_new:Npx \clist_length:n #1
  {
    \exp_not:N \int_eval:n
      {
        0
        \exp_not:N \clist_length_n_aux:w \c_space_tl
        #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
\cs_new:Npx \clist_length_n_aux:w #1 ,
  {
    \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
    \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
    \exp_not:N \clist_length_n_aux:w \c_space_tl
  }
\cs_generate_variant:Nn \clist_length:N { c }
```

(*End definition for* `\clist_length:N` *and* `\clist_length:c`*. These functions are documented on page* ??.)

`\clist_item:Nn`
`\clist_item:cn`
`\clist_item_aux:nnNn`
`\clist_item_N_loop:nw`

To avoid needing to test the end of the list at each step, we first compute the ⟨*length*⟩ of the list. If the item number is less than $-\langle length\rangle$ or more than $\langle length\rangle - 1$, the result is empty. If it is negative, but not less than $-\langle length\rangle$, add the ⟨*length*⟩ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached zero, otherwise, decrease the counter and repeat.

```
\cs_new:Npn \clist_item:Nn #1#2
  {
    \exp_args:Nfo \clist_item_aux:nnNn
```

```
      { \clist_length:N #1 }
      #1
      \clist_item_N_loop:nw
      {#2}
  }
\cs_new:Npn \clist_item_aux:nnNn #1#2#3#4
  {
    \int_compare:nNnTF {#4} < \c_zero
      {
        \int_compare:nNnTF {#4} < { - #1 }
          { \use_none_delimit_by_q_stop:w }
          { \exp_args:Nf #3 { \int_eval:n { #4 + #1 } } }
      }
      {
        \int_compare:nNnTF {#4} < {#1}
          { #3 {#4} }
          { \use_none_delimit_by_q_stop:w }
      }
    #2, \q_stop
  }
\cs_new:Npn \clist_item_N_loop:nw #1 #2,
  {
    \int_compare:nNnTF {#1} = \c_zero
      { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
      { \exp_args:Nf \clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \clist_item:Nn { c }
```

(*End definition for* `\clist_item:Nn` *and* `\clist_item:cn`*. These functions are documented on page* **??***.*)

`\clist_item:nn`
`\clist_item_n_aux:nw`
`\clist_item_n_loop:nw`
`\clist_item_n_end:n`
`\clist_item_n_strip:w`

This starts in the same way as `\clist_item:Nn` by checking the length of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```
\cs_new:Npn \clist_item:nn #1#2
  {
    \exp_args:Nf \clist_item_aux:nnNn
      { \clist_length:n {#1} }
      {#1}
      \clist_item_n_aux:nw
      {#2}
  }
\cs_new:Npn \clist_item_n_aux:nw #1
  { \clist_item_n_loop:nw {#1} \prg_do_nothing: }
\cs_new:Npn \clist_item_n_loop:nw #1 #2,
  {
    \exp_args:No \tl_if_blank:nTF {#2}
      { \clist_item_n_loop:nw {#1} \prg_do_nothing: }
      {
        \int_compare:nNnTF {#1} = \c_zero
          { \exp_args:No \clist_item_n_end:n {#2} }
```

```
        {
          \exp_args:Nf \clist_item_n_loop:nw
            { \int_eval:n { #1 - 1 } }
            \prg_do_nothing:
        }
    }
  }
\cs_new:Npn \clist_item_n_end:n #1 #2 \q_stop
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \clist_item_n_strip:w
    \tl_trim_spaces:n {#1} ,
  }
\cs_new:Npn \clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(*End definition for* `\clist_item:nn`*. This function is documented on page* **??**.)

`\clist_set_from_seq:NN`
`\clist_set_from_seq:cN`
`\clist_set_from_seq:Nc`
`\clist_set_from_seq:cc`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:cN`
`\clist_gset_from_seq:Nc`
`\clist_gset_from_seq:cc`
`\clist_set_from_seq_aux:NNNN`
`\clist_wrap_item:n`

Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with `\exp_not:n`, and a comma. Items which contain a comma or a space are surrounded by an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\cs_new_protected:Npn \clist_set_from_seq:NN
  { \clist_set_from_seq_aux:NNNN \clist_clear:N  \tl_set:Nx  }
\cs_new_protected:Npn \clist_gset_from_seq:NN
  { \clist_set_from_seq_aux:NNNN \clist_gclear:N \tl_gset:Nx }
\cs_new_protected:Npn \clist_set_from_seq_aux:NNNN #1#2#3#4
  {
    \seq_if_empty:NTF #4
      { #1 #3 }
      {
        #2 #3
          {
            \exp_last_unbraced:Nf \use_none:n
              { \seq_map_function:NN #4 \clist_wrap_item:n }
          }
      }
  }
\cs_new:Npn \clist_wrap_item:n #1
  {
    ,
    \tl_if_empty:oTF { \clist_set_from_seq_aux:w #1 ~ , #1 ~ }
      { \exp_not:n   {#1}   }
      { \exp_not:n { {#1} } }
  }
\cs_new:Npn \clist_set_from_seq_aux:w #1 , #2 ~ { }
\cs_generate_variant:Nn \clist_set_from_seq:NN  {     Nc }
\cs_generate_variant:Nn \clist_set_from_seq:NN  { c , cc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN {     Nc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(*End definition for* `\clist_set_from_seq:NN` *and others. These functions are documented on page* **??**.)

\clist_const:Nn
\clist_const:cn
\clist_const:Nx
\clist_const:cx

Creating and initializing a constant comma list is done in a way similar to `\clist_set:Nn` and `\clist_gset:Nn`, being careful to strip spaces.

```
\cs_new_protected:Npn \clist_const:Nn #1#2
  { \tl_const:Nx #1 { \clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(*End definition for* `\clist_const:Nn` *and others. These functions are documented on page* **??**.)

\clist_if_empty_p:n
\clist_if_empty:nTF
\clist_if_empty_n_aux:w
\clist_if_empty_n_aux:wNw

As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces (besides, this particular variant of the emptyness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary will grab `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.

```
\prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
  {
    \clist_if_empty_n_aux:w ? #1
    , \q_mark \prg_return_false:
    , \q_mark \prg_return_true:
    \q_stop
  }
\cs_new:Npn \clist_if_empty_n_aux:w #1 ,
  {
    \tl_if_empty:oTF { \use_none:nn #1 ? }
      { \clist_if_empty_n_aux:w ? }
      { \clist_if_empty_n_aux:wNw }
  }
\cs_new:Npn \clist_if_empty_n_aux:wNw #1 \q_mark #2#3 \q_stop {#2}
```

(*End definition for* `\clist_if_empty:n`*. These functions are documented on page 115.*)

194.12 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

\clist_top:NN
\clist_top:cN

These are old stack functions.

```
⟨*deprecated⟩
\cs_new_eq:NN \clist_top:NN \clist_get:NN
\cs_new_eq:NN \clist_top:cN \clist_get:cN
⟨/deprecated⟩
```

(*End definition for* `\clist_top:NN` *and* `\clist_top:cN`*. These functions are documented on page* **??**.)

\clist_remove_element:Nn
\clist_gremove_element:Nn

An older name for `\clist_remove_all:Nn`.

```
⟨*deprecated⟩
\cs_new_eq:NN \clist_remove_element:Nn  \clist_remove_all:Nn
\cs_new_eq:NN \clist_gremove_element:Nn \clist_gremove_all:Nn
⟨/deprecated⟩
```

(*End definition for* `\clist_remove_element:Nn` *and* `\clist_gremove_element:Nn`*. These functions are documented on page* **??**.)

\clist_display:N
\clist_display:c

An older name for \clist_show:N.

```
⟨*deprecated⟩
\cs_new_eq:NN \clist_display:N \clist_show:N
\cs_new_eq:NN \clist_display:c \clist_show:c
⟨/deprecated⟩
```

(*End definition for* \clist_display:N *and* \clist_display:c*. These functions are documented on page* ??.)

Deprecated on 2011-09-05, for removal by 2011-12-31.

\clist_trim_spaces:N
\clist_trim_spaces:c
\clist_gtrim_spaces:N
\clist_gtrim_spaces:c

Since clist items are now always stripped from their surrounding spaces, it is redundant to provide these functions. The \clist_trim_spaces:n function is now internal, deprecated for use outside the kernel.

```
⟨*deprecated⟩
\cs_new_protected:Npn \clist_trim_spaces:N  #1 { \clist_set:No  #1 {#1} }
\cs_new_protected:Npn \clist_gtrim_spaces:N #1 { \clist_gset:No #1 {#1} }
\cs_generate_variant:Nn \clist_trim_spaces:N  { c }
\cs_generate_variant:Nn \clist_gtrim_spaces:N { c }
⟨/deprecated⟩
```

(*End definition for* \clist_trim_spaces:N *and others. These functions are documented on page* ??.)

```
⟨/initex | package⟩
```

195 l3prop implementation

The following test files are used for this code: m3prop001.

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

A property list is a macro whose top-level expansion is for the form "\q_prop ⟨key_0⟩ \q_prop {⟨$value_0$⟩} \q_prop . . . \q_prop ⟨key_{n-1}⟩ \q_prop {⟨$value_{n-1}$⟩} \q_prop". The trailing \q_prop is always present for performance reasons: this means that empty property lists are not actually empty.

\q_prop

A private quark is used as a marker between entries.

```
\quark_new:N \q_prop
```

(*End definition for* \q_prop*. This function is documented on page* 121.)

\c_empty_prop

An empty prop contains exactly one \q_prop.

```
\tl_const:Nn \c_empty_prop { \q_prop }
```

(*End definition for* \c_empty_prop*. This variable is documented on page* 121.)

195.1 Allocation and initialisation

\prop_new:N
\prop_new:c

Internally, property lists are token lists, but an empty prop is not an empty tl, so we need to do things by hand.

```
\cs_new_protected:Npn \prop_new:N #1 { \cs_new_eq:NN  #1  \c_empty_prop }
\cs_new_protected:Npn \prop_new:c #1 { \cs_new_eq:cN {#1} \c_empty_prop }
```

(*End definition for* `\prop_new:N` *and* `\prop_new:c`*. These functions are documented on page* **??**.)

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

The same idea for clearing

```
\cs_new_protected:Npn \prop_clear:N #1  { \cs_set_eq:NN   #1  \c_empty_prop }
\cs_new_protected:Npn \prop_clear:c #1  { \cs_set_eq:cN  {#1} \c_empty_prop }
\cs_new_protected:Npn \prop_gclear:N #1 { \cs_gset_eq:NN  #1  \c_empty_prop }
\cs_new_protected:Npn \prop_gclear:c #1 { \cs_gset_eq:cN {#1} \c_empty_prop }
```

(*End definition for* `\prop_clear:N` *and* `\prop_clear:c`*. These functions are documented on page* **??**.)

\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c

Once again a simple copy from the token list functions.

```
\cs_new_protected:Npn \prop_clear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_clear_new:N { c }
\cs_new_protected:Npn \prop_gclear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(*End definition for* `\prop_clear_new:N` *and* `\prop_clear_new:c`*. These functions are documented on page* **??**.)

\prop_set_eq:NN
\prop_set_eq:cN
\prop_set_eq:Nc
\prop_set_eq:cc
\prop_gset_eq:NN
\prop_gset_eq:cN
\prop_gset_eq:Nc
\prop_gset_eq:cc

Once again, these are simply copies from the token list functions.

```
\cs_new_eq:NN \prop_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \prop_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \prop_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \prop_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\prop_set_eq:NN` *and others. These functions are documented on page* **??**.)

195.2 Accessing data in property lists

\prop_split:NnTF
\prop_split_aux:NnTF
\prop_split_aux:nnnn
\prop_split_aux:w

This function is used by most of the module, and hence must be fast. The aim here is to split a property list at a given key into the part before the key–value pair, the value associated with the key and the part after the key–value pair. To do this, the key is first detokenized (to avoid repeatedly doing this), then a delimited function is constructed to match the key. It will match `\q_prop` ⟨*detokenized key*⟩ `\q_prop {`⟨*value*⟩`}` ⟨*extra argument*⟩, effectively separating an ⟨*extract1*⟩ before the key in the property list and an ⟨*extract2*⟩ after the key.

If the key is present in the property list, then ⟨*extra argument*⟩ is simply `\q_prop`, and `\prop_split_aux:nnnn` will gobble this and the false branch (`#4`), leaving the correct code on the input stream. More precisely, it leaves the user code (true branch), followed by three groups, {⟨*extract*$_1$⟩} {⟨*value*⟩} {⟨*extract*$_2$⟩}. In order for ⟨*extract1*⟩⟨*extract2*⟩ to be a well-formed property list, ⟨*extract1*⟩ has a leading and trailing `\q_prop`, retaining exactly the structure of a property list, while ⟨*extract2*⟩ omits the leading `\q_prop`.

If the key is not there, then ⟨*extra argument*⟩ is `? \use_ii:nn { }`, and `\prop_split_aux:nnnn ? \u` removes the three brace groups that just follow. Then `\use_ii:nn` removes the true branch, leaving the false branch, with no trailing material.

```
\cs_new_protected:Npn \prop_split:NnTF #1#2
  { \exp_args:NNo \prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
\cs_new_protected:Npn \prop_split_aux:NnTF #1#2
  {
    \cs_set_protected:Npn \prop_split_aux:w
      ##1 \q_prop #2 \q_prop ##2 ##3 ##4 \q_mark ##5 \q_stop
      { \prop_split_aux:nnnn ##3 { {##1 \q_prop } {##2} {##4} } }
    \exp_after:wN \prop_split_aux:w #1 \q_mark
        \q_prop #2 \q_prop { } { ? \use_ii:nn { } } \q_mark \q_stop
  }
\cs_new:Npn \prop_split_aux:nnnn #1#2#3#4 { #3 #2 }
\cs_new_protected:Npn \prop_split_aux:w { }
```

(*End definition for* `\prop_split:NnTF`*. This function is documented on page* *121**.*)

`\prop_split:Nnn`

The goal here is to provide a common interface for both true and false branches of `\prop_split:NnTF`. In both cases, the code given by the user will be placed in front of three brace groups, {⟨*extract*$_1$⟩} {⟨*value*⟩} {⟨*extract*$_2$⟩}. If the key was missing from the property list, then ⟨*extract1*⟩ is the full property list, ⟨*value*⟩ is `\q_no_value`, and ⟨*extract2*⟩ is empty. Otherwise, ⟨*extract1*⟩ is the part of the property list before the ⟨*key*⟩, and has the structure of a property list, ⟨*value*⟩ is the value corresponding to the ⟨*key*⟩, and ⟨*extract2*⟩ (the part after the ⟨*key*⟩) is missing the leading `\q_prop`.

```
\cs_new_protected:Npn \prop_split:Nnn #1#2#3
  {
    \prop_split:NnTF #1 {#2}
    {#3}
    { \exp_args:Nno \use:n {#3} {#1} { \q_no_value } { } }
  }
```

(*End definition for* `\prop_split:Nnn`*. This function is documented on page* *121**.*)

`\prop_del:Nn`
`\prop_del:NV`
`\prop_del:cn`
`\prop_del:cV`
`\prop_gdel:Nn`
`\prop_gdel:NV`
`\prop_gdel:cn`
`\prop_gdel:cV`
`\prop_del_aux:NNnnn`

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```
\cs_new_protected:Npn \prop_del:Nn #1#2
  { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_set:Nn #1 } { } }
\cs_new_protected:Npn \prop_gdel:Nn #1#2
  { \prop_split:NnTF #1 {#2} { \prop_del_aux:NNnnn \tl_gset:Nn #1 } { } }
\cs_new_protected:Npn \prop_del_aux:NNnnn #1#2#3#4#5
  { #1 #2 { #3 #5 } }
\cs_generate_variant:Nn \prop_del:Nn  {     NV }
\cs_generate_variant:Nn \prop_del:Nn  { c , cV }
\cs_generate_variant:Nn \prop_gdel:Nn {     NV }
\cs_generate_variant:Nn \prop_gdel:Nn { c , cV }
```

(*End definition for* `\prop_del:Nn` *and others. These functions are documented on page* **??***.*)

\prop_get:NnN
\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:NoN
\prop_get_aux:Nnnn

Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```
\cs_new_protected:Npn \prop_get:NnN #1#2#3
  {
    \prop_split:NnTF #1 {#2}
      { \prop_get_aux:Nnnn #3 }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_get_aux:Nnnn #1#2#3#4
  { \tl_set:Nn #1 {#3} }
\cs_generate_variant:Nn \prop_get:NnN {     NV , No }
\cs_generate_variant:Nn \prop_get:NnN { c , cV , co }
```

(*End definition for* `\prop_get:NnN` *and others. These functions are documented on page* **??**.)

\prop_pop:NnN
\prop_pop:NoN
\prop_pop:cnN
\prop_pop:coN
\prop_gpop:NnN
\prop_gpop:NoN
\prop_gpop:cnN
\prop_gpop:coN
\prop_pop_aux:NNNnnn

Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```
\cs_new_protected:Npn \prop_pop:NnN #1#2#3
  {
    \prop_split:NnTF #1 {#2}
      { \prop_pop_aux:NNNnnn \tl_set:Nn #1 #3 }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_gpop:NnN #1#2#3
  {
    \prop_split:NnTF #1 {#2}
      { \prop_pop_aux:NNNnnn \tl_gset:Nn #1 #3 }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_pop_aux:NNNnnn #1#2#3#4#5#6
  {
    \tl_set:Nn #3 {#5}
    #1 #2 { #4 #6 }
  }
\cs_generate_variant:Nn \prop_pop:NnN  {     No }
\cs_generate_variant:Nn \prop_pop:NnN  { c , co }
\cs_generate_variant:Nn \prop_gpop:NnN {     No }
\cs_generate_variant:Nn \prop_gpop:NnN { c , co }
```

(*End definition for* `\prop_pop:NnN` *and others. These functions are documented on page* **??**.)

\prop_put:Nnn
\prop_put:NnV
\prop_put:Nno
\prop_put:Nnx
\prop_put:NVn
\prop_put:NVV
\prop_put:Non
\prop_put:Noo
\prop_put:cnn
\prop_put:cnV
\prop_put:cno
\prop_put:cnx
\prop_put:cVn
\prop_put:cVV
\prop_put:con
\prop_put:coo
\prop_gput:Nnn
\prop_gput:NnV

Putting a key–value pair in a property list starts by splitting to remove any existing value. The property list is then reconstructed with the two remaining parts `#5` and `#7` first, followed by the new or updated entry.

```
\cs_new_protected:Npn \prop_put:Nnn { \prop_put_aux:NNnn \tl_set:Nx }
\cs_new_protected:Npn \prop_gput:Nnn { \prop_put_aux:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \prop_put_aux:NNnn #1#2#3#4
  {
    \prop_split:Nnn #2 {#3} { \prop_put_aux:NNnnnn #1 #2 {#3} {#4} }
```

```
  }
\cs_new_protected:Npn \prop_put_aux:NNnnnnn #1#2#3#4#5#6#7
  {
    #1 #2
      {
        \exp_not:n { #5 #7 }
        \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop }
      }
  }
\cs_generate_variant:Nn \prop_put:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_put:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
\cs_generate_variant:Nn \prop_gput:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_gput:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
```

(*End definition for* `\prop_put:Nnn` *and others. These functions are documented on page* **??**.)

`\prop_put_if_new:Nnn`
`\prop_put_if_new:cnn`
`\prop_gput_if_new:Nnn`
`\prop_gput_if_new:cnn`

Adding conditionally also splits. If the key is already present, the three brace groups given by `\prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```
\cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
  { \prop_put_if_new_aux:NNnn \tl_put_right:Nx }
\cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
  { \prop_put_if_new_aux:NNnn \tl_gput_right:Nx }
\cs_new_protected:Npn \prop_put_if_new_aux:NNnn #1#2#3#4
  {
    \prop_split:NnTF #2 {#3}
      { \use_none:nnn }
      {
        #1 #2
          { \tl_to_str:n {#3} \exp_not:n { \q_prop {#4} \q_prop } }
      }
  }
\cs_generate_variant:Nn \prop_put_if_new:Nnn  { c }
\cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }
```

(*End definition for* `\prop_put_if_new:Nnn` *and* `\prop_put_if_new:cnn`. *These functions are documented on page* **??**.)

195.3 Property list conditionals

`\prop_if_exist_p:N`
`\prop_if_exist_p:c`
`\prop_if_exist:NTF`
`\prop_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \prop_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \prop_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \prop_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \prop_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \prop_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \prop_if_exist:cT  \cs_if_exist:cT
```

```
\cs_new_eq:NN \prop_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \prop_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\prop_if_exist:N` *and* `\prop_if_exist:c`*. These functions are documented on page* **??**.)

`\prop_if_empty_p:N`
`\prop_if_empty_p:c`
`\prop_if_empty:NTF`
`\prop_if_empty:cTF`

The test here uses `\c_empty_prop` as it is not really empty!

```
\prg_new_conditional:Npnn \prop_if_empty:N #1 { p, T , F , TF }
  {
    \if_meaning:w #1 \c_empty_prop
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \prop_if_empty_p:N {c}
\cs_generate_variant:Nn \prop_if_empty:NTF {c}
\cs_generate_variant:Nn \prop_if_empty:NT  {c}
\cs_generate_variant:Nn \prop_if_empty:NF  {c}
```

(*End definition for* `\prop_if_empty:N` *and* `\prop_if_empty:c`*. These functions are documented on page* **??**.)

`\prop_if_in_p:Nn`
`\prop_if_in_p:NV`
`\prop_if_in_p:No`
`\prop_if_in_p:cn`
`\prop_if_in_p:cV`
`\prop_if_in_p:co`
`\prop_if_in:NnTF`
`\prop_if_in:NVTF`
`\prop_if_in:NoTF`
`\prop_if_in:cnTF`
`\prop_if_in:cVTF`
`\prop_if_in:coTF`
`\prop_if_in_aux:nwn`
`\prop_if_in_aux:N`

Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```
\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
  {
    \prop_split:NnTF #1 {#2}
      {
        \prg_return_true:
        \use_none:nnn
      }
    { \prg_return_false: }
  }
```

but `\prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq:xx`, which is expandable. To terminate the mapping, we add the key that is search for at the end of the property list. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq:xx`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. When ending, we test the next token: it is either `\q_prop` or `\q_recursion_tail` in the case of a missing key. Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```
\prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
  {
    \exp_last_unbraced:Noo \prop_if_in_aux:nwn
      { \tl_to_str:n {#2} } #1
      \tl_to_str:n {#2} \q_prop { }
```

```
    \q_recursion_tail
    \prg_break_point:n { }
  }
\cs_new:Npn \prop_if_in_aux:nwn #1 \q_prop #2 \q_prop #3
  {
    \str_if_eq:xxTF {#1} {#2}
      { \prop_if_in_aux:N }
      { \prop_if_in_aux:nwn {#1} }
  }
\cs_new:Npn \prop_if_in_aux:N #1
  {
    \if_meaning:w \q_prop #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
    \prop_map_break:
  }
\cs_generate_variant:Nn \prop_if_in_p:Nn {     NV , No }
\cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnT  { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnF  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }
```

(*End definition for* `\prop_if_in:Nn` *and others. These functions are documented on page* ??.)

195.4 Recovering values from property lists with branching

`\prop_get:NnNTF`
`\prop_get:NVNTF`
`\prop_get:NoNTF`
`\prop_get:cnNTF`
`\prop_get:cVNTF`
`\prop_get:coNTF`
`\prop_get_aux_true:Nnnn`

Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```
\prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
  {
    \prop_split:NnTF #1 {#2}
      { \prop_get_aux_true:Nnnn #3 }
      { \prg_return_false: }
  }
\cs_new_protected:Npn \prop_get_aux_true:Nnnn #1#2#3#4
  {
    \tl_set:Nn #1 {#3}
    \prg_return_true:
  }
\cs_generate_variant:Nn \prop_get:NnNT  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNF  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNTF {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNT  { c , cV , co }
\cs_generate_variant:Nn \prop_get:NnNF  { c , cV , co }
```

```
\cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }
```

(*End definition for* `\prop_get:NnN` *and others. These functions are documented on page* ??.)

195.5 Mapping to property lists

`\prop_map_function:NN`
`\prop_map_function:Nc`
`\prop_map_function:cN`
`\prop_map_function:cc`
`\prop_map_function_aux:Nwn`

The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`.

```
\cs_new:Npn \prop_map_function:NN #1#2
  {
    \exp_last_unbraced:NNo \prop_map_function_aux:Nwn #2
      #1 \q_recursion_tail \q_prop { }
    \prg_break_point:n { }
  }
\cs_new:Npn \prop_map_function_aux:Nwn #1 \q_prop #2 \q_prop #3
  {
    \if_meaning:w \q_recursion_tail #2
      \exp_after:wN \prop_map_break:
    \fi:
    #1 {#2} {#3}
    \prop_map_function_aux:Nwn #1
  }
\cs_generate_variant:Nn \prop_map_function:NN {     Nc }
\cs_generate_variant:Nn \prop_map_function:NN { c , cc }
```

(*End definition for* `\prop_map_function:NN` *and others. These functions are documented on page* ??.)

`\prop_map_inline:Nn`
`\prop_map_inline:cn`

Mapping in line requires a nesting level counter.

```
\cs_new_protected:Npn \prop_map_inline:Nn #1#2
  {
    \int_gincr:N \g_prg_map_int
    \cs_gset:cpn { prop_map_inline_ \int_use:N \g_prg_map_int :nn }
      ##1##2 {#2}
    \exp_last_unbraced:Nco \prop_map_function_aux:Nwn
      { prop_map_inline_ \int_use:N \g_prg_map_int :nn }
      #1
      \q_recursion_tail \q_prop { }
    \prg_break_point:n { \int_gdecr:N \g_prg_map_int }
  }
\cs_generate_variant:Nn \prop_map_inline:Nn { c }
```

(*End definition for* `\prop_map_inline:Nn` *and* `\prop_map_inline:cn`*. These functions are documented on page* ??.)

`\prop_map_break:`
`\prop_map_break:n`

The break statements are simply copies.

```
\cs_new_eq:NN \prop_map_break:  \prg_map_break:
\cs_new_eq:NN \prop_map_break:n \prg_map_break:n
```

(*End definition for* `\prop_map_break:`*. This function is documented on page 120.*)

195.6 Viewing property lists

\prop_show:N
\prop_show:c

Apply the general \msg_aux_show:Nnx. Contrarily to sequences and comma lists, we use \msg_aux_show:nn to format both the key and the value for each pair.

```
\cs_new_protected:Npn \prop_show:N #1
  {
    \msg_aux_show:Nnx
      #1
      { prop }
      { \prop_map_function:NN #1 \msg_aux_show:nn }
  }
\cs_generate_variant:Nn \prop_show:N { c }
```

(*End definition for* \prop_show:N *and* \prop_show:c*. These functions are documented on page* **??**.)

195.7 Experimental functions

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*
\prop_gpop:cnN*TF*
\prop_gpop:cnN*TF*
\prop_pop_aux_true:NNNnnn

Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, \prg_return_true: is used after the assignments.

```
\prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
  {
    \prop_split:NnTF #1 {#2}
      { \prop_pop_aux_true:NNNnnn \tl_set:Nn #1 #3 }
      { \prg_return_false: }
  }
\prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
  {
    \prop_split:NnTF #1 {#2}
      { \prop_pop_aux_true:NNNnnn \tl_gset:Nn #1 #3 }
      { \prg_return_false: }
  }
\cs_new_protected:Npn \prop_pop_aux_true:NNNnnn #1#2#3#4#5#6
  {
    \tl_set:Nn #3 {#5}
    #1 #2 { #4 #6 }
    \prg_return_true:
  }
\cs_generate_variant:Nn \prop_pop:NnNT   { c }
\cs_generate_variant:Nn \prop_pop:NnNF   { c }
\cs_generate_variant:Nn \prop_pop:NnNTF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNT  { c }
\cs_generate_variant:Nn \prop_gpop:NnNF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNTF { c }
```

(*End definition for* \prop_pop:NnN *and others. These functions are documented on page* **??**.)

\prop_map_tokens:Nn
\prop_map_tokens:cn
\prop_map_tokens_aux:nwn

The mapping grabs one key–value pair at a time, and stops when reaching the marker key \q_recursion_tail, which cannot appear in normal keys since those are strings. The odd construction \use:n {#1} allows #1 to contain any token.

```
\cs_new:Npn \prop_map_tokens:Nn #1#2
  {
    \exp_last_unbraced:Nno \prop_map_tokens_aux:nwn {#2} #1
    \q_recursion_tail \q_prop { }
    \prg_break_point:n { }
  }
\cs_new:Npn \prop_map_tokens_aux:nwn #1 \q_prop #2 \q_prop #3
  {
    \if_meaning:w \q_recursion_tail #2
      \exp_after:wN \prop_map_break:
    \fi:
    \use:n {#1} {#2} {#3}
    \prop_map_tokens_aux:nwn {#1}
  }
\cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(*End definition for* `\prop_map_tokens:Nn` *and* `\prop_map_tokens:cn`*. These functions are documented on page* **??**.)

`\prop_get:Nn`
`\prop_get:cn`
`\prop_get_Nn_aux:nwn`

Getting the value corresponding to a key in a property list in an expandable fashion is a simple instance of mapping some tokens. Map the function `\prop_get_aux:nnn` which takes as its three arguments the ⟨*key*⟩ that we are looking for, the current ⟨*key*⟩ and the current ⟨*value*⟩. If the ⟨*keys*⟩ match, the ⟨*value*⟩ is returned. If none of the keys match, this expands to nothing.

```
\cs_new:Npn \prop_get:Nn #1#2
  {
    \exp_last_unbraced:Noo \prop_get_Nn_aux:nwn
      { \tl_to_str:n {#2} } #1
      \tl_to_str:n {#2} \q_prop { }
    \prg_break_point:n { }
  }
\cs_new:Npn \prop_get_Nn_aux:nwn #1 \q_prop #2 \q_prop #3
  {
    \str_if_eq:xxTF {#1} {#2}
      { \prg_map_break:n { \exp_not:n {#3} } }
      { \prop_get_Nn_aux:nwn {#1} }
  }
\cs_generate_variant:Nn \prop_get:Nn { c }
```

(*End definition for* `\prop_get:Nn` *and* `\prop_get:cn`*. These functions are documented on page* **??**.)

195.8 Deprecated interfaces

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\prop_display:N`
`\prop_display:c`

An older name for `\prop_show:N`.

```
⟨*deprecated⟩
\cs_new_eq:NN \prop_display:N \prop_show:N
\cs_new_eq:NN \prop_display:c \prop_show:c
⟨/deprecated⟩
```

(*End definition for* `\prop_display:N` *and* `\prop_display:c`*. These functions are documented on page* **??**.)

`\prop_gget:NnN`
`\prop_gget:NVN`
`\prop_gget:cnN`
`\prop_gget:cVN`
`\prop_gget_aux:Nnnn`

Getting globally is no longer supported: this is a conceptual change, so the necessary code for the transition is provided directly.

```
⟨*deprecated⟩
\cs_new_protected:Npn \prop_gget:NnN #1#2#3
  { \prop_split:Nnn #1 {#2} { \prop_gget_aux:Nnnn #3 } }
\cs_new_protected:Npn \prop_gget_aux:Nnnn #1#2#3#4
  { \tl_gset:Nn #1 {#3} }
\cs_generate_variant:Nn \prop_gget:NnN {     NV }
\cs_generate_variant:Nn \prop_gget:NnN { c , cV }
⟨/deprecated⟩
```

(*End definition for* `\prop_gget:NnN` *and others. These functions are documented on page* **??**.)

`\prop_get_gdel:NnN`

This name seems very odd.

```
⟨*deprecated⟩
\cs_new_eq:NN \prop_get_gdel:NnN \prop_gpop:NnN
⟨/deprecated⟩
```

(*End definition for* `\prop_get_gdel:NnN`*. This function is documented on page* **??**.)

`\prop_if_in:ccTF`

A hang-over from an ancient implementation

```
⟨*deprecated⟩
\cs_generate_variant:Nn \prop_if_in:NnT  { cc }
\cs_generate_variant:Nn \prop_if_in:NnF  { cc }
\cs_generate_variant:Nn \prop_if_in:NnTF { cc }
⟨/deprecated⟩
```

(*End definition for* `\prop_if_in:ccTF`*. This function is documented on page* **??**.)

`\prop_gput:ccx`

Another one.

```
⟨*deprecated⟩
\cs_generate_variant:Nn \prop_gput:Nnn { ccx }
⟨/deprecated⟩
```

(*End definition for* `\prop_gput:ccx`*. This function is documented on page* **??**.)

`\prop_if_eq_p:NN`
`\prop_if_eq_p:Nc`
`\prop_if_eq_p:cN`
`\prop_if_eq_p:cc`
`\prop_if_eq:NNTF`
`\prop_if_eq:NcTF`
`\prop_if_eq:cNTF`
`\prop_if_eq:ccTF`

These ones do no even make sense!

```
⟨*deprecated⟩
\prg_new_eq_conditional:NNn \prop_if_eq:NN \tl_if_eq:NN { p , T , F , TF }
\prg_new_eq_conditional:NNn \prop_if_eq:cN \tl_if_eq:cN { p , T , F , TF }
\prg_new_eq_conditional:NNn \prop_if_eq:Nc \tl_if_eq:Nc { p , T , F , TF }
\prg_new_eq_conditional:NNn \prop_if_eq:cc \tl_if_eq:cc { p , T , F , TF }
⟨/deprecated⟩
```

(*End definition for* `\prop_if_eq:NN` *and others. These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

196 l3box implementation

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

196.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box_new:N
\box_new:c

Defining a new ⟨*box*⟩ register: remember that box 255 is not generally available.

```
⟨*package⟩
\cs_new_protected:Npn \box_new:N #1
  {
    \chk_if_free_cs:N #1
    \newbox #1
  }
⟨/package⟩
\cs_generate_variant:Nn \box_new:N { c }
```

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

Clear a ⟨*box*⟩ register.

```
\cs_new_protected:Npn \box_clear:N #1
  { \box_set_eq:NN  #1 \c_empty_box }
\cs_new_protected:Npn \box_gclear:N #1
  { \box_gset_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box_clear:N  { c }
\cs_generate_variant:Nn \box_gclear:N { c }
```

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

Clear or new.

```
\cs_new_protected:Npn \box_clear_new:N #1
  { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\cs_new_protected:Npn \box_gclear_new:N #1
  { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\cs_generate_variant:Nn \box_clear_new:N  { c }
\cs_generate_variant:Nn \box_gclear_new:N { c }
```

\box_set_eq:NN
\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc
\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

Assigning the contents of a box to be another box.

```
\cs_new_protected:Npn \box_set_eq:NN #1#2
  { \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box_gset_eq:NN
  { \tex_global:D \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_eq:NN  { cN , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq:NN { cN , Nc , cc }
```

\box_set_eq_clear:NN
\box_set_eq_clear:cN
\box_set_eq_clear:Nc
\box_set_eq_clear:cc
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc

Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```
\cs_new_protected:Npn \box_set_eq_clear:NN #1#2
  { \tex_setbox:D #1 \tex_box:D #2 }
\cs_new_protected:Npn \box_gset_eq_clear:NN
  { \tex_global:D  \box_set_eq_clear:NN }
\cs_generate_variant:Nn \box_set_eq_clear:NN  { cN , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq_clear:NN { cN , Nc , cc }
```

\box_if_exist_p:N
\box_if_exist_p:c
\box_if_exist:NTF
\box_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\cs_new_eq:NN \box_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \box_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \box_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \box_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \box_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \box_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \box_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \box_if_exist_p:c \cs_if_exist_p:c
```

196.2 Measuring and setting box dimensions

\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.

```
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N { c }
\cs_generate_variant:Nn \box_dp:N { c }
\cs_generate_variant:Nn \box_wd:N { c }
```

\box_set_ht:Nn
\box_set_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```
\cs_new_protected:Npn \box_set_dp:Nn #1#2
  { \box_dp:N #1 \dim_eval:w #2 \dim_eval_end: }
\cs_new_protected:Npn \box_set_ht:Nn #1#2
  { \box_ht:N #1 \dim_eval:w #2 \dim_eval_end: }
\cs_new_protected:Npn \box_set_wd:Nn #1#2
  { \box_wd:N #1 \dim_eval:w #2 \dim_eval_end: }
\cs_generate_variant:Nn \box_set_ht:Nn { c }
\cs_generate_variant:Nn \box_set_dp:Nn { c }
\cs_generate_variant:Nn \box_set_wd:Nn { c }
```

196.3 Using boxes

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

Using a ⟨*box*⟩. These are just TeX primitives with meaningful names.

```
\cs_new_eq:NN \box_use_clear:N \tex_box:D
\cs_new_eq:NN \box_use:N \tex_copy:D
\cs_generate_variant:Nn \box_use_clear:N { c }
\cs_generate_variant:Nn \box_use:N { c }
```

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

Move box material in different directions.

```
\cs_new_protected:Npn \box_move_left:nn #1#2
  { \tex_moveleft:D \dim_eval:w #1 \dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_right:nn #1#2
  { \tex_moveright:D \dim_eval:w #1 \dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_up:nn #1#2
  { \tex_raise:D \dim_eval:w #1 \dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_down:nn #1#2
  { \tex_lower:D \dim_eval:w #1 \dim_eval_end: #2 }
```

196.4 Box conditionals

\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.

```
\cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N*TF*
\box_if_horizontal:c*TF*
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N*TF*
\box_if_vertical:c*TF*

```
\prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
  { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
  { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\cs_generate_variant:Nn \box_if_horizontal:NT  { c }
\cs_generate_variant:Nn \box_if_horizontal:NF  { c }
\cs_generate_variant:Nn \box_if_horizontal:NTF { c }
\cs_generate_variant:Nn \box_if_vertical_p:N { c }
\cs_generate_variant:Nn \box_if_vertical:NT  { c }
\cs_generate_variant:Nn \box_if_vertical:NF  { c }
\cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:N*TF*
\box_if_empty:c*TF*

Testing if a ⟨*box*⟩ is empty/void.

```
\prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
  { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_empty_p:N { c }
\cs_generate_variant:Nn \box_if_empty:NT  { c }
\cs_generate_variant:Nn \box_if_empty:NF  { c }
\cs_generate_variant:Nn \box_if_empty:NTF { c }
```

(*End definition for* `\box_new:N` *and* `\box_new:c`*. These functions are documented on page* **??**.)

196.5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

Set a box to the previous box.

```
\cs_new_protected:Npn \box_set_to_last:N #1
  { \tex_setbox:D #1 \tex_lastbox:D }
\cs_new_protected:Npn \box_gset_to_last:N
  { \tex_global:D \box_set_to_last:N }
\cs_generate_variant:Nn \box_set_to_last:N  { c }
\cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(*End definition for* `\box_set_to_last:N` *and* `\box_set_to_last:c`*. These functions are documented on page* ??*.*)

196.6 Constant boxes

`\c_empty_box`

```
⟨*package⟩
\cs_new_eq:NN \c_empty_box \voidb@x
⟨/package⟩
⟨*initex⟩
\box_new:N \c_empty_box
⟨/initex⟩
```

(*End definition for* `\c_empty_box`*. This variable is documented on page 127.*)

196.7 Scratch boxes

`\l_tmpa_box`
`\l_tmpb_box`

```
⟨*package⟩
\cs_new_eq:NN \l_tmpa_box \@tempboxa
⟨/package⟩
⟨*initex⟩
\box_new:N \l_tmpa_box
⟨/initex⟩
\box_new:N \l_tmpb_box
```

(*End definition for* `\l_tmpa_box` *and* `\l_tmpb_box`*. These variables are documented on page 127.*)

196.8 Viewing box contents

`\box_show:N`
`\box_show:c`

Check that the variable exists, then show the contents of the box and write it into the log file. The spurious `\use:n` gives a nicer output.

```
\cs_new_protected:Npn \box_show:N #1
  {
    \box_if_exist:NTF #1
      { \tex_showbox:D \use:n {#1} }
      {
        \msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_generate_variant:Nn \box_show:N { c }
```

(*End definition for* `\box_show:N` *and* `\box_show:c`*. These functions are documented on page* ??*.*)

`\box_show:Nnn`
`\box_show:cnn`
`\box_show_full:N`
`\box_show_full:c`

Show the contents of a box and write it into the log file, after setting the parameters `\showboxbreadth` and `\showboxdepth` to the values provided by the user.

```
\cs_new_protected:Npn \box_show:Nnn #1#2#3
  {
    \group_begin:
      \int_set:Nn \tex_showboxbreadth:D {#2}
```

```
      \int_set:Nn \tex_showboxdepth:D {#3}
      \int_set_eq:NN \tex_tracingonline:D \c_one
      \box_show:N #1
    \group_end:
  }
\cs_generate_variant:Nn \box_show:Nnn { c }
\cs_new_protected:Npn \box_show_full:N #1
  { \box_show:Nnn #1 { \c_max_int } { \c_max_int } }
\cs_generate_variant:Nn \box_show_full:N { c }
```

(*End definition for* `\box_show:Nnn` *and* `\box_show:cnn`*. These functions are documented on page* **??**.)

196.9 Horizontal mode boxes

`\hbox:n`

(*The test suite for this command, and others in this file, is* m3box002.lvt.)

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox:n { \tex_hbox:D \scan_stop: }
```

(*End definition for* `\hbox:n`*. This function is documented on page 127.*)

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

```
\cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
\cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
\cs_generate_variant:Nn \hbox_set:Nn { c }
\cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(*End definition for* `\hbox_set:Nn` *and* `\hbox_set:cn`*. These functions are documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
  { \tex_setbox:D #1 \tex_hbox:D to \dim_eval:w #2 \dim_eval_end: {#3} }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnn
  { \tex_global:D \hbox_set_to_wd:Nnn }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(*End definition for* `\hbox_set_to_wd:Nnn` *and* `\hbox_set_to_wd:cnn`*. These functions are documented on page* **??**.)

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_set_end:`
`\hbox_gset_end:`

Storing material in a horizontal box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \hbox_set:Nw  #1
  { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
\cs_new_protected:Npn \hbox_gset:Nw
  { \tex_global:D \hbox_set:Nw }
\cs_generate_variant:Nn \hbox_set:Nw  { c }
\cs_generate_variant:Nn \hbox_gset:Nw { c }
\cs_new_eq:NN \hbox_set_end:  \c_group_end_token
\cs_new_eq:NN \hbox_gset_end: \c_group_end_token
```

(*End definition for* `\hbox_set:Nw` *and* `\hbox_set:cw`*. These functions are documented on page* **??**.)

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

Renamed September 2011.

```
\cs_new_eq:NN \hbox_set_inline_begin:N  \hbox_set:Nw
\cs_new_eq:NN \hbox_set_inline_begin:c  \hbox_set:cw
\cs_new_eq:NN \hbox_set_inline_end:     \hbox_set_end:
\cs_new_eq:NN \hbox_gset_inline_begin:N \hbox_gset:Nw
\cs_new_eq:NN \hbox_gset_inline_begin:c \hbox_gset:cw
\cs_new_eq:NN \hbox_gset_inline_end:    \hbox_gset_end:
```

(*End definition for* `\hbox_set_inline_begin:N` *and* `\hbox_set_inline_begin:c`*. These functions are documented on page* **??**.)

\hbox_to_wd:nn
\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox_to_wd:nn #1#2
   { \tex_hbox:D to \dim_eval:w #1 \dim_eval_end: {#2} }
\cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_skip {#1} }
```

(*End definition for* `\hbox_to_wd:nn`*. This function is documented on page 128.*)

\hbox_overlap_left:n
\hbox_overlap_right:n

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```
\cs_new_protected:Npn \hbox_overlap_left:n  #1
  { \hbox_to_zero:n { \tex_hss:D #1 } }
\cs_new_protected:Npn \hbox_overlap_right:n #1
  { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(*End definition for* `\hbox_overlap_left:n` *and* `\hbox_overlap_right:n`*. These functions are documented on page 128.*)

\hbox_unpack:N
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
\cs_generate_variant:Nn \hbox_unpack:N { c }
\cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(*End definition for* `\hbox_unpack:N` *and* `\hbox_unpack:c`*. These functions are documented on page* **??**.)

196.10 Vertical mode boxes

TeX ends these boxes directly with the internal *end_graf* routine. This means that there is no `\par` at the end of vertical boxes unless we insert one.

\vbox:n *The following test files are used for this code: m3box003.lvt.*

\vbox_top:n *The following test files are used for this code: m3box003.lvt.*

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox:n #1     { \tex_vbox:D { #1 \par } }
\cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }
```

(*End definition for* `\vbox:n`*. This function is documented on page 129.*)

\vbox_to_ht:nn
\vbox_to_zero:n
\vbox_to_ht:nn
\vbox_to_zero:n

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox_to_ht:nn #1#2
  { \tex_vbox:D to \dim_eval:w #1 \dim_eval_end: { #2 \par } }
\cs_new_protected:Npn \vbox_to_zero:n #1
  { \tex_vbox:D to \c_zero_dim { #1 \par } }
```

(*End definition for* `\vbox_to_ht:nn` *and* `\vbox_to_zero:n`*. These functions are documented on page* *129*.)

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

Storing material in a vertical box with a natural height.

```
\cs_new_protected:Npn \vbox_set:Nn #1#2
  { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
\cs_new_protected:Npn \vbox_gset:Nn  { \tex_global:D \vbox_set:Nn }
\cs_generate_variant:Nn \vbox_set:Nn  { c }
\cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(*End definition for* `\vbox_set:Nn` *and* `\vbox_set:cn`*. These functions are documented on page* **??**.)

\vbox_set_top:Nn
\vbox_set_top:cn
\vbox_gset_top:Nn
\vbox_gset_top:cn

Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
\cs_new_protected:Npn \vbox_set_top:Nn #1#2
  { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
\cs_new_protected:Npn \vbox_gset_top:Nn
  { \tex_global:D \vbox_set_top:Nn }
\cs_generate_variant:Nn \vbox_set_top:Nn { c }
\cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(*End definition for* `\vbox_set_top:Nn` *and* `\vbox_set_top:cn`*. These functions are documented on page* **??**.)

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

Storing material in a vertical box with a specified height.

```
\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
  { \tex_setbox:D #1 \tex_vbox:D to \dim_eval:w #2 \dim_eval_end: { #3 \par } }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnn
  { \tex_global:D \vbox_set_to_ht:Nnn }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnn  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(*End definition for* `\vbox_set_to_ht:Nnn` *and* `\vbox_set_to_ht:cnn`*. These functions are documented on page* **??**.)

\vbox_set:Nw
\vbox_set:cw
\vbox_gset:Nw
\vbox_gset:cw
\vbox_set_end:
\vbox_gset_end:

Storing material in a vertical box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \vbox_set:Nw #1
  { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
\cs_new_protected:Npn \vbox_gset:Nw
  { \tex_global:D \vbox_set:Nw }
\cs_generate_variant:Nn \vbox_set:Nw  { c }
\cs_generate_variant:Nn \vbox_gset:Nw { c }
\cs_new_protected:Npn \vbox_set_end:
  {
    \par
    \c_group_end_token
  }
\cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(*End definition for* `\vbox_set:Nw` *and* `\vbox_set:cw`*. These functions are documented on page* **??**.)

`\vbox_set_inline_begin:N`
`\vbox_set_inline_begin:c`
`\vbox_gset_inline_begin:N`
`\vbox_gset_inline_begin:c`
`\vbox_set_inline_end:`
`\vbox_gset_inline_end:`

Renamed September 2011.

```
\cs_new_eq:NN \vbox_set_inline_begin:N  \vbox_set:Nw
\cs_new_eq:NN \vbox_set_inline_begin:c  \vbox_set:cw
\cs_new_eq:NN \vbox_set_inline_end:  \vbox_set_end:
\cs_new_eq:NN \vbox_gset_inline_begin:N \vbox_gset:Nw
\cs_new_eq:NN \vbox_gset_inline_begin:c \vbox_gset:cw
\cs_new_eq:NN \vbox_gset_inline_end: \vbox_gset_end:
```

(*End definition for* `\vbox_set_inline_begin:N` *and* `\vbox_set_inline_begin:c`*. These functions are documented on page* **??**.)

`\vbox_unpack:N`
`\vbox_unpack:c`
`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\cs_generate_variant:Nn \vbox_unpack:N { c }
\cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(*End definition for* `\vbox_unpack:N` *and* `\vbox_unpack:c`*. These functions are documented on page* **??**.)

`\vbox_set_split_to_ht:NNn`

Splitting a vertical box in two.

```
\cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
  { \tex_setbox:D #1 \tex_vsplit:D #2 to \dim_eval:w #3 \dim_eval_end: }
```

(*End definition for* `\vbox_set_split_to_ht:NNn`*. This function is documented on page 130.*)

196.11 Affine transformations

`\l_box_angle_fp`

When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
\fp_new:N \l_box_angle_fp
```

(*End definition for* `\l_box_angle_fp`*. This variable is documented on page* **??**.)

`\l_box_cos_fp`
`\l_box_sin_fp`

These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\fp_new:N \l_box_cos_fp
\fp_new:N \l_box_sin_fp
```

(*End definition for* `\l_box_cos_fp` *and* `\l_box_sin_fp`*. These variables are documented on page* **??**.)

`\l_box_top_dim`
`\l_box_bottom_dim`
`\l_box_left_dim`
`\l_box_right_dim`

These are the positions of the four edges of a box before manipulation.

```
\dim_new:N \l_box_top_dim
\dim_new:N \l_box_bottom_dim
\dim_new:N \l_box_left_dim
\dim_new:N \l_box_right_dim
```

(*End definition for* `\l_box_top_dim` *and others. These variables are documented on page* **??**.)

`\l_box_top_new_dim`
`\l_box_bottom_new_dim`
`\l_box_left_new_dim`
`\l_box_right_new_dim`

These are the positions of the four edges of a box after manipulation.

```
\dim_new:N \l_box_top_new_dim
\dim_new:N \l_box_bottom_new_dim
\dim_new:N \l_box_left_new_dim
\dim_new:N \l_box_right_new_dim
```

(*End definition for* `\l_box_top_new_dim` *and others. These variables are documented on page* **??**.)

`\l_box_internal_box`
`\l_box_internal_fp`

Scratch space.

```
\box_new:N \l_box_internal_box
\fp_new:N \l_box_internal_fp
```

(*End definition for* `\l_box_internal_box` *and* `\l_box_internal_fp`*. These variables are documented on page* **??**.)

`\l_box_x_fp`
`\l_box_y_fp`
`\l_box_x_new_fp`
`\l_box_y_new_fp`

Used as the input and output values for a point when manipulation the location.

```
\fp_new:N \l_box_x_fp
\fp_new:N \l_box_y_fp
\fp_new:N \l_box_x_new_fp
\fp_new:N \l_box_y_new_fp
```

(*End definition for* `\l_box_x_fp` *and others. These variables are documented on page* **??**.)

`\box_rotate:Nn`
`\box_rotate_aux:N`
`\box_rotate_set_sin_cos:`
`\box_rotate_x:nnN`
`\box_rotate_y:nnN`
`\box_rotate_quadrant_one:`
`\box_rotate_quadrant_two:`
`\box_rotate_quadrant_three:`
`\box_rotate_quadrant_four:`

Rotation of a box starts with working out the relevant sine and cosine. There is then a check to avoid doing any real work for the trivial rotation.

```
\cs_new_protected:Npn \box_rotate:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \fp_set:Nn \l_box_angle_fp {#2}
          \box_rotate_set_sin_cos:
          \fp_compare:NNNTF \l_box_sin_fp = \c_zero_fp
            {
               \fp_compare:NNNTF \l_box_cos_fp = \c_one_fp
                 { \box_use:N #1 }
                 { \box_rotate_aux:N #1 }
            }
            { \box_rotate_aux:N #1 }
        \group_end:
      }
  }
```

The edges of the box are then recorded: the left edge will always be at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
\cs_new_protected:Npn \box_rotate_aux:N #1
  {
    \dim_set:Nn \l_box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l_box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l_box_left_dim
```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B, C, D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and

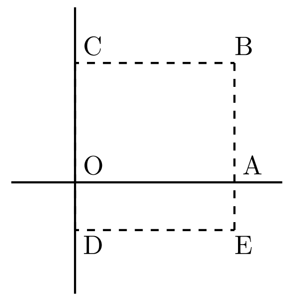

Figure 1: Co-ordinates of a box prior to rotation.

angle α:

$$
\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}
$$

The "extra" horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x-coordinate 0. This is desirable as TEX boxes must have the reference point at the left edge of the box. (As O is always $(0, 0)$, this part of the calculation is omitted here.)

```
    \fp_compare:NNNTF \l_box_sin_fp > \c_zero_fp
      {
        \fp_compare:NNNTF \l_box_cos_fp > \c_zero_fp
          { \box_rotate_quadrant_one: }
          { \box_rotate_quadrant_two: }
      }
      {
        \fp_compare:NNNTF \l_box_cos_fp < \c_zero_fp
          { \box_rotate_quadrant_three: }
          { \box_rotate_quadrant_four: }
      }
```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TEX reference point. So the content of the box is moved such that the reference point of the rotated box will be in the same place as the original.

```
    \hbox_set:Nn \l_box_internal_box { \box_use:N #1 }
    \hbox_set:Nn \l_box_internal_box
      {
        \tex_kern:D -\l_box_left_new_dim
        \hbox:n
          {
            \driver_box_rotate_begin:
            \box_use:N \l_box_internal_box
            \driver_box_rotate_end:
```

```
        }
    }
```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```
    \box_set_ht:Nn \l_box_internal_box {  \l_box_top_new_dim }
    \box_set_dp:Nn \l_box_internal_box { -\l_box_bottom_new_dim }
    \box_set_wd:Nn \l_box_internal_box
      { \l_box_right_new_dim - \l_box_left_new_dim }
    \box_use:N \l_box_internal_box
  }
```

A simple conversion from degrees to radians followed by calculation of the sine and cosine.

```
\cs_new_protected:Npn \box_rotate_set_sin_cos:
  {
    \fp_set_eq:NN \l_box_internal_fp \l_box_angle_fp
    \fp_div:Nn \l_box_internal_fp { 180 }
    \fp_mul:Nn \l_box_internal_fp { \c_pi_fp }
    \fp_sin:Nn \l_box_sin_fp { \l_box_internal_fp }
    \fp_cos:Nn \l_box_cos_fp { \l_box_internal_fp }
  }
```

These functions take a general point (#1, #2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```
\cs_new_protected:Npn \box_rotate_x:nnN #1#2#3
  {
    \fp_set_from_dim:Nn \l_box_x_fp {#1}
    \fp_set_from_dim:Nn \l_box_y_fp {#2}
    \fp_set_eq:NN \l_box_x_new_fp \l_box_x_fp
    \fp_set_eq:NN \l_box_internal_fp   \l_box_y_fp
    \fp_mul:Nn \l_box_x_new_fp { \l_box_cos_fp }
    \fp_mul:Nn \l_box_internal_fp   { \l_box_sin_fp }
    \fp_sub:Nn \l_box_x_new_fp { \l_box_internal_fp }
    \dim_set:Nn #3 { \fp_to_dim:N \l_box_x_new_fp }
  }
\cs_new_protected:Npn \box_rotate_y:nnN #1#2#3
  {
    \fp_set_from_dim:Nn \l_box_x_fp {#1}
    \fp_set_from_dim:Nn \l_box_y_fp {#2}
    \fp_set_eq:NN \l_box_y_new_fp \l_box_y_fp
    \fp_set_eq:NN \l_box_internal_fp   \l_box_x_fp
    \fp_mul:Nn \l_box_y_new_fp { \l_box_cos_fp }
    \fp_mul:Nn \l_box_internal_fp   { \l_box_sin_fp }
    \fp_add:Nn \l_box_y_new_fp { \l_box_internal_fp }
    \dim_set:Nn #3 { \fp_to_dim:N \l_box_y_new_fp }
}
```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y-values, whereas the left and right edges need the x-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```
\cs_new_protected:Npn \box_rotate_quadrant_one:
  {
    \box_rotate_y:nnN \l_box_right_dim \l_box_top_dim
      \l_box_top_new_dim
    \box_rotate_y:nnN \l_box_left_dim  \l_box_bottom_dim
      \l_box_bottom_new_dim
    \box_rotate_x:nnN \l_box_left_dim  \l_box_top_dim
      \l_box_left_new_dim
    \box_rotate_x:nnN \l_box_right_dim \l_box_bottom_dim
      \l_box_right_new_dim
  }
\cs_new_protected:Npn \box_rotate_quadrant_two:
  {
    \box_rotate_y:nnN \l_box_right_dim \l_box_bottom_dim
      \l_box_top_new_dim
    \box_rotate_y:nnN \l_box_left_dim  \l_box_top_dim
      \l_box_bottom_new_dim
    \box_rotate_x:nnN \l_box_right_dim  \l_box_top_dim
      \l_box_left_new_dim
    \box_rotate_x:nnN \l_box_left_dim   \l_box_bottom_dim
      \l_box_right_new_dim
  }
\cs_new_protected:Npn \box_rotate_quadrant_three:
  {
    \box_rotate_y:nnN \l_box_left_dim  \l_box_bottom_dim
      \l_box_top_new_dim
    \box_rotate_y:nnN \l_box_right_dim \l_box_top_dim
      \l_box_bottom_new_dim
    \box_rotate_x:nnN \l_box_right_dim \l_box_bottom_dim
      \l_box_left_new_dim
    \box_rotate_x:nnN \l_box_left_dim   \l_box_top_dim
      \l_box_right_new_dim
  }
\cs_new_protected:Npn \box_rotate_quadrant_four:
  {
    \box_rotate_y:nnN \l_box_left_dim  \l_box_top_dim
      \l_box_top_new_dim
    \box_rotate_y:nnN \l_box_right_dim \l_box_bottom_dim
      \l_box_bottom_new_dim
    \box_rotate_x:nnN \l_box_left_dim  \l_box_bottom_dim
      \l_box_left_new_dim
    \box_rotate_x:nnN \l_box_right_dim \l_box_top_dim
      \l_box_right_new_dim
  }
```

(*End definition for* `\box_rotate:Nn`*. This function is documented on page 125.*)

`\l_box_scale_x_fp`
`\l_box_scale_y_fp`

Scaling is potentially-different in the two axes.

```
\fp_new:N \l_box_scale_x_fp
\fp_new:N \l_box_scale_y_fp
```

(*End definition for* `\l_box_scale_x_fp` *and* `\l_box_scale_y_fp`*. These variables are documented on page* ??.)

`\box_resize:Nnn`
`\box_resize:cnn`
`\box_resize_aux:Nnn`

Resizing a box starts by working out the various dimensions of the existing box.

```
\cs_new_protected:Npn \box_resize:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l_box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l_box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l_box_left_dim
```

The x-scaling and resulting box size is easy enough to work out: the dimension is that given as `#2`, and the scale is simply the new width divided by the old one.

```
          \fp_set_from_dim:Nn \l_box_scale_x_fp {#2}
          \fp_set_from_dim:Nn \l_box_internal_fp { \l_box_right_dim }
          \fp_div:Nn \l_box_scale_x_fp { \l_box_internal_fp }
```

The y-scaling needs both the height and the depth of the current box.

```
          \fp_set_from_dim:Nn \l_box_scale_y_fp {#3}
          \fp_set_from_dim:Nn \l_box_internal_fp
            { \l_box_top_dim - \l_box_bottom_dim }
          \fp_div:Nn \l_box_scale_y_fp { \l_box_internal_fp }
```

At this stage, check for trivial scaling. If both scalings are unity, then the code does nothing. Otherwise, pass on to the auxiliary function to find the new dimensions.

```
          \fp_compare:NNNTF \l_box_scale_x_fp = \c_one_fp
            {
              \fp_compare:NNNTF \l_box_scale_y_fp = \c_one_fp
                { \box_use:N #1 }
                { \box_resize_aux:Nnn #1 {#2} {#3} }
            }
            { \box_resize_aux:Nnn #1 {#2} {#3} }
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize:Nnn { c }
```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. This is done using the absolute value of the scale so that the new edge is in the correct place. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```
\cs_new_protected:Npn \box_resize_aux:Nnn #1#2#3
  {
    \dim_compare:nNnTF {#2} > \c_zero_dim
      { \dim_set:Nn \l_box_right_new_dim {#2} }
      { \dim_set:Nn \l_box_right_new_dim { \c_zero_dim - ( #2 ) } }
    \dim_compare:nNnTF {#3} > \c_zero_dim
      {
        \dim_set:Nn \l_box_top_new_dim
          { \fp_use:N \l_box_scale_y_fp \l_box_top_dim }
        \dim_set:Nn \l_box_bottom_new_dim
          { \fp_use:N \l_box_scale_y_fp \l_box_bottom_dim }
      }
      {
        \dim_set:Nn \l_box_top_new_dim
          { - \fp_use:N \l_box_scale_y_fp \l_box_top_dim }
        \dim_set:Nn \l_box_bottom_new_dim
          { - \fp_use:N \l_box_scale_y_fp \l_box_bottom_dim }
      }
    \box_resize_common:N #1
  }
```

(*End definition for* `\box_resize:Nnn` *and* `\box_resize:cnn`*. These functions are documented on page* ??.)

`\box_resize_to_ht_plus_dp:Nn`
`\box_resize_to_ht_plus_dp:cn`
`\box_resize_to_wd:Nn`
`\box_resize_to_wd:cn`

Scaling to a total height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```
\cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l_box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l_box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l_box_left_dim
          \fp_set_from_dim:Nn \l_box_scale_y_fp {#2}
          \fp_set_from_dim:Nn \l_box_internal_fp
            { \l_box_top_dim - \l_box_bottom_dim }
          \fp_div:Nn \l_box_scale_y_fp { \l_box_internal_fp }
          \fp_set_eq:NN \l_box_scale_x_fp \l_box_scale_y_fp
          \fp_compare:NNNTF \l_box_scale_y_fp = \c_one_fp
            { \box_use:N #1 }
            { \box_resize_aux:Nnn #1 {#2} {#2} }
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
\cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
```

```
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l_box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l_box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l_box_left_dim
          \fp_set_from_dim:Nn \l_box_scale_x_fp {#2}
          \fp_set_from_dim:Nn \l_box_internal_fp { \l_box_right_dim }
          \fp_div:Nn \l_box_scale_x_fp { \l_box_internal_fp }
          \fp_set_eq:NN \l_box_scale_y_fp \l_box_scale_x_fp
          \fp_compare:NNNTF \l_box_scale_x_fp = \c_one_fp
            { \box_use:N #1 }
            { \box_resize_aux:Nnn #1 {#2} {#2} }
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
```

(*End definition for* `\box_resize_to_ht_plus_dp:Nn` *and* `\box_resize_to_ht_plus_dp:cn`*. These functions are documented on page* **??**.)

`\box_scale:Nnn`
`\box_scale:cnn`
`\box_scale_aux:Nnn`

When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The dimension scaling operations are carried out using the TeX mechanism as it avoids needing to use `fp` operations.

```
\cs_new_protected:Npn \box_scale:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \fp_set:Nn \l_box_scale_x_fp {#2}
          \fp_set:Nn \l_box_scale_y_fp {#3}
          \dim_set:Nn \l_box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l_box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l_box_left_dim
          \fp_compare:NNNTF \l_box_scale_x_fp = \c_one_fp
            {
              \fp_compare:NNNTF \l_box_scale_y_fp = \c_one_fp
                { \box_use:N #1 }
                { \box_scale_aux:Nnn #1 {#2} {#3} }
            }
            { \box_scale_aux:Nnn #1 {#2} {#3} }
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_scale:Nnn { c }
```

```
\cs_new_protected:Npn \box_scale_aux:Nnn #1#2#3
  {
    \fp_compare:NNNTF \l_box_scale_y_fp > \c_zero_fp
      {
        \dim_set:Nn \l_box_top_new_dim    { #3 \l_box_top_dim }
        \dim_set:Nn \l_box_bottom_new_dim { #3 \l_box_bottom_dim }
      }
      {
        \dim_set:Nn  \l_box_top_new_dim    { -#3 \l_box_bottom_dim }
        \dim_set:Nn  \l_box_bottom_new_dim { -#3 \l_box_top_dim }
      }
    \fp_compare:NNNTF \l_box_scale_x_fp > \c_zero_fp
      { \l_box_right_new_dim #2 \l_box_right_dim }
      { \l_box_right_new_dim -#2 \l_box_right_dim }
    \box_resize_common:N #1
  }
```

(*End definition for* `\box_scale:Nnn` *and* `\box_scale:cnn`*. These functions are documented on page* **??**.)

`\box_resize_common:N` The main resize function places in input into a box which will start of with zero width, and includes the handles for engine rescaling.

```
\cs_new_protected:Npn \box_resize_common:N #1
  {
    \hbox_set:Nn \l_box_internal_box
      {
        \driver_box_scale_begin:
        \hbox_overlap_right:n { \box_use:N #1 }
        \driver_box_scale_end:
      }
```

The new height and depth can be applied directly.

```
    \box_set_ht:Nn \l_box_internal_box { \l_box_top_new_dim }
    \box_set_dp:Nn \l_box_internal_box { \l_box_bottom_new_dim }
```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```
    \fp_compare:NNNTF \l_box_scale_x_fp < \c_zero_fp
      {
        \hbox_to_wd:nn { \l_box_right_new_dim }
          {
            \tex_kern:D \l_box_right_new_dim
            \box_use:N \l_box_internal_box
            \tex_hss:D
          }
      }
      {
        \box_set_wd:Nn \l_box_internal_box { \l_box_right_new_dim }
        \box_use:N \l_box_internal_box
      }
```

```
  }
```

(*End definition for* `\box_resize_common:N`*. This function is documented on page* **??**.)

196.12 Viewing part of a box

`\box_clip:N`
`\box_clip:c`

A wrapper around the driver-dependent code.

```
\cs_new_protected:Npn \box_clip:N #1
  { \hbox_set:Nn #1 { \driver_box_use_clip:N #1 } }
\cs_generate_variant:Nn \box_clip:N { c }
```

(*End definition for* `\box_clip:N` *and* `\box_clip:c`*. These functions are documented on page* **??**.)

`\box_trim:Nnnnn`
`\box_trim:cnnnn`

Trimming from the left- and right-hand edges of the box is easy. The total width is set to remove from the right, and a skip will shift the material to remove from the left.

```
\cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
  {
    \box_set_wd:Nn #1 { \box_wd:N #1 - \dim_eval:n {#4} - \dim_eval:n {#2} }
    \hbox_set:Nn #1
      {
        \skip_horizontal:n { - \dim_eval:n {#2} }
        \box_use:N #1
      }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim.

```
    \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
      { \box_set_dp:Nn #1 { \box_dp:N #1 - \dim_eval:n {#3} } }
      {
        \hbox_set:Nn #1
          {
            \box_move_down:nn { \dim_eval:n {#3} - \box_dp:N #1 }
              { \box_use:N #1 }
          }
        \box_set_dp:Nn #1 \c_zero_dim
      }
    \dim_compare:nNnTF { \box_ht:N #1 } > {#5}
      { \box_set_ht:Nn #1 { \box_ht:N #1 - \dim_eval:n {#5} } }
      {
        \hbox_set:Nn #1
          {
            \box_move_up:nn { \dim_eval:n {#5} - \box_ht:N #1 }
              { \box_use:N #1 }
          }
        \box_set_ht:Nn #1 \c_zero_dim
      }
  }
\cs_generate_variant:Nn \box_trim:Nnnnn { c }
```

(*End definition for* `\box_trim:Nnnnn` *and* `\box_trim:cnnnn`*. These functions are documented on page* **??**.)

\box_viewport:Nnnnn
\box_viewport:cnnnn

The same general logic as for clipping, but with absolute dimensions. Thus again width is easy and height is harder.

```
\cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
  {
    \box_set_wd:Nn #1 { \dim_eval:n {#4} - \dim_eval:n {#2} }
    \hbox_set:Nn #1
      {
        \skip_horizontal:n { - \dim_eval:n {#2} }
        \box_use:N #1
      }
    \dim_compare:nNnTF {#3} > \c_zero_dim
      {
        \hbox_set:Nn #1 { \box_move_down:nn {#3} { \box_use:N #1 } }
        \box_set_dp:Nn #1 \c_zero_dim
      }
      { \box_set_dp:Nn #1 { - \dim_eval:n {#3} } }
    \dim_compare:nNnTF {#5} > \c_zero_dim
      { \box_set_ht:Nn #1 {#5} }
      {
        \hbox_set:Nn #1
          { \box_move_up:nn { -\dim_eval:n {#5} } { \box_use:N #1 } }
        \box_set_ht:Nn #1 \c_zero_dim
      }
  }
\cs_generate_variant:Nn \box_viewport:Nnnnn { c }
```

(*End definition for* `\box_viewport:Nnnnn` *and* `\box_viewport:cnnnn`*. These functions are documented on page* **??**.)

196.13 Deprecated functions

\l_last_box

Deprecated 2011-11-13, for removal by 2012-02-28.

```
\cs_new_eq:NN \l_last_box \tex_lastbox:D
```

(*End definition for* `\l_last_box`*. This variable is documented on page* **??**.)

```
⟨/initex | package⟩
```

197 l3coffins Implementation

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

197.1 Coffins: data structures and general variables

\l_coffin_internal_box
\l_coffin_internal_dim
\l_coffin_internal_fp
\l_coffin_internal_tl

Scratch variables.

```
\box_new:N \l_coffin_internal_box
\dim_new:N \l_coffin_internal_dim
\fp_new:N  \l_coffin_internal_fp
\tl_new:N  \l_coffin_internal_tl
```

(*End definition for* `\l_coffin_internal_box`*. This function is documented on page* **??**.)

`\c_coffin_corners_prop`

The "corners"; of a coffin define the real content, as opposed to the TEX bounding box. They all start off in the same place, of course.

```
\prop_new:N \c_coffin_corners_prop
\prop_put:Nnn \c_coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c_coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c_coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c_coffin_corners_prop { br } { { 0 pt } { 0 pt } }
```

(*End definition for* `\c_coffin_corners_prop`*. This variable is documented on page* **??**.)

`\c_coffin_poles_prop`

Pole positions are given for horizontal, vertical and reference-point based values.

```
\prop_new:N \c_coffin_poles_prop
\tl_set:Nn \l_coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
\prop_put:Nno \c_coffin_poles_prop { l }  { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { hc } { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { r }  { \l_coffin_internal_tl }
\tl_set:Nn \l_coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
\prop_put:Nno \c_coffin_poles_prop { b }  { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { vc } { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { t }  { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { B }  { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { H }  { \l_coffin_internal_tl }
\prop_put:Nno \c_coffin_poles_prop { T }  { \l_coffin_internal_tl }
```

(*End definition for* `\c_coffin_poles_prop`*. This variable is documented on page* **??**.)

`\l_coffin_calc_a_fp`
`\l_coffin_calc_b_fp`
`\l_coffin_calc_c_fp`
`\l_coffin_calc_d_fp`
`\l_coffin_calc_result_fp`

Used for calculations of intersections and in other internal places.

```
\fp_new:N \l_coffin_calc_a_fp
\fp_new:N \l_coffin_calc_b_fp
\fp_new:N \l_coffin_calc_c_fp
\fp_new:N \l_coffin_calc_d_fp
\fp_new:N \l_coffin_calc_result_fp
```

(*End definition for* `\l_coffin_calc_a_fp`*. This function is documented on page* **??**.)

`\l_coffin_error_bool`

For propagating errors so that parts of the code can work around them.

```
\bool_new:N \l_coffin_error_bool
```

(*End definition for* `\l_coffin_error_bool`*. This variable is documented on page* **??**.)

`\l_coffin_offset_x_dim`
`\l_coffin_offset_y_dim`

The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

```
\dim_new:N \l_coffin_offset_x_dim
\dim_new:N \l_coffin_offset_y_dim
```

(*End definition for* `\l_coffin_offset_x_dim`*. This function is documented on page* **??**.)

\l_coffin_pole_a_tl
\l_coffin_pole_b_tl

Needed for finding the intersection of two poles.

```
\tl_new:N \l_coffin_pole_a_tl
\tl_new:N \l_coffin_pole_b_tl
```

(*End definition for* `\l_coffin_pole_a_tl`. *This function is documented on page* **??**.)

\l_coffin_sin_fp
\l_coffin_cos_fp

Used for rotations to get the sine and cosine values.

```
\fp_new:N \l_coffin_sin_fp
\fp_new:N \l_coffin_cos_fp
```

(*End definition for* `\l_coffin_sin_fp`. *This function is documented on page* **??**.)

\l_coffin_x_dim
\l_coffin_y_dim
\l_coffin_x_prime_dim
\l_coffin_y_prime_dim

For calculating intersections and so forth.

```
\dim_new:N \l_coffin_x_dim
\dim_new:N \l_coffin_y_dim
\dim_new:N \l_coffin_x_prime_dim
\dim_new:N \l_coffin_y_prime_dim
```

(*End definition for* `\l_coffin_x_dim`. *This function is documented on page* **??**.)

\l_coffin_x_fp
\l_coffin_y_fp
\l_coffin_x_prime_fp
\l_coffin_y_prime_fp

Used for calculations where there are clear x- and y-components, for example during vector rotation.

```
\fp_new:N \l_coffin_x_fp
\fp_new:N \l_coffin_y_fp
\fp_new:N \l_coffin_x_prime_fp
\fp_new:N \l_coffin_y_prime_fp
```

(*End definition for* `\l_coffin_x_fp`. *This function is documented on page* **??**.)

\l_coffin_Depth_dim
\l_coffin_Height_dim
\l_coffin_TotalHeight_dim
\l_coffin_Width_dim

Dimensions for the various parts of a coffin.

```
\dim_new:N \l_coffin_Depth_dim
\dim_new:N \l_coffin_Height_dim
\dim_new:N \l_coffin_TotalHeight_dim
\dim_new:N \l_coffin_Width_dim
```

(*End definition for* `\l_coffin_Depth_dim`. *This function is documented on page* **??**.)

\coffin_saved_Depth:
\coffin_saved_Height:
\coffin_saved_TotalHeight:
\coffin_saved_Width:

Used to save the meaning of `\Depth`, `\Height`, `\TotalHeight` and `\Width`.

```
\cs_new_nopar:Npn \coffin_saved_Depth:       { }
\cs_new_nopar:Npn \coffin_saved_Height:      { }
\cs_new_nopar:Npn \coffin_saved_TotalHeight: { }
\cs_new_nopar:Npn \coffin_saved_Width:       { }
```

(*End definition for* `\coffin_saved_Depth:`. *This function is documented on page* **??**.)

197.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

\coffin_if_exist:NT Several of the higher-level coffin functions will give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```
\cs_new_protected:Npn \coffin_if_exist:NT #1#2
  {
    \cs_if_exist:NTF #1
      {
        \cs_if_exist:cTF { l_coffin_poles_ \int_value:w #1 _prop }
          { #2 }
          {
            \msg_kernel_error:nnx { coffins } { unknown-coffin }
              { \token_to_str:N #1 }
          }
      }
      {
        \msg_kernel_error:nnx { coffins } { unknown-coffin }
          { \token_to_str:N #1 }
      }
  }
```

(*End definition for* `\coffin_if_exist:NT`*. This function is documented on page* **??**.)

\coffin_clear:N
\coffin_clear:c

Clearing coffins means emptying the box and resetting all of the structures.

```
\cs_new_protected:Npn \coffin_clear:N #1
  {
    \coffin_if_exist:NT #1
      {
        \box_clear:N #1
        \coffin_reset_structure:N #1
      }
  }
\cs_generate_variant:Nn \coffin_clear:N { c }
```

(*End definition for* `\coffin_clear:N` *and* `\coffin_clear:c`*. These functions are documented on page* **??**.)

\coffin_new:N
\coffin_new:c

Creating a new coffin means making the underlying box and adding the data structures. These are created globally, as there is a need to avoid any strange effects if the coffin is created inside a group. This means that the usual rule about `\l_...` variables has to be broken.

```
\cs_new_protected:Npn \coffin_new:N #1
  {
    \box_new:N #1
    \prop_clear_new:c { l_coffin_corners_ \int_value:w #1 _prop }
    \prop_clear_new:c { l_coffin_poles_   \int_value:w #1 _prop }
    \prop_gset_eq:cN { l_coffin_corners_ \int_value:w #1 _prop }
```

```
      \c_coffin_corners_prop
    \prop_gset_eq:cN { l_coffin_poles_ \int_value:w #1 _prop }
      \c_coffin_poles_prop
  }
\cs_generate_variant:Nn \coffin_new:N { c }
```

(*End definition for* `\coffin_new:N` *and* `\coffin_new:c`*. These functions are documented on page* **??**.)

`\hcoffin_set:Nn`
`\hcoffin_set:cn`

Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
\cs_new_protected:Npn \hcoffin_set:Nn #1#2
  {
    \coffin_if_exist:NT #1
      {
        \hbox_set:Nn #1
          {
            \color_group_begin:
              \color_ensure_current:
              #2
            \color_group_end:
          }
        \coffin_reset_structure:N #1
        \coffin_update_poles:N #1
        \coffin_update_corners:N #1
      }
  }
\cs_generate_variant:Nn \hcoffin_set:Nn { c }
```

(*End definition for* `\hcoffin_set:Nn` *and* `\hcoffin_set:cn`*. These functions are documented on page* **??**.)

`\vcoffin_set:Nnn`
`\vcoffin_set:cnn`

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box.

```
\cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
  {
    \coffin_if_exist:NT #1
      {
        \vbox_set:Nn #1
          {
            \dim_set:Nn \tex_hsize:D {#2}
            \color_group_begin:
              \color_ensure_current:
              #3
            \color_group_end:
          }
        \coffin_reset_structure:N #1
        \coffin_update_poles:N #1
        \coffin_update_corners:N #1
        \vbox_set_top:Nn \l_coffin_internal_box { \vbox_unpack:N #1 }
        \coffin_set_pole:Nnx #1 { T }
```

```
          {
            { 0 pt }
            { \dim_eval:n { \box_ht:N #1 - \box_ht:N \l_coffin_internal_box } }
            { 1000 pt }
            { 0 pt }
          }
        \box_clear:N \l_coffin_internal_box
      }
  }
\cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(*End definition for* `\vcoffin_set:Nnn` *and* `\vcoffin_set:cnn`*. These functions are documented on page* **??**.)

`\hcoffin_set:Nw`
`\hcoffin_set:cw`
`\hcoffin_set_end:`

These are the "begin"/"end" versions of the above: watch the grouping!

```
\cs_new_protected:Npn \hcoffin_set:Nw #1
  {
    \coffin_if_exist:NT #1
      {
        \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
          \cs_set_protected_nopar:Npn \hcoffin_set_end:
            {
                \color_group_end:
              \hbox_set_end:
              \coffin_reset_structure:N #1
              \coffin_update_poles:N #1
              \coffin_update_corners:N #1
            }
      }
  }
\cs_new_protected_nopar:Npn \hcoffin_set_end: { }
\cs_generate_variant:Nn \hcoffin_set:Nw { c }
```

(*End definition for* `\hcoffin_set:Nw` *and* `\hcoffin_set:cw`*. These functions are documented on page* **??**.)

`\vcoffin_set:Nnw`
`\vcoffin_set:cnw`
`\vcoffin_set_end:`

The same for vertical coffins.

```
\cs_new_protected:Npn \vcoffin_set:Nnw #1#2
  {
    \coffin_if_exist:NT #1
      {
        \vbox_set:Nw #1
          \dim_set:Nn \tex_hsize:D {#2}
          \color_group_begin: \color_ensure_current:
          \cs_set_protected:Npn \vcoffin_set_end:
            {
                \color_group_end:
              \vbox_set_end:
              \coffin_reset_structure:N #1
              \coffin_update_poles:N #1
              \coffin_update_corners:N #1
```

```
                \vbox_set_top:Nn \l_coffin_internal_box { \vbox_unpack:N #1 }
                \coffin_set_pole:Nnx #1 { T }
                  {
                    { 0 pt }
                    {
                      \dim_eval:n { \box_ht:N #1 - \box_ht:N \l_coffin_internal_box }
                    }
                    { 1000 pt }
                    { 0 pt }
                  }
                \box_clear:N \l_coffin_internal_box
              }
        }
  }
\cs_new_protected_nopar:Npn \vcoffin_set_end: { }
\cs_generate_variant:Nn \vcoffin_set:Nnw { c }
```

(*End definition for* `\vcoffin_set:Nnw` *and* `\vcoffin_set:cnw`*. These functions are documented on page* **??**.)

`\coffin_set_eq:NN`
`\coffin_set_eq:Nc`
`\coffin_set_eq:cN`
`\coffin_set_eq:cc`

Setting two coffins equal is just a wrapper around other functions.

```
\cs_new_protected:Npn \coffin_set_eq:NN #1#2
  {
    \coffin_if_exist:NT #1
      {
        \box_set_eq:NN #1 #2
        \coffin_set_eq_structure:NN #1 #2
      }
  }
\cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(*End definition for* `\coffin_set_eq:NN` *and others. These functions are documented on page* **??**.)

`\c_empty_coffin`
`\l_coffin_aligned_coffin`
`\l_coffin_aligned_internal_coffin`

Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available.

```
\coffin_new:N \c_empty_coffin
\hbox_set:Nn  \c_empty_coffin { }
\coffin_new:N \l_coffin_aligned_coffin
\coffin_new:N \l_coffin_aligned_internal_coffin
```

(*End definition for* `\c_empty_coffin`*. This function is documented on page* **??**.)

197.3 Measuring coffins

`\coffin_dp:N`
`\coffin_dp:c`
`\coffin_ht:N`
`\coffin_ht:c`
`\coffin_wd:N`
`\coffin_wd:c`

Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```
\cs_new_eq:NN \coffin_dp:N \box_dp:N
\cs_new_eq:NN \coffin_dp:c \box_dp:c
\cs_new_eq:NN \coffin_ht:N \box_ht:N
\cs_new_eq:NN \coffin_ht:c \box_ht:c
```

```
\cs_new_eq:NN \coffin_wd:N \box_wd:N
\cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(*End definition for* `\coffin_dp:N` *and others. These functions are documented on page* **??**.)

197.4 Coffins: handle and pole management

`\coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
\cs_new_protected:Npn \coffin_get_pole:NnN #1#2#3
  {
    \prop_get:cnNF
      { l_coffin_poles_ \int_value:w #1 _prop } {#2} #3
      {
        \msg_kernel_error:nnxx { coffins } { unknown-coffin-pole }
          {#2} { \token_to_str:N #1 }
        \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
      }
  }
```

(*End definition for* `\coffin_get_pole:NnN`*. This function is documented on page* **??**.)

`\coffin_reset_structure:N` Resetting the structure is a simple copy job.

```
\cs_new_protected:Npn \coffin_reset_structure:N #1
  {
    \prop_set_eq:cN { l_coffin_corners_ \int_value:w #1 _prop }
      \c_coffin_corners_prop
    \prop_set_eq:cN { l_coffin_poles_ \int_value:w #1 _prop }
      \c_coffin_poles_prop
  }
```

(*End definition for* `\coffin_reset_structure:N`*. This function is documented on page* **??**.)

`\coffin_set_eq_structure:NN`
`\coffin_gset_eq_structure:NN`

Setting coffin structures equal simply means copying the property list.

```
\cs_new_protected:Npn \coffin_set_eq_structure:NN #1#2
  {
    \prop_set_eq:cc { l_coffin_corners_ \int_value:w #1 _prop }
      { l_coffin_corners_ \int_value:w #2 _prop }
    \prop_set_eq:cc { l_coffin_poles_ \int_value:w #1 _prop }
      { l_coffin_poles_ \int_value:w #2 _prop }
  }
\cs_new_protected:Npn \coffin_gset_eq_structure:NN #1#2
  {
    \prop_gset_eq:cc { l_coffin_corners_ \int_value:w #1 _prop }
      { l_coffin_corners_ \int_value:w #2 _prop }
    \prop_gset_eq:cc { l_coffin_poles_ \int_value:w #1 _prop }
      { l_coffin_poles_ \int_value:w #2 _prop }
  }
```

(*End definition for* `\coffin_set_eq_structure:NN` *and* `\coffin_gset_eq_structure:NN`*. These functions are documented on page* **??**.)

\coffin_set_user_dimensions:N
\coffin_end_user_dimensions:
\Depth
\Height
\TotalHeight
\Width

These make design-level names for the dimensions of a coffin easy to get at.

```
\cs_new_protected:Npn \coffin_set_user_dimensions:N #1
  {
    \cs_set_eq:NN \coffin_saved_Height:      \Height
    \cs_set_eq:NN \coffin_saved_Depth:       \Depth
    \cs_set_eq:NN \coffin_saved_TotalHeight: \TotalHeight
    \cs_set_eq:NN \coffin_saved_Width:       \Width
    \cs_set_eq:NN \Height      \l_coffin_Height_dim
    \cs_set_eq:NN \Depth       \l_coffin_Depth_dim
    \cs_set_eq:NN \TotalHeight \l_coffin_TotalHeight_dim
    \cs_set_eq:NN \Width       \l_coffin_Width_dim
    \dim_set:Nn \Height      { \box_ht:N #1 }
    \dim_set:Nn \Depth       { \box_dp:N #1 }
    \dim_set:Nn \TotalHeight { \box_ht:N #1 + \box_dp:N #1 }
    \dim_set:Nn \Width       { \box_wd:N #1 }
  }
\cs_new_protected_nopar:Npn \coffin_end_user_dimensions:
  {
    \cs_set_eq:NN \Height      \coffin_saved_Height:
    \cs_set_eq:NN \Depth       \coffin_saved_Depth:
    \cs_set_eq:NN \TotalHeight \coffin_saved_TotalHeight:
    \cs_set_eq:NN \Width       \coffin_saved_Width:
  }
```

(*End definition for* `\coffin_set_user_dimensions:N`*. This function is documented on page* **??**.)

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_set_pole:Nnn
\coffin_set_pole:Nnx

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```
\cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
  {
    \coffin_if_exist:NT #1
      {
        \coffin_set_user_dimensions:N #1
        \coffin_set_pole:Nnx #1 {#2}
          {
            { 0 pt } { \dim_eval:n {#3} }
            { 1000 pt } { 0 pt }
          }
        \coffin_end_user_dimensions:
      }
  }
\cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
  {
    \coffin_if_exist:NT #1
      {
        \coffin_set_user_dimensions:N #1
        \coffin_set_pole:Nnx #1 {#2}
          {
            { \dim_eval:n {#3} } { 0 pt }
```

```
            { 0 pt } { 1000 pt }
          }
        \coffin_end_user_dimensions:
      }
  }
\cs_new_protected:Npn \coffin_set_pole:Nnn #1#2#3
  { \prop_put:cnn { l_coffin_poles_ \int_value:w #1 _prop } {#2} {#3} }
\cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
\cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
\cs_generate_variant:Nn \coffin_set_pole:Nnn { Nnx }
```

(*End definition for* `\coffin_set_horizontal_pole:Nnn` *and* `\coffin_set_horizontal_pole:cnn`*. These functions are documented on page* **??**.)

`\coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TEX box.

```
\cs_new_protected:Npn \coffin_update_corners:N #1
  {
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } { tl }
      { { 0 pt } { \dim_use:N \box_ht:N #1 } }
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } { tr }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } { bl }
      { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } { br }
      { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
  }
```

(*End definition for* `\coffin_update_corners:N`*. This function is documented on page* **??**.)

`\coffin_update_poles:N` This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```
\cs_new_protected:Npn \coffin_update_poles:N #1
  {
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } { hc }
      {
        { \dim_eval:n { 0.5 \box_wd:N #1 } }
        { 0 pt } { 0 pt } { 1000 pt }
      }
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } { r }
      {
        { \dim_use:N \box_wd:N #1 }
        { 0 pt } { 0 pt } { 1000 pt }
      }
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } { vc }
      {
        { 0 pt }
        { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
        { 1000 pt }
```

```
        { 0 pt }
      }
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } { t }
      {
        { 0 pt }
        { \dim_use:N \box_ht:N #1 }
        { 1000 pt }
        { 0 pt }
      }
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } { b }
      {
        { 0 pt }
        { \dim_eval:n { - \box_dp:N #1 } }
        { 1000 pt }
        { 0 pt }
      }
  }
```

(*End definition for* `\coffin_update_poles:N`. *This function is documented on page* **??**.)

197.5 Coffins: calculation of pole intersections

\coffin_calculate_intersection:Nnn
\coffin_calculate_intersection:nnnnnnnn
\coffin_calculate_intersection_aux:nnnnnN

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```
\cs_new_protected:Npn \coffin_calculate_intersection:Nnn #1#2#3
  {
    \coffin_get_pole:NnN #1 {#2} \l_coffin_pole_a_tl
    \coffin_get_pole:NnN #1 {#3} \l_coffin_pole_b_tl
    \bool_set_false:N \l_coffin_error_bool
    \exp_last_two_unbraced:Noo
      \coffin_calculate_intersection:nnnnnnnn
        \l_coffin_pole_a_tl \l_coffin_pole_b_tl
    \bool_if:NT \l_coffin_error_bool
      {
        \msg_kernel_error:nn { coffins } { no-pole-intersection }
        \dim_zero:N \l_coffin_x_dim
        \dim_zero:N \l_coffin_y_dim
      }
  }
```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d'). These are arguments 1–4 and 5–8, respectively. In both cases a and b are the co-ordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c'. However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' will be zero and a special case is needed.

```
\cs_new_protected:Npn \coffin_calculate_intersection:nnnnnnnn
  #1#2#3#4#5#6#7#8
  {
```

```
    \dim_compare:nNnTF {#3} = { \c_zero_dim }
```

The case where the first pole is vertical. So the x-component of the interaction will be at a. There is then a test on the second pole: if it is also vertical then there is an error.

```
      {
        \dim_set:Nn \l_coffin_x_dim {#1}
        \dim_compare:nNnTF {#7} = \c_zero_dim
          { \bool_set_true:N \l_coffin_error_bool }
```

The second pole may still be horizontal, in which case the y-component of the intersection will be b'. If not,

$$y = \frac{d'}{c'}(x - a') + b'$$

with the x-component already known to be `#1`. This calculation is done as a generalised auxiliary.

```
          {
            \dim_compare:nNnTF {#8} = \c_zero_dim
              { \dim_set:Nn \l_coffin_y_dim {#6} }
              {
                \coffin_calculate_intersection_aux:nnnnnN
                  {#1} {#5} {#6} {#7} {#8} \l_coffin_y_dim
              }
          }
      }
```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x- and y-components interchanged.

```
      {
        \dim_compare:nNnTF {#4} = \c_zero_dim
          {
            \dim_set:Nn \l_coffin_y_dim {#2}
            \dim_compare:nNnTF {#8} = { \c_zero_dim }
              { \bool_set_true:N \l_coffin_error_bool }
              {
                \dim_compare:nNnTF {#7} = \c_zero_dim
                  { \dim_set:Nn \l_coffin_x_dim {#5} }
```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'}(y - b') + a'$$

which is again handled by the same auxiliary.

```
                  {
                    \coffin_calculate_intersection_aux:nnnnnN
                      {#2} {#6} {#5} {#8} {#7} \l_coffin_x_dim
                  }
              }
          }
```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```
        {
          \dim_compare:nNnTF {#7} = \c_zero_dim
            {
              \dim_set:Nn \l_coffin_x_dim {#5}
              \coffin_calculate_intersection_aux:nnnnnN
                {#5} {#1} {#2} {#3} {#4} \l_coffin_y_dim
            }
            {
              \dim_compare:nNnTF {#8} = \c_zero_dim
                {
                  \dim_set:Nn \l_coffin_x_dim {#6}
                  \coffin_calculate_intersection_aux:nnnnnN
                    {#6} {#2} {#1} {#4} {#3} \l_coffin_x_dim
                }
```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```
              {
                  \fp_set_from_dim:Nn \l_coffin_calc_a_fp {#3}
                  \fp_set_from_dim:Nn \l_coffin_calc_b_fp {#4}
                  \fp_set_from_dim:Nn \l_coffin_calc_c_fp {#7}
                  \fp_set_from_dim:Nn \l_coffin_calc_d_fp {#8}
                  \fp_div:Nn \l_coffin_calc_b_fp \l_coffin_calc_a_fp
                  \fp_div:Nn \l_coffin_calc_d_fp \l_coffin_calc_c_fp
                  \fp_compare:nNnTF
                    \l_coffin_calc_b_fp = \l_coffin_calc_d_fp
                    { \bool_set_true:N \l_coffin_error_bool }
```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y-values is then worked out using the standard auxiliary starting from the x-position.

```
                    {
                      \fp_set_from_dim:Nn \l_coffin_calc_result_fp {#6}
                      \fp_set_from_dim:Nn \l_coffin_calc_a_fp {#2}
                      \fp_sub:Nn \l_coffin_calc_result_fp
                        { \l_coffin_calc_a_fp }
                      \fp_set_from_dim:Nn \l_coffin_calc_a_fp {#1}
                      \fp_mul:Nn \l_coffin_calc_a_fp
                        { \l_coffin_calc_b_fp }
                      \fp_add:Nn \l_coffin_calc_result_fp
                        { \l_coffin_calc_a_fp }
```

```
                        \fp_set_from_dim:Nn \l_coffin_calc_a_fp {#5}
                        \fp_mul:Nn \l_coffin_calc_a_fp
                          { \l_coffin_calc_d_fp }
                        \fp_sub:Nn \l_coffin_calc_result_fp
                          { \l_coffin_calc_a_fp }
                        \fp_sub:Nn \l_coffin_calc_b_fp
                          { \l_coffin_calc_d_fp }
                        \fp_div:Nn \l_coffin_calc_result_fp
                          { \l_coffin_calc_b_fp }
                        \dim_set:Nn \l_coffin_x_dim
                          { \fp_to_dim:N \l_coffin_calc_result_fp }
                        \coffin_calculate_intersection_aux:nnnnnN
                          { \l_coffin_x_dim }
                          {#5} {#6} {#8} {#7} \l_coffin_y_dim
                      }
                  }
              }
          }
      }
  }
```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \frac{\#5}{\#4} (\#1 - \#2) + \#3$$

Thus `#4` and `#5` should be the directions of the pole while `#2` and `#3` are co-ordinates.

```
\cs_new_protected:Npn \coffin_calculate_intersection_aux:nnnnnN
  #1#2#3#4#5#6
  {
    \fp_set_from_dim:Nn \l_coffin_calc_result_fp {#1}
    \fp_set_from_dim:Nn \l_coffin_calc_a_fp {#2}
    \fp_set_from_dim:Nn \l_coffin_calc_b_fp {#3}
    \fp_set_from_dim:Nn \l_coffin_calc_c_fp {#4}
    \fp_set_from_dim:Nn \l_coffin_calc_d_fp {#5}
    \fp_sub:Nn \l_coffin_calc_result_fp { \l_coffin_calc_a_fp }
    \fp_div:Nn \l_coffin_calc_result_fp { \l_coffin_calc_d_fp }
    \fp_mul:Nn \l_coffin_calc_result_fp { \l_coffin_calc_c_fp }
    \fp_add:Nn \l_coffin_calc_result_fp { \l_coffin_calc_b_fp }
    \dim_set:Nn #6 { \fp_to_dim:N \l_coffin_calc_result_fp }
  }
```

(*End definition for `\coffin_calculate_intersection:Nnn`. This function is documented on page* **??**.)

197.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnncnnnn`
`\coffin_join:cnncnnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```
\cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
```

```
  {
    \coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l_coffin_aligned_coffin
```

Correct the placement of the reference point. If the x-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which will show up if it is wider than the sum of the x-offset and the width of the second box. So a second kern may be needed.

```
    \hbox_set:Nn \l_coffin_aligned_coffin
      {
        \dim_compare:nNnT { \l_coffin_offset_x_dim } < \c_zero_dim
          { \tex_kern:D -\l_coffin_offset_x_dim }
        \hbox_unpack:N \l_coffin_aligned_coffin
        \dim_set:Nn \l_coffin_internal_dim
          { \l_coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
        \dim_compare:nNnT \l_coffin_internal_dim < \c_zero_dim
          { \tex_kern:D -\l_coffin_internal_dim }
      }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
   \coffin_reset_structure:N \l_coffin_aligned_coffin
    \prop_clear:c
      {  l_coffin_corners_ \int_value:w \l_coffin_aligned_coffin _ prop }
    \coffin_update_poles:N \l_coffin_aligned_coffin
```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That will then depend on whether any shift was needed.

```
    \dim_compare:nNnTF \l_coffin_offset_x_dim < \c_zero_dim
      {
        \coffin_offset_poles:Nnn #1 { -\l_coffin_offset_x_dim } { 0 pt }
        \coffin_offset_poles:Nnn #4 { 0 pt } { \l_coffin_offset_y_dim }
        \coffin_offset_corners:Nnn #1 { -\l_coffin_offset_x_dim } { 0 pt }
        \coffin_offset_corners:Nnn #4 { 0 pt } { \l_coffin_offset_y_dim }
      }
      {
        \coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
        \coffin_offset_poles:Nnn #4
          { \l_coffin_offset_x_dim } { \l_coffin_offset_y_dim }
        \coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
        \coffin_offset_corners:Nnn #4
          { \l_coffin_offset_x_dim } { \l_coffin_offset_y_dim }
      }
    \coffin_update_vertical_poles:NNN #1 #4 \l_coffin_aligned_coffin
    \coffin_set_eq:NN #1 \l_coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_join:NnnNnnnn` *and others. These functions are documented on page* **??**.)

\coffin_attach:NnnNnnnn
\coffin_attach:cnnNnnnn
\coffin_attach:Nnncnnnn
\coffin_attach:cnncnnnn
\coffin_attach_mark:NnnNnnnn

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```
\cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l_coffin_aligned_coffin
    \box_set_ht:Nn \l_coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l_coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l_coffin_aligned_coffin { \box_wd:N #1 }
    \coffin_reset_structure:N \l_coffin_aligned_coffin
    \prop_set_eq:cc
      { l_coffin_corners_ \int_value:w \l_coffin_aligned_coffin _prop }
      { l_coffin_corners_ \int_value:w #1 _prop }
    \coffin_update_poles:N  \l_coffin_aligned_coffin
    \coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
    \coffin_offset_poles:Nnn #4
      { \l_coffin_offset_x_dim } { \l_coffin_offset_y_dim }
    \coffin_update_vertical_poles:NNN #1 #4 \l_coffin_aligned_coffin
    \coffin_set_eq:NN #1 \l_coffin_aligned_coffin
  }
\cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l_coffin_aligned_coffin
    \box_set_ht:Nn \l_coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l_coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l_coffin_aligned_coffin { \box_wd:N #1 }
    \box_set_eq:NN #1 \l_coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_attach:NnnNnnnn` *and others. These functions are documented on page* **??**.)

\coffin_align:NnnNnnnnN

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the 'primed' storage area to be used for the second coffin. The 'real' box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result will depend on how the bounding box is being handled.

```
\cs_new_protected:Npn \coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \coffin_calculate_intersection:Nnn #4 {#5} {#6}
    \dim_set:Nn \l_coffin_x_prime_dim { \l_coffin_x_dim }
    \dim_set:Nn \l_coffin_y_prime_dim { \l_coffin_y_dim }
    \coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l_coffin_offset_x_dim
```

```
      { \l_coffin_x_dim - \l_coffin_x_prime_dim + #7 }
    \dim_set:Nn \l_coffin_offset_y_dim
      { \l_coffin_y_dim - \l_coffin_y_prime_dim + #8 }
    \hbox_set:Nn \l_coffin_aligned_internal_coffin
      {
        \box_use:N #1
        \tex_kern:D -\box_wd:N #1
        \tex_kern:D \l_coffin_offset_x_dim
        \box_move_up:nn { \l_coffin_offset_y_dim } { \box_use:N #4 }
      }
    \coffin_set_eq:NN #9 \l_coffin_aligned_internal_coffin
  }
```

(*End definition for* `\coffin_align:NnnNnnnnN`*. This function is documented on page* **??**.)

`\coffin_offset_poles:Nnn`
`\coffin_offset_pole:Nnnnnnn`

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a `-` means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that `-` should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```
\cs_new_protected:Npn \coffin_offset_poles:Nnn #1#2#3
  {
    \prop_map_inline:cn { l_coffin_poles_ \int_value:w #1 _prop }
      { \coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
  {
    \dim_set:Nn \l_coffin_x_dim { #3 + #7 }
    \dim_set:Nn \l_coffin_y_dim { #4 + #8 }
    \tl_if_in:nnTF {#2} { - }
      { \tl_set:Nn \l_coffin_internal_tl { {#2} } }
      { \tl_set:Nn \l_coffin_internal_tl { { #1 - #2 } } }
    \exp_last_unbraced:NNo \coffin_set_pole:Nnx \l_coffin_aligned_coffin
      { \l_coffin_internal_tl }
      {
        { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for* `\coffin_offset_poles:Nnn`*. This function is documented on page* **??**.)

`\coffin_offset_corners:Nnn`
`\coffin_offset_corners:Nnnnn`

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```
\cs_new_protected:Npn \coffin_offset_corners:Nnn #1#2#3
  {
    \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
      { \coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
  }
```

```
\cs_new_protected:Npn \coffin_offset_corner:Nnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx
      { l_coffin_corners_ \int_value:w \l_coffin_aligned_coffin _prop }
      { #1 - #2 }
      {
        { \dim_eval:n { #3 + #5 } }
        { \dim_eval:n { #4 + #6 } }
      }
  }
```

(*End definition for* `\coffin_offset_corners:Nnn`. *This function is documented on page* **??**.)

`\coffin_update_vertical_poles:NNN`
`\coffin_update_T:nnnnnnnnN`
`\coffin_update_B:nnnnnnnnN`

The `T` and `B` poles will need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```
\cs_new_protected:Npn \coffin_update_vertical_poles:NNN #1#2#3
  {
    \coffin_get_pole:NnN #3 { #1 -T } \l_coffin_pole_a_tl
    \coffin_get_pole:NnN #3 { #2 -T } \l_coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \coffin_update_T:nnnnnnnnN
      \l_coffin_pole_a_tl \l_coffin_pole_b_tl #3
    \coffin_get_pole:NnN #3 { #1 -B } \l_coffin_pole_a_tl
    \coffin_get_pole:NnN #3 { #2 -B } \l_coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \coffin_update_B:nnnnnnnnN
      \l_coffin_pole_a_tl \l_coffin_pole_b_tl #3
  }
\cs_new_protected:Npn \coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \coffin_set_pole:Nnx #9 { T }
          { { 0 pt } {#6} { 1000 pt } { 0 pt } }
      }
      {
        \coffin_set_pole:Nnx #9 { T }
          { { 0 pt } {#2} { 1000 pt } { 0 pt } }
      }
  }
\cs_new_protected:Npn \coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \coffin_set_pole:Nnx #9 { B }
          { { 0 pt } {#2}  { 1000 pt } { 0 pt } }
      }
      {
        \coffin_set_pole:Nnx #9 { B }
          { { 0 pt } {#6} { 1000 pt } { 0 pt } }
      }
```

```
  }
```

(*End definition for* `\coffin_update_vertical_poles:NNN`*. This function is documented on page* **??**.)

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
\cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
  {
    \coffin_align:NnnNnnnnN \c_empty_coffin { H } { l }
      #1 {#2} {#3} {#4} {#5} \l_coffin_aligned_coffin
    \hbox_unpack:N \c_empty_box
    \box_use:N \l_coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(*End definition for* `\coffin_typeset:Nnnnn` *and* `\coffin_typeset:cnnnn`*. These functions are documented on page* **??**.)

197.7 Rotating coffins

`\l_coffin_bounding_prop`

A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
\prop_new:N \l_coffin_bounding_prop
```

(*End definition for* `\l_coffin_bounding_prop`*. This variable is documented on page* **??**.)

`\l_coffin_bounding_shift_dim`

The shift of the bounding box of a coffin from the real content.

```
\dim_new:N \l_coffin_bounding_shift_dim
```

(*End definition for* `\l_coffin_bounding_shift_dim`*. This variable is documented on page* **??**.)

`\l_coffin_left_corner_dim`
`\l_coffin_right_corner_dim`
`\l_coffin_bottom_corner_dim`
`\l_coffin_top_corner_dim`

These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
\dim_new:N \l_coffin_left_corner_dim
\dim_new:N \l_coffin_right_corner_dim
\dim_new:N \l_coffin_bottom_corner_dim
\dim_new:N \l_coffin_top_corner_dim
```

(*End definition for* `\l_coffin_left_corner_dim`*. This function is documented on page* **??**.)

`\coffin_rotate:Nn`
`\coffin_rotate:cn`

Rotating a coffin requires several steps which can be conveniently run together. The first step is to convert the angle given in degrees to one in radians. This is then used to set `\l_coffin_sin_fp` and `\l_coffin_cos_fp`, which are carried through unchanged for the rest of the procedure.

```
\cs_new_protected:Npn \coffin_rotate:Nn #1#2
  {
    \fp_set:Nn \l_coffin_internal_fp {#2}
    \fp_div:Nn \l_coffin_internal_fp { 180 }
    \fp_mul:Nn \l_coffin_internal_fp { \c_pi_fp }
    \fp_sin:Nn \l_coffin_sin_fp { \l_coffin_internal_fp }
    \fp_cos:Nn \l_coffin_cos_fp { \l_coffin_internal_fp }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
    \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
      { \coffin_rotate_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l_coffin_poles_ \int_value:w #1 _prop }
      { \coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
    \coffin_set_bounding:N #1
    \prop_map_inline:Nn \l_coffin_bounding_prop
      { \coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
    \coffin_find_corner_maxima:N #1
    \coffin_find_bounding_shift:
    \box_rotate:Nn #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x-direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y-direction is dealt with by moving the box down by any depth it has acquired.

```
    \hbox_set:Nn #1
      {
        \tex_kern:D \l_coffin_bounding_shift_dim
        \tex_kern:D -\l_coffin_left_corner_dim
        \box_move_down:nn { \l_coffin_bottom_corner_dim }
          { \box_use:N #1 }
      }
```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content.

```
    \box_set_ht:Nn #1
      { \l_coffin_top_corner_dim - \l_coffin_bottom_corner_dim }
    \box_set_dp:Nn #1 { 0 pt }
    \box_set_wd:Nn #1
      { \l_coffin_right_corner_dim - \l_coffin_left_corner_dim }
```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```
    \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
      { \coffin_shift_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l_coffin_poles_ \int_value:w #1 _prop }
      { \coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
  }
\cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

(*End definition for* `\coffin_rotate:Nn` *and* `\coffin_rotate:cn`*. These functions are documented on page* **??***.*)

`\coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```
\cs_new_protected:Npn \coffin_set_bounding:N #1
  {
   \prop_put:Nnx \l_coffin_bounding_prop { tl }
      { { 0 pt } { \dim_use:N \box_ht:N #1 } }
    \prop_put:Nnx \l_coffin_bounding_prop { tr }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
    \dim_set:Nn \l_coffin_internal_dim { - \box_dp:N #1 }
    \prop_put:Nnx \l_coffin_bounding_prop { bl }
      { { 0 pt } { \dim_use:N \l_coffin_internal_dim } }
    \prop_put:Nnx \l_coffin_bounding_prop { br }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l_coffin_internal_dim } }
  }
```

(*End definition for* `\coffin_set_bounding:N`*. This function is documented on page* **??***.*)

`\coffin_rotate_bounding:nnn` `\coffin_rotate_corner:Nnnn` Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```
\cs_new_protected:Npn \coffin_rotate_bounding:nnn #1#2#3
  {
    \coffin_rotate_vector:nnNN {#2} {#3} \l_coffin_x_dim \l_coffin_y_dim
    \prop_put:Nnx \l_coffin_bounding_prop {#1}
      { { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim } }
  }
\cs_new_protected:Npn \coffin_rotate_corner:Nnnn #1#2#3#4
  {
    \coffin_rotate_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } {#2}
      { { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim } }
  }
```

(*End definition for* `\coffin_rotate_bounding:nnn`*. This function is documented on page* **??***.*)

`\coffin_rotate_pole:Nnnnnn` Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```
\cs_new_protected:Npn \coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
  {
    \coffin_rotate_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
    \coffin_rotate_vector:nnNN {#5} {#6}
      \l_coffin_x_prime_dim \l_coffin_y_prime_dim
    \coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim }
        { \dim_use:N \l_coffin_x_prime_dim }
        { \dim_use:N \l_coffin_y_prime_dim }
      }
  }
```

(*End definition for* `\coffin_rotate_pole:Nnnnnn`*. This function is documented on page* **??**.)

`\coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l_coffin_cos_fp` and `\l_coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```
\cs_new_protected:Npn \coffin_rotate_vector:nnNN #1#2#3#4
  {
    \fp_set_from_dim:Nn \l_coffin_x_fp {#1}
    \fp_set_from_dim:Nn \l_coffin_y_fp {#2}
    \fp_set_eq:NN \l_coffin_x_prime_fp \l_coffin_x_fp
    \fp_set_eq:NN \l_coffin_internal_fp     \l_coffin_y_fp
    \fp_mul:Nn \l_coffin_x_prime_fp { \l_coffin_cos_fp }
    \fp_mul:Nn \l_coffin_internal_fp     { \l_coffin_sin_fp }
    \fp_sub:Nn \l_coffin_x_prime_fp { \l_coffin_internal_fp }
    \fp_set_eq:NN \l_coffin_y_prime_fp \l_coffin_y_fp
    \fp_set_eq:NN \l_coffin_internal_fp     \l_coffin_x_fp
    \fp_mul:Nn \l_coffin_y_prime_fp { \l_coffin_cos_fp }
    \fp_mul:Nn \l_coffin_internal_fp     { \l_coffin_sin_fp }
    \fp_add:Nn \l_coffin_y_prime_fp { \l_coffin_internal_fp }
    \dim_set:Nn #3 { \fp_to_dim:N \l_coffin_x_prime_fp }
    \dim_set:Nn #4 { \fp_to_dim:N \l_coffin_y_prime_fp }
  }
```

(*End definition for* `\coffin_rotate_vector:nnNN`*. This function is documented on page* **??**.)

`\coffin_find_corner_maxima:N`
`\coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```
\cs_new_protected:Npn \coffin_find_corner_maxima:N #1
  {
    \dim_set:Nn \l_coffin_top_corner_dim   { -\c_max_dim }
    \dim_set:Nn \l_coffin_right_corner_dim { -\c_max_dim }
    \dim_set:Nn \l_coffin_bottom_corner_dim { \c_max_dim }
    \dim_set:Nn \l_coffin_left_corner_dim   { \c_max_dim }
    \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
      { \coffin_find_corner_maxima_aux:nn ##2 }
  }
\cs_new_protected:Npn \coffin_find_corner_maxima_aux:nn #1#2
  {
    \dim_set_min:Nn \l_coffin_left_corner_dim   {#1}
    \dim_set_max:Nn \l_coffin_right_corner_dim  {#1}
    \dim_set_min:Nn \l_coffin_bottom_corner_dim {#2}
    \dim_set_max:Nn \l_coffin_top_corner_dim    {#2}
  }
```

(*End definition for* `\coffin_find_corner_maxima:N`*. This function is documented on page* **??**.)

\coffin_find_bounding_shift:
\coffin_find_bounding_shift_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```
\cs_new_protected_nopar:Npn \coffin_find_bounding_shift:
  {
    \dim_set:Nn \l_coffin_bounding_shift_dim { \c_max_dim }
    \prop_map_inline:Nn \l_coffin_bounding_prop
      { \coffin_find_bounding_shift_aux:nn ##2 }
  }
\cs_new_protected:Npn \coffin_find_bounding_shift_aux:nn #1#2
  { \dim_set_min:Nn \l_coffin_bounding_shift_dim {#1} }
```

(*End definition for* `\coffin_find_bounding_shift:`. *This function is documented on page* **??**.)

\coffin_shift_corner:Nnnn
\coffin_shift_pole:Nnnnnn

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x- and y-components. For the poles, this means that the direction vector is unchanged.

```
\cs_new_protected:Npn \coffin_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l_coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l_coffin_bottom_corner_dim } }
      }
  }
\cs_new_protected:Npn \coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l_coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l_coffin_bottom_corner_dim } }
        {#5} {#6}
      }
  }
```

(*End definition for* `\coffin_shift_corner:Nnnn`. *This function is documented on page* **??**.)

197.8 Resizing coffins

\l_coffin_scale_x_fp
\l_coffin_scale_y_fp

Storage for the scaling factors in x and y, respectively.

```
\fp_new:N \l_coffin_scale_x_fp
\fp_new:N \l_coffin_scale_y_fp
```

(*End definition for* `\l_coffin_scale_x_fp`. *This function is documented on page* **??**.)

\l_coffin_scaled_total_height_dim
\l_coffin_scaled_width_dim

When scaling, the values given have to be turned into absolute values.

```
\dim_new:N \l_coffin_scaled_total_height_dim
\dim_new:N \l_coffin_scaled_width_dim
```

(*End definition for* `\l_coffin_scaled_total_height_dim`. *This function is documented on page* **??**.)

\coffin_resize:Nnn
\coffin_resize:cnn

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```
\cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
  {
    \coffin_set_user_dimensions:N #1
    \box_resize:Nnn #1 {#2} {#3}
    \fp_set_from_dim:Nn \l_coffin_scale_x_fp {#2}
    \fp_set_from_dim:Nn \l_coffin_internal_fp { \Width }
    \fp_div:Nn \l_coffin_scale_x_fp { \l_coffin_internal_fp }
    \fp_set_from_dim:Nn \l_coffin_scale_y_fp {#3}
    \fp_set_from_dim:Nn \l_coffin_internal_fp { \TotalHeight }
    \fp_div:Nn \l_coffin_scale_y_fp { \l_coffin_internal_fp }
    \coffin_resize_common:Nnn #1 {#2} {#3}
  }
\cs_generate_variant:Nn \coffin_resize:Nnn { c }
```

(*End definition for* `\coffin_resize:Nnn` *and* `\coffin_resize:cnn`*. These functions are documented on page* **??**.)

\coffin_resize_common:Nnn

The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
\cs_new_protected:Npn \coffin_resize_common:Nnn #1#2#3
  {
    \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
      { \coffin_scale_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l_coffin_poles_ \int_value:w #1 _prop }
      { \coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative x-scaling values will place the poles in the wrong location: this is corrected here.

```
    \fp_compare:NNNT \l_coffin_scale_x_fp < \c_zero_fp
      {
        \prop_map_inline:cn { l_coffin_corners_ \int_value:w #1 _prop }
          { \coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
        \prop_map_inline:cn { l_coffin_poles_ \int_value:w #1 _prop }
          { \coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
      }
    \coffin_end_user_dimensions:
  }
```

(*End definition for* `\coffin_resize_common:Nnn`*. This function is documented on page* **??**.)

\coffin_scale:Nnn
\coffin_scale:cnn

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the `fp` module.

```
\cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
  {
    \box_scale:Nnn #1 {#2} {#3}
```

```
    \coffin_set_user_dimensions:N #1
    \fp_set:Nn \l_coffin_scale_x_fp {#2}
    \fp_set:Nn \l_coffin_scale_y_fp {#3}
    \fp_compare:NNNTF \l_coffin_scale_y_fp > \c_zero_fp
      { \l_coffin_scaled_total_height_dim #3 \TotalHeight }
      { \l_coffin_scaled_total_height_dim -#3 \TotalHeight }
    \fp_compare:NNNTF \l_coffin_scale_x_fp > \c_zero_fp
      { \l_coffin_scaled_width_dim -#2 \Width }
      { \l_coffin_scaled_width_dim #2 \Width }
    \coffin_resize_common:Nnn #1
      { \l_coffin_scaled_width_dim } { \l_coffin_scaled_total_height_dim }
  }
\cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

(*End definition for* `\coffin_scale:Nnn` *and* `\coffin_scale:cnn`*. These functions are documented on page* **??**.)

`\coffin_scale_vector:nnNN`

This functions scales a vector from the origin using the pre-set scale factors in x and y. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```
\cs_new_protected:Npn \coffin_scale_vector:nnNN #1#2#3#4
  {
    \fp_set_from_dim:Nn \l_coffin_internal_fp {#1}
    \fp_mul:Nn \l_coffin_internal_fp { \l_coffin_scale_x_fp }
    \dim_set:Nn #3 { \fp_to_dim:N \l_coffin_internal_fp }
    \fp_set_from_dim:Nn \l_coffin_internal_fp {#2}
    \fp_mul:Nn \l_coffin_internal_fp { \l_coffin_scale_y_fp }
    \dim_set:Nn #4 { \fp_to_dim:N \l_coffin_internal_fp }
  }
```

(*End definition for* `\coffin_scale_vector:nnNN`*. This function is documented on page* **??**.)

`\coffin_scale_corner:Nnnn`
`\coffin_scale_pole:Nnnnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```
\cs_new_protected:Npn \coffin_scale_corner:Nnnn #1#2#3#4
  {
    \coffin_scale_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } {#2}
      { { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim } }
  }
\cs_new_protected:Npn \coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
  {
    \coffin_scale_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
    \coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for* `\coffin_scale_corner:Nnnn`*. This function is documented on page* **??**.)

`\coffin_x_shift_corner:Nnnn`
`\coffin_x_shift_pole:Nnnnnn`

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```
\cs_new_protected:Npn \coffin_x_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l_coffin_corners_ \int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
    }
  }
\cs_new_protected:Npn \coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx { l_coffin_poles_ \int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n #3 + \box_wd:N #1 } {#4}
        {#5} {#6}
      }
  }
```

(*End definition for* `\coffin_x_shift_corner:Nnnn`. *This function is documented on page* **??**.)

197.9 Coffin diagnostics

`\l_coffin_display_coffin`
`\l_coffin_display_coord_coffin`
`\l_coffin_display_pole_coffin`

Used for printing coffins with data structures attached.

```
\coffin_new:N \l_coffin_display_coffin
\coffin_new:N \l_coffin_display_coord_coffin
\coffin_new:N \l_coffin_display_pole_coffin
```

(*End definition for* `\l_coffin_display_coffin`. *This function is documented on page* **??**.)

`\l_coffin_display_handles_prop`

This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
\prop_new:N \l_coffin_display_handles_prop
\prop_put:Nnn \l_coffin_display_handles_prop { tl }
  { { b } { r } { -1 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { thc }
  { { b } { hc } { 0 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { tr }
  { { b } { l } { 1 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { vcl }
  { { vc } { r } { -1 } { 0 } }
\prop_put:Nnn \l_coffin_display_handles_prop { vchc }
  { { vc } { hc } { 0 } { 0 } }
\prop_put:Nnn \l_coffin_display_handles_prop { vcr }
  { { vc } { l } { 1 } { 0 } }
\prop_put:Nnn \l_coffin_display_handles_prop { bl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { bhc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { br }
  { { t } { l } { 1 } { -1 } }
```

```
\prop_put:Nnn \l_coffin_display_handles_prop { Tl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Thc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Tr }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Hl }
  { { vc } { r } { -1 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Hhc }
  { { vc } { hc } { 0 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Hr }
  { { vc } { l } { 1 } { 1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Bl }
  { { b } { r } { -1 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Bhc }
  { { b } { hc } { 0 } { -1 } }
\prop_put:Nnn \l_coffin_display_handles_prop { Br }
  { { b } { l } { 1 } { -1 } }
```

(*End definition for* `\l_coffin_display_handles_prop`. *This variable is documented on page* **??**.)

`\l_coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```
\dim_new:N  \l_coffin_display_offset_dim
\dim_set:Nn \l_coffin_display_offset_dim { 2 pt }
```

(*End definition for* `\l_coffin_display_offset_dim`. *This variable is documented on page* **??**.)

`\l_coffin_display_x_dim`
`\l_coffin_display_y_dim`

As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```
\dim_new:N \l_coffin_display_x_dim
\dim_new:N \l_coffin_display_y_dim
```

(*End definition for* `\l_coffin_display_x_dim`. *This function is documented on page* **??**.)

`\l_coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a "nice" output.

```
\prop_new:N \l_coffin_display_poles_prop
```

(*End definition for* `\l_coffin_display_poles_prop`. *This variable is documented on page* **??**.)

`\l_coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```
\tl_new:N  \l_coffin_display_font_tl
⟨*initex⟩
\tl_set:Nn \l_coffin_display_font_tl { } % TODO
⟨/initex⟩
⟨*package⟩
\tl_set:Nn \l_coffin_display_font_tl { \sffamily \tiny }
⟨/package⟩
```

(*End definition for* `\l_coffin_display_font_tl`. *This variable is documented on page* **??**.)

`\coffin_mark_handle:Nnnn`
`\coffin_mark_handle:cnnn`
`\coffin_mark_handle_aux:nnnnNnn`

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
\cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
  {
    \hcoffin_set:Nn \l_coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
        \rule { 1 pt } { 1 pt }
⟨/package⟩
      }
    \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
      \l_coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
    \hcoffin_set:Nn \l_coffin_display_coord_coffin
      {
⟨*initex⟩
        % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
⟨/package⟩
        \l_coffin_display_font_tl
        ( \tl_to_str:n { #2 , #3 } )
      }
    \prop_get:NnN \l_coffin_display_handles_prop
      { #2 #3 } \l_coffin_internal_tl
    \quark_if_no_value:NTF \l_coffin_internal_tl
      {
        \prop_get:NnN \l_coffin_display_handles_prop
          { #3 #2 } \l_coffin_internal_tl
        \quark_if_no_value:NTF \l_coffin_internal_tl
          {
            \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
              \l_coffin_display_coord_coffin { l } { vc }
                { 1 pt } { 0 pt }
          }
          {
            \exp_last_unbraced:No \coffin_mark_handle_aux:nnnnNnn
              \l_coffin_internal_tl #1 {#2} {#3}
          }
      }
      {
        \exp_last_unbraced:No \coffin_mark_handle_aux:nnnnNnn
          \l_coffin_internal_tl #1 {#2} {#3}
```

```
      }
  }
\cs_new_protected:Npn \coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
  {
    \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
      \l_coffin_display_coord_coffin {#1} {#2}
      { #3 \l_coffin_display_offset_dim }
      { #4 \l_coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(*End definition for* `\coffin_mark_handle:Nnnn` *and* `\coffin_mark_handle:cnnn`*. These functions are documented on page* **??**.)

`\coffin_display_handles:Nn`
`\coffin_display_handles:cn`
`\coffin_display_handles_aux:nnnnnn`
`\coffin_display_handles_aux:nnnn`
`\coffin_display_attach:Nnnnn`

Printing the poles starts by removing any duplicates, for which the `H` poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```
\cs_new_protected:Npn \coffin_display_handles:Nn #1#2
  {
    \hcoffin_set:Nn \l_coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#2}
        \rule { 1 pt } { 1 pt }
⟨/package⟩
      }
    \prop_set_eq:Nc \l_coffin_display_poles_prop
      { l_coffin_poles_ \int_value:w #1 _prop }
    \coffin_get_pole:NnN #1 { H } \l_coffin_pole_a_tl
    \coffin_get_pole:NnN #1 { T } \l_coffin_pole_b_tl
    \tl_if_eq:NNT \l_coffin_pole_a_tl \l_coffin_pole_b_tl
      { \prop_del:Nn \l_coffin_display_poles_prop { T } }
    \coffin_get_pole:NnN #1 { B } \l_coffin_pole_b_tl
    \tl_if_eq:NNT \l_coffin_pole_a_tl \l_coffin_pole_b_tl
      { \prop_del:Nn \l_coffin_display_poles_prop { B } }
    \coffin_set_eq:NN \l_coffin_display_coffin #1
    \prop_map_inline:Nn \l_coffin_display_poles_prop
      {
        \prop_del:Nn \l_coffin_display_poles_prop {##1}
        \coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
      }
    \box_use:N \l_coffin_display_coffin
  }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
\cs_new_protected:Npn \coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
  {
    \prop_map_inline:Nn \l_coffin_display_poles_prop
      {
        \bool_set_false:N \l_coffin_error_bool
        \coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
        \bool_if:NF \l_coffin_error_bool
          {
            \dim_set:Nn \l_coffin_display_x_dim { \l_coffin_x_dim }
            \dim_set:Nn \l_coffin_display_y_dim { \l_coffin_y_dim }
            \coffin_display_attach:Nnnnn
              \l_coffin_display_pole_coffin { hc } { vc }
              { 0 pt } { 0 pt }
            \hcoffin_set:Nn \l_coffin_display_coord_coffin
              {
⟨*initex⟩
                % TODO
⟨/initex⟩
⟨*package⟩
                \color {#6}
⟨/package⟩
                \l_coffin_display_font_tl
                ( \tl_to_str:n { #1 , ##1 } )
              }
            \prop_get:NnN \l_coffin_display_handles_prop
              { #1 ##1 } \l_coffin_internal_tl
            \quark_if_no_value:NTF \l_coffin_internal_tl
              {
                \prop_get:NnN \l_coffin_display_handles_prop
                  { ##1 #1 } \l_coffin_internal_tl
                \quark_if_no_value:NTF \l_coffin_internal_tl
                  {
                    \coffin_display_attach:Nnnnn
                      \l_coffin_display_coord_coffin { l } { vc }
                      { 1 pt } { 0 pt }
                  }
                  {
                    \exp_last_unbraced:No
                      \coffin_display_handles_aux:nnnn
                      \l_coffin_internal_tl
                  }
              }
              {
                \exp_last_unbraced:No \coffin_display_handles_aux:nnnn
                  \l_coffin_internal_tl
              }
          }
      }
  }
\cs_new_protected:Npn \coffin_display_handles_aux:nnnn #1#2#3#4
```

```
  {
    \coffin_display_attach:Nnnnn
      \l_coffin_display_coord_coffin {#1} {#2}
      { #3 \l_coffin_display_offset_dim }
      { #4 \l_coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_display_handles:Nn { c }
```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```
\cs_new_protected:Npn \coffin_display_attach:Nnnnn #1#2#3#4#5
  {
    \coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l_coffin_x_prime_dim { \l_coffin_x_dim }
    \dim_set:Nn \l_coffin_y_prime_dim { \l_coffin_y_dim }
    \dim_set:Nn \l_coffin_offset_x_dim
      { \l_coffin_display_x_dim - \l_coffin_x_prime_dim + #4 }
    \dim_set:Nn \l_coffin_offset_y_dim
      { \l_coffin_display_y_dim - \l_coffin_y_prime_dim + #5 }
    \hbox_set:Nn \l_coffin_aligned_coffin
      {
        \box_use:N \l_coffin_display_coffin
        \tex_kern:D -\box_wd:N \l_coffin_display_coffin
        \tex_kern:D \l_coffin_offset_x_dim
        \box_move_up:nn { \l_coffin_offset_y_dim } { \box_use:N #1 }
      }
    \box_set_ht:Nn \l_coffin_aligned_coffin
      { \box_ht:N \l_coffin_display_coffin }
    \box_set_dp:Nn \l_coffin_aligned_coffin
      { \box_dp:N \l_coffin_display_coffin }
    \box_set_wd:Nn \l_coffin_aligned_coffin
      { \box_wd:N \l_coffin_display_coffin }
    \box_set_eq:NN \l_coffin_display_coffin \l_coffin_aligned_coffin
  }
```

(*End definition for* `\coffin_display_handles:Nn` *and* `\coffin_display_handles:cn`*. These functions are documented on page 135.*)

`\coffin_show_structure:N`
`\coffin_show_structure:c`

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```
\cs_new_protected:Npn \coffin_show_structure:N #1
  {
    \coffin_if_exist:NT #1
      {
        \msg_aux_show:Nnx #1 { coffins }
          {
            \prop_map_function:cN
              { l_coffin_poles_ \int_value:w #1 _prop }
              \msg_aux_show_unbraced:nn
          }
```

```
      }
  }
\cs_generate_variant:Nn \coffin_show_structure:N { c }
```

(*End definition for* `\coffin_show_structure:N` *and* `\coffin_show_structure:c`*. These functions are documented on page* **??***.*)

197.10 Messages

```
\msg_kernel_new:nnnn { coffins } { no-pole-intersection }
  { No~intersection~between~coffin~poles. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
    but~they~do~not~have~a~unique~meeting~point:~
    the~value~(0~pt,~0~pt)~will~be~used.
  }
\msg_kernel_new:nnnn { coffins } { unknown-coffin }
  { Unknown~coffin~'#1'. }
  { The~coffin~'#1'~was~never~defined. }
\msg_kernel_new:nnnn { coffins } { unknown-coffin-pole }
  { Pole~'#1'~unknown~for~coffin~'#2'. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
    but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
  }
\msg_kernel_new:nnn { coffins } { show }
  {
    Size~of~coffin~\token_to_str:N #1 : \\
    > ~ ht~=~\dim_use:N \box_ht:N #1 \\
    > ~ dp~=~\dim_use:N \box_dp:N #1 \\
    > ~ wd~=~\dim_use:N \box_wd:N #1 \\
    Poles~of~coffin~\token_to_str:N #1 :
  }
```

⟨/initex | package⟩

198 l3color Implementation

⟨*initex | package⟩

⟨*package⟩

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

⟨/package⟩

`\color_group_begin:`
`\color_group_end:`

Grouping for colour is almost the same as using the basic `\group_begin:` and `\group_end:` functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```
\cs_new_eq:NN \color_group_begin: \group_begin:
```

```
\cs_new_protected_nopar:Npn \color_group_end:
  {
      \tex_par:D
    \group_end:
  }
```

(*End definition for* `\color_group_begin:` *and* `\color_group_end:`*. These functions are documented on page* **??**.)

`\color_ensure_current:` A driver-independent wrapper for setting the foreground colour to the current colour "now".

```
⟨*initex⟩
\cs_new_protected_nopar:Npn \color_ensure_current:
  { \driver_color_ensure_current: }
⟨/initex⟩
⟨*package⟩
\cs_new_protected_nopar:Npn \color_ensure_current: { \set@color }
⟨/package⟩
```

(*End definition for* `\color_ensure_current:`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

199 l3msg implementation

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

`\l_msg_internal_tl` A general scratch for the module.

```
\tl_new:N \l_msg_internal_tl
```

(*End definition for* `\l_msg_internal_tl`*. This variable is documented on page* **??**.)

199.1 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

`\c_msg_text_prefix_tl`
`\c_msg_more_text_prefix_tl` Locations for the text of messages.

```
\tl_const:Nn \c_msg_text_prefix_tl      { msg~text~>~ }
\tl_const:Nn \c_msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(*End definition for* `\c_msg_text_prefix_tl` *and* `\c_msg_more_text_prefix_tl`*. These variables are documented on page* **??**.)

\msg_if_exist_p:nn
\msg_if_exist:nnTF

Test whether the control sequence containing the message text exists or not.

```
\prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c_msg_text_prefix_tl #1 / #2 }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* `\msg_if_exist:nn`. *These functions are documented on page 138.*)

\chk_if_free_msg:nn

This auxiliary is similar to `\chk_if_free_cs:N`, and is used when defining messages with `\msg_new:nnnn`. It could be inlined in `\msg_new:nnnn`, but the experimental l3trace module needs to disable this check when reloading a package with the extra tracing information.

```
\cs_new_protected:Npn \chk_if_free_msg:nn #1#2
  {
    \msg_if_exist:nnT {#1} {#2}
      {
        \msg_kernel_error:nnxx { msg } { message-already-defined }
          {#1} {#2}
      }
  }
⟨*package⟩
\tex_ifodd:D \l@expl@log@functions@bool
  \cs_gset_protected:Npn \chk_if_free_msg:nn #1#2
    {
      \msg_if_exist:nnT {#1} {#2}
        {
          \msg_kernel_error:nnxx { msg } { message-already-defined }
            {#1} {#2}
        }
      \iow_log:x { Defining~message~#1/#2~ \msg_line_context: }
    }
\fi:
⟨/package⟩
```

(*End definition for* `\chk_if_free_msg:nn`.)

\msg_new:nnnn
\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
\cs_new_protected:Npn \msg_new:nnnn #1#2
  {
    \chk_if_free_msg:nn {#1} {#2}
    \msg_gset:nnnn {#1} {#2}
  }
\cs_new_protected:Npn \msg_new:nnn #1#2#3
  { \msg_new:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
  {
    \cs_set:cpn { \c_msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_set:cpn { \c_msg_more_text_prefix_tl #1 / #2 }
```

```
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_set:nnn #1#2#3
  { \msg_set:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
  {
    \cs_gset:cpn { \c_msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_gset:cpn { \c_msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_gset:nnn #1#2#3
  { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(*End definition for* `\msg_new:nnnn` *and* `\msg_new:nnn`*. These functions are documented on page* **??**.)

199.2 Messages: support functions and text

`\c_msg_coding_error_text_tl`
`\c_msg_continue_text_tl`
`\c_msg_critical_text_tl`
`\c_msg_fatal_text_tl`
`\c_msg_help_text_tl`
`\c_msg_no_info_text_tl`
`\c_msg_on_line_tl`
`\c_msg_return_text_tl`
`\c_msg_trouble_text_tl`

Simple pieces of text for messages.

```
\tl_const:Nn \c_msg_coding_error_text_tl
  {
    This~is~a~coding~error.
    \\ \\
  }
\tl_const:Nn \c_msg_continue_text_tl
  { Type~<return>~to~continue }
\tl_const:Nn \c_msg_critical_text_tl
  { Reading~the~current~file~will~stop }
\tl_const:Nn \c_msg_fatal_text_tl
  { This~is~a~fatal~error:~LaTeX~will~abort }
\tl_const:Nn \c_msg_help_text_tl
  { For~immediate~help~type~H~<return> }
\tl_const:Nn \c_msg_no_info_text_tl
  {
    LaTeX~does~not~know~anything~more~about~this~error,~sorry.
    \c_msg_return_text_tl
  }
\tl_const:Nn \c_msg_on_line_text_tl { on~line }
\tl_const:Nn \c_msg_return_text_tl
  {
    \\ \\
    Try~typing~<return>~to~proceed.
    \\
    If~that~doesn't~work,~type~X~<return>~to~quit.
  }
\tl_const:Nn \c_msg_trouble_text_tl
  {
    \\ \\
    More~errors~will~almost~certainly~follow: \\
    the~LaTeX~run~should~be~aborted.
```

```
  }
```

(*End definition for* `\c_msg_coding_error_text_tl` *and others. These variables are documented on page* 138.)

`\msg_newline:`
`\msg_two_newlines:`

New lines are printed in the same way as for low-level file writing.

```
\cs_new_nopar:Npn \msg_newline: { ^^J }
\cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }
```

(*End definition for* `\msg_newline:` *and* `\msg_two_newlines:`. *These functions are documented on page* ??.)

`\msg_line_number`
`\msg_line_context:`

For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not `new`.

```
\cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
\cs_gset_nopar:Npn \msg_line_context:
  {
    \c_msg_on_line_text_tl
    \c_space_tl
    \msg_line_number:
  }
```

(*End definition for* `\msg_line_number` *and* `\msg_line_context:`. *These functions are documented on page* ??.)

199.3 Showing messages: low level mechanism

`\msg_interrupt:xxx`

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TEX's `\errhelp` register before issuing an `\errmessage`.

```
\cs_new_protected:Npn \msg_interrupt:xxx #1#2#3
  {
    \tl_if_empty:nTF {#3}
      {
        \msg_interrupt_wrap:xx { \\ \c_msg_no_info_text_tl }
          {#1 \\\\ #2 \\\\ \c_msg_continue_text_tl }
      }
      {
        \msg_interrupt_wrap:xx { \\ #3 }
          {#1 \\\\ #2 \\\\ \c_msg_help_text_tl }
      }
  }
```

(*End definition for* `\msg_interrupt:xxx`. *This function is documented on page* 142.)

`\msg_interrupt_wrap:xx`
`\msg_interrupt_more_text:n`

First setup TEX's `\errhelp` register with the extra help `#1`, then build a nice-looking error message with `#2`. Everything is done using `x`-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\msg_interrupt_more_text:n` recieves its argument as a line-wrapped string, which is thus unaffected by expansion.

```
\cs_new_protected:Npn \msg_interrupt_wrap:xx #1#2
```

```
  {
    \iow_wrap:xnnnN {#1} { | ~ } { 2 } { } \msg_interrupt_more_text:n
    \iow_wrap:xnnnN {#2} { ! ~ } { 2 } { } \msg_interrupt_text:n
  }
\cs_new_protected:Npn \msg_interrupt_more_text:n #1
  {
    \exp_args:Nx \tex_errhelp:D
      {
        |''''''''''''''''''''''''''''''''''''''''''''''''
        #1 \iow_newline:
        |................................................
      }
  }
```

(*End definition for* `\msg_interrupt_wrap:xx`. *This function is documented on page 142.*)

\msg_interrupt_text:n
\c_msg_hide_tl<dots>

The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made "invisible": TeX's own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with an odd `\c_msg_hide_tl<dots>` which fills the output with dots. The trailing closing brace is turned into a space to hide it as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line will bee inserted after the message is entirely cleaned up.

```
\group_begin:
  \char_set_lccode:nn {`\{} {`\ }
  \char_set_lccode:nn {`\}} {`\ }
  \char_set_lccode:nn {`\&} {`\!}
  \char_set_catcode_active:N \&
  \char_set_catcode_letter:N \.
  \tl_new:N
    \c_msg_hide_tl..........................................
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \msg_interrupt_text:n #1
      {
        \iow_term:x
          {
            \iow_newline:
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            \iow_newline:
            !
          }
        \group_begin:
          \cs_set_protected_nopar:Npn &
            {
              \tex_errmessage:D
                {
```

```
                    #1
                    \c_msg_hide_tl..........................................
                  }
              }
            \exp_after:wN
          \group_end:
          &
        }
    }
```

(*End definition for* `\msg_interrupt_text:n`. *This function is documented on page* **??**.)

`\msg_log:x`
`\msg_term:x`

Printing to the log or terminal without a stop is rather easier. A bit of simple visual work sets things off nicely.

```
\cs_new_protected:Npn \msg_log:x #1
  {
    \iow_log:x { ................................................. }
    \iow_wrap:xnnnN { . ~ #1} { . ~ } { 2 } { }
      \iow_log:x
    \iow_log:x { ................................................. }
  }
\cs_new_protected:Npn \msg_term:x #1
  {
    \iow_term:x { ************************************************* }
    \iow_wrap:xnnnN { * ~ #1} { * ~ } { 2 } { }
      \iow_term:x
    \iow_term:x { ************************************************* }
  }
```

(*End definition for* `\msg_log:x`. *This function is documented on page 142.*)

199.4 Displaying messages

LaTeX is handling error messages and so the TeX ones are disabled.

```
\int_gset:Nn \tex_errorcontextlines:D { -1 }
```

`\msg_fatal_text:n`
`\msg_critical_text:n`
`\msg_error_text:n`
`\msg_warning_text:n`
`\msg_info_text:n`

A function for issuing messages: both the text and order could in principal vary.

```
\cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
\cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
\cs_new:Npn \msg_error_text:n #1 { #1~error }
\cs_new:Npn \msg_warning_text:n #1 { #1~warning }
\cs_new:Npn \msg_info_text:n #1 { #1~info }
```

(*End definition for* `\msg_fatal_text:n` *and others. These functions are documented on page 139.*)

`\msg_see_documentation_text:n`

Contextual footer information.

```
\cs_new:Npn \msg_see_documentation_text:n #1
  { \\ \\ See~the~#1~documentation~for~further~information. }
```

(*End definition for* `\msg_see_documentation_text:n`. *This function is documented on page* **??**.)

\msg_class_set:nn Setting up a message class does two tasks. Any existing redirection is cleared, and the various message functions are created to simply use the code stored for the message.

```
\cs_new_protected:Npn \msg_class_set:nn #1#2
  {
    \prop_clear_new:c { l_msg_redirect_ #1 _prop }
    \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
      { \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6} }
    \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
    \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
    \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
    \cs_set_protected:cpx { msg_ #1 :nn } ##1##2
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { } }
  }
```

(*End definition for* `\msg_class_set:nn`. *This function is documented on page 139.*)

\msg_if_more_text_p:N
\msg_if_more_text_p:c
\msg_if_more_text:NTF
\msg_if_more_text:cTF
\msg_no_more_text:xxxx

A test to see if any more text is available, using a permanently-empty text function.

```
\prg_new_conditional:Npnn \msg_if_more_text:N #1 { p , T , F , TF }
  {
    \cs_if_eq:NNTF #1 \msg_no_more_text:xxxx
      { \prg_return_false:  }
      { \prg_return_true: }
  }
\cs_new:Npn \msg_no_more_text:xxxx #1#2#3#4 { }
\cs_generate_variant:Nn \msg_if_more_text_p:N { c }
\cs_generate_variant:Nn \msg_if_more_text:NT  { c }
\cs_generate_variant:Nn \msg_if_more_text:NF  { c }
\cs_generate_variant:Nn \msg_if_more_text:NTF { c }
```

(*End definition for* `\msg_if_more_text:N` *and* `\msg_if_more_text:c`. *These functions are documented on page* **??**.)

\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nn

For fatal errors, after the error message TeX bails out.

```
\msg_class_set:nn { fatal }
  {
    \msg_interrupt:xxx
      { \msg_fatal_text:n {#1} : ~ "#2" }
      {
        \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c_msg_fatal_text_tl }
    \tex_end:D
  }
```

(*End definition for* `\msg_fatal:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_critical:nnxxxx
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx
\msg_critical:nn

Not quite so bad: just end the current file.

```
\msg_class_set:nn { critical }
  {
    \msg_interrupt:xxx
      { \msg_critical_text:n {#1} : ~ "#2" }
      {
        \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c_msg_critical_text_tl }
    \tex_endinput:D
  }
```

(*End definition for* `\msg_critical:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
\msg_error:nn

For an error, the interrupt routine is called, then any recovery code is tried.

```
\msg_class_set:nn { error }
  {
    \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl #1 / #2 }
      {
        \msg_interrupt:xxx
          { \msg_error_text:n {#1} : ~ "#2" }
          {
            \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
            \msg_see_documentation_text:n {#1}
          }
          { \use:c { \c_msg_more_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
     }
     {
        \msg_interrupt:xxx
          { \msg_error_text:n {#1} : ~ "#2" }
          {
            \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
            \msg_see_documentation_text:n {#1}
          }
          { }
     }
  }
```

(*End definition for* `\msg_error:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_warning:nnxxxx
\msg_warning:nnxxx
\msg_warning:nnxx
\msg_warning:nnx
\msg_warning:nn

Warnings are printed to the terminal.

```
\msg_class_set:nn { warning }
  {
    \msg_term:x
      {
        \msg_warning_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_warning:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_info:nnxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn

Information only goes into the log.

```
\msg_class_set:nn { info }
  {
    \msg_log:x
      {
        \msg_info_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_info:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_log:nnxxxx
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn

"Log" data is very similar to information, but with no extras added.

```
\msg_class_set:nn { log }
  {
    \msg_log:x
      { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
  }
```

(*End definition for* `\msg_log:nnxxxx` *and others. These functions are documented on page* **??**.)

\msg_none:nnxxxx
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn

The `none` message type is needed so that input can be gobbled.

```
\msg_class_set:nn { none } { }
```

(*End definition for* `\msg_none:nnxxxx` *and others. These functions are documented on page* **??**.)

\l_msg_class_tl
\l_msg_current_class_tl

Support variables needed for the redirection system.

```
\tl_new:N \l_msg_class_tl
\tl_new:N \l_msg_current_class_tl
```

(*End definition for* `\l_msg_class_tl` *and* `\l_msg_current_class_tl`*. These variables are documented on page* **??**.)

\l_msg_redirect_classes_prop
\l_msg_redirect_names_prop

For filtering messages, a list of all messages and of those which have to be modified is required.

```
\prop_new:N \l_msg_redirect_classes_prop
\prop_new:N \l_msg_redirect_names_prop
```

(*End definition for* `\l_msg_redirect_classes_prop` *and* `\l_msg_redirect_names_prop`*. These variables are documented on page* **??**.)

\l_msg_redirect_prop

For redirection of individually-named messages

```
\prop_new:N \l_msg_redirect_prop
```

(*End definition for* `\l_msg_redirect_prop`*. This variable is documented on page* **??**.)

\l_msg_use_direct_bool

Used to force redirection when a name is given directly.

```
\bool_new:N \l_msg_use_direct_bool
```

(*End definition for* `\l_msg_use_direct_bool`*. This variable is documented on page* **??**.)

\msg_use:nnnnxxxx
\msg_use_code:
\msg_use_or_change_class:
\msg_use_aux_i:nn
\msg_use_aux_ii:nn
\msg_use_aux_iii:w
\msg_use_aux_iv:w
\msg_use_aux_v:

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around.

```
\cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8
  {
    \msg_if_exist:nnTF {#3} {#4}
      {
        \cs_if_exist:cTF { msg_ #1 :nnxxxx }
          {
            \tl_set:Nn \l_msg_current_class_tl {#1}
            \cs_set_protected_nopar:Npx \msg_use_code: { \exp_not:n {#2} }
            \cs_set_protected_nopar:Npn \msg_use_or_change_class:
              {
                \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl
                  { \msg_use_code: }
                  {
                    \use:c { msg_ \l_msg_class_tl :nnxxxx }
                      {#3} {#4} {#5} {#6} {#7} {#8}
                  }
              }
            \bool_if:NTF \l_msg_use_direct_bool
              {
                \bool_set_false:N \l_msg_use_direct_bool
                \msg_use_code:
              }
              { \msg_use_aux_i:nn {#3} {#4} }
          }
          { \msg_kernel_error:nnx { msg } { message-class-unknown } {#1} }
      }
      { \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4} }
  }
\cs_new_protected_nopar:Npn \msg_use_code: { }
\cs_new_protected_nopar:Npn \msg_use_or_change_class: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted.

```
\cs_new_protected:Npn \msg_use_aux_i:nn #1#2
  {
    \prop_get:NnNTF \l_msg_redirect_prop { #1 / #2 } \l_msg_class_tl
      {
        \bool_set_true:N \l_msg_use_direct_bool
        \msg_use_or_change_class:
      }
      { \msg_use_aux_ii:nn {#1} {#2} }
  }
```

Next check if there is a redirection by module or by submodule.

```
\cs_new_protected:Npn \msg_use_aux_ii:nn #1#2
  {
    \prop_get:coNTF { l_msg_redirect_ \l_msg_current_class_tl _ prop }
```

```
      { \msg_use_aux_iii:w #1 / #2 / \q_stop } \l_msg_class_tl
      { \msg_use_or_change_class: }
      {
        \prop_get:coNTF { l_msg_redirect_ \l_msg_current_class_tl _ prop }
          { \msg_use_aux_iv:w #1 / #2 \q_stop } \l_msg_class_tl
        { \msg_use_or_change_class: }
        { \msg_use_aux_v: }
      }
  }
\cs_new:Npn \msg_use_aux_iii:w  #1 / #2 / #3 \q_stop { #1 / #2 }
\cs_new:Npn \msg_use_aux_iv:w #1 / #2 \q_stop { #1 }
```

Finally test for redirection of an entire class.

```
\cs_new_protected:Npn \msg_use_aux_v:
  {
    \prop_get:cnNF { l_msg_redirect_ \l_msg_current_class_tl _ prop }
      { * } \l_msg_class_tl
      { \tl_set_eq:NN \l_msg_class_tl \l_msg_current_class_tl }
    \msg_use_or_change_class:
  }
```

(*End definition for* `\msg_use:nnnnxxxx`*. This function is documented on page* **??**.)

\msg_redirect_class:nn
\msg_redirect_class_aux:nnn
\msg_redirect_class_aux:nVn

Converts class one into class two.

```
\cs_new_protected:Npn \msg_redirect_class:nn #1#2
  {
    \cs_if_exist:cTF { msg_ #1 :nnxxxx }
      {
        \cs_if_exist:cTF { msg_ #2 :nnxxxx }
          {
            \tl_set:Nn \l_msg_current_class_tl {#1}
            \msg_redirect_class_aux:nnn {#1} {#2} {#2}
          }
          { \msg_kernel_error:nnx { msg } { message-class-unknown } {#2} }
      }
      { \msg_kernel_error:nnx { msg } { message-class-unknown } {#1} }
  }
\cs_new_protected:Npn \msg_redirect_class_aux:nnn #1#2#3
  {
    \prop_get:cnNTF { l_msg_redirect_ #2 _prop } { * } \l_msg_class_tl
      {
        \tl_if_eq:NNTF \l_msg_class_tl \l_msg_current_class_tl
          { \msg_kernel_error:nnxx { msg } { message-loop } {#1} {#3} }
          { \msg_redirect_class_aux:nVn {#1} \l_msg_class_tl {#3} }
      }
      { \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#3} }
  }
\cs_generate_variant:Nn \msg_redirect_class_aux:nnn { nV }
```

(*End definition for* `\msg_redirect_class:nn`*. This function is documented on page 141.*)

\msg_redirect_module:nnn
\msg_redirect_module_aux:nnnn
\msg_redirect_module_aux:nnVn

For when all messages of a class should be altered for a given module.

```
\cs_new_protected:Npn \msg_redirect_module:nnn #1#2#3
  {
    \cs_if_exist:cTF { msg_ #2 :nnxxxx }
      {
        \cs_if_exist:cTF { msg_ #3 :nnxxxx }
          {
            \tl_set:Nn \l_msg_current_class_tl {#1}
            \msg_redirect_module_aux:nnnn {#1} {#2} {#3} {#3}
          }
          { \msg_kernel_error:nnx { msg } { message-class-unknown } {#3} }
      }
      { \msg_kernel_error:nnx { msg } { message-class-unknown } {#2} }
  }
\cs_new_protected:Npn \msg_redirect_module_aux:nnnn #1#2#3#4
  {
    \prop_get:cnNTF { l_msg_redirect_ #3 _prop } {#1} \l_msg_class_tl
      {
        \tl_if_eq:NNTF \l_msg_class_tl \l_msg_current_class_tl
          { \msg_kernel_error:nnxx { msg } { message-loop } {#2} {#4} }
          { \msg_redirect_moduel_aux:nVn {#1} {#2} \l_msg_class_tl {#4} }
      }
      { \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#4} }
  }
\cs_generate_variant:Nn \msg_redirect_module_aux:nnnn { nnV }
```

(*End definition for* `\msg_redirect_module:nnn`*. This function is documented on page 141.*)

\msg_redirect_name:nnn

Named message will always use the given class even if that class is redirected further.

```
\cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
  {
    \cs_if_exist:cTF { msg_ #3 :nnxxxx }
      { \prop_put:Nnn \l_msg_redirect_prop { #1 / #2 } {#3} }
      { \msg_kernel_error:nnx { msg } { message-class-unknown } {#3} }
  }
```

(*End definition for* `\msg_redirect_name:nnn`*. This function is documented on page 141.*)

199.5 Kernel-specific functions

\msg_kernel_new:nnnn
\msg_kernel_new:nnn
\msg_kernel_set:nnnn
\msg_kernel_set:nnn

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```
\cs_new_protected:Npn \msg_kernel_new:nnnn #1#2
  { \msg_new:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \msg_kernel_new:nnn #1#2
  { \msg_new:nnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \msg_kernel_set:nnnn #1#2
  { \msg_set:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \msg_kernel_set:nnn #1#2
  { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(*End definition for* `\msg_kernel_new:nnnn`*. This function is documented on page* **??**.)

`\msg_kernel_fatal:nnxxxx`
`\msg_kernel_fatal:nnxxx`
`\msg_kernel_fatal:nnxx`
`\msg_kernel_fatal:nnx`
`\msg_kernel_fatal:nn`

Fatal kernel errors cannot be re-defined.

```
\cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6
  {
    \msg_interrupt:xxx
      { \msg_fatal_text:n { LaTeX } : ~ "#1 / #2" }
      {
        \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
          {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n { LaTeX3 }
      }
      { \c_msg_fatal_text_tl }
    \tex_end:D
  }
\cs_new_protected:Npn \msg_kernel_fatal:nnxxx #1#2#3#4#5
  {\msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
\cs_new_protected:Npn \msg_kernel_fatal:nnxx #1#2#3#4
  { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} {#4} { } { } }
\cs_new_protected:Npn \msg_kernel_fatal:nnx #1#2#3
  { \msg_kernel_fatal:nnxxxx {#1} {#2} {#3} { } { } { } }
\cs_new_protected:Npn \msg_kernel_fatal:nn #1#2
  { \msg_kernel_fatal:nnxxxx {#1} {#2} { } { } { } { } }
```

(*End definition for* `\msg_kernel_fatal:nnxxxx`*. This function is documented on page* **??**.)

`\msg_kernel_error:nnxxxx`
`\msg_kernel_error:nnxxx`
`\msg_kernel_error:nnxx`
`\msg_kernel_error:nnx`
`\msg_kernel_error:nn`

Neither can kernel errors.

```
\cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6
  {
    \msg_if_more_text:cTF { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
      {
        \msg_interrupt:xxx
          { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
          {
            \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
              {#3} {#4} {#5} {#6}
            \msg_see_documentation_text:n { LaTeX3 }
          }
          {
            \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 }
              {#3} {#4} {#5} {#6}
          }
      }
      {
        \msg_interrupt:xxx
          { \msg_error_text:n { LaTeX } : ~ " #1 / #2 " }
          {
            \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
              {#3} {#4} {#5} {#6}
            \msg_see_documentation_text:n { LaTeX3 }
```

```
        }
        { }
    }
  }
\cs_new_protected:Npn \msg_kernel_error:nnxxx #1#2#3#4#5
  {\msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
\cs_gset_protected:Npn \msg_kernel_error:nnxx #1#2#3#4
  { \msg_kernel_error:nnxxxx {#1} {#2} {#3} {#4} { } { } }
\cs_gset_protected:Npn \msg_kernel_error:nnx #1#2#3
  { \msg_kernel_error:nnxxxx {#1} {#2} {#3} { } { } { } }
\cs_gset_protected:Npn \msg_kernel_error:nn #1#2
  { \msg_kernel_error:nnxxxx {#1} {#2} { } { } { } { } }
```

(*End definition for* `\msg_kernel_error:nnxxxx`*. This function is documented on page* **??**.)

`\msg_kernel_warning:nnxxxx`
`\msg_kernel_warning:nnxxx`
`\msg_kernel_warning:nnxx`
`\msg_kernel_warning:nnx`
`\msg_kernel_warning:nn`
`\msg_kernel_info:nnxxxx`
`\msg_kernel_info:nnxxx`
`\msg_kernel_info:nnxx`
`\msg_kernel_info:nnx`
`\msg_kernel_info:nn`

Kernel messages which can be redirected.

```
\prop_new:N \l_msg_redirect_kernel_warning_prop
\cs_new_protected:Npn \msg_kernel_warning:nnxxxx #1#2#3#4#5#6
  {
    \msg_use:nnnnxxxx { warning }
      {
        \msg_term:x
          {
            \msg_warning_text:n { LaTeX } : ~ " #1 / #2 " \\ \\
            \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
              {#3} {#4} {#5} {#6}
          }
      }
      { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
  }
\cs_new_protected:Npn \msg_kernel_warning:nnxxx #1#2#3#4#5
  { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
\cs_new_protected:Npn \msg_kernel_warning:nnxx #1#2#3#4
  { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} {#4} { } { } }
\cs_new_protected:Npn \msg_kernel_warning:nnx #1#2#3
  { \msg_kernel_warning:nnxxxx {#1} {#2} {#3} { } { } { } }
\cs_new_protected:Npn \msg_kernel_warning:nn #1#2
  { \msg_kernel_warning:nnxxxx {#1} {#2} { } { } { } { } }
\prop_new:N \l_msg_redirect_kernel_info_prop
\cs_new_protected:Npn \msg_kernel_info:nnxxxx #1#2#3#4#5#6
  {
    \msg_use:nnnnxxxx { info }
      {
        \msg_log:x
          {
            \msg_info_text:n { LaTeX } : ~ " #1 / #2 " \\ \\
            \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
              {#3} {#4} {#5} {#6}
          }
      }
      { LaTeX } { #1 / #2 } {#3} {#4} {#5} {#6}
```

```
  }
\cs_new_protected:Npn \msg_kernel_info:nnxxx #1#2#3#4#5
  { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} {#5} { } }
\cs_new_protected:Npn \msg_kernel_info:nnxx #1#2#3#4
  { \msg_kernel_info:nnxxxx {#1} {#2} {#3} {#4} { } { } }
\cs_new_protected:Npn \msg_kernel_info:nnx #1#2#3
  { \msg_kernel_info:nnxxxx {#1} {#2} {#3} { } { } { } }
\cs_new_protected:Npn \msg_kernel_info:nn #1#2
  { \msg_kernel_info:nnxxxx {#1} {#2} { } { } { } { } }
```

(*End definition for* `\msg_kernel_warning:nnxxxx`*. This function is documented on page* **??**.)

Error messages needed to actually implement the message system itself.

```
\msg_kernel_new:nnnn { msg } { message-already-defined }
  { Message~'#2'~for~module~'#1'~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~define~a~new~message~called~'#2'\
    by~the~module~'#1':~this~message~already~exists.
    \c_msg_return_text_tl
  }
\msg_kernel_new:nnnn { msg } { message-unknown }
  { Unknown~message~'#2'~for~module~'#1'. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~display~a~message~called~'#2'\\
    by~the~module~'#1'~module:~this~message~does~not~exist.
    \c_msg_return_text_tl
  }
\msg_kernel_new:nnnn { msg } { message-class-unknown }
  { Unknown~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
    this~was~never~defined.
    \c_msg_return_text_tl
  }
\msg_kernel_new:nnnn { msg } { redirect-loop }
  { Message~redirection~loop~for~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\\
    The~original~message~here~has~been~lost.
    \c_msg_return_text_tl
  }
```

Messages for earlier kernel modules.

```
\msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
  { Function~'#1'~cannot~be~defined~with~#2~arguments. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~a~function~'#1'~with~
    #2~arguments. \\
    TeX~allows~between~0~and~9~arguments~for~a~single~function.
```

```
  }
\msg_kernel_new:nnnn { kernel } { command-already-defined }
  { Control~sequence~#1~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
    but~this~name~has~already~been~used~elsewhere. \\ \\
    The~current~meaning~is:\\
    \ \ #2
  }
\msg_kernel_new:nnnn { kernel } { command-not-defined }
  { Control~sequence~#1~undefined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
    been~defined~yet.
  }
\msg_kernel_new:nnnn { kernel } { out-of-registers }
  { No~room~for~a~new~#1. }
  {
    TeX~only~supports~\int_use:N \c_max_register_int \
    of~each~type.~All~the~#1~registers~have~been~used.~
    This~run~will~be~aborted~now.
  }
\msg_kernel_new:nnnn { kernel } { variable-not-defined }
  { Variable~#1~undefined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
    been~defined~yet.
  }
\msg_kernel_new:nnnn { seq } { empty-sequence }
  { Empty~sequence~#1. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~recover~an~entry~from~a~sequence~that~
    has~no~content:~that~cannot~happen!
  }
\msg_kernel_new:nnnn { tl } { empty-search-pattern }
  { Empty~search~pattern. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~%
    would~lead~to~an~infinite~loop!
  }
\msg_kernel_new:nnnn { scan } { already-defined }
  { Scan~mark~#1~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
```

```
    but~this~name~has~already~been~used~for~a~scan~mark.
  }
```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
\msg_kernel_new:nnn { seq } { misused }
  { A~sequence~was~misused. }
\msg_kernel_new:nnn { kernel } { bad-var }
  { Erroneous~variable~#1 used! }
\msg_kernel_new:nnn { prg } { zero-step }
  { Zero~step~size~for~stepwise~function~#1. }
\msg_kernel_new:nnn { prg } { replicate-neg }
  { Negative~argument~for~\prg_replicate:nn. }
\msg_kernel_new:nnn { kernel } { unknown-comparison }
  { Relation~symbol~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
```

Messages used by the "show" functions.

```
\msg_kernel_new:nnn { seq } { show }
  {
    The~sequence~\token_to_str:N #1~
    \seq_if_empty:NTF #1
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
\msg_kernel_new:nnn { prop } { show }
  {
    The~property~list~\token_to_str:N #1~
    \prop_if_empty:NTF #1
      { is~empty }
      { contains~the~pairs~(without~outer~braces): }
  }
\msg_kernel_new:nnn { clist } { show }
  {
    The~comma~list~
    \str_if_eq:nnF {#1} { \l_clist_internal_clist } { \token_to_str:N #1~}
    \clist_if_empty:NTF #1
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
\msg_kernel_new:nnn { ior } { show-no-stream }
  { No~input~streams~are~open }
\msg_kernel_new:nnn { ior } { show-open-streams }
  { The~following~input~streams~are~in~use: }
\msg_kernel_new:nnn { iow } { show-no-stream }
  { No~output~streams~are~open }
\msg_kernel_new:nnn { iow } { show-open-streams }
  { The~following~output~streams~are~in~use: }
```

199.6 Expandable errors

\msg_expandable_error:n

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```
<argument> \LaTeX3 error:
                                           The error message.
```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error:` ⟨*error message*⟩. Then `\msg_expandable_error_aux:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\c_zero` prevents losing braces around the user-inserted text if any, and stops the expansion of `\romannumeral`.

```
\group_begin:
\char_set_catcode_math_superscript:N \^
\char_set_lccode:nn {`^} {`\ }
\char_set_lccode:nn {`L} {`L}
\char_set_lccode:nn {`T} {`T}
\char_set_lccode:nn {`X} {`X}
\tl_to_lowercase:n
  {
    \cs_new:Npx \msg_expandable_error:n #1
      {
        \exp_not:n
          {
            \tex_romannumeral:D
            \exp_after:wN \exp_after:wN
            \exp_after:wN \msg_expandable_error_aux:w
            \exp_after:wN \exp_after:wN
            \exp_after:wN \c_zero
          }
        \exp_not:N \use:n { \exp_not:c { LaTeX3~error: } ^ #1 } ^
      }
    \cs_new:Npn \msg_expandable_error_aux:w #1 ^ #2 ^ { #1 }
  }
\group_end:
```

(*End definition for* `\msg_expandable_error:n`*. This function is documented on page 144.*)

\msg_expandable_kernel_error:nnnnnn
\msg_expandable_kernel_error:nnnnn
\msg_expandable_kernel_error:nnnn
\msg_expandable_kernel_error:nnn
\msg_expandable_kernel_error:nn

The command built from the csname `\c_msg_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `\msg_expandable_error:n`.

```
\cs_new:Npn \msg_expandable_kernel_error:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nf \msg_expandable_error:n
      {
        \exp_args:NNc \exp_after:wN \exp_stop_f:
          { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
          {#3} {#4} {#5} {#6}
```

```
      }
  }
\cs_new:Npn \msg_expandable_kernel_error:nnnnn #1#2#3#4#5
  {
    \msg_expandable_kernel_error:nnnnnn
      {#1} {#2} {#3} {#4} {#5} { }
  }
\cs_new:Npn \msg_expandable_kernel_error:nnnn #1#2#3#4
  {
    \msg_expandable_kernel_error:nnnnnn
      {#1} {#2} {#3} {#4} { } { }
  }
\cs_new:Npn \msg_expandable_kernel_error:nnn #1#2#3
  {
    \msg_expandable_kernel_error:nnnnnn
      {#1} {#2} {#3} { } { } { }
  }
\cs_new:Npn \msg_expandable_kernel_error:nn #1#2
  {
    \msg_expandable_kernel_error:nnnnnn
      {#1} {#2} { } { } { } { }
  }
```

(*End definition for* `\msg_expandable_kernel_error:nnnnnn` *and others. These functions are documented on page* **??**.)

199.7 Showing variables

Functions defined in this section are used for diagnostic functions in l3clist, l3io, l3prop, l3seq, xtemplate

`\msg_aux_use:nn`
`\msg_aux_use:nnxxxx`

Print the text of a message to the terminal, without formatting.

```
\cs_new_protected:Npn \msg_aux_use:nn #1#2
  { \msg_aux_use:nnxxxx {#1} {#2} { } { } { } { } }
\cs_new_protected:Npn \msg_aux_use:nnxxxx #1#2#3#4#5#6
  {
    \iow_wrap:xnnnN
      {
        \use:c { \c_msg_text_prefix_tl #1 / #2 }
          {#3} {#4} {#5} {#6}
      }
      { } \c_zero { } \iow_term:x
  }
```

(*End definition for* `\msg_aux_use:nn`*. This function is documented on page* **??**.)

`\msg_aux_show:Nnx`
`\msg_aux_show:x`
`\msg_aux_show:w`

The arguments of `\msg_aux_show:Nnx` are

- The ⟨*variable*⟩ to be shown.
- The `TF` emptyness conditional for that type of variables.

- The type of the variable.
- A mapping of the form \seq_map_function:NN ⟨*variable*⟩ \msg_aux_show:n, which produces the formatted string.

We remove a new line and >␣ from the first item using a w-type auxiliary, and the fact that f-expansion removes a space. To avoid a low-level TeX error if there is an empty argument, a simple test is used to keep the output "clean". The odd \exp_after:wN and trailing \prg_do_nothing: improve the output slightly.

```
\cs_new_protected:Npn \msg_aux_show:Nnx #1#2#3
  {
    \cs_if_exist:NTF #1
      {
        \msg_aux_use:nnxxxx { LaTeX / #2 } { show } {#1} { } { } { }
        \msg_aux_show:x {#3}
      }
      {
        \msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \msg_aux_show:x #1
  {
    \tl_set:Nx \l_msg_internal_tl {#1}
    \tl_if_empty:NT \l_msg_internal_tl
      { \tl_set:Nx \l_msg_internal_tl { > } }
    \exp_args:Nf \etex_showtokens:D
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \msg_aux_show:w
        \exp_after:wN \l_msg_internal_tl
        \exp_after:wN
      }
    \prg_do_nothing:
  }
\cs_new:Npn \msg_aux_show:w #1 > { }
```

(*End definition for* \msg_aux_show:Nnx*. This function is documented on page 144.*)

\msg_aux_show:n
\msg_aux_show:nn
\msg_aux_show_unbraced:nn

Each item in the variable is formatted using one of the following functions.

```
\cs_new:Npn \msg_aux_show:n #1
  {
    \iow_newline: > \c_space_tl \c_space_tl { \exp_not:n {#1} }
  }
\cs_new:Npn \msg_aux_show:nn #1#2
  {
    \iow_newline: > \c_space_tl \c_space_tl { \exp_not:n {#1} }
    \c_space_tl \c_space_tl => \c_space_tl \c_space_tl { \exp_not:n {#2} }
  }
\cs_new:Npn \msg_aux_show_unbraced:nn #1#2
```

```
  {
    \iow_newline: > \c_space_tl \c_space_tl \exp_not:n {#1}
    \c_space_tl \c_space_tl => \c_space_tl \c_space_tl \exp_not:n {#2}
  }
```

(*End definition for* `\msg_aux_show:n`. *This function is documented on page* 144.)

199.8 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

`\msg_class_new:nn`

This is only ever used in a `set` fashion.

```
⟨*deprecated⟩
\cs_new_eq:NN \msg_class_new:nn \msg_class_set:nn
⟨/deprecated⟩
```

(*End definition for* `\msg_class_new:nn`. *This function is documented on page* **??**.)

`\msg_trace:nnxxxx`
`\msg_trace:nnxxx`
`\msg_trace:nnxx`
`\msg_trace:nnx`
`\msg_trace:nn`

The performance here is never going to be good enough for tracing code, so let's be realistic.

```
⟨*deprecated⟩
\cs_new_eq:NN \msg_trace:nnxxxx \msg_log:nnxxxx
\cs_new_eq:NN \msg_trace:nnxxx  \msg_log:nnxxx
\cs_new_eq:NN \msg_trace:nnxx   \msg_log:nnxx
\cs_new_eq:NN \msg_trace:nnx    \msg_log:nnx
\cs_new_eq:NN \msg_trace:nn     \msg_log:nn
⟨/deprecated⟩
```

(*End definition for* `\msg_trace:nnxxxx` *and others. These functions are documented on page* **??**.)

`\msg_generic_new:nnn`
`\msg_generic_new:nn`
`\msg_generic_set:nnn`
`\msg_generic_set:nn`
`\msg_direct_interrupt:xxxxx`
`\msg_direct_log:xx`
`\msg_direct_term:xx`

These were all too low-level.

```
⟨*deprecated⟩
\cs_new_protected:Npn \msg_generic_new:nnn #1#2#3 { \deprecated }
\cs_new_protected:Npn \msg_generic_new:nn  #1#2   { \deprecated }
\cs_new_protected:Npn \msg_generic_set:nnn #1#2#3 { \deprecated }
\cs_new_protected:Npn \msg_generic_set:nn  #1#2   { \deprecated }
\cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 { \deprecated }
\cs_new_protected:Npn \msg_direct_log:xx #1#2  { \deprecated }
\cs_new_protected:Npn \msg_direct_term:xx #1#2 { \deprecated }
⟨/deprecated⟩
```

(*End definition for* `\msg_generic_new:nnn`. *This function is documented on page* **??**.)

`\msg_kernel_bug:x`
`\c_msg_kernel_bug_text_tl`
`\c_msg_kernel_bug_more_text_tl`

```
⟨*deprecated⟩
\cs_set_protected:Npn \msg_kernel_bug:x #1
  {
    \msg_interrupt:xxx { \c_msg_kernel_bug_text_tl }
      {
        #1
       \msg_see_documentation_text:n { LaTeX3 }
      }
```

```
      { \c_msg_kernel_bug_more_text_tl }
  }
\tl_const:Nn \c_msg_kernel_bug_text_tl
  { This~is~a~LaTeX~bug:~check~coding! }
\tl_const:Nn \c_msg_kernel_bug_more_text_tl
  {
    There~is~a~coding~bug~somewhere~around~here. \\
    This~probably~needs~examining~by~an~expert.
    \c_msg_return_text_tl
  }
⟨/deprecated⟩
```

(*End definition for* `\msg_kernel_bug:x`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

200 l3keys Implementation

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

200.1 Low-level interface

For historical reasons this code uses the 'keyval' module prefix.

`\g_keyval_level_int`

For nesting purposes an integer is needed for the current level.

```
\int_new:N \g_keyval_level_int
```

(*End definition for* `\g_keyval_level_int`*. This variable is documented on page* **??**.)

`\l_keyval_key_tl`
`\l_keyval_value_tl`

The current key name and value.

```
\tl_new:N \l_keyval_key_tl
\tl_new:N \l_keyval_value_tl
```

(*End definition for* `\l_keyval_key_tl` *and* `\l_keyval_value_tl`*. These variables are documented on page* **??**.)

`\l_keyval_sanitise_tl`
`\l_keyval_parse_tl`

Token list variables for dealing with awkward category codes in the input.

```
\tl_new:N \l_keyval_sanitise_tl
\tl_new:N \l_keyval_parse_tl
```

(*End definition for* `\l_keyval_sanitise_tl`*. This function is documented on page* **??**.)

`\keyval_parse:n`

The parsing function first deals with the category codes for `=` and `,`, so that there are no odd events. The input is then handed off to the element by element system.

```
\group_begin:
  \char_set_catcode_active:n { `\= }
  \char_set_catcode_active:n { `\, }
  \char_set_lccode:nn { `\8 } { `\= }
```

```
  \char_set_lccode:nn { `\9 } { `\, }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \keyval_parse:n #1
      {
        \group_begin:
          \tl_clear:N \l_keyval_sanitise_tl
          \tl_set:Nn \l_keyval_sanitise_tl {#1}
          \tl_replace_all:Nnn \l_keyval_sanitise_tl { = } { 8 }
          \tl_replace_all:Nnn \l_keyval_sanitise_tl { , } { 9 }
          \tl_clear:N \l_keyval_parse_tl
          \exp_after:wN \keyval_parse_elt:w \exp_after:wN
            \q_no_value \l_keyval_sanitise_tl 9 \q_nil 9
        \exp_after:wN \group_end:
        \l_keyval_parse_tl
      }
  }
```

(*End definition for* `\keyval_parse:n`. *This function is documented on page* **??**.)

`\keyval_parse_elt:w`

Each item to be parsed will have `\q_no_value` added to the front. Hence the blank test here can always be used to find a totally empty argument. If this is the case, the system loops round. If there is something to parse, there is a check for the `\q_nil` marker and if not a hand-off.

```
\cs_new_protected:Npn \keyval_parse_elt:w #1 ,
  {
    \tl_if_blank:oTF { \use_none:n #1 }
      { \keyval_parse_elt:w \q_no_value }
      {
        \quark_if_nil:oF { \use_ii:nn #1 }
          {
            \keyval_split_key_value:w #1 = = \q_stop
            \keyval_parse_elt:w \q_no_value
          }
      }
  }
```

(*End definition for* `\keyval_parse_elt:w`. *This function is documented on page* **??**.)

`\keyval_split_key_value:w`
`\keyval_split_key_value_aux:wTF`

The key and value are handled separately. First the key is grabbed and saved as `\l_keyval_key_tl`. Then a check is need to see if there is a value at all: if not then the key name is simply added to the output. If there is a value then there is a check to ensure that there was only one `=` in the input (remembering some extra ones are around at the moment to prevent errors). All being well, there is an hand-off to find the value: the `\q_nil` is there to prevent loss of braces.

```
\cs_new_protected:Npn \keyval_split_key_value:w #1 = #2 \q_stop
  {
    \keyval_split_key:w #1 \q_stop
    \str_if_eq:nnTF {#2} { = }
```

```
      {
        \tl_put_right:Nx \l_keyval_parse_tl
          {
            \exp_not:c
              { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n }
              { \exp_not:o \l_keyval_key_tl }
          }
      }
      {
        \keyval_split_key_value_aux:wTF #2 \q_no_value \q_stop
          { \keyval_split_value:w \q_nil #2 }
          { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
      }
  }
\cs_new:Npn \keyval_split_key_value_aux:wTF #1 = #2#3 \q_stop
  { \tl_if_head_eq_meaning:nNTF {#3} \q_no_value }
```

(*End definition for* `\keyval_split_key_value:w`*. This function is documented on page* **??**.)

`\keyval_split_key:w` The aim here is to remove spaces and also exactly one set of braces. There is also a quark to remove, hence the `\use_none:n` appearing before application of `\tl_trim_spaces:n`.

```
\cs_new_protected:Npn \keyval_split_key:w #1 \q_stop
  {
    \tl_set:Nx \l_keyval_key_tl
      { \exp_after:wN \tl_trim_spaces:n \exp_after:wN { \use_none:n #1 } }
  }
```

(*End definition for* `\keyval_split_key:w`*. This function is documented on page* **??**.)

`\keyval_split_value:w` Here the value has to be separated from the equals signs and the leading `\q_nil` added in to keep the brace levels. Fist the processing function can be added to the output list. If there is no value, setting `\l_keyval_value_tl` with three groups removed will leave nothing at all, and so an empty group can be added to the parsed list. On the other hand, if the value is entirely contained within a set of braces then `\l_keyval_value_tl` will contain `\q_nil` only. In that case, strip off the leading quark using `\use_ii:nnn`, which also deals with any spaces.

```
\cs_new_protected:Npn \keyval_split_value:w #1 = =
  {
    \tl_put_right:Nx \l_keyval_parse_tl
      {
        \exp_not:c
          { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn }
          { \exp_not:o \l_keyval_key_tl }
      }
    \tl_set:Nx \l_keyval_value_tl
      { \exp_not:o { \use_none:nnn #1 \q_nil \q_nil } }
    \tl_if_empty:NTF \l_keyval_value_tl
      { \tl_put_right:Nn \l_keyval_parse_tl { { } } }
      {
        \quark_if_nil:NTF \l_keyval_value_tl
          {
```

```
          \tl_put_right:Nx \l_keyval_parse_tl
            { { \exp_not:o { \use_ii:nnn #1 \q_nil } } }
        }
        { \keyval_split_value_aux:w #1 \q_stop }
    }
  }
```

A similar idea to the key code: remove the spaces from each end and deal with one set of braces.

```
\cs_new_protected:Npn \keyval_split_value_aux:w \q_nil #1 \q_stop
  {
    \tl_set:Nx \l_keyval_value_tl { \tl_trim_spaces:n {#1} }
    \tl_put_right:Nx \l_keyval_parse_tl
      { { \exp_not:o \l_keyval_value_tl } }
  }
```

(*End definition for* `\keyval_split_value:w`. *This function is documented on page* **??**.)

`\keyval_parse:NNn` The outer parsing routine just sets up the processing functions and hands off.

```
\cs_new_protected:Npn \keyval_parse:NNn #1#2#3
  {
    \int_gincr:N \g_keyval_level_int
    \cs_gset_eq:cN
      { keyval_key_no_value_elt_ \int_use:N \g_keyval_level_int :n } #1
    \cs_gset_eq:cN
      { keyval_key_value_elt_ \int_use:N \g_keyval_level_int :nn }   #2
    \keyval_parse:n {#3}
    \int_gdecr:N \g_keyval_level_int
  }
```

(*End definition for* `\keyval_parse:NNn`. *This function is documented on page 155.*)

One message for the low level parsing system.

```
\msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
  { Misplaced~equals~sign~in~key-value~input~\msg_line_number: }
  {
    LaTeX~is~attempting~to~parse~some~key-value~input~but~found~
    two~equals~signs~not~separated~by~a~comma.
  }
```

200.2 Constants and variables

`\c_keys_code_root_tl` `\c_keys_vars_root_tl` The prefixes for the code and variables of the keys themselves.

```
\tl_const:Nn \c_keys_code_root_tl { key~code~>~ }
\tl_const:Nn \c_keys_vars_root_tl { key~var~>~ }
```

(*End definition for* `\c_keys_code_root_tl` *and* `\c_keys_vars_root_tl`. *These variables are documented on page* **??**.)

`\c_keys_props_root_tl` The prefix for storing properties.

```
\tl_const:Nn \c_keys_props_root_tl { key~prop~>~ }
```

(*End definition for* `\c_keys_props_root_tl`. *This variable is documented on page* **??**.)

\c_keys_value_forbidden_tl
\c_keys_value_required_tl

Two marker token lists.

```
\tl_const:Nn \c_keys_value_forbidden_tl { forbidden }
\tl_const:Nn \c_keys_value_required_tl  { required }
```

(*End definition for* `\c_keys_value_forbidden_tl` *and* `\c_keys_value_required_tl`*. These variables are documented on page* **??**.)

\l_keys_choice_int
\l_keys_choices_tl

Publicly accessible data on which choice is being used when several are generated as a set.

```
\int_new:N \l_keys_choice_int
\tl_new:N \l_keys_choices_tl
```

(*End definition for* `\l_keys_choice_int` *and* `\l_keys_choices_tl`*. These variables are documented on page* **??**.)

\l_keys_key_tl

The name of a key itself: needed when setting keys.

```
\tl_new:N \l_keys_key_tl
```

(*End definition for* `\l_keys_key_tl`*. This variable is documented on page 153.*)

\l_keys_module_tl

The module for an entire set of keys.

```
\tl_new:N \l_keys_module_tl
```

(*End definition for* `\l_keys_module_tl`*. This variable is documented on page* **??**.)

\l_keys_no_value_bool

A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```
\bool_new:N \l_keys_no_value_bool
```

(*End definition for* `\l_keys_no_value_bool`*. This variable is documented on page* **??**.)

\l_keys_path_tl

The "path" of the current key is stored here: this is available to the programmer and so is public.

```
\tl_new:N \l_keys_path_tl
```

(*End definition for* `\l_keys_path_tl`*. This variable is documented on page 153.*)

\l_keys_property_tl

The "property" begin set for a key at definition time is stored here.

```
\tl_new:N \l_keys_property_tl
```

(*End definition for* `\l_keys_property_tl`*. This variable is documented on page* **??**.)

\l_keys_unknown_clist

Used when setting only known keys to store those left over.

```
\tl_new:N \l_keys_unknown_clist
```

(*End definition for* `\l_keys_unknown_clist`*. This variable is documented on page* **??**.)

\l_keys_value_tl

The value given for a key: may be empty if no value was given.

```
\tl_new:N \l_keys_value_tl
```

(*End definition for* `\l_keys_value_tl`*. This variable is documented on page 153.*)

200.3 The key defining mechanism

\keys_define:nn
\keys_define_aux:nnn
\keys_define_aux:onn

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```
\cs_new_protected:Npn \keys_define:nn
  { \keys_define_aux:onn \l_keys_module_tl }
\cs_new_protected:Npn \keys_define_aux:nnn #1#2#3
  {
    \tl_set:Nx \l_keys_module_tl { \tl_to_str:n {#2} }
    \keyval_parse:NNn \keys_define_elt:n \keys_define_elt:nn {#3}
    \tl_set:Nn \l_keys_module_tl {#1}
  }
\cs_generate_variant:Nn \keys_define_aux:nnn { o }
```

(*End definition for* `\keys_define:nn`. *This function is documented on page* *146*.)

\keys_define_elt:n
\keys_define_elt:nn
\keys_define_elt_aux:nn

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
\cs_new_protected:Npn \keys_define_elt:n #1
  {
    \bool_set_true:N \l_keys_no_value_bool
    \keys_define_elt_aux:nn {#1} { }
  }
\cs_new_protected:Npn \keys_define_elt:nn #1#2
  {
    \bool_set_false:N \l_keys_no_value_bool
    \keys_define_elt_aux:nn {#1} {#2}
  }
\cs_new_protected:Npn \keys_define_elt_aux:nn #1#2
  {
    \keys_property_find:n {#1}
    \cs_if_exist:cTF { \c_keys_props_root_tl \l_keys_property_tl }
      { \keys_define_key:n {#2} }
      {
        \msg_kernel_error:nnxx { keys } { property-unknown }
          { \l_keys_property_tl } { \l_keys_path_tl }
      }
  }
```

(*End definition for* `\keys_define_elt:n`. *This function is documented on page* **??**.)

\keys_property_find:n
\keys_property_find_aux:w

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```
\cs_new_protected:Npn \keys_property_find:n #1
  {
    \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }
    \tl_if_in:nnTF {#1} { . }
      { \keys_property_find_aux:w #1 \q_stop }
```

```
      { \msg_kernel_error:nnx { keys } { key-no-property } {#1} }
  }
\cs_new_protected:Npn \keys_property_find_aux:w #1 . #2 \q_stop
  {
    \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
    \tl_if_in:nnTF {#2} { . }
      {
        \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
        \keys_property_find_aux:w #2 \q_stop
      }
      { \tl_set:Nn \l_keys_property_tl { . #2 } }
  }
```

(*End definition for* `\keys_property_find:n`*. This function is documented on page* **??**.)

`\keys_define_key:n`
`\keys_define_key_aux:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a `:` as if it is missing the earlier tests will have failed.

```
\cs_new_protected:Npn \keys_define_key:n #1
  {
    \bool_if:NTF \l_keys_no_value_bool
      {
        \exp_after:wN \keys_define_key_aux:w
          \l_keys_property_tl \q_stop
          { \use:c { \c_keys_props_root_tl \l_keys_property_tl } }
          {
            \msg_kernel_error:nnxx { keys }
              { property-requires-value } { \l_keys_property_tl }
              { \l_keys_path_tl }
          }
      }
      { \use:c { \c_keys_props_root_tl \l_keys_property_tl } {#1} }
  }
\cs_new_protected:Npn \keys_define_key_aux:w #1 : #2 \q_stop
  { \tl_if_empty:nTF {#2} }
```

(*End definition for* `\keys_define_key:n`*. This function is documented on page* **??**.)

200.4 Turning properties into actions

`\keys_bool_set:NN`

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```
\cs_new:Npn \keys_bool_set:NN #1#2
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \keys_choice_make:
    \keys_cmd_set:nx { \l_keys_path_tl / true }
      { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
    \keys_cmd_set:nx { \l_keys_path_tl / false }
```

```
      { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
    \keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \msg_kernel_error:nnx { keys } { boolean-values-only }
          { \l_keys_key_tl }
      }
    \keys_default_set:n { true }
  }
```

(*End definition for* `\keys_bool_set:NN`*. This function is documented on page* **??**.)

`\keys_bool_set_inverse:NN` Inverse boolean setting is much the same.

```
\cs_new:Npn \keys_bool_set_inverse:NN #1#2
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \keys_choice_make:
    \keys_cmd_set:nx { \l_keys_path_tl / true }
      { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
    \keys_cmd_set:nx { \l_keys_path_tl / false }
      { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
    \keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \msg_kernel_error:nnx { keys } { boolean-values-only }
          { \l_keys_key_tl }
      }
    \keys_default_set:n { true }
  }
```

(*End definition for* `\keys_bool_set_inverse:NN`*. This function is documented on page* **??**.)

`\keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```
\cs_new_protected_nopar:Npn \keys_choice_make:
  {
    \keys_cmd_set:nn { \l_keys_path_tl }
      { \keys_choice_find:n {##1} }
    \keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \msg_kernel_error:nnxx { keys } { choice-unknown }
          { \l_keys_path_tl } {##1}
      }
  }
```

(*End definition for* `\keys_choice_make:`*. This function is documented on page* **??**.)

`\keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```
\cs_new_protected:Npn \keys_choices_make:nn #1#2
  {
    \keys_choice_make:
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#1}
      {
```

```
      \keys_cmd_set:nx { \l_keys_path_tl / ##1 }
        {
          \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
          \int_set:Nn \exp_not:N \l_keys_choice_int
            { \int_use:N \l_keys_choice_int }
          \exp_not:n {#2}
        }
      \int_incr:N \l_keys_choice_int
    }
  }
```

(*End definition for* `\keys_choices_make:nn`*. This function is documented on page* **??**.)

`\keys_choices_generate:n` `\keys_choices_generate_aux:n` Creating multiple-choices means setting up the "indicator" code, then applying whatever the user wanted.

```
\cs_new_protected:Npn \keys_choices_generate:n #1
  {
    \cs_if_exist:cTF
      { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
      {
        \keys_choice_make:
        \int_zero:N \l_keys_choice_int
        \clist_map_function:nN {#1} \keys_choices_generate_aux:n
      }
      {
        \msg_kernel_error:nnx { keys }
          { generate-choices-before-code } { \l_keys_path_tl }
      }
  }
\cs_new_protected:Npn \keys_choices_generate_aux:n #1
  {
    \keys_cmd_set:nx { \l_keys_path_tl / #1 }
      {
        \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
        \int_set:Nn \exp_not:N \l_keys_choice_int
          { \int_use:N \l_keys_choice_int }
        \exp_not:v
          { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
      }
    \int_incr:N \l_keys_choice_int
  }
```

(*End definition for* `\keys_choices_generate:n`*. This function is documented on page* **??**.)

`\keys_choice_code_store:x` The code for making multiple choices is stored in a token list.

```
\cs_new_protected:Npn \keys_choice_code_store:x #1
  {
    \cs_if_exist:cF
      { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
      {
        \tl_new:c
```

```
          { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
      }
    \tl_set:cx { \c_keys_vars_root_tl \l_keys_path_tl .choice~code }
      {#1}
  }
```

(*End definition for* `\keys_choice_code_store:x`. *This function is documented on page* **??**.)

`\keys_cmd_set:nn` `\keys_cmd_set:nx` `\keys_cmd_set_aux:n`

Creating a new command means tidying up the properties and then making the internal function which actually does the work.

```
\cs_new_protected:Npn \keys_cmd_set:nn #1#2
  {
    \keys_cmd_set_aux:n {#1}
    \cs_set:cpn { \c_keys_code_root_tl #1 } ##1 {#2}
  }
\cs_new_protected:Npn \keys_cmd_set:nx #1#2
  {
    \keys_cmd_set_aux:n {#1}
    \cs_set:cpx { \c_keys_code_root_tl #1 } ##1 {#2}
  }
\cs_new_protected:Npn \keys_cmd_set_aux:n #1
  {
    \tl_clear_new:c { \c_keys_vars_root_tl #1 .default }
    \tl_set:cn { \c_keys_vars_root_tl #1 .default } { \q_no_value }
    \tl_clear_new:c { \c_keys_vars_root_tl #1 .req }
  }
```

(*End definition for* `\keys_cmd_set:nn` *and* `\keys_cmd_set:nx`. *These functions are documented on page* **??**.)

`\keys_default_set:n` `\keys_default_set:V`

Setting a default value is easy.

```
\cs_new_protected:Npn \keys_default_set:n #1
  { \tl_set:cn { \c_keys_vars_root_tl \l_keys_path_tl .default } {#1} }
\cs_generate_variant:Nn \keys_default_set:n { V }
```

(*End definition for* `\keys_default_set:n` *and* `\keys_default_set:V`. *These functions are documented on page* **??**.)

`\keys_meta_make:n` `\keys_meta_make:x`

To create a meta-key, simply set up to pass data through.

```
\cs_new_protected:Npn \keys_meta_make:n #1
  {
    \exp_args:NNo \keys_cmd_set:nn \l_keys_path_tl
      { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }
  }
\cs_new_protected:Npn \keys_meta_make:x #1
  {
    \keys_cmd_set:nx { \l_keys_path_tl }
      { \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1} }
  }
```

(*End definition for* `\keys_meta_make:n` *and* `\keys_meta_make:x`. *These functions are documented on page* **??**.)

\keys_multichoice_find:n
\keys_multichoice_make:
\keys_multichoices_make:nn

Choices where several values can be selected are very similar to normal exclusive choices. There is just a slight change in implementation to map across a comma-separated list. This then requires that the appropriate set up takes place elsewhere.

```
\cs_new:Npn \keys_multichoice_find:n #1
  { \clist_map_function:nN {#1} \keys_choice_find:n }
\cs_new_protected_nopar:Npn \keys_multichoice_make:
  {
    \keys_cmd_set:nn { \l_keys_path_tl }
      { \keys_multichoice_find:n {##1} }
    \keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \msg_kernel_error:nnxx { keys } { choice-unknown }
          { \l_keys_path_tl } {##1}
      }
  }
\cs_new_protected:Npn \keys_multichoices_make:nn #1#2
  {
    \keys_multichoice_make:
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#1}
      {
        \keys_cmd_set:nx { \l_keys_path_tl / ##1 }
          {
            \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
            \int_set:Nn \exp_not:N \l_keys_choice_int
              { \int_use:N \l_keys_choice_int }
            \exp_not:n {#2}
          }
        \int_incr:N \l_keys_choice_int
      }
  }
```

(*End definition for* `\keys_multichoice_find:n`*. This function is documented on page* **??**.)

\keys_value_requirement:n

Values can be required or forbidden by having the appropriate marker set.

```
\cs_new_protected:Npn \keys_value_requirement:n #1
  {
    \tl_set_eq:cc
      { \c_keys_vars_root_tl \l_keys_path_tl .req }
      { c_keys_value_ #1 _tl }
  }
```

(*End definition for* `\keys_value_requirement:n`*. This function is documented on page* **??**.)

\keys_variable_set:NnNN
\keys_variable_set:cnNN
\keys_variable_set:NnN
\keys_variable_set:cnN

Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed. The three-argument version is set up so that the use of `{ }` as an `N`-type variable is only done once!

```
\cs_new_protected:Npn \keys_variable_set:NnNN #1#2#3#4
  {
    \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
    \keys_cmd_set:nx { \l_keys_path_tl }
```

```
      { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1} }
  }
\cs_new_protected:Npn \keys_variable_set:NnN #1#2#3
  { \keys_variable_set:NnNN #1 {#2} { } #3 }
\cs_generate_variant:Nn \keys_variable_set:NnNN { c }
\cs_generate_variant:Nn \keys_variable_set:NnN  { c }
```

(*End definition for* `\keys_variable_set:NnNN` *and* `\keys_variable_set:cnNN`*. These functions are documented on page* **??**.)

200.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

`.bool_set:N` `.bool_gset:N` One function for this.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .bool_set:N } #1
  { \keys_bool_set:NN #1 { } }
\cs_new_protected:cpn { \c_keys_props_root_tl .bool_gset:N } #1
  { \keys_bool_set:NN #1 g }
```

(*End definition for* `.bool_set:N`*. This function is documented on page 147.*)

`.bool_set_inverse:N` `.bool_gset_inverse:N` One function for this.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .bool_set_inverse:N } #1
  { \keys_bool_set_inverse:NN #1 { } }
\cs_new_protected:cpn { \c_keys_props_root_tl .bool_gset_inverse:N } #1
  { \keys_bool_set_inverse:NN #1 g }
```

(*End definition for* `.bool_set_inverse:N`*. This function is documented on page 147.*)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
\cs_new_protected_nopar:cpn { \c_keys_props_root_tl .choice: }
  { \keys_choice_make: }
```

(*End definition for* `.choice:`*. This function is documented on page* **??**.)

`.choices:nn` For auto-generation of a series of mutually-exclusive choices. Here, `#1` will consist of two separate arguments, hence the slightly odd-looking implementation.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .choices:nn } #1
  { \keys_choices_make:nn #1 }
```

(*End definition for* `.choices:nn`*. This function is documented on page 147.*)

`.code:n` `.code:x` Creating code is simply a case of passing through to the underlying `set` function.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .code:n } #1
  { \keys_cmd_set:nn { \l_keys_path_tl } {#1} }
\cs_new_protected:cpn { \c_keys_props_root_tl .code:x } #1
  { \keys_cmd_set:nx { \l_keys_path_tl } {#1} }
```

(*End definition for* `.code:n` *and* `.code:x`*. These functions are documented on page 148.*)

`.choice_code:n`
`.choice_code:x`

Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:n } #1
  { \keys_choice_code_store:x { \exp_not:n {#1} } }
\cs_new_protected:cpn { \c_keys_props_root_tl .choice_code:x } #1
  { \keys_choice_code_store:x {#1} }
```

(*End definition for* `.choice_code:n` *and* `.choice_code:x`. *These functions are documented on page* *147*.)

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

```
\cs_new_protected:cpn { \c_keys_props_root_tl .clist_set:N } #1
  { \keys_variable_set:NnN #1 { clist } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .clist_set:c } #1
  { \keys_variable_set:cnN {#1} { clist } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .clist_gset:N } #1
  { \keys_variable_set:NnNN #1 { clist } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .clist_gset:c } #1
  { \keys_variable_set:cnNN {#1} { clist } g n }
```

(*End definition for* `.clist_set:N` *and* `.clist_set:c`. *These functions are documented on page* *147*.)

`.default:n`
`.default:V`

Expansion is left to the internal functions.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .default:n } #1
  { \keys_default_set:n {#1} }
\cs_new_protected:cpn { \c_keys_props_root_tl .default:V } #1
  { \keys_default_set:V #1 }
```

(*End definition for* `.default:n` *and* `.default:V`. *These functions are documented on page* *148*.)

`.dim_set:N`
`.dim_set:c`
`.dim_gset:N`
`.dim_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .dim_set:N } #1
  { \keys_variable_set:NnN #1 { dim } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .dim_set:c } #1
  { \keys_variable_set:cnN {#1} { dim } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .dim_gset:N } #1
  { \keys_variable_set:NnNN #1 { dim } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .dim_gset:c } #1
  { \keys_variable_set:cnNN {#1} { dim } g n }
```

(*End definition for* `.dim_set:N` *and* `.dim_set:c`. *These functions are documented on page* *148*.)

`.fp_set:N`
`.fp_set:c`
`.fp_gset:N`
`.fp_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .fp_set:N } #1
  { \keys_variable_set:NnN #1 { fp } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .fp_set:c } #1
  { \keys_variable_set:cnN {#1} { fp } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .fp_gset:N } #1
  { \keys_variable_set:NnNN #1 { fp } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .fp_gset:c } #1
  { \keys_variable_set:cnNN {#1} { fp } g n }
```

(*End definition for* `.fp_set:N` *and* `.fp_set:c`. *These functions are documented on page* *148*.)

`.generate_choices:n`

Making choices is easy.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .generate_choices:n } #1
  { \keys_choices_generate:n {#1} }
```

(*End definition for* `.generate_choices:n`. *This function is documented on page 149.*)

`.int_set:N`
`.int_set:c`
`.int_gset:N`
`.int_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .int_set:N } #1
  { \keys_variable_set:NnN #1 { int } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .int_set:c } #1
  { \keys_variable_set:cnN {#1} { int } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .int_gset:N } #1
  { \keys_variable_set:NnNN #1 { int } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .int_gset:c } #1
  { \keys_variable_set:cnNN {#1} { int } g n }
```

(*End definition for* `.int_set:N` *and* `.int_set:c`. *These functions are documented on page 149.*)

`.meta:n`
`.meta:x`

Making a meta is handled internally.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .meta:n } #1
  { \keys_meta_make:n {#1} }
\cs_new_protected:cpn { \c_keys_props_root_tl .meta:x } #1
  { \keys_meta_make:x {#1} }
```

(*End definition for* `.meta:n` *and* `.meta:x`. *These functions are documented on page 149.*)

`.multichoice:`
`.multichoices:nn`

The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
\cs_new_protected_nopar:cpn { \c_keys_props_root_tl .multichoice: }
  { \keys_multichoice_make: }
\cs_new_protected:cpn { \c_keys_props_root_tl .multichoices:nn } #1
  { \keys_multichoices_make:nn #1 }
```

(*End definition for* `.multichoice:`. *This function is documented on page* **??**.)

`.skip_set:N`
`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:N } #1
  { \keys_variable_set:NnN #1 { skip } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:c } #1
  { \keys_variable_set:cnN {#1} { skip } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:N } #1
  { \keys_variable_set:NnNN #1 { skip } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:c } #1
  { \keys_variable_set:cnNN {#1} { skip } g n }
```

(*End definition for* `.skip_set:N` *and* `.skip_set:c`. *These functions are documented on page 149.*)

`.tl_set:N`
`.tl_set:c`
`.tl_gset:N`
`.tl_gset:c`
`.tl_set_x:N`
`.tl_set_x:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:N } #1
   { \keys_variable_set:NnN #1 { tl } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:c } #1
  { \keys_variable_set:cnN {#1} { tl } n }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:N } #1
  { \keys_variable_set:NnN #1 { tl } x }
```

```
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:c } #1
  { \keys_variable_set:cnN {#1} { tl } x }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:N } #1
  { \keys_variable_set:NnNN #1 { tl } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:c } #1
  { \keys_variable_set:cnNN {#1} { tl } g n }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:N } #1
  { \keys_variable_set:NnNN #1 { tl } g x }
\cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:c } #1
  { \keys_variable_set:cnNN {#1} { tl } g x }
```

(*End definition for* `.tl_set:N` *and* `.tl_set:c`*. These functions are documented on page 150.*)

`.value_forbidden:`
`.value_required:`

These are very similar, so both call the same function.

```
\cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_forbidden: }
  { \keys_value_requirement:n { forbidden } }
\cs_new_protected_nopar:cpn { \c_keys_props_root_tl .value_required: }
  { \keys_value_requirement:n { required } }
```

(*End definition for* `.value_forbidden:`*. This function is documented on page* **??**.)

200.6 Setting keys

`\keys_set:nn`
`\keys_set:nV`
`\keys_set:nv`
`\keys_set:no`
`\keys_set_aux:nnn`
`\keys_set_aux:onn`

A simple wrapper again.

```
\cs_new_protected:Npn \keys_set:nn
  { \keys_set_aux:onn { \l_keys_module_tl } }
\cs_new_protected:Npn \keys_set_aux:nnn #1#2#3
  {
    \tl_set:Nx \l_keys_module_tl { \tl_to_str:n {#2} }
    \keyval_parse:NNn \keys_set_elt:n \keys_set_elt:nn {#3}
    \tl_set:Nn \l_keys_module_tl {#1}
  }
\cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
\cs_generate_variant:Nn \keys_set_aux:nnn { o }
```

(*End definition for* `\keys_set:nn` *and others. These functions are documented on page* **??**.)

`\keys_set_known:nnN`
`\keys_set_known:nVN`
`\keys_set_known:nvN`
`\keys_set_known:noN`
`\keys_set_known_aux:nnnN`
`\keys_set_known_aux:onnN`

```
\cs_new_protected:Npn \keys_set_known:nnN
  { \keys_set_known_aux:onnN { \l_keys_module_tl } }
\cs_new_protected:Npn \keys_set_known_aux:nnnN #1#2#3#4
  {
    \tl_set:Nx \l_keys_module_tl { \tl_to_str:n {#2} }
    \clist_clear:N \l_keys_unknown_clist
    \cs_set_eq:NN \keys_execute_unknown: \keys_execute_unknown_alt:
    \keyval_parse:NNn \keys_set_elt:n \keys_set_elt:nn {#3}
    \cs_set_eq:NN \keys_execute_unknown: \keys_execute_unknown_std:
    \tl_set:Nn \l_keys_module_tl {#1}
    \clist_set_eq:NN #4 \l_keys_unknown_clist
  }
\cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\cs_generate_variant:Nn \keys_set_known_aux:nnnN { o }
```

(*End definition for* `\keys_set_known:nnN` *and others. These functions are documented on page* **??**.)

`\keys_set_elt:n`
`\keys_set_elt:nn`
`\keys_set_elt_aux:nn`

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```
\cs_new_protected:Npn \keys_set_elt:n #1
  {
    \bool_set_true:N \l_keys_no_value_bool
    \keys_set_elt_aux:nn {#1} { }
  }
\cs_new_protected:Npn \keys_set_elt:nn #1#2
  {
    \bool_set_false:N \l_keys_no_value_bool
    \keys_set_elt_aux:nn {#1} {#2}
  }
\cs_new_protected:Npn \keys_set_elt_aux:nn #1#2
  {
    \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
    \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }
    \keys_value_or_default:n {#2}
    \bool_if:nTF
      {
        \keys_if_value_p:n { required } &&
        \l_keys_no_value_bool
      }
      {
        \msg_kernel_error:nnx { keys } { value-required }
          { \l_keys_path_tl }
      }
      {
        \bool_if:nTF
          {
              \keys_if_value_p:n { forbidden } &&
            ! \l_keys_no_value_bool
          }
          {
            \msg_kernel_error:nnxx { keys } { value-forbidden }
              { \l_keys_path_tl } { \l_keys_value_tl }
          }
          { \keys_execute: }
      }
  }
```

(*End definition for* `\keys_set_elt:n` *and* `\keys_set_elt:nn`*. These functions are documented on page* **??**.)

`\keys_value_or_default:n`

If a value is given, return it as `#1`, otherwise send a default if available.

```
\cs_new_protected:Npn \keys_value_or_default:n #1
  {
    \tl_set:Nn \l_keys_value_tl {#1}
```

```
    \bool_if:NT \l_keys_no_value_bool
      {
        \quark_if_no_value:cF { \c_keys_vars_root_tl \l_keys_path_tl .default }
          {
            \cs_if_exist:cT { \c_keys_vars_root_tl \l_keys_path_tl .default }
              {
                \tl_set_eq:Nc \l_keys_value_tl
                  { \c_keys_vars_root_tl \l_keys_path_tl .default }
              }
          }
      }
  }
```

(*End definition for* `\keys_value_or_default:n`*. This function is documented on page* **??**.)

`\keys_if_value_p:n`

To test if a value is required or forbidden. A simple check for the existence of the appropriate marker.

```
\prg_new_conditional:Npnn \keys_if_value:n #1 { p }
  {
    \tl_if_eq:ccTF { c_keys_value_ #1 _tl }
      { \c_keys_vars_root_tl \l_keys_path_tl .req }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_value_p:n`*. This function is documented on page* **??**.)

`\keys_execute:`
`\keys_execute_unknown:`
`\keys_execute_unknown_std:`
`\keys_execute_unknown_alt:`
`\keys_execute:nn`

Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain.

```
\cs_new_nopar:Npn \keys_execute:
  { \keys_execute:nn { \l_keys_path_tl } { \keys_execute_unknown: } }
\cs_new_nopar:Npn \keys_execute_unknown:
  {
    \keys_execute:nn { \l_keys_module_tl / unknown }
      {
        \msg_kernel_error:nnxx { keys } { key-unknown }
          { \l_keys_path_tl } { \l_keys_module_tl }
      }
  }
\cs_new_eq:NN \keys_execute_unknown_std: \keys_execute_unknown:
\cs_new_nopar:Npn \keys_execute_unknown_alt:
  {
    \clist_put_right:Nx \l_keys_unknown_clist
      {
        \exp_not:o \l_keys_key_tl
        \bool_if:NF \l_keys_no_value_bool
          { = { \exp_not:o \l_keys_value_tl } }
      }
  }
\cs_new:Npn \keys_execute:nn #1#2
  {
```

```
    \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
      {
        \exp_args:Nc \exp_args:No { \c_keys_code_root_tl #1 }
          \l_keys_value_tl
      }
      {#2}
  }
```

(*End definition for* `\keys_execute:`. *This function is documented on page* **??**.)

`\keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```
\cs_new:Npn \keys_choice_find:n #1
  {
    \keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
      { \keys_execute:nn { \l_keys_path_tl / unknown } { } }
  }
```

(*End definition for* `\keys_choice_find:n`. *This function is documented on page* **??**.)

200.7 Utilities

`\keys_if_exist_p:nn`
`\keys_if_exist:nnTF`

A utility for others to see if a key exists.

```
\prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c_keys_code_root_tl #1 / #2 }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_exist:nn`. *These functions are documented on page 154.*)

`\keys_if_choice_exist_p:nnn`
`\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nn(TF)`.

```
\prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c_keys_code_root_tl #1 / #2 / #3 }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_choice_exist:nnn`. *These functions are documented on page* **??**.)

`\keys_show:nn` Showing a key is just a question of using the correct name.

```
\cs_new:Npn \keys_show:nn #1#2
  { \cs_show:c { \c_keys_code_root_tl #1 / \tl_to_str:n {#2} } }
```

(*End definition for* `\keys_show:nn`. *This function is documented on page 154.*)

200.8 Messages

For when there is a need to complain.

```
\msg_kernel_new:nnnn { keys } { boolean-values-only }
  { Key~’#1’~accepts~boolean~values~only. }
  { The~key~’#1’~only~accepts~the~values~’true’~and~’false’. }
\msg_kernel_new:nnnn { keys } { choice-unknown }
  { Choice~’#2’~unknown~for~key~’#1’. }
  {
    The~key~’#1’~takes~a~limited~number~of~values.\\
    The~input~given,~’#2’,~is~not~on~the~list~accepted.
  }
\msg_kernel_new:nnnn { keys } { generate-choices-before-code }
  { No~code~available~to~generate~choices~for~key~’#1’. }
  {
    \c_msg_coding_error_text_tl
    Before~using~.generate_choices:n~the~code~should~be~defined~
    with~’.choice_code:n’~or~’.choice_code:x’.
  }
\msg_kernel_new:nnnn { keys } { key-no-property }
  { No~property~given~in~definition~of~key~’#1’. }
  {
    \c_msg_coding_error_text_tl
    Inside~\keys_define:nn each~key~name
    needs~a~property:  \\
    ~ ~ #1 .<property> \\
    LaTeX~did~not~find~a~’.’~to~indicate~the~start~of~a~property.
  }
\msg_kernel_new:nnnn { keys } { key-unknown }
  { The~key~’#1’~is~unknown~and~is~being~ignored. }
  {
    The~module~’#2’~does~not~have~a~key~called~#1’.\\
    Check~that~you~have~spelled~the~key~name~correctly.
  }
\msg_kernel_new:nnnn { keys } { option-unknown }
  { Unknown~option~’#1’~for~package~#2. }
  {
    LaTeX~has~been~asked~to~set~an~option~called~’#1’~
    but~the~#2~package~has~not~created~an~option~with~this~name.
  }
\msg_kernel_new:nnnn { keys } { property-requires-value }
  { The~property~’#1’~requires~a~value. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~set~property~’#2’~for~key~’#1’.\\
    No~value~was~given~for~the~property,~and~one~is~required.
  }
\msg_kernel_new:nnnn { keys } { property-unknown }
  { The~key~property~’#1’~is~unknown. }
  {
```

```
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
    this~property~is~not~defined.
  }
\msg_kernel_new:nnnn { keys } { value-forbidden }
  { The~key~'#1'~does~not~taken~a~value. }
  {
    The~key~'#1'~should~be~given~without~a~value.\\
    LaTeX~will~ignore~the~given~value~'#2'.
  }
\msg_kernel_new:nnnn { keys } { value-required }
  { The~key~'#1'~requires~a~value. }
  {
    The~key~'#1'~must~have~a~value.\\
    No~value~was~present:~the~key~will~be~ignored.
  }
```

200.9 Deprecated functions

Deprecated on 2011-05-27, for removal by 2011-08-31.

\KV_process_space_removal_sanitize:NNn
\KV_process_space_removal_no_sanitize:NNn
\KV_process_no_space_removal_no_sanitize:NNn

There is just one function for this now.

```
⟨*deprecated⟩
\cs_new_eq:NN \KV_process_space_removal_sanitize:NNn       \keyval_parse:NNn
\cs_new_eq:NN \KV_process_space_removal_no_sanitize:NNn    \keyval_parse:NNn
\cs_new_eq:NN \KV_process_no_space_removal_no_sanitize:NNn \keyval_parse:NNn
⟨/deprecated⟩
```

(*End definition for* \KV_process_space_removal_sanitize:NNn*. This function is documented on page* ??.)

```
⟨/initex | package⟩
```

201 l3file implementation

The following test files are used for this code: m3file001.

```
⟨*initex | package⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

201.1 File operations

\g_file_current_name_tl

The name of the current file should be available at all times.

```
\tl_new:N \g_file_current_name_tl
```

For the format the file name needs to be picked up at the start of the file. In package mode the current file name is collected from LaTeX 2_ε.

```
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
  }
⟨/initex⟩
⟨*package⟩
\tl_gset_eq:NN \g_file_current_name_tl \@currname
⟨/package⟩
```

(*End definition for* `\g_file_current_name_tl`. *This variable is documented on page 156.*)

`\g_file_stack_seq` The input list of files is stored as a sequence stack.

```
\seq_new:N \g_file_stack_seq
```

(*End definition for* `\g_file_stack_seq`. *This variable is documented on page 162.*)

`\g_file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list.

```
\seq_new:N \g_file_record_seq
```

The current file name should be included in the file list!

```
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
  }
⟨/initex⟩
```

(*End definition for* `\g_file_record_seq`. *This variable is documented on page 162.*)

`\l_file_internal_name_tl` Used to return the fully-qualified name of a file.

```
\tl_new:N \l_file_internal_name_tl
```

(*End definition for* `\l_file_internal_name_tl`. *This variable is documented on page 162.*)

`\l_file_search_path_seq` The current search path.

```
\seq_new:N \l_file_search_path_seq
```

(*End definition for* `\l_file_search_path_seq`. *This variable is documented on page 162.*)

`\l_file_internal_saved_path_seq` The current search path has to be saved for package use.

```
⟨*package⟩
\seq_new:N \l_file_internal_saved_path_seq
⟨/package⟩
```

(*End definition for* `\l_file_internal_saved_path_seq`. *This variable is documented on page 162.*)

\l_file_internal_seq

Scratch space for comma list conversion in package mode.

```
⟨*package⟩
\seq_new:N \l_file_internal_seq
⟨/package⟩
```

(*End definition for* `\l_file_internal_seq`. *This variable is documented on page* *163*.)

\file_name_sanitize:nn

For converting a token list to a string where active characters are treated as strings from the start.

```
\cs_new_protected:Npn \file_name_sanitize:nn #1#2
  {
    \group_begin:
      \seq_map_inline:Nn \l_char_active_seq
        { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
      \tl_set:Nx \l_file_internal_name_tl {#1}
      \tl_set:Nx \l_file_internal_name_tl
        { \tl_to_str:N \l_file_internal_name_tl }
      \tl_if_in:NnTF \l_file_internal_name_tl { ~ }
        {
          \msg_kernel_error:nnx { file } { space-in-file-name }
            { \l_file_internal_name_tl }
        }
      \use:x
        {
          \group_end:
          \exp_not:n {#2} { \l_file_internal_name_tl }
        }
  }
```

(*End definition for* `\file_name_sanitize:nn`. *This function is documented on page* *163*.)

\file_add_path:nN
\file_add_path_aux:nN
\file_add_path_search:nN

The way to test if a file exists is to try to open it: if it does not exist then TeX will report end-of-file. For files which are in the current directory, this is straight-forward. For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, `#2` is returned empty.

```
\cs_new_protected:Npn \file_add_path:nN #1
  { \file_name_sanitize:nn {#1} { \file_add_path_aux:nN } }
\cs_new_protected:Npn \file_add_path_aux:nN #1#2
  {
    \ior_open_unsafe:Nn \g_file_internal_ior {#1}
    \ior_if_eof:NTF \g_file_internal_ior
      { \file_add_path_search:nN {#1} #2 }
      {
        \ior_close:N \g_file_internal_ior
        \tl_set:Nn #2 {#1}
      }
  }
\cs_new_protected:Npn \file_add_path_search:nN #1#2
  {
```

```
    \tl_set:Nn #2 { \q_no_value }
⟨*package⟩
    \cs_if_exist:NT \input@path
      {
        \seq_set_eq:NN \l_file_internal_saved_path_seq \l_file_search_path_seq
        \seq_set_from_clist:NN \l_file_internal_seq \input@path
        \seq_concat:NNN \l_file_search_path_seq
          \l_file_search_path_seq \l_file_internal_seq
      }
⟨/package⟩
    \seq_map_inline:Nn \l_file_search_path_seq
      {
        \ior_open_unsafe:Nn \g_file_internal_ior { ##1 #1 }
        \ior_if_eof:NF \g_file_internal_ior
          {
            \tl_set:Nx #2 { ##1 #1 }
            \seq_map_break:
          }
      }
⟨*package⟩
    \cs_if_exist:NT \input@path
      { \seq_set_eq:NN \l_file_search_path_seq \l_file_internal_saved_path_seq }
⟨/package⟩
    \ior_close:N \g_file_internal_ior
  }
```

(*End definition for* `\file_add_path:nN`*. This function is documented on page* 156*.*)

`\file_if_exist:nTF`

The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path will contain something, whereas if the file was not located then the return value will be empty.

```
\prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
  {
    \file_add_path:nN {#1} \l_file_internal_name_tl
    \quark_if_no_value:NTF \l_file_internal_name_tl
      { \prg_return_false: }
      { \prg_return_true: }
  }
```

(*End definition for* `\file_if_exist:nTF`*. This function is documented on page* 156*.*)

`\file_input:n`
`\file_input_aux:n`
`\file_input_aux:V`
`\file_input_error:n`

Loading a file is done in a safe way, checking first that the file exists and loading only if it does.

```
\cs_new_protected:Npn \file_input:n #1
  {
    \file_add_path:nN {#1} \l_file_internal_name_tl
    \quark_if_no_value:NTF \l_file_internal_name_tl
      { \file_name_sanitize:nn {#1} { \file_input_error:n } }
      { \file_input_aux:V \l_file_internal_name_tl }
  }
\cs_new_protected:Npn \file_input_aux:n #1
```

```
  {
⟨*initex⟩
    \seq_gput_right:Nn \g_file_record_seq {#1}
⟨/initex⟩
⟨*package⟩
    \@addtofilelist {#1}
⟨/package⟩
    \seq_gpush:Nn \g_file_stack_seq \g_file_current_name_tl
    \tl_gset:Nn \g_file_current_name_tl {#1}
    \tex_input:D #1 \c_space_tl
    \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
  }
\cs_generate_variant:Nn \file_input_aux:n { V }
\cs_new_protected:Npn \file_input_error:n #1
  { \msg_kernel_error:nnx { file } { file-not-found } {#1} }
```

(*End definition for* `\file_input:n`*. This function is documented on page 157.*)

`\file_path_include:n`
`\file_path_remove:n`

Wrapper functions to manage the search path.

```
\cs_new_protected:Npn \file_path_include:n #1
  {
    \seq_if_in:NnF  \l_file_search_path_seq {#1}
      { \seq_put_right:Nn \l_file_search_path_seq {#1} }
  }
\cs_new_protected:Npn \file_path_remove:n #1
  { \seq_remove_all:Nn \l_file_search_path_seq {#1} }
```

(*End definition for* `\file_path_include:n`*. This function is documented on page 157.*)

`\file_list:`

A function to list all files used to the log.

```
\cs_new_protected_nopar:Npn \file_list:
  {
    \seq_remove_duplicates:N \g_file_record_seq
    \iow_log:n { *~File~List~* }
    \seq_map_inline:Nn \g_file_record_seq { \iow_log:n {##1} }
    \iow_log:n { ************* }
  }
```

(*End definition for* `\file_list:`*. This function is documented on page* **??**.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```
⟨*package⟩
\AtBeginDocument
  {
    \seq_set_from_clist:NN \l_file_internal_seq \@filelist
    \seq_gconcat:NNN \g_file_record_seq \g_file_record_seq \l_file_internal_seq
  }
⟨/package⟩
```

201.2 Input–output variables constants

\c_term_ior

Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
\cs_new_eq:NN \c_term_ior \c_sixteen
```

(*End definition for* \c_term_ior*. This variable is documented on page 162.*)

\c_log_iow
\c_term_iow

Here we allocate two output streams for writing to the transcript file only (\c_log_iow) and to both the terminal and transcript file (\c_term_iow).

```
\cs_new_eq:NN \c_log_iow  \c_minus_one
\cs_new_eq:NN \c_term_iow \c_sixteen
```

(*End definition for* \c_log_iow *and* \c_term_iow*. These variables are documented on page 162.*)

\c_iow_streams_tl
\c_ior_streams_tl

The list of streams available, by number.

```
\tl_const:Nn \c_iow_streams_tl
  {
    \c_zero
    \c_one
    \c_two
    \c_three
    \c_four
    \c_five
    \c_six
    \c_seven
    \c_eight
    \c_nine
    \c_ten
    \c_eleven
    \c_twelve
    \c_thirteen
    \c_fourteen
    \c_fifteen
  }
\cs_new_eq:NN \c_ior_streams_tl \c_iow_streams_tl
```

(*End definition for* \c_iow_streams_tl *and* \c_ior_streams_tl*. These variables are documented on page* **??**.)

\g_iow_streams_prop
\g_ior_streams_prop

The allocations for streams are stored in property lists, which are set up to have a "full" set of allocations from the start. In package mode, a few slots are always taken, so these are blocked off from use.

```
\prop_new:N \g_iow_streams_prop
\prop_new:N \g_ior_streams_prop
⟨*package⟩
\prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e~reserved }
\prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e~reserved }
\prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e~reserved }
\prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e~reserved }
⟨/package⟩
```

(*End definition for* `\g_iow_streams_prop` *and* `\g_ior_streams_prop`*. These variables are documented on page* **??**.)

`\l_iow_stream_int`
`\l_ior_stream_int`

Used to track the number allocated to the stream being created: this is taken from the property list but does alter.

```
\int_new:N \l_iow_stream_int
\cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int
```

(*End definition for* `\l_iow_stream_int` *and* `\l_ior_stream_int`*. These variables are documented on page* **??**.)

201.3 Stream management

`\ior_raw_new:N`
`\ior_raw_new:c`
`\iow_raw_new:N`
`\iow_raw_new:c`

The lowest level for stream management is actually creating raw TeX streams. As these are very limited (even with ε-TeX), this should not be addressed directly.

```
⟨*initex⟩
\alloc_setup_type:nnn { ior } \c_zero \c_sixteen
\cs_new_protected:Npn \ior_raw_new:N #1
  { \alloc_reg:nNN { ior } \tex_chardef:D #1 }
\alloc_setup_type:nnn { iow } \c_zero \c_sixteen
\cs_new_protected:Npn \iow_raw_new:N #1
  { \alloc_reg:nNN { iow } \tex_chardef:D #1 }
⟨/initex⟩
⟨*package⟩
\cs_set_eq:NN \iow_raw_new:N \newwrite
\cs_set_eq:NN \ior_raw_new:N \newread
⟨/package⟩
\cs_generate_variant:Nn \ior_raw_new:N { c }
\cs_generate_variant:Nn \iow_raw_new:N { c }
```

(*End definition for* `\ior_raw_new:N` *and* `\ior_raw_new:c`*. These functions are documented on page* **??**.)

`\ior_new:N`
`\ior_new:c`
`\iow_new:N`
`\iow_new:c`

Reserving a new stream is done by defining the name as equal to using the terminal.

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*End definition for* `\ior_new:N` *and others. These functions are documented on page* **??**.)

`\g_file_internal_ior`

Delayed from above so that the mechanisms are in place.

```
\ior_new:N \g_file_internal_ior
```

(*End definition for* `\g_file_internal_ior`*. This variable is documented on page* **??**.)

`\ior_open:Nn`
`\ior_open:cn`
`\iow_open:Nn`
`\iow_open:cn`
`\ior_open_aux:Nn`
`\ior_open:NnTF`
`\ior_open_aux:NnTF`
`\ior_open_unsafe:Nn`
`\ior_open_unsafe:No`
`\iow_open_unsafe:Nn`

In both cases, opening a stream starts with a call to the closing function: this is safest. There is then a loop through the allocation number list to find the first free stream number. When one is found the allocation can take place, the information can be stored and finally the file can actually be opened. Before any actual file operations there is a precaution against special characters in file names. For reading files, there is an intermediate auxiliary to allow path addition, keeping the internal function fast and avoiding an infinite loop.

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \file_name_sanitize:nn {#2} { \ior_open_aux:Nn #1 } }
\cs_generate_variant:Nn \ior_open:Nn { c }
\cs_new_protected:Npn \iow_open:Nn #1#2
  { \file_name_sanitize:nn {#2} { \iow_open_unsafe:Nn #1 } }
\cs_generate_variant:Nn \iow_open:Nn { c }
\cs_new_protected:Npn \ior_open_aux:Nn #1#2
  {
    \file_add_path:nN {#2} \l_file_internal_name_tl
    \quark_if_no_value:NTF \l_file_internal_name_tl
      { \file_input_error:n {#2} }
      { \ior_open_unsafe:No #1 \l_file_internal_name_tl }
  }
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  { \file_name_sanitize:nn {#2} { \ior_open_aux:NnTF #1 } }
\cs_new_protected:Npn \ior_open_aux:NnTF #1#2
  {
    \file_add_path:nN {#2} \l_file_internal_name_tl
    \quark_if_no_value:NTF \l_file_internal_name_tl
      { \prg_return_false: }
      {
        \ior_open_unsafe:No #1 \l_file_internal_name_tl
        \prg_return_true:
      }
  }
\cs_generate_variant:Nn \ior_open:NnT  { c }
\cs_generate_variant:Nn \ior_open:NnF  { c }
\cs_generate_variant:Nn \ior_open:NnTF { c }
\cs_new_protected:Npn \ior_open_unsafe:Nn #1#2
  {
    \ior_close:N #1
    \int_set:Nn \l_ior_stream_int \c_sixteen
    \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
    \int_compare:nNnTF \l_ior_stream_int = \c_sixteen
      { \msg_kernel_fatal:nn { ior } { streams-exhausted } }
      {
        \ior_stream_alloc:N #1
        \prop_gput:NVn \g_ior_streams_prop \l_ior_stream_int {#2}
        \tex_openin:D #1#2 \scan_stop:
      }
  }
\cs_generate_variant:Nn \ior_open_unsafe:Nn { No }
\cs_new_protected:Npn \iow_open_unsafe:Nn #1#2
  {
    \iow_close:N #1
    \int_set:Nn \l_iow_stream_int \c_sixteen
    \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
    \int_compare:nNnTF \l_iow_stream_int = \c_sixteen
      { \msg_kernel_fatal:nn { iow } { streams-exhausted } }
      {
```

```
        \iow_stream_alloc:N #1
        \prop_gput:NVn \g_iow_streams_prop \l_iow_stream_int {#2}
        \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
      }
  }
```

(*End definition for* `\ior_open:Nn` *and others. These functions are documented on page 163.*)

`\ior_alloc_read:n`
`\iow_alloc_write:n`

These functions are used to see if a particular stream is available. The property list contains file names for streams in use, so any unused ones are for the taking.

```
\cs_new_protected:Npn \iow_alloc_write:n #1
  {
    \prop_if_in:NnF \g_iow_streams_prop {#1}
      {
        \int_set:Nn \l_iow_stream_int {#1}
        \tl_map_break:
      }
  }
\cs_new_protected:Npn \ior_alloc_read:n #1
  {
    \prop_if_in:NnF \g_iow_streams_prop {#1}
      {
        \int_set:Nn \l_ior_stream_int {#1}
        \tl_map_break:
      }
  }
```

(*End definition for* `\ior_alloc_read:n`*. This function is documented on page 163.*)

`\iow_stream_alloc:N`
`\ior_stream_alloc:N`
`\iow_stream_alloc_aux:`
`\ior_stream_alloc_aux:`
`\g_iow_internal_iow`
`\g_ior_internal_ior`

Allocating a raw stream is much easier in IniTeX mode than for the package. For the format, all streams will be allocated by l3file and so there is a simple check to see if a raw stream is actually available. On the other hand, for the package there will be non-managed streams. So if the managed one is not open, a check is made to see if some other managed stream is available before deciding to open a new one. If a new one is needed, we get the number allocated by LaTeX 2_ε to get "back on track" with allocation.

```
\iow_new:N \g_iow_internal_iow
\ior_new:N \g_ior_internal_ior
\cs_new_protected:Npn \iow_stream_alloc:N #1
  {
    \cs_if_exist:cF { g_iow_ \int_use:N \l_iow_stream_int _iow }
      {
⟨*package⟩
        \iow_stream_alloc_aux:
        \int_compare:nNnT \l_iow_stream_int = \c_sixteen
          {
            \iow_raw_new:N \g_iow_internal_iow
            \int_set:Nn \l_iow_stream_int { \g_iow_internal_iow }
            \cs_gset_eq:cN
              { g_iow_ \int_use:N \l_iow_stream_int _iow } \g_iow_internal_iow
          }
⟨/package⟩
```

```
⟨*initex⟩
    \iow_raw_new:c { g_iow_ \int_use:N \l_iow_stream_int _iow }
⟨/initex⟩
   }
  \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _iow }
 }
⟨*package⟩
\cs_new_protected_nopar:Npn \iow_stream_alloc_aux:
 {
  \int_incr:N \l_iow_stream_int
  \int_compare:nNnT \l_iow_stream_int < \c_sixteen
   {
    \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _iow }
     {
      \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
       { \iow_stream_alloc_aux: }
     }
     { \iow_stream_alloc_aux: }
   }
 }
⟨/package⟩
\cs_new_protected:Npn \ior_stream_alloc:N #1
 {
  \cs_if_exist:cF { g_ior_ \int_use:N \l_ior_stream_int _ior }
   {
⟨*package⟩
    \ior_stream_alloc_aux:
    \int_compare:nNnT \l_ior_stream_int = \c_sixteen
     {
      \ior_raw_new:N \g_ior_internal_ior
      \int_set:Nn \l_ior_stream_int { \g_ior_internal_ior }
      \cs_gset_eq:cN
       { g_ior_ \int_use:N \l_iow_stream_int _ior } \g_ior_internal_ior
     }
⟨/package⟩
⟨*initex⟩
    \ior_raw_new:c { g_ior_ \int_use:N \l_ior_stream_int _ior }
⟨/initex⟩
   }
  \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int _ior }
 }
⟨*package⟩
\cs_new_protected_nopar:Npn \ior_stream_alloc_aux:
 {
  \int_incr:N \l_ior_stream_int
  \int_compare:nNnT \l_ior_stream_int < \c_sixteen
   {
    \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int _ior }
     {
      \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
```

```
              { \ior_stream_alloc_aux: }
          }
          { \ior_stream_alloc_aux: }
      }
  }
⟨/package⟩
```

(*End definition for* `\iow_stream_alloc:N` *and* `\ior_stream_alloc:N`*. These functions are documented on page* **??**.)

`\ior_close:N`
`\ior_close:c`
`\iow_close:N`
`\iow_close:c`

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \cs_if_exist:NT #1
      {
        \int_compare:nNnF #1 = \c_minus_one
          {
            \int_compare:nNnF #1 = \c_sixteen
              { \tex_closein:D #1 }
            \prop_gdel:NV \g_ior_streams_prop #1
            \cs_gset_eq:NN #1 \c_term_ior
          }
      }
  }
\cs_new_protected:Npn \iow_close:N #1
  {
    \cs_if_exist:NT #1
      {
        \int_compare:nNnF #1 = \c_minus_one
          {
            \int_compare:nNnF #1 = \c_sixteen
              { \tex_closein:D #1 }
            \prop_gdel:NV \g_iow_streams_prop #1
            \cs_gset_eq:NN #1 \c_term_iow
          }
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*End definition for* `\ior_close:N` *and others. These functions are documented on page* **??**.)

`\ior_list_streams:`

Show the property lists, but with some "pretty printing". See the l3msg module. If there are no open read streams, issue the message `show-no-stream`, and show an empty token list. If there are open read streams, format them with `\msg_aux_show_unbraced:nn`, and with the message `show-open-streams`.

```
\cs_new_protected_nopar:Npn \ior_list_streams:
  { \ior_list_streams_aux:Nn \g_ior_streams_prop { ior } }
\cs_new_protected_nopar:Npn \iow_list_streams:
```

```
  { \ior_list_streams_aux:Nn \g_iow_streams_prop { iow } }
\cs_new_protected:Npn \ior_list_streams_aux:Nn #1#2
  {
    \msg_aux_use:nn { LaTeX / #2 }
      { \prop_if_empty:NTF #1 { show-no-stream } { show-open-streams } }
    \msg_aux_show:x
      { \prop_map_function:NN #1 \msg_aux_show_unbraced:nn }
  }
```

(*End definition for* `\ior_list_streams:`. *This function is documented on page* **??**.)

Text for the error messages.

```
\msg_kernel_new:nnnn { iow } { streams-exhausted }
  { Output~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
    All~16 are currently~in~use,~and~something~wanted~to~open
    another~one.
  }
\msg_kernel_new:nnnn { ior } { streams-exhausted }
  { Input~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
    All~16 are currently~in~use,~and~something~wanted~to~open
    another~one.
  }
```

201.4 Deferred writing

`\iow_shipout_x:Nn`
`\iow_shipout_x:Nx`

First the easy part, this is the primitive.

```
\cs_new_eq:NN \iow_shipout_x:Nn \tex_write:D
\cs_generate_variant:Nn \iow_shipout_x:Nn { Nx }
```

(*End definition for* `\iow_shipout_x:Nn` *and* `\iow_shipout_x:Nx`*. These functions are documented on page* **??**.)

`\iow_shipout:Nn`
`\iow_shipout:Nx`

With ε-TEX available deferred writing is easy.

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \iow_shipout_x:Nn #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { Nx }
```

(*End definition for* `\iow_shipout:Nn` *and* `\iow_shipout:Nx`*. These functions are documented on page* **??**.)

201.5 Immediate writing

`\iow_now:Nx`

An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```
\cs_new_protected_nopar:Npn \iow_now:Nx { \tex_immediate:D \iow_shipout_x:Nn }
```

(*End definition for* `\iow_now:Nx`. *This function is documented on page* **??**.)

\iow_now:Nn This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error.

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  { \iow_now:Nx #1 { \exp_not:n {#2} } }
```

(*End definition for* `\iow_now:Nn`*. This function is documented on page 159.*)

\iow_log:n
\iow_log:x
\iow_term:n
\iow_term:x

Writing to the log and the terminal directly are relatively easy.

```
\cs_set_protected_nopar:Npn \iow_log:x  { \iow_now:Nx \c_log_iow  }
\cs_new_protected_nopar:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
\cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
```

(*End definition for* `\iow_log:n` *and* `\iow_log:x`*. These functions are documented on page* **??**.)

201.6 Special characters for writing

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

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

(*End definition for* `\iow_newline:`*. This function is documented on page* **??**.)

\iow_char:N Function to write any escaped char to an output stream.

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End definition for* `\iow_char:N`*. This function is documented on page 160.*)

201.7 Hard-wrapping lines based on length

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

\l_iow_line_length_int This is the "raw" length of a line which can be written to a file. The standard value is the line length typically used by TEXLive and MikTEX.

```
\int_new:N  \l_iow_line_length_int
\int_set:Nn \l_iow_line_length_int { 78 }
```

(*End definition for* `\l_iow_line_length_int`*. This function is documented on page 161.*)

\l_iow_target_length_int This stores the target line length: the full length minus any part for a leader at the start of each line.

```
\int_new:N \l_iow_target_length_int
```

(*End definition for* `\l_iow_target_length_int`*.*)

\l_iow_current_line_int
\l_iow_current_word_int
\l_iow_current_indentation_int

These store the number of characters in the line and word currently being constructed, and the current indentation, respectively.

```
\int_new:N \l_iow_current_line_int
\int_new:N \l_iow_current_word_int
\int_new:N \l_iow_current_indentation_int
```

(*End definition for* `\l_iow_current_line_int`*,* `\l_iow_current_word_int`*, and* `\l_iow_current_indentation_int`*.*)

\l_iow_current_line_tl
\l_iow_current_word_tl
\l_iow_current_indentation_tl

These hold the current line of text and current word, and a number of spaces for indentation, respectively.

```
\tl_new:N \l_iow_current_line_tl
\tl_new:N \l_iow_current_word_tl
\tl_new:N \l_iow_current_indentation_tl
```

(*End definition for* `\l_iow_current_line_tl` *,* `\l_iow_current_word_tl` *, and* `\l_iow_current_indentation_tl` *.*)

\l_iow_wrap_tl

Used for the expansion step before detokenizing.

```
\tl_new:N \l_iow_wrap_tl
```

(*End definition for* `\l_iow_wrap_tl`*.*)

\l_iow_wrapped_tl

The output from wrapping text: fully expanded and with lines which are not overly long.

```
\tl_new:N \l_iow_wrapped_tl
```

(*End definition for* `\l_iow_wrapped_tl`*.*)

\l_iow_line_start_bool

Boolean to avoid adding a space at the beginning of forced newlines.

```
\bool_new:N \l_iow_line_start_bool
```

(*End definition for* `\l_iow_line_start_bool`*.*)

\c_catcode_other_space_tl

Lowercase a character with category code 12 to produce an "other" space. We can do everything within the group, because `\tl_const:Nn` defines its argument globally.

```
\group_begin:
  \char_set_catcode_other:N \*
  \char_set_lccode:nn {`\*} {`\ }
  \tl_to_lowercase:n { \tl_const:Nn \c_catcode_other_space_tl { * } }
\group_end:
```

(*End definition for* `\c_catcode_other_space_tl`*.*)

\c_iow_wrap_marker_tl
\c_iow_wrap_end_marker_tl
\c_iow_wrap_newline_marker_tl
\c_iow_wrap_indent_marker_tl
\c_iow_wrap_unindent_marker_tl
\iow_wrap_new_marker:n

Every special action of the wrapping code is preceeded by the same recognizable string, `\c_iow_wrap_marker_tl`. Upon seeing that "word", the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c_iow_wrap_marker_tl` look nicer. Note that `\iow_wrap_new_marker:n` does not survive the group, but all constants are defined globally.

```
\group_begin:
  \int_set_eq:NN \tex_escapechar:D \c_minus_one
  \tl_const:Nx \c_iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
  \cs_set:Npn \iow_wrap_new_marker:n #1
    {
      \tl_const:cx { c_iow_wrap_ #1 _marker_tl }
        {
          \c_catcode_other_space_tl
          \c_iow_wrap_marker_tl
          \c_catcode_other_space_tl
          #1
          \c_catcode_other_space_tl
        }
    }
```

```
    \iow_wrap_new_marker:n { end }
    \iow_wrap_new_marker:n { newline }
    \iow_wrap_new_marker:n { indent }
    \iow_wrap_new_marker:n { unindent }
  \group_end:
```

(*End definition for* `\c_iow_wrap_marker_tl`. *This function is documented on page* *161*.)

`\iow_indent:n`
`\iow_indent_expandable:n`

We give a dummy (protected) definition to `\iow_indent:n` when outside messages. Within wrapped message, it places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. Note that there will be no forced line-break, so the indentation only changes when the next line is started.

```
\cs_new_protected:Npn \iow_indent:n #1 { }
\cs_new:Npx \iow_indent_expandable:n #1
  {
    \c_iow_wrap_indent_marker_tl
    #1
    \c_iow_wrap_unindent_marker_tl
  }
```

(*End definition for* `\iow_indent:n`. *This function is documented on page* *161*.)

`\iow_wrap:xnnnN`

The main wrapping function works as follows. The target number of characters in a line is calculated, before fully-expanding the input such that `\\` and `\␣` are converted into the appropriate values. There is then a loop over each word in the input, which will do the actual wrapping. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The argument `#4` is available for additional set up steps for the output. The definition of `\\` and `\␣` use an "other" space rather than a normal space, because the latter might be absorbed by TeX to end a number or other `f`-type expansions. The `\tl_to_str:N` step converts the "other" space back to a normal space.

```
\cs_new_protected:Npn \iow_wrap:xnnnN #1#2#3#4#5
  {
    \group_begin:
      \int_set:Nn \l_iow_target_length_int { \l_iow_line_length_int - ( #3 ) }
      \int_zero:N \l_iow_current_indentation_int
      \tl_clear:N \l_iow_current_indentation_tl
      \int_zero:N \l_iow_current_line_int
      \tl_clear:N \l_iow_current_line_tl
      \tl_clear:N \l_iow_wrap_tl
      \bool_set_true:N \l_iow_line_start_bool
      \int_set_eq:NN \tex_escapechar:D \c_minus_one
      \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
      \cs_set_nopar:Npx \# { \token_to_str:N \# }
      \cs_set_nopar:Npx \} { \token_to_str:N \} }
      \cs_set_nopar:Npx \% { \token_to_str:N \% }
      \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
      \int_set:Nn \tex_escapechar:D { 92 }
      \cs_set_eq:NN \\ \c_iow_wrap_newline_marker_tl
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_indent:n \iow_indent_expandable:n
```

```
      #4
⟨*initex⟩
      \tl_set:Nx \l_iow_wrap_tl {#1}
⟨/initex⟩
⟨*package⟩
      \protected@edef \l_iow_wrap_tl {#1}
⟨/package⟩
      \cs_set:Npn \\ { \iow_newline: #2 }
      \use:x
        {
          \iow_wrap_loop:w
          \tl_to_str:N \l_iow_wrap_tl
          \tl_to_str:N \c_iow_wrap_end_marker_tl
          \c_space_tl \c_space_tl
          \exp_not:N \q_stop
        }
    \exp_args:NNo \group_end:
    #5 \l_iow_wrapped_tl
  }
```

(*End definition for* `\iow_wrap:xnnnN`*. This function is documented on page 161.*)

`\iow_wrap_loop:w`

The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```
\cs_new_protected:Npn \iow_wrap_loop:w #1 ~ %
  {
    \tl_set:Nn \l_iow_current_word_tl {#1}
    \tl_if_eq:NNTF \l_iow_current_word_tl \c_iow_wrap_marker_tl
      { \iow_wrap_special:w }
      { \iow_wrap_word: }
  }
```

(*End definition for* `\iow_wrap_loop:w`*.*)

`\iow_wrap_word:`
`\iow_wrap_word_fits:`
`\iow_wrap_word_newline:`

For a normal word, update the line length, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line. Otherwise, the current line is added to the result, with the run-on text. The current word (and its length) are then put in the new line.

```
\cs_new_protected_nopar:Npn \iow_wrap_word:
  {
    \int_set:Nn \l_iow_current_word_int
      { \str_length_skip_spaces:N \l_iow_current_word_tl }
    \int_add:Nn \l_iow_current_line_int { \l_iow_current_word_int }
    \int_compare:nNnTF \l_iow_current_line_int < \l_iow_target_length_int
      { \iow_wrap_word_fits: }
      { \iow_wrap_word_newline: }
    \iow_wrap_loop:w
  }
\cs_new_protected_nopar:Npn \iow_wrap_word_fits:
  {
```

```
    \bool_if:NTF \l_iow_line_start_bool
      {
        \bool_set_false:N \l_iow_line_start_bool
        \tl_put_right:Nx \l_iow_current_line_tl
          { \l_iow_current_indentation_tl \l_iow_current_word_tl }
        \int_add:Nn \l_iow_current_line_int
          { \l_iow_current_indentation_int }
      }
      {
        \tl_put_right:Nx \l_iow_current_line_tl
          { ~ \l_iow_current_word_tl }
        \int_incr:N \l_iow_current_line_int
      }
  }
\cs_new_protected_nopar:Npn \iow_wrap_word_newline:
  {
    \tl_put_right:Nx \l_iow_wrapped_tl
      { \l_iow_current_line_tl \\ }
    \int_set:Nn \l_iow_current_line_int
      {
        \l_iow_current_word_int
        + \l_iow_current_indentation_int
      }
    \tl_set:Nx \l_iow_current_line_tl
      { \l_iow_current_indentation_tl \l_iow_current_word_tl }
  }
```

(*End definition for* `\iow_wrap_word:`. *This function is documented on page 161.*)

`\iow_wrap_special:w`
`\iow_wrap_newline:w`
`\iow_wrap_indent:w`
`\iow_wrap_unindent:w`
`\iow_wrap_end:w`

When the "special" marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker. Forced newlines are almost identical to those caused by overflow, except that here the word is empty. To indent more, add four spaces to the start of the indentation token list. To reduce indentation, rebuild the indentation token list using `\prg_replicate:nn`. At the end, we simply save the last line (without the run-on text), and prevent the loop.

```
\cs_new_protected:Npn \iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
  {
    \use:c { iow_wrap_#1: }
    \str_if_eq:xxTF { #2~#3 } { ~ \c_iow_wrap_marker_tl }
      { \iow_wrap_special:w }
      { \iow_wrap_loop:w #2 ~ #3 ~ }
  }
\cs_new_protected_nopar:Npn \iow_wrap_newline:
  {
    \tl_put_right:Nx \l_iow_wrapped_tl
      { \l_iow_current_line_tl \\ }
    \int_zero:N \l_iow_current_line_int
    \tl_clear:N \l_iow_current_line_tl
    \bool_set_true:N \l_iow_line_start_bool
```

```
  }
\cs_new_protected_nopar:Npx \iow_wrap_indent:
  {
    \int_add:Nn \l_iow_current_indentation_int \c_four
    \tl_put_right:Nx \exp_not:N \l_iow_current_indentation_tl
      { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
  }
\cs_new_protected_nopar:Npn \iow_wrap_unindent:
  {
    \int_sub:Nn \l_iow_current_indentation_int \c_four
    \tl_set:Nx \l_iow_current_indentation_tl
      { \prg_replicate:nn \l_iow_current_indentation_int { ~ } }
  }
\cs_new_protected_nopar:Npn \iow_wrap_end:
  {
    \tl_put_right:Nx \l_iow_wrapped_tl
      { \l_iow_current_line_tl }
    \use_none_delimit_by_q_stop:w
  }
```

(*End definition for* `\iow_wrap_special:w`. *This function is documented on page 161.*)

`\str_length_skip_spaces:N`
`\str_length_skip_spaces:n`
`\str_length_loop:NNNNNNNNN`

The wrapping code requires to measure the number of character in each word. This could be done with `\tl_length:n`, but it is ten times faster (literally) to use the code below.

```
\cs_new_nopar:Npn \str_length_skip_spaces:N
  { \exp_args:No \str_length_skip_spaces:n }
\cs_new:Npn \str_length_skip_spaces:n #1
  {
    \int_value:w \int_eval:w
      \exp_after:wN \str_length_loop:NNNNNNNNN \tl_to_str:n {#1}
        { X8 } { X7 } { X6 } { X5 } { X4 } { X3 } { X2 } { X1 } { X0 } \q_stop
    \int_eval_end:
  }
\cs_new:Npn \str_length_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_catcode:w X #9
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \else:
      9 +
      \exp_after:wN \str_length_loop:NNNNNNNNN
    \fi:
  }
```

(*End definition for* `\str_length_skip_spaces:N`. *This function is documented on page 161.*)

201.8 Reading input

`\if_eof:w`

The primitive conditional

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*End definition for* `\if_eof:w`. *This function is documented on page 163.*)

\ior_if_eof_p:N
\ior_if_eof:NTF

To test if some particular input stream is exhausted the following conditional is provided.

```
\prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \if_int_compare:w #1 = \c_sixteen
          \prg_return_true:
        \else:
          \if_eof:w #1
            \prg_return_true:
          \else:
            \prg_return_false:
          \fi:
        \fi:
      }
      { \prg_return_true: }
  }
```

(*End definition for* `\ior_if_eof:N`. *These functions are documented on page 159.*)

\ior_to:NN
\ior_gto:NN

And here we read from files.

```
\cs_new_protected:Npn \ior_to:NN #1#2
  { \tex_read:D #1 to #2 }
\cs_new_protected:Npn \ior_gto:NN #1#2
  { \tex_global:D \tex_read:D #1 to #2 }
```

(*End definition for* `\ior_to:NN` *and* `\ior_gto:NN`. *These functions are documented on page 159.*)

\ior_str_to:NN
\ior_str_gto:NN

Reading as strings is also a primitive wrapper.

```
\cs_new_protected:Npn \ior_str_to:NN #1#2
  { \etex_readline:D #1 to #2 }
\cs_new_protected:Npn \ior_str_gto:NN #1#2
  { \tex_global:D \etex_readline:D #1 to #2 }
```

(*End definition for* `\ior_str_to:NN` *and* `\ior_str_gto:NN`. *These functions are documented on page 159.*)

201.9 Experimental functions

\ior_map_inline:Nn
\ior_str_map_inline:Nn
\ior_str_map_inline_aux:NNn
\ior_str_map_inline_aux:NNNn
\ior_str_map_inline_loop:NNN
\l_ior_internal_tl

Mapping to an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N`. This mapping cannot be nested as the stream has only one "current line".

```
\cs_new_protected_nopar:Npn \ior_map_inline:Nn
  { \ior_map_inline_aux:NNn \ior_to:NN }
\cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
  { \ior_map_inline_aux:NNn \ior_str_to:NN }
\cs_new_protected_nopar:Npn \ior_map_inline_aux:NNn
  {
    \exp_args:Nc \ior_map_inline_aux:NNNn
      { ior_map_ \int_use:N \g_prg_map_int :n }
```

```
  }
\cs_new_protected:Npn \ior_map_inline_aux:NNNn #1#2#3#4
  {
    \cs_set:Npn #1 ##1 {#4}
    \int_gincr:N \g_prg_map_int
    \ior_if_eof:NF #3 { \ior_map_inline_loop:NNN #1#2#3 }
    \prg_break_point:n { \int_gdecr:N \g_prg_map_int }
  }
\cs_new_protected:Npn \ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l_ior_internal_tl
    \ior_if_eof:NF #3
      {
        \exp_args:No #1 \l_ior_internal_tl
        \ior_map_inline_loop:NNN #1#2#3
      }
  }
\tl_new:N  \l_ior_internal_tl
```

(*End definition for* `\ior_map_inline:Nn` *and* `\ior_str_map_inline:Nn`. *These functions are documented on page* **??**.)

201.10 Messages

```
\msg_kernel_new:nnnn { file } { file-not-found }
  { File~'#1'~not~found. }
  {
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\msg_kernel_new:nnnn { file } { space-in-file-name }
  { Space~in~file~name~'#1'. }
  {
    Spaces~are~not~permitted~in~files~loaded~by~LaTeX: \\
    Further~errors~may~follow!
  }
```

201.11 Deprecated functions

Deprecated on 2012-02-10, for removal by 2012-05-31.

`\iow_now_when_avail:Nn`
`\iow_now_when_avail:Nx`

For writing only if the stream requested is open at all.

```
\cs_new_protected:Npn \iow_now_when_avail:Nn #1
  { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 } }
\cs_new_protected:Npn \iow_now_when_avail:Nx #1
  { \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 } }
```

(*End definition for* `\iow_now_when_avail:Nn` *and* `\iow_now_when_avail:Nx`. *These functions are documented on page* **??**.)

Deprecated on 2011-05-27, for removal by 2011-08-31.

\iow_now_buffer_safe:Nn
\iow_now_buffer_safe:Nx

This is much more easily done using the wrapping system: there is an expansion there, so a bit of a hack is needed.

```
⟨*deprecated⟩
\cs_new_protected:Npn \iow_now_buffer_safe:Nn #1#2
  { \iow_wrap:xnnnN { \exp_not:n {#2} } { } \c_zero { } \iow_now:Nn #1 }
\cs_new_protected:Npn \iow_now_buffer_safe:Nx #1#2
  { \iow_wrap:xnnnN {#2} { } \c_zero { } \iow_now:Nn #1 }
⟨/deprecated⟩
```

(*End definition for* \iow_now_buffer_safe:Nn *and* \iow_now_buffer_safe:Nx*. These functions are documented on page* **??**.)

\ior_open_streams:
\iow_open_streams:

Slightly misleading names.

```
⟨*deprecated⟩
\cs_new_eq:NN \ior_open_streams: \ior_list_streams:
\cs_new_eq:NN \iow_open_streams: \iow_list_streams:
⟨/deprecated⟩
```

(*End definition for* \ior_open_streams:*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

202 l3fp Implementation

The following test files are used for this code: m3fp003.lvt.

```
⟨*initex | package⟩
```

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

202.1 Constants

\c_forty_four
\c_one_million
\c_one_hundred_million
\c_five_hundred_million
\c_one_thousand_million

There is some speed to gain by moving numbers into fixed positions.

```
\int_const:Nn \c_forty_four { 44 }
\int_const:Nn \c_one_million { 1 000 000 }
\int_const:Nn \c_one_hundred_million { 100 000 000 }
\int_const:Nn \c_five_hundred_million { 500 000 000 }
\int_const:Nn \c_one_thousand_million { 1 000 000 000 }
```

(*End definition for* \c_forty_four*. This function is documented on page* **??**.)

\c_fp_pi_by_four_decimal_int
\c_fp_pi_by_four_extended_int
\c_fp_pi_decimal_int
\c_fp_pi_extended_int
\c_fp_two_pi_decimal_int
\c_fp_two_pi_extended_int

Parts of π for trigonometric range reduction, implemented as `int` variables for speed.

```
\int_new:N  \c_fp_pi_by_four_decimal_int
\int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
\int_new:N  \c_fp_pi_by_four_extended_int
\int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
\int_new:N  \c_fp_pi_decimal_int
\int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
```

```
\int_new:N  \c_fp_pi_extended_int
\int_set:Nn \c_fp_pi_extended_int { 589 793 238 }
\int_new:N  \c_fp_two_pi_decimal_int
\int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
\int_new:N  \c_fp_two_pi_extended_int
\int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }
```

(*End definition for* `\c_fp_pi_by_four_decimal_int`. *This function is documented on page* **??**.)

`\c_e_fp` The value e as a "machine number".

```
\tl_const:Nn \c_e_fp { + 2.718281828 e 0 }
```

(*End definition for* `\c_e_fp`. *This variable is documented on page 170.*)

`\c_one_fp` The constant value 1: used for fast comparisons.

```
\tl_const:Nn \c_one_fp { + 1.000000000 e 0 }
```

(*End definition for* `\c_one_fp`. *This variable is documented on page 170.*)

`\c_pi_fp` The value π as a "machine number".

```
\tl_const:Nn \c_pi_fp { + 3.141592654 e 0 }
```

(*End definition for* `\c_pi_fp`. *This variable is documented on page 170.*)

`\c_undefined_fp` A marker for undefined values.

```
\tl_const:Nn \c_undefined_fp { X 0.000000000 e 0 }
```

(*End definition for* `\c_undefined_fp`. *This variable is documented on page 170.*)

`\c_zero_fp` The constant zero value.

```
\tl_const:Nn \c_zero_fp { + 0.000000000 e 0 }
```

(*End definition for* `\c_zero_fp`. *This variable is documented on page 170.*)

202.2 Variables

`\l_fp_arg_tl` A token list to store the formalised representation of the input for transcendental functions.

```
\tl_new:N \l_fp_arg_tl
```

(*End definition for* `\l_fp_arg_tl`. *This variable is documented on page* **??**.)

`\l_fp_count_int` A counter for things like the number of divisions possible.

```
\int_new:N \l_fp_count_int
```

(*End definition for* `\l_fp_count_int`. *This variable is documented on page* **??**.)

`\l_fp_div_offset_int` When carrying out division, an offset is used for the results to get the decimal part correct.

```
\int_new:N \l_fp_div_offset_int
```

(*End definition for* `\l_fp_div_offset_int`. *This variable is documented on page* **??**.)

\l_fp_exp_integer_int
\l_fp_exp_decimal_int
\l_fp_exp_extended_int
\l_fp_exp_exponent_int

Used for the calculation of exponent values.

```
\int_new:N \l_fp_exp_integer_int
\int_new:N \l_fp_exp_decimal_int
\int_new:N \l_fp_exp_extended_int
\int_new:N \l_fp_exp_exponent_int
```

(*End definition for* `\l_fp_exp_integer_int`*. This function is documented on page* **??**.)

\l_fp_input_a_sign_int
\l_fp_input_a_integer_int
\l_fp_input_a_decimal_int
\l_fp_input_a_exponent_int
\l_fp_input_b_sign_int
\l_fp_input_b_integer_int
\l_fp_input_b_decimal_int
\l_fp_input_b_exponent_int

Storage for the input: two storage areas as there are at most two inputs.

```
\int_new:N \l_fp_input_a_sign_int
\int_new:N \l_fp_input_a_integer_int
\int_new:N \l_fp_input_a_decimal_int
\int_new:N \l_fp_input_a_exponent_int
\int_new:N \l_fp_input_b_sign_int
\int_new:N \l_fp_input_b_integer_int
\int_new:N \l_fp_input_b_decimal_int
\int_new:N \l_fp_input_b_exponent_int
```

(*End definition for* `\l_fp_input_a_sign_int`*. This function is documented on page* **??**.)

\l_fp_input_a_extended_int
\l_fp_input_b_extended_int

For internal use, “extended” floating point numbers are needed.

```
\int_new:N \l_fp_input_a_extended_int
\int_new:N \l_fp_input_b_extended_int
```

(*End definition for* `\l_fp_input_a_extended_int`*. This function is documented on page* **??**.)

\l_fp_mul_a_i_int
\l_fp_mul_a_ii_int
\l_fp_mul_a_iii_int
\l_fp_mul_a_iv_int
\l_fp_mul_a_v_int
\l_fp_mul_a_vi_int
\l_fp_mul_b_i_int
\l_fp_mul_b_ii_int
\l_fp_mul_b_iii_int
\l_fp_mul_b_iv_int
\l_fp_mul_b_v_int
\l_fp_mul_b_vi_int

Multiplication requires that the decimal part is split into parts so that there are no overflows.

```
\int_new:N \l_fp_mul_a_i_int
\int_new:N \l_fp_mul_a_ii_int
\int_new:N \l_fp_mul_a_iii_int
\int_new:N \l_fp_mul_a_iv_int
\int_new:N \l_fp_mul_a_v_int
\int_new:N \l_fp_mul_a_vi_int
\int_new:N \l_fp_mul_b_i_int
\int_new:N \l_fp_mul_b_ii_int
\int_new:N \l_fp_mul_b_iii_int
\int_new:N \l_fp_mul_b_iv_int
\int_new:N \l_fp_mul_b_v_int
\int_new:N \l_fp_mul_b_vi_int
```

(*End definition for* `\l_fp_mul_a_i_int`*. This function is documented on page* **??**.)

\l_fp_mul_output_int
\l_fp_mul_output_tl

Space for multiplication results.

```
\int_new:N \l_fp_mul_output_int
\tl_new:N  \l_fp_mul_output_tl
```

(*End definition for* `\l_fp_mul_output_int`*. This function is documented on page* **??**.)

\l_fp_output_sign_int
\l_fp_output_integer_int
\l_fp_output_decimal_int
\l_fp_output_exponent_int

Output is stored in the same way as input.

```
\int_new:N \l_fp_output_sign_int
\int_new:N \l_fp_output_integer_int
\int_new:N \l_fp_output_decimal_int
\int_new:N \l_fp_output_exponent_int
```

(*End definition for* `\l_fp_output_sign_int`*. This function is documented on page* **??***.*)

`\l_fp_output_extended_int` Again, for calculations an extended part.

```
\int_new:N \l_fp_output_extended_int
```

(*End definition for* `\l_fp_output_extended_int`*. This variable is documented on page* **??***.*)

`\l_fp_round_carry_bool` To indicate that a digit needs to be carried forward.

```
\bool_new:N \l_fp_round_carry_bool
```

(*End definition for* `\l_fp_round_carry_bool`*. This variable is documented on page* **??***.*)

`\l_fp_round_decimal_tl` A temporary store when rounding, to build up the decimal part without needing to do any maths.

```
\tl_new:N \l_fp_round_decimal_tl
```

(*End definition for* `\l_fp_round_decimal_tl`*. This variable is documented on page* **??***.*)

`\l_fp_round_position_int`
`\l_fp_round_target_int` Used to check the position for rounding.

```
\int_new:N \l_fp_round_position_int
\int_new:N \l_fp_round_target_int
```

(*End definition for* `\l_fp_round_position_int`*. This function is documented on page* **??***.*)

`\l_fp_sign_tl` There are places where the sign needs to be set up "early", so that the registers can be re-used.

```
\tl_new:N \l_fp_sign_tl
```

(*End definition for* `\l_fp_sign_tl`*. This variable is documented on page* **??***.*)

`\l_fp_split_sign_int` When splitting the input it is fastest to use a fixed name for the sign part, and to transfer it after the split is complete.

```
\int_new:N \l_fp_split_sign_int
```

(*End definition for* `\l_fp_split_sign_int`*. This variable is documented on page* **??***.*)

`\l_fp_internal_int` A scratch `int`: used only where the value is not carried forward.

```
\int_new:N \l_fp_internal_int
```

(*End definition for* `\l_fp_internal_int`*. This variable is documented on page* **??***.*)

`\l_fp_internal_tl` A scratch token list variable for expanding material.

```
\tl_new:N \l_fp_internal_tl
```

(*End definition for* `\l_fp_internal_tl`*. This variable is documented on page* **??***.*)

`\l_fp_trig_octant_int` To track which octant the trigonometric input is in.

```
\int_new:N \l_fp_trig_octant_int
```

(*End definition for* `\l_fp_trig_octant_int`*. This variable is documented on page* **??***.*)

`\l_fp_trig_sign_int`
`\l_fp_trig_decimal_int`
`\l_fp_trig_extended_int` Used for the calculation of trigonometric values.

```
\int_new:N \l_fp_trig_sign_int
\int_new:N \l_fp_trig_decimal_int
\int_new:N \l_fp_trig_extended_int
```

(*End definition for* `\l_fp_trig_sign_int`*. This function is documented on page* **??***.*)

202.3 Parsing numbers

\fp_read:N
\fp_read_aux:w

Reading a stored value is made easier as the format is designed to match the delimited function. This is always used to read the first value (register `a`).

```
\cs_new_protected:Npn \fp_read:N #1
  { \exp_after:wN \fp_read_aux:w #1 \q_stop }
\cs_new_protected:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop
  {
    \if:w #1 -
      \l_fp_input_a_sign_int \c_minus_one
    \else:
      \l_fp_input_a_sign_int \c_one
    \fi:
    \l_fp_input_a_integer_int  #2 \scan_stop:
    \l_fp_input_a_decimal_int  #3 \scan_stop:
    \l_fp_input_a_exponent_int #4 \scan_stop:
  }
```

(*End definition for* `\fp_read:N`. *This function is documented on page* **??**.)

\fp_split:Nn
\fp_split_sign:
\fp_split_exponent:
\fp_split_aux_i:w
\fp_split_aux_ii:w
\fp_split_aux_iii:w
\fp_split_decimal:w
\fp_split_decimal_aux:w

The aim here is to use as much of TeX's mechanism as possible to pick up the numerical input without any mistakes. In particular, negative numbers have to be filtered out first in case the integer part is 0 (in which case TeX would drop the `-` sign). That process has to be done in a loop for cases where the sign is repeated. Finding an exponent is relatively easy, after which the next phase is to find the integer part, which will terminate with a `.`, and trigger the decimal-finding code. The later will allow the decimal to be too long, truncating the result.

```
\cs_new_protected:Npn \fp_split:Nn #1#2
  {
    \tl_set:Nx \l_fp_internal_tl {#2}
    \tl_set_rescan:Nno \l_fp_internal_tl { \char_set_catcode_ignore:n { 32 } }
      { \l_fp_internal_tl }
    \l_fp_split_sign_int \c_one
    \fp_split_sign:
    \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int
    \exp_after:wN \fp_split_exponent:w \l_fp_internal_tl e e \q_stop #1
  }
\cs_new_protected_nopar:Npn \fp_split_sign:
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_after:wN \tl_head:w \l_fp_internal_tl ? \q_stop } { - }
        = \c_zero
      \tl_set:Nx \l_fp_internal_tl
        {
          \exp_after:wN
            \tl_tail:w \l_fp_internal_tl \prg_do_nothing: \q_stop
        }
      \l_fp_split_sign_int -\l_fp_split_sign_int
      \exp_after:wN \fp_split_sign:
    \else:
```

```
      \if_int_compare:w \pdftex_strcmp:D
        { \exp_after:wN \tl_head:w \l_fp_internal_tl ? \q_stop } { + }
          = \c_zero
        \tl_set:Nx \l_fp_internal_tl
          {
            \exp_after:wN
              \tl_tail:w \l_fp_internal_tl \prg_do_nothing: \q_stop
          }
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_split_sign:
       \fi:
    \fi:
  }
\cs_new_protected:Npn \fp_split_exponent:w #1 e #2 e #3 \q_stop #4
  {
    \use:c { l_fp_input_ #4 _exponent_int }
      \int_eval:w 0 #2 \scan_stop:
    \tex_afterassignment:D \fp_split_aux_i:w
    \use:c { l_fp_input_ #4 _integer_int }
      \int_eval:w 0 #1 . . \q_stop #4
  }
\cs_new_protected:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop
  { \fp_split_aux_ii:w #2 000000000 \q_stop }
\cs_new_protected:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9
  { \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9} }
\cs_new_protected:Npn \fp_split_aux_iii:w #1#2 \q_stop
  {
    \l_fp_internal_int 1 #1 \scan_stop:
    \exp_after:wN \fp_split_decimal:w
      \int_use:N \l_fp_internal_int 000000000 \q_stop
  }
\cs_new_protected:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9
  { \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9} }
\cs_new_protected:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4
  {
    \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
    \if_int_compare:w
      \int_eval:w
        \use:c { l_fp_input_ #4 _integer_int } +
        \use:c { l_fp_input_ #4 _decimal_int }
      \scan_stop:
        = \c_zero
      \use:c { l_fp_input_ #4 _sign_int } \c_one
    \fi:
    \if_int_compare:w
      \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
    \else:
      \exp_after:wN \fp_overflow_msg:
    \fi:
  }
```

(*End definition for* `\fp_split:Nn`*. This function is documented on page* **??**.)

\fp_standardise:NNNN
\fp_standardise_aux:NNNN
\fp_standardise_aux:
\fp_standardise_aux:w

The idea here is to shift the input into a known exponent range. This is done using TeX tokens where possible, as this is faster than arithmetic.

```
\cs_new_protected:Npn \fp_standardise:NNNN #1#2#3#4
  {
    \if_int_compare:w
      \int_eval:w #2 + #3 = \c_zero
      #1 \c_one
      #4 \c_zero
      \exp_after:wN \use_none:nnnn
    \else:
      \exp_after:wN \fp_standardise_aux:NNNN
    \fi:
    #1#2#3#4
  }
\cs_new_protected:Npn \fp_standardise_aux:NNNN #1#2#3#4
  {
    \cs_set_protected_nopar:Npn \fp_standardise_aux:
      {
        \if_int_compare:w #2 = \c_zero
          \tex_advance:D #3 \c_one_thousand_million
          \exp_after:wN \fp_standardise_aux:w
            \int_use:N #3 \q_stop
           \exp_after:wN \fp_standardise_aux:
         \fi:
      }
    \cs_set_protected:Npn
      \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
      {
        #2 ##2 \scan_stop:
        #3 ##3##4##5##6##7##8##9 0 \scan_stop:
        \tex_advance:D #4 \c_minus_one
      }
    \fp_standardise_aux:
    \cs_set_protected_nopar:Npn \fp_standardise_aux:
      {
        \if_int_compare:w #2 > \c_nine
          \tex_advance:D #2 \c_one_thousand_million
          \exp_after:wN \use_i:nn \exp_after:wN
            \fp_standardise_aux:w \int_use:N #2
           \exp_after:wN \fp_standardise_aux:
         \fi:
      }
    \cs_set_protected:Npn
      \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9
      {
        #2 ##1##2##3##4##5##6##7##8 \scan_stop:
        \tex_advance:D #3 \c_one_thousand_million
        \tex_divide:D #3 \c_ten
        \tl_set:Nx \l_fp_internal_tl
```

```
          {
            ##9
            \exp_after:wN \use_none:n \int_use:N #3
          }
        #3 \l_fp_internal_tl \scan_stop:
        \tex_advance:D #4 \c_one
      }
    \fp_standardise_aux:
    \if_int_compare:w #4 < \c_one_hundred
      \if_int_compare:w #4 > -\c_one_hundred
      \else:
        #1 \c_one
        #2 \c_zero
        #3 \c_zero
        #4 \c_zero
      \fi:
    \else:
      \exp_after:wN \fp_overflow_msg:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_standardise_aux: { }
\cs_new_protected_nopar:Npn \fp_standardise_aux:w { }
```

(*End definition for* `\fp_standardise:NNNN`*. This function is documented on page* **??**.)

202.4 Internal utilities

`\fp_level_input_exponents:`
`\fp_level_input_exponents_a:`
`\fp_level_input_exponents_a:NNNNNNNNN`
`\fp_level_input_exponents_b:`
`\fp_level_input_exponents_b:NNNNNNNNN`

The routines here are similar to those used to standardise the exponent. However, the aim here is different: the two exponents need to end up the same.

```
\cs_new_protected_nopar:Npn \fp_level_input_exponents:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
      \exp_after:wN \fp_level_input_exponents_a:
    \else:
      \exp_after:wN \fp_level_input_exponents_b:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_level_input_exponents_a:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
      \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
        \fp_level_input_exponents_a:NNNNNNNNN
          \int_use:N \l_fp_input_b_integer_int
      \exp_after:wN \fp_level_input_exponents_a:
    \fi:
  }
\cs_new_protected:Npn \fp_level_input_exponents_a:NNNNNNNNN
  #1#2#3#4#5#6#7#8#9
  {
```

```
    \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
    \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
    \tex_divide:D \l_fp_input_b_decimal_int \c_ten
    \tl_set:Nx \l_fp_internal_tl
      {
        #9
        \exp_after:wN \use_none:n
          \int_use:N \l_fp_input_b_decimal_int
      }
    \l_fp_input_b_decimal_int \l_fp_internal_tl \scan_stop:
    \tex_advance:D \l_fp_input_b_exponent_int \c_one
  }
\cs_new_protected_nopar:Npn \fp_level_input_exponents_b:
  {
    \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
      \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
        \fp_level_input_exponents_b:NNNNNNNNN
          \int_use:N \l_fp_input_a_integer_int
      \exp_after:wN \fp_level_input_exponents_b:
    \fi:
  }
\cs_new_protected:Npn \fp_level_input_exponents_b:NNNNNNNNN
  #1#2#3#4#5#6#7#8#9
  {
    \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
    \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \tex_divide:D \l_fp_input_a_decimal_int \c_ten
    \tl_set:Nx \l_fp_internal_tl
      {
        #9
        \exp_after:wN \use_none:n
          \int_use:N \l_fp_input_a_decimal_int
      }
    \l_fp_input_a_decimal_int \l_fp_internal_tl \scan_stop:
    \tex_advance:D \l_fp_input_a_exponent_int \c_one
  }
```

(*End definition for* `\fp_level_input_exponents:`. *This function is documented on page* **??**.)

`\fp_tmp:w` Used for output of results, cutting down on `\exp_after:wN`. This is just a place holder definition.

```
\cs_new_protected:Npn \fp_tmp:w #1#2 { }
```

(*End definition for* `\fp_tmp:w`.)

202.5 Operations for fp variables

The format of `fp` variables is tightly defined, so that they can be read quickly by the internal code. The format is a single sign token, a single number, the decimal point, nine decimal numbers, an `e` and finally the exponent. This final part may vary in length.

When stored, floating points will always be stored with a value in the integer position unless the number is zero.

\fp_new:N
\fp_new:c

Fixed-points always have a value, and of course this has to be initialised globally.

```
\cs_new_protected:Npn \fp_new:N #1
  {
    \tl_new:N #1
    \tl_gset_eq:NN #1 \c_zero_fp
  }
\cs_generate_variant:Nn \fp_new:N { c }
```

(*End definition for* `\fp_new:N` *and* `\fp_new:c`*. These functions are documented on page* **??**.)

\fp_const:Nn
\fp_const:cn

A simple wrapper.

```
\cs_new_protected:Npn \fp_const:Nn #1#2
  {
    \fp_new:N #1
    \fp_gset:Nn #1 {#2}
  }
\cs_generate_variant:Nn \fp_const:Nn { c }
```

(*End definition for* `\fp_const:Nn` *and* `\fp_const:cn`*. These functions are documented on page* **??**.)

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Zeroing fixed-points is pretty obvious.

```
\cs_new_protected:Npn \fp_zero:N #1
  { \tl_set_eq:NN #1 \c_zero_fp }
\cs_new_protected:Npn \fp_gzero:N #1
  { \tl_gset_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_zero:N { c }
\cs_generate_variant:Nn \fp_gzero:N { c }
```

(*End definition for* `\fp_zero:N` *and* `\fp_zero:c`*. These functions are documented on page* **??**.)

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Create a floating point if needed, otherwise clear it.

```
\cs_new_protected:Npn \fp_zero_new:N  #1
  { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\cs_new_protected:Npn \fp_gzero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
\cs_generate_variant:Nn \fp_zero_new:N  { c }
\cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(*End definition for* `\fp_zero_new:N` *and others. These functions are documented on page* **??**.)

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn
\fp_set_aux:NNn

To trap any input errors, a very simple version of the parser is run here. This will pick up any invalid characters at this stage, saving issues later. The splitting approach is the same as the more advanced function later.

```
\cs_new_protected_nopar:Npn \fp_set:Nn  { \fp_set_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gset:Nn { \fp_set_aux:NNn \tl_gset:Nn }
\cs_new_protected:Npn \fp_set_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_split:Nn a {#3}
```

```
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
                -
              \else:
                +
              \fi:
              \int_use:N \l_fp_input_a_integer_int
              .
              \exp_after:wN \use_none:n
                \int_use:N \l_fp_input_a_decimal_int
              e
              \int_use:N \l_fp_input_a_exponent_int
            }
        }
    \fp_tmp:w
  }
\cs_generate_variant:Nn \fp_set:Nn  { c }
\cs_generate_variant:Nn \fp_gset:Nn { c }
```

(*End definition for `\fp_set:Nn` and `\fp_set:cn`. These functions are documented on page* **??**.)

`\fp_set_from_dim:Nn`
`\fp_set_from_dim:cn`
`\fp_gset_from_dim:Nn`
`\fp_gset_from_dim:cn`
`\fp_set_from_dim_aux:NNn`
`\fp_set_from_dim_aux:w`
`\l_fp_internal_dim`
`\l_fp_internal_skip`

Here, dimensions are converted to fixed-points *via* a temporary variable. This ensures that they always convert as points. The code is then essentially the same as for `\fp_set:Nn`, but with the dimension passed so that it will be striped of the `pt` on the way through. The passage through a skip is used to remove any rubber part.

```
\cs_new_protected_nopar:Npn \fp_set_from_dim:Nn
  { \fp_set_from_dim_aux:NNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn
  { \fp_set_from_dim_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \fp_set_from_dim_aux:NNn #1#2#3
  {
    \group_begin:
      \l_fp_internal_skip \etex_glueexpr:D #3 \scan_stop:
      \l_fp_internal_dim \l_fp_internal_skip
      \fp_split:Nn a
        {
          \exp_after:wN \fp_set_from_dim_aux:w
            \dim_use:N \l_fp_internal_dim
        }
      \fp_standardise:NNNN
```

```
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
                -
              \else:
                +
              \fi:
              \int_use:N \l_fp_input_a_integer_int
              .
              \exp_after:wN \use_none:n
                \int_use:N \l_fp_input_a_decimal_int
              e
              \int_use:N \l_fp_input_a_exponent_int
            }
        }
    \fp_tmp:w
  }
\cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w
  {
    \cs_set:Npn \exp_not:N \fp_set_from_dim_aux:w
      ##1 \tl_to_str:n { pt } {##1}
  }
\fp_set_from_dim_aux:w
\cs_generate_variant:Nn \fp_set_from_dim:Nn  { c }
\cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
\dim_new:N \l_fp_internal_dim
\skip_new:N \l_fp_internal_skip
```

(*End definition for* `\fp_set_from_dim:Nn` *and* `\fp_set_from_dim:cn`*. These functions are documented on page* **??**.)

`\fp_set_eq:NN`
`\fp_set_eq:cN`
`\fp_set_eq:Nc`
`\fp_set_eq:cc`
`\fp_gset_eq:NN`
`\fp_gset_eq:cN`
`\fp_gset_eq:Nc`
`\fp_gset_eq:cc`

Pretty simple, really.

```
\cs_new_eq:NN \fp_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \fp_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \fp_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \fp_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \fp_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\fp_set_eq:NN` *and others. These functions are documented on page* **??**.)

\fp_show:N
\fp_show:c

Simple showing of the underlying variable.

```
\cs_new_eq:NN \fp_show:N \tl_show:N
\cs_new_eq:NN \fp_show:c \tl_show:c
```

(*End definition for* \fp_show:N *and* \fp_show:c*. These functions are documented on page* **??**.)

\fp_use:N
\fp_use:c
\fp_use_aux:w
\fp_use_none:w
\fp_use_small:w
\fp_use_large:w
\fp_use_large_aux_i:w
\fp_use_large_aux_1:w
\fp_use_large_aux_2:w
\fp_use_large_aux_3:w
\fp_use_large_aux_4:w
\fp_use_large_aux_5:w
\fp_use_large_aux_6:w
\fp_use_large_aux_7:w
\fp_use_large_aux_8:w
\fp_use_large_aux_i:w
\fp_use_large_aux_ii:w

The idea of the \fp_use:N function to convert the stored value into something suitable for TeX to use as a number in an expandable manner. The first step is to deal with the sign, then work out how big the input is.

```
\cs_new:Npn \fp_use:N #1
  { \exp_after:wN \fp_use_aux:w #1 \q_stop }
\cs_generate_variant:Nn \fp_use:N { c }
\cs_new:Npn \fp_use_aux:w #1#2 e #3 \q_stop
  {
    \if:w #1 -
      -
    \fi:
    \if_int_compare:w #3 > \c_zero
      \exp_after:wN \fp_use_large:w
    \else:
      \if_int_compare:w #3 < \c_zero
        \exp_after:wN \exp_after:wN \exp_after:wN
          \fp_use_small:w
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_use_none:w
      \fi:
    \fi:
    #2 e #3 \q_stop
  }
```

When the exponent is zero, the input is simply returned as output.

```
\cs_new:Npn \fp_use_none:w #1 e #2 \q_stop {#1}
```

For small numbers (less than 1) the correct number of zeros have to be inserted, but the decimal point is easy.

```
\cs_new:Npn \fp_use_small:w #1 . #2 e #3 \q_stop
  {
    0 .
    \prg_replicate:nn { -#3 - 1 } { 0 }
    #1#2
  }
```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```
\cs_new:Npn \fp_use_large:w #1 . #2 e #3 \q_stop
  {
    \if_int_compare:w #3 < \c_ten
      \exp_after:wN \fp_use_large_aux_i:w
    \else:
      \exp_after:wN \fp_use_large_aux_ii:w
    \fi:
```

```
    #1#2 e #3 \q_stop
  }
\cs_new:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop
  {
    #1
    \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
  }
\cs_new:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
\cs_new:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop
  { #1#2 . #3 }
\cs_new:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop
  { #1#2#3 . #4 }
\cs_new:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop
  { #1#2#3#4 . #5 }
\cs_new:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop
  { #1#2#3#4#5 . #6 }
\cs_new:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
  { #1#2#3#4#5#6 . #7 }
\cs_new:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
  { #1#2#3#4#6#7 . #8 }
\cs_new:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop
  { #1#2#3#4#5#6#7#8 . #9 }
\cs_new:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
\cs_new:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop
  {
    #1
    \prg_replicate:nn { #2 - 9 } { 0 }
    .
  }
```

(*End definition for* `\fp_use:N` *and* `\fp_use:c`*. These functions are documented on page* **??**.)

`\fp_if_exist_p:N`
`\fp_if_exist_p:c`
`\fp_if_exist:NTF`
`\fp_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\cs_new_eq:NN \fp_if_exist:NTF \cs_if_exist:NTF
\cs_new_eq:NN \fp_if_exist:NT  \cs_if_exist:NT
\cs_new_eq:NN \fp_if_exist:NF  \cs_if_exist:NF
\cs_new_eq:NN \fp_if_exist_p:N \cs_if_exist_p:N
\cs_new_eq:NN \fp_if_exist:cTF \cs_if_exist:cTF
\cs_new_eq:NN \fp_if_exist:cT  \cs_if_exist:cT
\cs_new_eq:NN \fp_if_exist:cF  \cs_if_exist:cF
\cs_new_eq:NN \fp_if_exist_p:c \cs_if_exist_p:c
```

(*End definition for* `\fp_if_exist:N` *and* `\fp_if_exist:c`*. These functions are documented on page* **??**.)

202.6 Transferring to other types

The `\fp_use:N` function converts a floating point variable to a form that can be used by TeX. Here, the functions are slightly different, as some information may be discarded.

`\fp_to_dim:N`
`\fp_to_dim:c`

A very simple wrapper.

```
\cs_new:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
```

```
\cs_generate_variant:Nn \fp_to_dim:N { c }
```

(*End definition for* `\fp_to_dim:N` *and* `\fp_to_dim:c`*. These functions are documented on page* **??**.)

`\fp_to_int:N`
`\fp_to_int:c`
`\fp_to_int_aux:w`
`\fp_to_int_none:w`
`\fp_to_int_small:w`
`\fp_to_int_large:w`
`\fp_to_int_large_aux_i:w`
`\fp_to_int_large_aux_1:w`
`\fp_to_int_large_aux_2:w`
`\fp_to_int_large_aux_3:w`
`\fp_to_int_large_aux_4:w`
`\fp_to_int_large_aux_5:w`
`\fp_to_int_large_aux_6:w`
`\fp_to_int_large_aux_7:w`
`\fp_to_int_large_aux_8:w`
`\fp_to_int_large_aux_i:w`
`\fp_to_int_large_aux:nnn`
`\fp_to_int_large_aux_ii:w`

Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

```
\cs_new:Npn \fp_to_int:N #1
  { \exp_after:wN \fp_to_int_aux:w #1 \q_stop }
\cs_generate_variant:Nn \fp_to_int:N { c }
\cs_new:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop
  {
    \if:w #1 -
      -
    \fi:
    \if_int_compare:w #3 < \c_zero
      \exp_after:wN \fp_to_int_small:w
    \else:
      \exp_after:wN \fp_to_int_large:w
    \fi:
    #2 e #3 \q_stop
  }
```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```
\cs_new:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop
  {
    \if_int_compare:w #3 > \c_one
    \else:
      \if_int_compare:w #1 < \c_five
        0
      \else:
        1
      \fi:
    \fi:
  }
```

For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```
\cs_new:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop
  {
    \if_int_compare:w #3 < \c_ten
      \exp_after:wN \fp_to_int_large_aux_i:w
    \else:
      \exp_after:wN \fp_to_int_large_aux_ii:w
    \fi:
    #1#2 e #3 \q_stop
  }
\cs_new:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop
  { \use:c { fp_to_int_large_aux_ #3 :w } #2 \q_stop {#1} }
\cs_new:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop
  { \fp_to_int_large_aux:nnn { #2 0 } {#1} }
```

```
\cs_new:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop
  { \fp_to_int_large_aux:nnn { #3 00 } {#1#2} }
\cs_new:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop
  { \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3} }
\cs_new:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop
  { \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4} }
\cs_new:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop
  { \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5} }
\cs_new:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop
  { \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6} }
\cs_new:cpn { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop
  { \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7} }
\cs_new:cpn { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop
  { \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8} }
\cs_new:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
\cs_new:Npn \fp_to_int_large_aux:nnn #1#2#3
  {
    \if_int_compare:w #1 < \c_five_hundred_million
      #3#2
    \else:
      \int_value:w \int_eval:w #3#2 + 1 \int_eval_end:
    \fi:
  }
\cs_new:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop
  {
    #1
    \prg_replicate:nn { #2 - 9 } { 0 }
  }
```

(*End definition for* `\fp_to_int:N` *and* `\fp_to_int:c`*. These functions are documented on page* **??***.*)

`\fp_to_tl:N`
`\fp_to_tl:c`
`\fp_to_tl_aux:w`
`\fp_to_tl_large:w`
`\fp_to_tl_large_aux_i:w`
`\fp_to_tl_large_aux_ii:w`
`\fp_to_tl_large_0:w`
`\fp_to_tl_large_1:w`
`\fp_to_tl_large_2:w`
`\fp_to_tl_large_3:w`
`\fp_to_tl_large_4:w`
`\fp_to_tl_large_5:w`
`\fp_to_tl_large_6:w`
`\fp_to_tl_large_7:w`
`\fp_to_tl_large_8:w`
`\fp_to_tl_large_8_aux:w`
`\fp_to_tl_large_9:w`
`\fp_to_tl_small:w`
`\fp_to_tl_small_one:w`
`\fp_to_tl_small_two:w`
`\fp_to_tl_small_aux:w`
`\fp_to_tl_large_zeros:NNNNNNNNN`
`\fp_to_tl_small_zeros:NNNNNNNNN`
`\fp_use_iix_ix:NNNNNNNNN`
`\fp_use_ix:NNNNNNNNN`
`\fp_use_i_to_vii:NNNNNNNNN`
`\fp_use_i_to_iix:NNNNNNNNN`

Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

```
\cs_new:Npn \fp_to_tl:N #1
  { \exp_after:wN \fp_to_tl_aux:w #1 \q_stop }
\cs_generate_variant:Nn \fp_to_tl:N { c }
\cs_new:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop
  {
    \if:w #1 -
      -
    \fi:
    \if_int_compare:w #3 < \c_zero
      \exp_after:wN \fp_to_tl_small:w
    \else:
      \exp_after:wN \fp_to_tl_large:w
    \fi:
    #2 e #3 \q_stop
  }
```

For "large" numbers (exponent ≥ 0) there are two cases. For very large exponents (≥ 10) life is easy: apart from dropping extra zeros there is no work to do. On the other hand, for intermediate exponent values the decimal needs to be moved, then zeros can be dropped.

```
\cs_new:Npn \fp_to_tl_large:w #1 e #2 \q_stop
  {
    \if_int_compare:w #2 < \c_ten
      \exp_after:wN \fp_to_tl_large_aux_i:w
    \else:
      \exp_after:wN \fp_to_tl_large_aux_ii:w
    \fi:
    #1 e #2 \q_stop
  }
\cs_new:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop
  { \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop }
\cs_new:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop
  {
    #1
    \fp_to_tl_large_zeros:NNNNNNNNN #2
    e #3
  }
\cs_new:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop
  {
    #1
    \fp_to_tl_large_zeros:NNNNNNNNN #2
  }
\cs_new:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop
  {
    #1#2
    \fp_to_tl_large_zeros:NNNNNNNNN #3 0
  }
\cs_new:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop
  {
    #1#2#3
    \fp_to_tl_large_zeros:NNNNNNNNN #4 00
  }
\cs_new:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop
  {
    #1#2#3#4
    \fp_to_tl_large_zeros:NNNNNNNNN #5 000
  }
\cs_new:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop
  {
    #1#2#3#4#5
    \fp_to_tl_large_zeros:NNNNNNNNN #6 0000
  }
\cs_new:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop
  {
    #1#2#3#4#5#6
    \fp_to_tl_large_zeros:NNNNNNNNN #7 00000
  }
\cs_new:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop
```

```
  {
    #1#2#3#4#5#6#7
    \fp_to_tl_large_zeros:NNNNNNNNN #8 000000
  }
\cs_new:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop
  {
    #1#2#3#4#5#6#7#8
    \fp_to_tl_large_zeros:NNNNNNNNN #9 0000000
  }
\cs_new:cpn { fp_to_tl_large_8:w } #1 .
  {
    #1
    \use:c { fp_to_tl_large_8_aux:w }
  }
\cs_new:cpn { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop
  {
    #1#2#3#4#5#6#7#8
    \fp_to_tl_large_zeros:NNNNNNNNN #9 00000000
  }
\cs_new:cpn { fp_to_tl_large_9:w } #1 . #2 \q_stop {#1#2}
```

Dealing with small numbers is a bit more complex as there has to be rounding. This makes life rather awkward, as there need to be a series of tests and calculations, as things cannot be stored in an expandable system.

```
\cs_new:Npn \fp_to_tl_small:w #1 e #2 \q_stop
  {
    \if_int_compare:w #2 = \c_minus_one
      \exp_after:wN \fp_to_tl_small_one:w
    \else:
      \if_int_compare:w #2 = -\c_two
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_two:w
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_to_tl_small_aux:w
      \fi:
    \fi:
    #1 e #2 \q_stop
  }
\cs_new:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop
  {
    \if_int_compare:w \fp_use_ix:NNNNNNNNN #2 > \c_four
      \if_int_compare:w
        \int_eval:w #1 \fp_use_i_to_iix:NNNNNNNNN #2 + 1
          < \c_one_thousand_million
        0.
        \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNNN
          \int_value:w \int_eval:w
              #1 \fp_use_i_to_iix:NNNNNNNNN #2 + 1
            \int_eval_end:
      \else:
        1
```

```
      \fi:
    \else:
      0. #1
      \fp_to_tl_small_zeros:NNNNNNNNN #2
    \fi:
  }
\cs_new:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop
  {
    \if_int_compare:w \fp_use_iix_ix:NNNNNNNNN #2 > \c_forty_four
      \if_int_compare:w
        \int_eval:w #1 \fp_use_i_to_vii:NNNNNNNNN #2 0 + \c_ten
          < \c_one_thousand_million
        0.0
        \exp_after:wN \fp_to_tl_small_zeros:NNNNNNNNN
          \int_value:w \int_eval:w
              #1 \fp_use_i_to_vii:NNNNNNNNN #2 0 + \c_ten
            \int_eval_end:
      \else:
        0.1
      \fi:
    \else:
      0.0
      #1
      \fp_to_tl_small_zeros:NNNNNNNNN #2
    \fi:
  }
\cs_new:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop
  {
    #1
    \fp_to_tl_large_zeros:NNNNNNNNN #2
    e #3
  }
```

Rather than a complex recursion, the tests for finding trailing zeros are written out long-hand. The difference between the two is only the need for a decimal marker.

```
\cs_new:Npn \fp_to_tl_large_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #9 = \c_zero
      \if_int_compare:w #8 = \c_zero
        \if_int_compare:w #7 = \c_zero
          \if_int_compare:w #6 = \c_zero
            \if_int_compare:w #5 = \c_zero
              \if_int_compare:w #4 = \c_zero
                \if_int_compare:w #3 = \c_zero
                  \if_int_compare:w #2 = \c_zero
                    \if_int_compare:w #1 = \c_zero
                    \else:
                      . #1
                    \fi:
                  \else:
```

```
                    . #1#2
                  \fi:
                \else:
                  . #1#2#3
                \fi:
              \else:
                . #1#2#3#4
              \fi:
            \else:
              . #1#2#3#4#5
            \fi:
          \else:
            . #1#2#3#4#5#6
          \fi:
        \else:
          . #1#2#3#4#5#6#7
        \fi:
      \else:
         . #1#2#3#4#5#6#7#8
      \fi:
    \else:
      . #1#2#3#4#5#6#7#8#9
    \fi:
  }
\cs_new:Npn \fp_to_tl_small_zeros:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #9 = \c_zero
      \if_int_compare:w #8 = \c_zero
        \if_int_compare:w #7 = \c_zero
          \if_int_compare:w #6 = \c_zero
            \if_int_compare:w #5 = \c_zero
              \if_int_compare:w #4 = \c_zero
                \if_int_compare:w #3 = \c_zero
                  \if_int_compare:w #2 = \c_zero
                    \if_int_compare:w #1 = \c_zero
                    \else:
                      #1
                    \fi:
                  \else:
                    #1#2
                  \fi:
                \else:
                  #1#2#3
                \fi:
              \else:
                #1#2#3#4
              \fi:
            \else:
              #1#2#3#4#5
            \fi:
```

```
          \else:
            #1#2#3#4#5#6
          \fi:
        \else:
          #1#2#3#4#5#6#7
        \fi:
      \else:
         #1#2#3#4#5#6#7#8
      \fi:
    \else:
      #1#2#3#4#5#6#7#8#9
    \fi:
  }
```

Some quick "return a few" functions.

```
\cs_new:Npn \fp_use_iix_ix:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
\cs_new:Npn \fp_use_ix:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
\cs_new:Npn \fp_use_i_to_vii:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {#1#2#3#4#5#6#7}
\cs_new:Npn \fp_use_i_to_iix:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {#1#2#3#4#5#6#7#8}
```

(*End definition for* `\fp_to_tl:N` *and* `\fp_to_tl:c`*. These functions are documented on page* **??**.)

202.7 Rounding numbers

The results may well need to be rounded. A couple of related functions to do this for a stored value.

`\fp_round_figures:Nn`
`\fp_round_figures:cn`
`\fp_ground_figures:Nn`
`\fp_ground_figures:cn`
`\fp_round_figures_aux:NNn`

Rounding to figures needs only an adjustment to the target by one (as the target is in decimal places).

```
\cs_new_protected_nopar:Npn \fp_round_figures:Nn
  { \fp_round_figures_aux:NNn \tl_set:Nn }
\cs_generate_variant:Nn \fp_round_figures:Nn { c }
\cs_new_protected_nopar:Npn \fp_ground_figures:Nn
  { \fp_round_figures_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_ground_figures:Nn { c }
\cs_new_protected:Npn \fp_round_figures_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \int_set:Nn \l_fp_round_target_int { #3 - 1 }
      \if_int_compare:w \l_fp_round_target_int < \c_ten
        \exp_after:wN \fp_round:
      \fi:
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
```

```
            {
              \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
                -
              \else:
                +
              \fi:
              \int_use:N \l_fp_input_a_integer_int
              .
              \exp_after:wN \use_none:n
                \int_use:N \l_fp_input_a_decimal_int
              e
              \int_use:N \l_fp_input_a_exponent_int
            }
        }
    \fp_tmp:w
  }
```

(*End definition for* `\fp_round_figures:Nn` *and* `\fp_round_figures:cn`*. These functions are documented on page* **??**.)

`\fp_round_places:Nn`
`\fp_round_places:cn`
`\fp_ground_places:Nn`
`\fp_ground_places:cn`
`\fp_round_places_aux:NNn`

Rounding to places needs an adjustment for the exponent value, which will mean that everything should be correct.

```
\cs_new_protected_nopar:Npn \fp_round_places:Nn
  { \fp_round_places_aux:NNn \tl_set:Nn }
\cs_generate_variant:Nn \fp_round_places:Nn { c }
\cs_new_protected_nopar:Npn \fp_ground_places:Nn
  { \fp_round_places_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_ground_places:Nn { c }
\cs_new_protected:Npn \fp_round_places_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \int_set:Nn \l_fp_round_target_int
        { #3 + \l_fp_input_a_exponent_int }
      \if_int_compare:w \l_fp_round_target_int < \c_ten
        \exp_after:wN \fp_round:
      \fi:
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
                -
              \else:
                +
              \fi:
              \int_use:N \l_fp_input_a_integer_int
              .
              \exp_after:wN \use_none:n
```

```
                \int_use:N \l_fp_input_a_decimal_int
              e
              \int_use:N \l_fp_input_a_exponent_int
            }
        }
    \fp_tmp:w
  }
```

(*End definition for* `\fp_round_places:Nn` *and* `\fp_round_places:cn`*. These functions are documented on page* **??**.)

`\fp_round:`
`\fp_round_aux:NNNNNNNNN`
`\fp_round_loop:N`

The rounding approach is the same for decimal places and significant figures. There are always nine decimal digits to round, so the code can be written to account for this. The basic logic is simply to find the rounding, track any carry digit and move along. At the end of the loop there is a possible shuffle if the integer part has become 10.

```
\cs_new_protected_nopar:Npn \fp_round:
  {
    \bool_set_false:N \l_fp_round_carry_bool
    \l_fp_round_position_int \c_eight
    \tl_clear:N \l_fp_round_decimal_tl
    \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \exp_after:wN \use_i:nn \exp_after:wN
      \fp_round_aux:NNNNNNNNN \int_use:N \l_fp_input_a_decimal_int
  }
\cs_new_protected:Npn \fp_round_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \fp_round_loop:N #9#8#7#6#5#4#3#2#1
    \bool_if:NT \l_fp_round_carry_bool
      { \tex_advance:D \l_fp_input_a_integer_int \c_one }
    \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
    \if_int_compare:w \l_fp_input_a_integer_int < \c_ten
    \else:
      \l_fp_input_a_integer_int \c_one
      \tex_divide:D \l_fp_input_a_decimal_int \c_ten
      \tex_advance:D \l_fp_input_a_exponent_int \c_one
    \fi:
  }
\cs_new_protected:Npn \fp_round_loop:N #1
  {
    \if_int_compare:w \l_fp_round_position_int < \l_fp_round_target_int
      \bool_if:NTF \l_fp_round_carry_bool
        { \l_fp_internal_int \int_eval:w #1 + \c_one \scan_stop: }
        { \l_fp_internal_int \int_eval:w #1 \scan_stop: }
      \if_int_compare:w \l_fp_internal_int = \c_ten
        \l_fp_internal_int \c_zero
      \else:
        \bool_set_false:N \l_fp_round_carry_bool
      \fi:
      \tl_set:Nx \l_fp_round_decimal_tl
        { \int_use:N \l_fp_internal_int \l_fp_round_decimal_tl }
```

```
    \else:
      \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
      \if_int_compare:w \l_fp_round_position_int = \l_fp_round_target_int
        \if_int_compare:w #1 > \c_four
          \bool_set_true:N \l_fp_round_carry_bool
        \fi:
      \fi:
    \fi:
    \tex_advance:D \l_fp_round_position_int \c_minus_one
    \if_int_compare:w \l_fp_round_position_int > \c_minus_one
      \exp_after:wN \fp_round_loop:N
    \fi:
  }
```

(*End definition for* `\fp_round:`. *This function is documented on page* **??**.)

202.8 Unary functions

\fp_abs:N
\fp_abs:c
\fp_gabs:N
\fp_gabs:c
\fp_abs_aux:NN

Setting the absolute value is easy: read the value, ignore the sign, return the result.

```
\cs_new_protected_nopar:Npn \fp_abs:N  { \fp_abs_aux:NN \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gabs:N { \fp_abs_aux:NN \tl_gset:Nn }
\cs_generate_variant:Nn \fp_abs:N  { c }
\cs_generate_variant:Nn \fp_gabs:N { c }
\cs_new_protected:Npn \fp_abs_aux:NN #1#2
  {
    \group_begin:
      \fp_read:N #2
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              +
              \int_use:N \l_fp_input_a_integer_int
              .
              \exp_after:wN \use_none:n
                \int_use:N \l_fp_input_a_decimal_int
              e
              \int_use:N \l_fp_input_a_exponent_int
            }
        }
    \fp_tmp:w
  }
```

(*End definition for* `\fp_abs:N` *and* `\fp_abs:c`. *These functions are documented on page* **??**.)

\fp_neg:N
\fp_neg:c
\fp_gneg:N
\fp_gneg:c
\fp_neg:NN

Just a bit more complex: read the input, reverse the sign and output the result.

```
\cs_new_protected_nopar:Npn \fp_neg:N  { \fp_neg_aux:NN \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gneg:N { \fp_neg_aux:NN \tl_gset:Nn }
\cs_generate_variant:Nn \fp_neg:N  { c }
```

```
\cs_generate_variant:Nn \fp_gneg:N { c }
\cs_new_protected:Npn \fp_neg_aux:NN #1#2
  {
    \group_begin:
      \fp_read:N #2
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \tl_set:Nx \l_fp_internal_tl
        {
          \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
            +
          \else:
            -
          \fi:
          \int_use:N \l_fp_input_a_integer_int
          .
          \exp_after:wN \use_none:n
            \int_use:N \l_fp_input_a_decimal_int
          e
          \int_use:N \l_fp_input_a_exponent_int
        }
    \exp_after:wN \group_end: \exp_after:wN
    #1 \exp_after:wN #2 \exp_after:wN { \l_fp_internal_tl }
  }
```

(*End definition for* `\fp_neg:N` *and* `\fp_neg:c`*. These functions are documented on page* **??**.)

202.9 Basic arithmetic

`\fp_add:Nn`
`\fp_add:cn`
`\fp_gadd:Nn`
`\fp_gadd:cn`
`\fp_add_aux:NNn`
`\fp_add_core:`
`\fp_add_sum:`
`\fp_add_difference:`

The various addition functions are simply different ways to call the single master function below. This pattern is repeated for the other arithmetic functions.

```
\cs_new_protected_nopar:Npn \fp_add:Nn  { \fp_add_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gadd:Nn { \fp_add_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_add:Nn   { c }
\cs_generate_variant:Nn \fp_gadd:Nn  { c }
```

Addition takes place using one of two paths. If the signs of the two parts are the same, they are simply combined. On the other hand, if the signs are different the calculation finds this difference.

```
\cs_new_protected:Npn \fp_add_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \fp_split:Nn b {#3}
      \fp_standardise:NNNN
        \l_fp_input_b_sign_int
        \l_fp_input_b_integer_int
        \l_fp_input_b_decimal_int
        \l_fp_input_b_exponent_int
      \fp_add_core:
    \fp_tmp:w #1#2
```

```
  }
\cs_new_protected_nopar:Npn \fp_add_core:
  {
    \fp_level_input_exponents:
    \if_int_compare:w
      \int_eval:w
        \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
        > \c_zero
      \exp_after:wN \fp_add_sum:
    \else:
      \exp_after:wN \fp_add_difference:
    \fi:
    \l_fp_output_exponent_int \l_fp_input_a_exponent_int
    \fp_standardise:NNNN
      \l_fp_output_sign_int
      \l_fp_output_integer_int
      \l_fp_output_decimal_int
      \l_fp_output_exponent_int
    \cs_set_protected:Npx \fp_tmp:w ##1##2
      {
        \group_end:
        ##1 ##2
          {
             \if_int_compare:w \l_fp_output_sign_int < \c_zero
              -
            \else:
              +
            \fi:
            \int_use:N \l_fp_output_integer_int
            .
            \exp_after:wN \use_none:n
              \int_value:w \int_eval:w
                 \l_fp_output_decimal_int + \c_one_thousand_million
            e
            \int_use:N \l_fp_output_exponent_int
          }
      }
  }
```
Finding the sum of two numbers is trivially easy.
```
\cs_new_protected_nopar:Npn \fp_add_sum:
  {
    \l_fp_output_sign_int \l_fp_input_a_sign_int
    \l_fp_output_integer_int
      \int_eval:w
        \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
      \scan_stop:
    \l_fp_output_decimal_int
      \int_eval:w
        \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
```

```
      \scan_stop:
    \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
    \else:
      \tex_advance:D \l_fp_output_integer_int \c_one
      \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
    \fi:
  }
```

When the signs of the two parts of the input are different, the absolute difference is worked out first. There is then a calculation to see which way around everything has worked out, so that the final sign is correct. The difference might also give a zero result with a negative sign, which is reversed as zero is regarded as positive.

```
\cs_new_protected_nopar:Npn \fp_add_difference:
  {
    \l_fp_output_integer_int
      \int_eval:w
        \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
      \scan_stop:
    \l_fp_output_decimal_int
      \int_eval:w
        \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int
      \scan_stop:
    \if_int_compare:w \l_fp_output_decimal_int < \c_zero
      \tex_advance:D \l_fp_output_integer_int \c_minus_one
      \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
    \fi:
    \if_int_compare:w \l_fp_output_integer_int < \c_zero
      \l_fp_output_sign_int \l_fp_input_b_sign_int
      \if_int_compare:w \l_fp_output_decimal_int = \c_zero
        \l_fp_output_integer_int -\l_fp_output_integer_int
      \else:
        \l_fp_output_decimal_int
          \int_eval:w
            \c_one_thousand_million - \l_fp_output_decimal_int
          \scan_stop:
        \l_fp_output_integer_int
           \int_eval:w
             - \l_fp_output_integer_int - \c_one
           \scan_stop:
      \fi:
    \else:
      \l_fp_output_sign_int \l_fp_input_a_sign_int
    \fi:
  }
```

(*End definition for* `\fp_add:Nn` *and* `\fp_add:cn`*. These functions are documented on page* **??**.)

`\fp_sub:Nn`
`\fp_sub:cn`
`\fp_gsub:Nn`
`\fp_gsub:cn`
`\fp_sub_aux:NNn`

Subtraction is essentially the same as addition, but with the sign of the second component reversed. Thus the core of the two function groups is the same, with just a little set up here.

```
\cs_new_protected_nopar:Npn \fp_sub:Nn  { \fp_sub_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gsub:Nn { \fp_sub_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_sub:Nn   { c }
\cs_generate_variant:Nn \fp_gsub:Nn  { c }
\cs_new_protected:Npn \fp_sub_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \fp_split:Nn b {#3}
      \fp_standardise:NNNN
        \l_fp_input_b_sign_int
        \l_fp_input_b_integer_int
        \l_fp_input_b_decimal_int
        \l_fp_input_b_exponent_int
      \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
      \fp_add_core:
    \fp_tmp:w #1#2
  }
```

(*End definition for* `\fp_sub:Nn` *and* `\fp_sub:cn`*. These functions are documented on page* **??**.)

`\fp_mul:Nn`
`\fp_mul:cn`
`\fp_gmul:Nn`
`\fp_gmul:cn`
`\fp_mul_aux:NNn`

The pattern is much the same for multiplication.

```
\cs_new_protected_nopar:Npn \fp_mul:Nn  { \fp_mul_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gmul:Nn { \fp_mul_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_mul:Nn  { c }
\cs_generate_variant:Nn \fp_gmul:Nn { c }
```

`\fp_mul_internal:`
`\fp_mul_split:NNNN`
`\fp_mul_split:w`
`\fp_mul_end_level:`
`\fp_mul_end_level:NNNNNNNNN`

The approach to multiplication is as follows. First, the two numbers are split into blocks of three digits. These are then multiplied together to find products for each group of three output digits. This is al written out in full for speed reasons. Between each block of three digits in the output, there is a carry step. The very lowest digits are not calculated, while

```
\cs_new_protected:Npn \fp_mul_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \fp_split:Nn b {#3}
      \fp_standardise:NNNN
        \l_fp_input_b_sign_int
        \l_fp_input_b_integer_int
        \l_fp_input_b_decimal_int
        \l_fp_input_b_exponent_int
      \fp_mul_internal:
      \l_fp_output_exponent_int
        \int_eval:w
          \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
        \scan_stop:
      \fp_standardise:NNNN
        \l_fp_output_sign_int
        \l_fp_output_integer_int
        \l_fp_output_decimal_int
        \l_fp_output_exponent_int
```

```
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \if_int_compare:w
                \int_eval:w
                  \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
                  < \c_zero
                \if_int_compare:w
                  \int_eval:w
                    \l_fp_output_integer_int + \l_fp_output_decimal_int
                    = \c_zero
                  +
                \else:
                  -
                \fi:
              \else:
                +
             \fi:
              \int_use:N \l_fp_output_integer_int
              .
              \exp_after:wN \use_none:n
                \int_value:w \int_eval:w
                   \l_fp_output_decimal_int + \c_one_thousand_million
              e
              \int_use:N \l_fp_output_exponent_int
            }
        }
    \fp_tmp:w
  }
```

Done separately so that the internal use is a bit easier.

```
\cs_new_protected_nopar:Npn \fp_mul_internal:
  {
    \fp_mul_split:NNNN \l_fp_input_a_decimal_int
      \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
    \fp_mul_split:NNNN \l_fp_input_b_decimal_int
      \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
    \l_fp_mul_output_int \c_zero
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int       \l_fp_mul_b_i_int
    \tex_divide:D \l_fp_mul_output_int \c_one_thousand
    \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int       \l_fp_input_b_integer_int
    \fp_mul_end_level:
```

```
    \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_input_b_integer_int
    \fp_mul_end_level:
    \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_input_b_integer_int
    \fp_mul_end_level:
    \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
    \fp_mul_end_level:
    \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
  }
```

The split works by making a 10 digit number, from which the first digit can then be dropped using a delimited argument. The groups of three digits are then assigned to the various parts of the input: notice that `##9` contains the last two digits of the smallest part of the input.

```
\cs_new_protected:Npn \fp_mul_split:NNNN #1#2#3#4
  {
    \tex_advance:D #1 \c_one_thousand_million
    \cs_set_protected:Npn \fp_mul_split_aux:w
       ##1##2##3##4##5##6##7##8##9 \q_stop {
        #2 ##2##3##4 \scan_stop:
        #3 ##5##6##7 \scan_stop:
        #4 ##8##9    \scan_stop:
      }
    \exp_after:wN \fp_mul_split_aux:w \int_use:N #1 \q_stop
    \tex_advance:D #1 -\c_one_thousand_million
  }
\cs_new_protected:Npn \fp_mul_product:NN #1#2
  {
    \l_fp_mul_output_int
      \int_eval:w \l_fp_mul_output_int + #1 * #2 \scan_stop:
  }
```

At the end of each output group of three, there is a transfer of information so that there is no danger of an overflow. This is done by expansion to keep the number of calculations down.

```
\cs_new_protected_nopar:Npn \fp_mul_end_level:
  {
    \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
    \exp_after:wN \use_i:nn \exp_after:wN
      \fp_mul_end_level:NNNNNNNNN \int_use:N \l_fp_mul_output_int
  }
\cs_new_protected:Npn \fp_mul_end_level:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
    \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
  }
```

(*End definition for* `\fp_mul:Nn` *and* `\fp_mul:cn`*. These functions are documented on page* **??**.)

`\fp_div:Nn`
`\fp_div:cn`
`\fp_gdiv:Nn`
`\fp_gdiv:cn`
`\fp_div_aux:NNn`

The pattern is much the same for multiplication.

```
\cs_new_protected_nopar:Npn \fp_div:Nn  { \fp_div_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gdiv:Nn { \fp_div_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_div:Nn  { c }
\cs_generate_variant:Nn \fp_gdiv:Nn { c }
```

`\fp_div_internal:`
`\fp_div_loop:`
`\fp_div_divide:`
`\fp_div_divide_aux:`
`\fp_div_store:`
`\fp_div_store_integer:`
`\fp_div_store_decimal:`

Division proper starts with a couple of tests. If the denominator is zero then a error is issued. On the other hand, if the numerator is zero then the result must be 0.0 and can be given with no further work.

```
\cs_new_protected:Npn \fp_div_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \fp_split:Nn b {#3}
      \fp_standardise:NNNN
        \l_fp_input_b_sign_int
        \l_fp_input_b_integer_int
        \l_fp_input_b_decimal_int
        \l_fp_input_b_exponent_int
      \if_int_compare:w
        \int_eval:w
          \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
          = \c_zero
        \cs_set_protected:Npx \fp_tmp:w ##1##2
          {
            \group_end:
            #1 \exp_not:N #2 { \c_undefined_fp }
          }
      \else:
        \if_int_compare:w
          \int_eval:w
            \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
            = \c_zero
          \cs_set_protected:Npx \fp_tmp:w ##1##2
            {
              \group_end:
              #1 \exp_not:N #2 { \c_zero_fp }
            }
        \else:
          \exp_after:wN \exp_after:wN \exp_after:wN \fp_div_internal:
        \fi:
      \fi:
    \fp_tmp:w #1#2
  }
```

The main division algorithm works by finding how many times `b` can be removed from `a`, storing the result and doing the subtraction. Input `a` is then multiplied by 10, and the process is repeated. The looping ends either when there is nothing left of `a` (*i.e.* an exact result) or when the code reaches the ninth decimal place. Most of the process takes place in the loop function below.

```
\cs_new_protected_nopar:Npn \fp_div_internal: {
  \l_fp_output_integer_int \c_zero
  \l_fp_output_decimal_int \c_zero
  \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
  \l_fp_div_offset_int \c_one_hundred_million
  \fp_div_loop:
  \l_fp_output_exponent_int
    \int_eval:w
      \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
    \scan_stop:
  \fp_standardise:NNNN
    \l_fp_output_sign_int
    \l_fp_output_integer_int
    \l_fp_output_decimal_int
    \l_fp_output_exponent_int
  \cs_set_protected:Npx \fp_tmp:w ##1##2
    {
      \group_end:
      ##1 ##2
        {
          \if_int_compare:w
            \int_eval:w
              \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
              < \c_zero
            \if_int_compare:w
              \int_eval:w
                \l_fp_output_integer_int + \l_fp_output_decimal_int
                = \c_zero
              +
            \else:
              -
            \fi:
          \else:
            +
          \fi:
          \int_use:N \l_fp_output_integer_int
          .
          \exp_after:wN \use_none:n
            \int_value:w \int_eval:w
               \l_fp_output_decimal_int + \c_one_thousand_million
             \int_eval_end:
          e
          \int_use:N \l_fp_output_exponent_int
        }
    }
}
```

The main loop implements the approach described above. The storing function is done as a function so that the integer and decimal parts can be done separately but rapidly.

```
\cs_new_protected_nopar:Npn \fp_div_loop:
  {
    \l_fp_count_int \c_zero
    \fp_div_divide:
    \fp_div_store:
    \tex_multiply:D \l_fp_input_a_integer_int \c_ten
    \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \exp_after:wN \fp_div_loop_step:w
      \int_use:N \l_fp_input_a_decimal_int \q_stop
    \if_int_compare:w
      \int_eval:w \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
        > \c_zero
        \if_int_compare:w \l_fp_div_offset_int > \c_zero
          \exp_after:wN \exp_after:wN \exp_after:wN
            \fp_div_loop:
        \fi:
    \fi:
  }
```

Checking to see if the numerator can be divides needs quite an involved check. Either the integer part has to be bigger for the numerator or, if it is not smaller then the decimal part of the numerator must not be smaller than that of the denominator. Once the test is right the rest is much as elsewhere.

```
\cs_new_protected_nopar:Npn \fp_div_divide:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
      \exp_after:wN \fp_div_divide_aux:
    \else:
      \if_int_compare:w \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
      \else:
        \if_int_compare:w
          \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
        \else:
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \fp_div_divide_aux:
        \fi:
      \fi:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_div_divide_aux:
  {
    \tex_advance:D \l_fp_count_int \c_one
    \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
    \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int
    \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero
      \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \fi:
```

```
    \fp_div_divide:
  }
```

Storing the number of each division is done differently for the integer and decimal. The integer is easy and a one-off, while the decimal also needs to account for the position of the digit to store.

```
\cs_new_protected_nopar:Npn \fp_div_store: { }
\cs_new_protected_nopar:Npn \fp_div_store_integer:
  {
    \l_fp_output_integer_int \l_fp_count_int
    \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:
  }
\cs_new_protected_nopar:Npn \fp_div_store_decimal:
  {
    \l_fp_output_decimal_int
      \int_eval:w
        \l_fp_output_decimal_int +
        \l_fp_count_int * \l_fp_div_offset_int
      \int_eval_end:
    \tex_divide:D \l_fp_div_offset_int \c_ten
  }
\cs_new_protected:Npn \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop
  {
    \l_fp_input_a_integer_int
      \int_eval:w #2 + \l_fp_input_a_integer_int  \int_eval_end:
    \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
  }
```

(*End definition for* `\fp_div:Nn` *and* `\fp_div:cn`*. These functions are documented on page* **??**.)

202.10 Arithmetic for internal use

For the more complex functions, it is only possible to deliver reliable 10 digit accuracy if the internal calculations are carried out to a higher degree of precision. This is done using a second set of functions so that the 'user' versions are not slowed down. These versions are also focussed on the needs of internal calculations. No error checking, sign checking or exponent levelling is done. For addition and subtraction, the arguments are:

- Integer part of input `a`.
- Decimal part of input `a`.
- Additional decimal part of input `a`.
- Integer part of input `b`.
- Decimal part of input `b`.
- Additional decimal part of input `b`.
- Integer part of output.

- Decimal part of output.
- Additional decimal part of output.

The situation for multiplication and division is a little different as they only deal with the decimal part.

\fp_add:NNNNNNNNN The internal sum is always exactly that: it is always a sum and there is no sign check.

```
\cs_new_protected:Npn \fp_add:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    #7 \int_eval:w #1 + #4 \int_eval_end:
    #8 \int_eval:w #2 + #5 \int_eval_end:
    #9 \int_eval:w #3 + #6 \int_eval_end:
    \if_int_compare:w #9 < \c_one_thousand_million
    \else:
      \tex_advance:D #8 \c_one
      \tex_advance:D #9 -\c_one_thousand_million
    \fi:
    \if_int_compare:w #8 < \c_one_thousand_million
    \else:
      \tex_advance:D #7 \c_one
      \tex_advance:D #8 -\c_one_thousand_million
    \fi:
  }
```

(*End definition for* `\fp_add:NNNNNNNNN`*. This function is documented on page* **??**.)

\fp_sub:NNNNNNNNN Internal subtraction is needed only when the first number is bigger than the second, so there is no need to worry about the sign. This is a good job as there are no arguments left. The flipping flag is used in the rare case where a sign change is possible.

```
\cs_new_protected:Npn \fp_sub:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    #7 \int_eval:w #1 - #4 \int_eval_end:
    #8 \int_eval:w #2 - #5 \int_eval_end:
    #9 \int_eval:w #3 - #6 \int_eval_end:
    \if_int_compare:w #9 < \c_zero
      \tex_advance:D #8 \c_minus_one
      \tex_advance:D #9 \c_one_thousand_million
    \fi:
    \if_int_compare:w #8 < \c_zero
      \tex_advance:D #7 \c_minus_one
      \tex_advance:D #8 \c_one_thousand_million
    \fi:
    \if_int_compare:w #7 < \c_zero
      \if_int_compare:w \int_eval:w #8 + #9 = \c_zero
        #7 -#7
      \else:
        \tex_advance:D #7 \c_one
        #8 \int_eval:w \c_one_thousand_million - #8 \int_eval_end:
        #9 \int_eval:w \c_one_thousand_million - #9 \int_eval_end:
      \fi:
```

```
    \fi:
  }
```

(*End definition for* `\fp_sub:NNNNNNNNN`*. This function is documented on page* **??**.)

`\fp_mul:NNNNNN` Decimal-part only multiplication but with higher accuracy than the user version.

```
\cs_new_protected:Npn \fp_mul:NNNNNN #1#2#3#4#5#6
  {
    \fp_mul_split:NNNN #1
      \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
    \fp_mul_split:NNNN #2
      \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
    \fp_mul_split:NNNN #3
      \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
    \fp_mul_split:NNNN #4
      \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
    \l_fp_mul_output_int \c_zero
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_mul_b_vi_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_mul_b_v_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int       \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int        \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_v_int         \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_vi_int        \l_fp_mul_b_i_int
    \tex_divide:D \l_fp_mul_output_int \c_one_thousand
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_v_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int       \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_v_int        \l_fp_mul_b_i_int
    \fp_mul_end_level:
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int       \l_fp_mul_b_i_int
    \fp_mul_end_level:
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_i_int
    \fp_mul_end_level:
    #6 0 \l_fp_mul_output_tl \scan_stop:
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_i_int
    \fp_mul_end_level:
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_i_int
    \fp_mul_end_level:
    \fp_mul_end_level:
    #5 0 \l_fp_mul_output_tl \scan_stop:
  }
```

(*End definition for* `\fp_mul:NNNNNN`*. This function is documented on page* **??**.)

`\fp_mul:NNNNNNNNN` For internal multiplication where the integer does need to be retained. This means of course that this code is quite slow, and so is only used when necessary.

```
\cs_new_protected:Npn \fp_mul:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \fp_mul_split:NNNN #2
      \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
    \fp_mul_split:NNNN #3
      \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
    \fp_mul_split:NNNN #5
      \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
    \fp_mul_split:NNNN #6
      \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
    \l_fp_mul_output_int \c_zero
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN \l_fp_mul_a_i_int         \l_fp_mul_b_vi_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int        \l_fp_mul_b_v_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int       \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int        \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_v_int         \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_vi_int        \l_fp_mul_b_i_int
    \tex_divide:D \l_fp_mul_output_int \c_one_thousand
    \fp_mul_product:NN #1                       \l_fp_mul_b_vi_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_v_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int       \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_v_int        \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_vi_int       #4
    \fp_mul_end_level:
    \fp_mul_product:NN #1                       \l_fp_mul_b_v_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int       \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_v_int        #4
    \fp_mul_end_level:
    \fp_mul_product:NN #1                       \l_fp_mul_b_iv_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_iv_int       #4
    \fp_mul_end_level:
    #9 0 \l_fp_mul_output_tl \scan_stop:
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN #1                       \l_fp_mul_b_iii_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_iii_int      #4
```

```
    \fp_mul_end_level:
    \fp_mul_product:NN #1                       \l_fp_mul_b_ii_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_ii_int       #4
    \fp_mul_end_level:
    \fp_mul_product:NN #1                       \l_fp_mul_b_i_int
    \fp_mul_product:NN \l_fp_mul_a_i_int        #4
    \fp_mul_end_level:
    #8 0 \l_fp_mul_output_tl \scan_stop:
    \tl_clear:N \l_fp_mul_output_tl
    \fp_mul_product:NN #1 #4
    \fp_mul_end_level:
    #7 0 \l_fp_mul_output_tl \scan_stop:
  }
```

(*End definition for* `\fp_mul:NNNNNNNNN`*. This function is documented on page* **??**.)

`\fp_div_integer:NNNNN` Here, division is always by an integer, and so it is possible to use TEX's native calculations rather than doing it in macros. The idea here is to divide the decimal part, find any remainder, then do the real division of the two parts before adding in what is needed for the remainder.

```
\cs_new_protected:Npn \fp_div_integer:NNNNN #1#2#3#4#5
  {
    \l_fp_internal_int #1
    \tex_divide:D \l_fp_internal_int #3
    \l_fp_internal_int \int_eval:w #1 - \l_fp_internal_int * #3 \int_eval_end:
    #4 #1
    \tex_divide:D #4 #3
    #5 #2
    \tex_divide:D #5 #3
    \tex_multiply:D \l_fp_internal_int \c_one_thousand
    \tex_divide:D \l_fp_internal_int #3
    #5 \int_eval:w #5 + \l_fp_internal_int * \c_one_million \int_eval_end:
    \if_int_compare:w #5 > \c_one_thousand_million
      \tex_advance:D #4 \c_one
      \tex_advance:D #5 -\c_one_thousand_million
    \fi:
  }
```

(*End definition for* `\fp_div_integer:NNNNN`*. This function is documented on page* **??**.)

`\fp_extended_normalise:`
`\fp_extended_normalise_aux_i:`
`\fp_extended_normalise_aux_i:w`
`\fp_extended_normalise_aux_ii:w`
`\fp_extended_normalise_aux_ii:`
`\fp_extended_normalise_aux:NNNNNNNNN`

The "extended" integers for internal use are mainly used in fixed-point mode. This comes up in a few places, so a generalised utility is made available to carry out the change. This function simply calls the two loops to shift the input to the point of having a zero exponent.

```
\cs_new_protected_nopar:Npn \fp_extended_normalise:
  {
    \fp_extended_normalise_aux_i:
    \fp_extended_normalise_aux_ii:
  }
\cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i:
```

```
  {
    \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero
      \tex_multiply:D \l_fp_input_a_integer_int \c_ten
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \exp_after:wN \fp_extended_normalise_aux_i:w
        \int_use:N \l_fp_input_a_decimal_int \q_stop
       \exp_after:wN \fp_extended_normalise_aux_i:
     \fi:
  }
\cs_new_protected:Npn \fp_extended_normalise_aux_i:w
  #1#2#3#4#5#6#7#8#9 \q_stop
  {
    \l_fp_input_a_integer_int
      \int_eval:w \l_fp_input_a_integer_int + #2 \scan_stop:
    \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
    \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
    \exp_after:wN \fp_extended_normalise_aux_ii:w
      \int_use:N \l_fp_input_a_extended_int \q_stop
  }
\cs_new_protected:Npn \fp_extended_normalise_aux_ii:w
  #1#2#3#4#5#6#7#8#9 \q_stop
  {
    \l_fp_input_a_decimal_int
      \int_eval:w \l_fp_input_a_decimal_int + #2 \scan_stop:
    \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
    \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
  }
\cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
        \fp_extended_normalise_ii_aux:NNNNNNNNN
        \int_use:N \l_fp_input_a_decimal_int
       \exp_after:wN \fp_extended_normalise_aux_ii:
     \fi:
  }
\cs_new_protected:Npn \fp_extended_normalise_ii_aux:NNNNNNNNN
  #1#2#3#4#5#6#7#8#9
  {
    \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
      \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
    \else:
      \tl_set:Nx \l_fp_internal_tl
        {
          \int_use:N \l_fp_input_a_integer_int
          #1#2#3#4#5#6#7#8
        }
      \l_fp_input_a_integer_int \c_zero
      \l_fp_input_a_decimal_int \l_fp_internal_tl \scan_stop:
```

```
    \fi:
    \tex_divide:D \l_fp_input_a_extended_int \c_ten
    \tl_set:Nx \l_fp_internal_tl
      {
        #9
        \int_use:N \l_fp_input_a_extended_int
      }
    \l_fp_input_a_extended_int \l_fp_internal_tl \scan_stop:
    \tex_advance:D \l_fp_input_a_exponent_int \c_one
  }
```

(*End definition for* `\fp_extended_normalise:`. *This function is documented on page* **??**.)

\fp_extended_normalise_output:
\fp_extended_normalise_output_aux_i:NNNNNNNNN
\fp_extended_normalise_output_aux_ii:NNNNNNNNN
\fp_extended_normalise_output_aux:N

At some stages in working out extended output, it is possible for the value to need shifting to keep the integer part in range. This only ever happens such that the integer needs to be made smaller.

```
\cs_new_protected_nopar:Npn \fp_extended_normalise_output:
  {
    \if_int_compare:w \l_fp_output_integer_int > \c_nine
      \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
        \fp_extended_normalise_output_aux_i:NNNNNNNNN
        \int_use:N \l_fp_output_integer_int
      \exp_after:wN \fp_extended_normalise_output:
    \fi:
  }
\cs_new_protected:Npn \fp_extended_normalise_output_aux_i:NNNNNNNNN
  #1#2#3#4#5#6#7#8#9
  {
    \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
    \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
    \tl_set:Nx \l_fp_internal_tl
      {
        #9
        \exp_after:wN \use_none:n
        \int_use:N \l_fp_output_decimal_int
      }
    \exp_after:wN \fp_extended_normalise_output_aux_ii:NNNNNNNNN
      \l_fp_internal_tl
  }
\cs_new_protected:Npn \fp_extended_normalise_output_aux_ii:NNNNNNNNN
  #1#2#3#4#5#6#7#8#9
  {
    \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
    \fp_extended_normalise_output_aux:N
  }
\cs_new_protected:Npn \fp_extended_normalise_output_aux:N #1
  {
    \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
    \tex_divide:D \l_fp_output_extended_int \c_ten
```

```
    \tl_set:Nx \l_fp_internal_tl
      {
        #1
        \exp_after:wN \use_none:n
          \int_use:N \l_fp_output_extended_int
      }
    \l_fp_output_extended_int \l_fp_internal_tl \scan_stop:
    \tex_advance:D \l_fp_output_exponent_int \c_one
  }
```

(*End definition for* `\fp_extended_normalise_output:`*. This function is documented on page* **??**.)

202.11 Trigonometric functions

`\fp_trig_normalise:`
`\fp_trig_normalise_aux:`
`\fp_trig_sub:NNN`

For normalisation, the code essentially switches to fixed-point arithmetic. There is a shift of the exponent, then repeated subtractions. The end result is a number in the range $-\pi < x \leq \pi$.

```
\cs_new_protected_nopar:Npn \fp_trig_normalise:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
      \l_fp_input_a_extended_int \c_zero
      \fp_extended_normalise:
      \fp_trig_normalise_aux:
      \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
        \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
        \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
      \fi:
       \exp_after:wN \fp_trig_octant:
    \else:
      \l_fp_input_a_sign_int    \c_one
      \l_fp_output_integer_int  \c_zero
      \l_fp_output_decimal_int  \c_zero
      \l_fp_output_exponent_int \c_zero
      \exp_after:wN \fp_trig_overflow_msg:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_trig_normalise_aux:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \c_three
      \fp_trig_sub:NNN
        \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
      \exp_after:wN \fp_trig_normalise_aux:
    \else:
      \if_int_compare:w \l_fp_input_a_integer_int > \c_two
        \if_int_compare:w \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
          \fp_trig_sub:NNN
            \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \fp_trig_normalise_aux:
```

```
        \fi:
      \fi:
    \fi:
  }
```

Here, there may be a sign change but there will never be any variation in the input. So a dedicated function can be used.

```
\cs_new_protected:Npn \fp_trig_sub:NNN #1#2#3
  {
    \l_fp_input_a_integer_int
      \int_eval:w \l_fp_input_a_integer_int - #1 \int_eval_end:
    \l_fp_input_a_decimal_int
      \int_eval:w \l_fp_input_a_decimal_int - #2 \int_eval_end:
    \l_fp_input_a_extended_int
      \int_eval:w \l_fp_input_a_extended_int - #3 \int_eval_end:
    \if_int_compare:w \l_fp_input_a_extended_int < \c_zero
      \tex_advance:D \l_fp_input_a_decimal_int \c_minus_one
      \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
    \fi:
    \if_int_compare:w \l_fp_input_a_decimal_int < \c_zero
      \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \fi:
    \if_int_compare:w \l_fp_input_a_integer_int < \c_zero
      \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
      \if_int_compare:w
        \int_eval:w
          \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
        = \c_zero
        \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
      \else:
        \l_fp_input_a_integer_int
           \int_eval:w
             - \l_fp_input_a_integer_int - \c_one
           \int_eval_end:
        \l_fp_input_a_decimal_int
          \int_eval:w
            \c_one_thousand_million - \l_fp_input_a_decimal_int
          \int_eval_end:
        \l_fp_input_a_extended_int
          \int_eval:w
            \c_one_thousand_million - \l_fp_input_a_extended_int
          \int_eval_end:
      \fi:
    \fi:
  }
```

(*End definition for* `\fp_trig_normalise:`*. This function is documented on page* **??**.)

`\fp_trig_octant:`
`\fp_trig_octant_aux_i:`
`\fp_trig_octant_aux_ii:`

Here, the input is further reduced into the range $0 < x \leq \pi/4$. This is pretty simple: check if $\pi/4$ can be taken off and if it can do it and loop. The check at the end is to "mop up" values which are so close to $\pi/4$ that they should be treated as such. The test for an even octant is needed as the 'remainder' needed is from the nearest $\pi/2$. The check for octant 4 is needed as an exact π input will otherwise end up in the wrong place!

```
\cs_new_protected_nopar:Npn \fp_trig_octant:
  {
    \l_fp_trig_octant_int \c_one
    \fp_trig_octant_aux_i:
    \if_int_compare:w \l_fp_input_a_decimal_int < \c_ten
      \l_fp_input_a_decimal_int  \c_zero
      \l_fp_input_a_extended_int \c_zero
    \fi:
    \if_int_odd:w \l_fp_trig_octant_int
    \else:
      \fp_sub:NNNNNNNNN
        \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
        \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
          \l_fp_input_a_extended_int
        \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
          \l_fp_input_a_extended_int
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_trig_octant_aux_i:
  {
    \if_int_compare:w \l_fp_trig_octant_int > \c_four
      \l_fp_trig_octant_int \c_four
      \l_fp_input_a_decimal_int  \c_fp_pi_by_four_decimal_int
      \l_fp_input_a_extended_int \c_fp_pi_by_four_extended_int
    \else:
      \exp_after:wN \fp_trig_octant_aux_ii:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_trig_octant_aux_ii:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
      \fp_sub:NNNNNNNNN
        \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
          \l_fp_input_a_extended_int
        \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
        \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
          \l_fp_input_a_extended_int
      \tex_advance:D \l_fp_trig_octant_int \c_one
      \exp_after:wN \fp_trig_octant_aux_i:
    \else:
      \if_int_compare:w
        \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
        \fp_sub:NNNNNNNNN
          \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
            \l_fp_input_a_extended_int
          \c_zero \c_fp_pi_by_four_decimal_int
```

```
            \c_fp_pi_by_four_extended_int
          \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
            \l_fp_input_a_extended_int
        \tex_advance:D \l_fp_trig_octant_int \c_one
        \exp_after:wN \exp_after:wN \exp_after:wN
          \fp_trig_octant_aux_i:
      \fi:
    \fi:
  }
```

(*End definition for* `\fp_trig_octant:`. *This function is documented on page* **??**.)

`\fp_sin:Nn`
`\fp_sin:cn`
`\fp_gsin:Nn`
`\fp_gsin:cn`
`\fp_sin_aux:NNn`
`\fp_sin_aux_i:`
`\fp_sin_aux_ii:`

Calculating the sine starts off in the usual way. There is a check to see if the value has already been worked out before proceeding further.

```
\cs_new_protected_nopar:Npn \fp_sin:Nn  { \fp_sin_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gsin:Nn { \fp_sin_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_sin:Nn   { c }
\cs_generate_variant:Nn \fp_gsin:Nn  { c }
```

The internal routine for sines does a check to see if the value is already known. This saves a lot of repetition when doing rotations. For very small values it is best to simply return the input as the sine: the cut-off is 1×10^{-5}.

```
\cs_new_protected:Npn \fp_sin_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \tl_set:Nx \l_fp_arg_tl
        {
          \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
            -
          \else:
            +
          \fi:
          \int_use:N \l_fp_input_a_integer_int
          .
          \exp_after:wN \use_none:n
            \int_value:w \int_eval:w
              \l_fp_input_a_decimal_int + \c_one_thousand_million
          e
          \int_use:N \l_fp_input_a_exponent_int
        }
      \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
        \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2 { \l_fp_arg_tl }
```

```
        }
      \else:
        \if_cs_exist:w
          c_fp_sin ( \l_fp_arg_tl ) _fp
        \cs_end:
        \else:
          \exp_after:wN \exp_after:wN \exp_after:wN
            \fp_sin_aux_i:
        \fi:
        \cs_set_protected_nopar:Npx \fp_tmp:w
          {
            \group_end:
            #1 \exp_not:N #2
              { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
          }
      \fi:
    \fp_tmp:w
  }
```

The internals for sine first normalise the input into an octant, then choose the correct set up for the Taylor series. The sign for the sine function is easy, so there is no worry about it. So the only thing to do is to get the output standardised.

```
\cs_new_protected_nopar:Npn \fp_sin_aux_i:
  {
    \fp_trig_normalise:
    \fp_sin_aux_ii:
    \if_int_compare:w \l_fp_output_integer_int = \c_one
      \l_fp_output_exponent_int \c_zero
    \else:
      \l_fp_output_integer_int \l_fp_output_decimal_int
      \l_fp_output_decimal_int \l_fp_output_extended_int
      \l_fp_output_exponent_int -\c_nine
    \fi:
    \fp_standardise:NNNN
      \l_fp_input_a_sign_int
      \l_fp_output_integer_int
      \l_fp_output_decimal_int
      \l_fp_output_exponent_int
    \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
    \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
      {
        \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
          +
        \else:
          -
        \fi:
        \int_use:N \l_fp_output_integer_int
        .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
```

```
            \l_fp_output_decimal_int + \c_one_thousand_million
        e
        \int_use:N \l_fp_output_exponent_int
      }
  }
\cs_new_protected_nopar:Npn \fp_sin_aux_ii:
  {
    \if_case:w \l_fp_trig_octant_int
    \or:
      \exp_after:wN \fp_trig_calc_sin:
    \or:
      \exp_after:wN \fp_trig_calc_cos:
    \or:
      \exp_after:wN \fp_trig_calc_cos:
    \or:
      \exp_after:wN \fp_trig_calc_sin:
    \fi:
  }
```

(*End definition for* `\fp_sin:Nn` *and* `\fp_sin:cn`*. These functions are documented on page* **??**.)

`\fp_cos:Nn`
`\fp_cos:cn`
`\fp_gcos:Nn`
`\fp_gcos:cn`
`\fp_cos_aux:NNn`
`\fp_cos_aux_i:`
`\fp_cos_aux_ii:`

Cosine is almost identical, but there is no short cut code here.

```
\cs_new_protected_nopar:Npn \fp_cos:Nn  { \fp_cos_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gcos:Nn { \fp_cos_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_cos:Nn   { c }
\cs_generate_variant:Nn \fp_gcos:Nn  { c }
\cs_new_protected:Npn \fp_cos_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \tl_set:Nx \l_fp_arg_tl
        {
          \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
            -
          \else:
            +
          \fi:
          \int_use:N \l_fp_input_a_integer_int
          .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
             \l_fp_input_a_decimal_int + \c_one_thousand_million
          e
          \int_use:N \l_fp_input_a_exponent_int
        }
      \if_cs_exist:w c_fp_cos ( \l_fp_arg_tl ) _fp \cs_end:
```

```
      \else:
        \exp_after:wN \fp_cos_aux_i:
      \fi:
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
        }
    \fp_tmp:w
  }
```

Almost the same as for sine: just a bit of correction for the sign of the output.

```
\cs_new_protected_nopar:Npn \fp_cos_aux_i:
  {
    \fp_trig_normalise:
    \fp_cos_aux_ii:
    \if_int_compare:w \l_fp_output_integer_int = \c_one
      \l_fp_output_exponent_int \c_zero
    \else:
      \l_fp_output_integer_int \l_fp_output_decimal_int
      \l_fp_output_decimal_int \l_fp_output_extended_int
      \l_fp_output_exponent_int -\c_nine
    \fi:
    \fp_standardise:NNNN
      \l_fp_input_a_sign_int
      \l_fp_output_integer_int
      \l_fp_output_decimal_int
      \l_fp_output_exponent_int
    \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
    \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
      {
        \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
          +
        \else:
          -
        \fi:
        \int_use:N \l_fp_output_integer_int
        .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
             \l_fp_output_decimal_int + \c_one_thousand_million
        e
        \int_use:N \l_fp_output_exponent_int
      }
  }
\cs_new_protected_nopar:Npn \fp_cos_aux_ii:
  {
    \if_case:w \l_fp_trig_octant_int
    \or:
```

```
      \exp_after:wN \fp_trig_calc_cos:
    \or:
      \exp_after:wN \fp_trig_calc_sin:
    \or:
      \exp_after:wN \fp_trig_calc_sin:
    \or:
      \exp_after:wN \fp_trig_calc_cos:
    \fi:
    \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
      \if_int_compare:w \l_fp_trig_octant_int > \c_two
        \l_fp_input_a_sign_int \c_minus_one
      \fi:
    \else:
      \if_int_compare:w \l_fp_trig_octant_int > \c_two
      \else:
        \l_fp_input_a_sign_int \c_one
      \fi:
    \fi:
  }
```

(*End definition for* `\fp_cos:Nn` *and* `\fp_cos:cn`*. These functions are documented on page* **??**.)

`\fp_trig_calc_cos:`
`\fp_trig_calc_sin:`
`\fp_trig_calc_Taylor:`

These functions actually do the calculation for sine and cosine.

```
\cs_new_protected_nopar:Npn \fp_trig_calc_cos:
  {
    \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
      \l_fp_output_integer_int \c_one
      \l_fp_output_decimal_int \c_zero
    \else:
      \l_fp_trig_sign_int \c_minus_one
      \fp_mul:NNNNNN
        \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
        \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
        \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \fp_div_integer:NNNNN
        \l_fp_trig_decimal_int \l_fp_trig_extended_int
        \c_two
        \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \l_fp_count_int \c_three
      \if_int_compare:w \l_fp_trig_extended_int = \c_zero
        \if_int_compare:w \l_fp_trig_decimal_int = \c_zero
          \l_fp_output_integer_int \c_one
          \l_fp_output_decimal_int \c_zero
          \l_fp_output_extended_int \c_zero
        \else:
          \l_fp_output_integer_int \c_zero
          \l_fp_output_decimal_int \c_one_thousand_million
          \l_fp_output_extended_int \c_zero
        \fi:
      \else:
        \l_fp_output_integer_int \c_zero
```

```
        \l_fp_output_decimal_int 999999999 \scan_stop:
        \l_fp_output_extended_int \c_one_thousand_million
      \fi:
      \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
      \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
      \exp_after:wN \fp_trig_calc_Taylor:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_trig_calc_sin:
  {
    \l_fp_output_integer_int \c_zero
    \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
      \l_fp_output_decimal_int \c_zero
    \else:
      \l_fp_output_decimal_int \l_fp_input_a_decimal_int
      \l_fp_output_extended_int \l_fp_input_a_extended_int
      \l_fp_trig_sign_int \c_one
      \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
      \l_fp_trig_extended_int \l_fp_input_a_extended_int
      \l_fp_count_int \c_two
      \exp_after:wN \fp_trig_calc_Taylor:
    \fi:
  }
```

This implements a Taylor series calculation for the trigonometric functions. Lots of shuffling about as TEX is not exactly a natural choice for this sort of thing.

```
\cs_new_protected_nopar:Npn \fp_trig_calc_Taylor:
  {
    \l_fp_trig_sign_int -\l_fp_trig_sign_int
    \fp_mul:NNNNNN
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
    \fp_mul:NNNNNN
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
    \fp_div_integer:NNNNN
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \l_fp_count_int
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
    \tex_advance:D \l_fp_count_int \c_one
    \fp_div_integer:NNNNN
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
      \l_fp_count_int
      \l_fp_trig_decimal_int \l_fp_trig_extended_int
    \tex_advance:D \l_fp_count_int \c_one
    \if_int_compare:w \l_fp_trig_decimal_int > \c_zero
      \if_int_compare:w \l_fp_trig_sign_int > \c_zero
        \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
```

```
        \tex_advance:D \l_fp_output_extended_int
          \l_fp_trig_extended_int
        \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million
        \else:
          \tex_advance:D \l_fp_output_decimal_int \c_one
          \tex_advance:D \l_fp_output_extended_int
            -\c_one_thousand_million
        \fi:
        \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
        \else:
          \tex_advance:D \l_fp_output_integer_int \c_one
          \tex_advance:D \l_fp_output_decimal_int
            -\c_one_thousand_million
        \fi:
      \else:
        \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
        \tex_advance:D \l_fp_output_extended_int
          -\l_fp_input_a_extended_int
        \if_int_compare:w \l_fp_output_extended_int < \c_zero
          \tex_advance:D \l_fp_output_decimal_int \c_minus_one
          \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
        \fi:
        \if_int_compare:w \l_fp_output_decimal_int < \c_zero
          \tex_advance:D \l_fp_output_integer_int \c_minus_one
          \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
        \fi:
      \fi:
       \exp_after:wN \fp_trig_calc_Taylor:
    \fi:
  }
```

(*End definition for* `\fp_trig_calc_cos:`*. This function is documented on page* **??**.)

`\fp_tan:Nn`
`\fp_tan:cn`
`\fp_gtan:Nn`
`\fp_gtan:cn`
`\fp_tan_aux:NNn`
`\fp_tan_aux_i:`
`\fp_tan_aux_ii:`
`\fp_tan_aux_iii:`
`\fp_tan_aux_iv:`

As might be expected, tangents are calculated from the sine and cosine by division. So there is a bit of set up, the two subsidiary pieces of work are done and then a division takes place. For small numbers, the same approach is used as for sines, with the input value simply returned as is.

```
\cs_new_protected_nopar:Npn \fp_tan:Nn  { \fp_tan_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gtan:Nn { \fp_tan_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_tan:Nn   { c }
\cs_generate_variant:Nn \fp_gtan:Nn  { c }
\cs_new_protected:Npn \fp_tan_aux:NNn #1#2#3
  {
    \group_begin:
     \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
```

```
      \tl_set:Nx \l_fp_arg_tl
        {
          \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
            -
          \else:
            +
          \fi:
          \int_use:N \l_fp_input_a_integer_int
          .
          \exp_after:wN \use_none:n
            \int_value:w \int_eval:w
              \l_fp_input_a_decimal_int + \c_one_thousand_million
          e
          \int_use:N \l_fp_input_a_exponent_int
        }
      \if_int_compare:w \l_fp_input_a_exponent_int < -\c_five
        \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2 { \l_fp_arg_tl }
        }
      \else:
        \if_cs_exist:w
          c_fp_tan ( \l_fp_arg_tl ) _fp
        \cs_end:
        \else:
          \exp_after:wN \exp_after:wN \exp_after:wN
            \fp_tan_aux_i:
        \fi:
        \cs_set_protected_nopar:Npx \fp_tmp:w
          {
            \group_end:
            #1 \exp_not:N #2
              { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
          }
      \fi:
    \fp_tmp:w
  }
```

The business of the calculation does not check for stored sines or cosines as there would then be an overhead to reading them back in. There is also no need to worry about "small" sine values as these will have been dealt with earlier. There is a two-step lead off so that undefined division is not even attempted.

```
\cs_new_protected_nopar:Npn \fp_tan_aux_i:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
      \exp_after:wN \fp_tan_aux_ii:
    \else:
      \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
        \c_zero_fp
```

```
      \exp_after:wN \fp_trig_overflow_msg:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_tan_aux_ii:
  {
    \fp_trig_normalise:
    \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
      \if_int_compare:w \l_fp_trig_octant_int > \c_two
        \l_fp_output_sign_int \c_minus_one
      \else:
        \l_fp_output_sign_int \c_one
      \fi:
    \else:
      \if_int_compare:w \l_fp_trig_octant_int > \c_two
        \l_fp_output_sign_int \c_one
      \else:
        \l_fp_output_sign_int \c_minus_one
      \fi:
    \fi:
    \fp_cos_aux_ii:
    \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
      \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
        \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
          \c_undefined_fp
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN
          \fp_tan_aux_iii:
      \fi:
    \else:
      \exp_after:wN \fp_tan_aux_iii:
    \fi:
  }
```

The division is done here using the same code as the standard division unit, shifting the digits in the calculated sine and cosine to maintain accuracy.

```
\cs_new_protected_nopar:Npn \fp_tan_aux_iii:
  {
    \l_fp_input_b_integer_int \l_fp_output_decimal_int
    \l_fp_input_b_decimal_int \l_fp_output_extended_int
    \l_fp_input_b_exponent_int -\c_nine
    \fp_standardise:NNNN
      \l_fp_input_b_sign_int
      \l_fp_input_b_integer_int
      \l_fp_input_b_decimal_int
      \l_fp_input_b_exponent_int
    \fp_sin_aux_ii:
    \l_fp_input_a_integer_int \l_fp_output_decimal_int
    \l_fp_input_a_decimal_int \l_fp_output_extended_int
    \l_fp_input_a_exponent_int -\c_nine
    \fp_standardise:NNNN
```

```
      \l_fp_input_a_sign_int
      \l_fp_input_a_integer_int
      \l_fp_input_a_decimal_int
      \l_fp_input_a_exponent_int
    \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
      \if_int_compare:w \l_fp_input_a_integer_int = \c_zero
        \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
          \c_zero_fp
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_tan_aux_iv:
      \fi:
    \else:
      \exp_after:wN \fp_tan_aux_iv:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_tan_aux_iv:
  {
    \l_fp_output_integer_int \c_zero
    \l_fp_output_decimal_int \c_zero
    \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
    \l_fp_div_offset_int \c_one_hundred_million
    \fp_div_loop:
    \l_fp_output_exponent_int
      \int_eval:w
        \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
      \int_eval_end:
    \fp_standardise:NNNN
      \l_fp_output_sign_int
      \l_fp_output_integer_int
      \l_fp_output_decimal_int
      \l_fp_output_exponent_int
    \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
    \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }
      {
        \if_int_compare:w \l_fp_output_sign_int > \c_zero
          +
        \else:
          -
        \fi:
        \int_use:N \l_fp_output_integer_int
        .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
             \l_fp_output_decimal_int + \c_one_thousand_million
        e
        \int_use:N \l_fp_output_exponent_int
      }
  }
```

(*End definition for `\fp_tan:Nn` and `\fp_tan:cn`. These functions are documented on page* **??**.)

202.12 Exponent and logarithm functions

\c_fp_exp_1_tl
\c_fp_exp_2_tl
\c_fp_exp_3_tl
\c_fp_exp_4_tl
\c_fp_exp_5_tl
\c_fp_exp_6_tl
\c_fp_exp_7_tl
\c_fp_exp_8_tl
\c_fp_exp_9_tl
\c_fp_exp_10_tl
\c_fp_exp_20_tl
\c_fp_exp_30_tl
\c_fp_exp_40_tl
\c_fp_exp_50_tl
\c_fp_exp_60_tl
\c_fp_exp_70_tl
\c_fp_exp_80_tl
\c_fp_exp_90_tl
\c_fp_exp_100_tl
\c_fp_exp_200_tl

Calculation of exponentials requires a number of precomputed values: first the positive integers.

```
\tl_const:cn { c_fp_exp_1_tl }   { { 2 } { 718281828 } { 459045235 } { 0 } }
\tl_const:cn { c_fp_exp_2_tl }   { { 7 } { 389056098 } { 930650227 } { 0 } }
\tl_const:cn { c_fp_exp_3_tl }   { { 2 } { 008553692 } { 318766774 } { 1 } }
\tl_const:cn { c_fp_exp_4_tl }   { { 5 } { 459815003 } { 314423908 } { 1 } }
\tl_const:cn { c_fp_exp_5_tl }   { { 1 } { 484131591 } { 025766034 } { 2 } }
\tl_const:cn { c_fp_exp_6_tl }   { { 4 } { 034287934 } { 927351226 } { 2 } }
\tl_const:cn { c_fp_exp_7_tl }   { { 1 } { 096633158 } { 428458599 } { 3 } }
\tl_const:cn { c_fp_exp_8_tl }   { { 2 } { 980957987 } { 041728275 } { 3 } }
\tl_const:cn { c_fp_exp_9_tl }   { { 8 } { 103083927 } { 575384008 } { 3 } }
\tl_const:cn { c_fp_exp_10_tl }  { { 2 } { 202646579 } { 480671652 } { 4 } }
\tl_const:cn { c_fp_exp_20_tl }  { { 4 } { 851651954 } { 097902280 } { 8 } }
\tl_const:cn { c_fp_exp_30_tl }  { { 1 } { 068647458 } { 152446215 } { 13 } }
\tl_const:cn { c_fp_exp_40_tl }  { { 2 } { 353852668 } { 370199854 } { 17 } }
\tl_const:cn { c_fp_exp_50_tl }  { { 5 } { 184705528 } { 587072464 } { 21 } }
\tl_const:cn { c_fp_exp_60_tl }  { { 1 } { 142007389 } { 815684284 } { 26 } }
\tl_const:cn { c_fp_exp_70_tl }  { { 2 } { 515438670 } { 919167006 } { 30 } }
\tl_const:cn { c_fp_exp_80_tl }  { { 5 } { 540622384 } { 393510053 } { 34 } }
\tl_const:cn { c_fp_exp_90_tl }  { { 1 } { 220403294 } { 317840802 } { 39 } }
\tl_const:cn { c_fp_exp_100_tl } { { 2 } { 688117141 } { 816135448 } { 43 } }
\tl_const:cn { c_fp_exp_200_tl } { { 7 } { 225973768 } { 125749258 } { 86 } }
```

(*End definition for* `\c_fp_exp_1_tl`. *This function is documented on page* **??**.)

\c_fp_exp_-1_tl
\c_fp_exp_-2_tl
\c_fp_exp_-3_tl
\c_fp_exp_-4_tl
\c_fp_exp_-5_tl
\c_fp_exp_-6_tl
\c_fp_exp_-7_tl
\c_fp_exp_-8_tl
\c_fp_exp_-9_tl
\c_fp_exp_-10_tl
\c_fp_exp_-20_tl
\c_fp_exp_-30_tl
\c_fp_exp_-40_tl
\c_fp_exp_-50_tl
\c_fp_exp_-60_tl
\c_fp_exp_-70_tl
\c_fp_exp_-80_tl
\c_fp_exp_-90_tl
\c_fp_exp_-100_tl
\c_fp_exp_-200_tl

Now the negative integers.

```
\tl_const:cn { c_fp_exp_-1_tl }   { { 3 } { 678794411 } { 71442322 }  { -1 } }
\tl_const:cn { c_fp_exp_-2_tl }   { { 1 } { 353352832 } { 366132692 } { -1 } }
\tl_const:cn { c_fp_exp_-3_tl }   { { 4 } { 978706836 } { 786394298 } { -2 } }
\tl_const:cn { c_fp_exp_-4_tl }   { { 1 } { 831563888 } { 873418029 } { -2 } }
\tl_const:cn { c_fp_exp_-5_tl }   { { 6 } { 737946999 } { 085467097 } { -3 } }
\tl_const:cn { c_fp_exp_-6_tl }   { { 2 } { 478752176 } { 666358423 } { -3 } }
\tl_const:cn { c_fp_exp_-7_tl }   { { 9 } { 118819655 } { 545162080 } { -4 } }
\tl_const:cn { c_fp_exp_-8_tl }   { { 3 } { 354626279 } { 025118388 } { -4 } }
\tl_const:cn { c_fp_exp_-9_tl }   { { 1 } { 234098040 } { 866795495 } { -4 } }
\tl_const:cn { c_fp_exp_-10_tl }  { { 4 } { 539992976 } { 248451536 } { -5 } }
\tl_const:cn { c_fp_exp_-20_tl }  { { 2 } { 061153622 } { 438557828 } { -9 } }
\tl_const:cn { c_fp_exp_-30_tl }  { { 9 } { 357622968 } { 840174605 } { -14 } }
\tl_const:cn { c_fp_exp_-40_tl }  { { 4 } { 248354255 } { 291588995 } { -18 } }
\tl_const:cn { c_fp_exp_-50_tl }  { { 1 } { 928749847 } { 963917783 } { -22 } }
\tl_const:cn { c_fp_exp_-60_tl }  { { 8 } { 756510762 } { 696520338 } { -27 } }
\tl_const:cn { c_fp_exp_-70_tl }  { { 3 } { 975449735 } { 908646808 } { -31 } }
\tl_const:cn { c_fp_exp_-80_tl }  { { 1 } { 804851387 } { 845415172 } { -35 } }
\tl_const:cn { c_fp_exp_-90_tl }  { { 8 } { 194012623 } { 990515430 } { -40 } }
\tl_const:cn { c_fp_exp_-100_tl } { { 3 } { 720075976 } { 020835963 } { -44 } }
\tl_const:cn { c_fp_exp_-200_tl } { { 1 } { 383896526 } { 736737530 } { -87 } }
```

(*End definition for* `\c_fp_exp_-1_tl`. *This function is documented on page* **??**.)

\fp_exp:Nn
\fp_exp:cn
\fp_gexp:Nn
\fp_gexp:cn
\fp_exp_aux:NNn
\fp_exp_internal:
\fp_exp_aux:
\fp_exp_integer:
\fp_exp_integer_tens:
\fp_exp_integer_units:
\fp_exp_integer_const:n
\fp_exp_integer_const:nnnn
\fp_exp_decimal:
\fp_exp_Taylor:
\fp_exp_const:Nx
\fp_exp_const:cx

The calculation of an exponent starts off starts in much the same way as the trigonometric functions: normalise the input, look for a pre-defined value and if one is not found hand off to the real workhorse function. The test for a definition of the result is used so that overflows do not result in any outcome being defined.

```
\cs_new_protected_nopar:Npn \fp_exp:Nn  { \fp_exp_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gexp:Nn { \fp_exp_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_exp:Nn  { c }
\cs_generate_variant:Nn \fp_gexp:Nn { c }
\cs_new_protected:Npn \fp_exp_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \l_fp_input_a_extended_int \c_zero
      \tl_set:Nx \l_fp_arg_tl
        {
          \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
            -
          \else:
            +
          \fi:
          \int_use:N \l_fp_input_a_integer_int
          .
          \exp_after:wN \use_none:n
            \int_value:w \int_eval:w
               \l_fp_input_a_decimal_int + \c_one_thousand_million
          e
          \int_use:N \l_fp_input_a_exponent_int
        }
      \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp \cs_end:
      \else:
        \exp_after:wN \fp_exp_internal:
      \fi:
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \if_cs_exist:w c_fp_exp ( \l_fp_arg_tl ) _fp
                \cs_end:
                \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
              \else:
                \c_zero_fp
              \fi:
            }
```

```
        }
    \fp_tmp:w
  }
```

The first real step is to convert the input into a fixed-point representation for further calculation: anything which is dropped here as too small would not influence the output in any case. There are a couple of overflow tests: the maximum

```
\cs_new_protected_nopar:Npn \fp_exp_internal:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int < \c_three
      \fp_extended_normalise:
      \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
        \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \fp_exp_aux:
        \else:
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \fp_exp_overflow_msg:
          \fi:
      \else:
        \if_int_compare:w \l_fp_input_a_integer_int < 230 \scan_stop:
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN  \fp_exp_aux:
        \else:
          \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
            { \c_zero_fp }
        \fi:
      \fi:
    \else:
      \exp_after:wN \fp_exp_overflow_msg:
    \fi:
  }
```

The main algorithm makes use of the fact that

$$\mathrm{e}^{nmp.q} = \mathrm{e}^{n}\mathrm{e}^{m}\mathrm{e}^{p}\mathrm{e}^{0.q}$$

and that there is a Taylor series that can be used to calculate $\mathrm{e}^{0.q}$. Thus the approach needed is in three parts. First, the exponent of the integer part of the input is found using the pre-calculated constants. Second, the Taylor series is used to find the exponent for the decimal part of the input. Finally, the two parts are multiplied together to give the result. As the normalisation code will already have dealt with any overflowing values, there are no further checks needed.

```
\cs_new_protected_nopar:Npn \fp_exp_aux:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
      \exp_after:wN \fp_exp_integer:
```

```
    \else:
      \l_fp_output_integer_int  \c_one
      \l_fp_output_decimal_int  \c_zero
      \l_fp_output_extended_int \c_zero
      \l_fp_output_exponent_int \c_zero
      \exp_after:wN \fp_exp_decimal:
    \fi:
  }
```

The integer part calculation starts with the hundreds. This is set up such that very large negative numbers can short-cut the entire procedure and simply return zero. In other cases, the code either recovers the exponent of the hundreds value or sets the appropriate storage to one (so that multiplication works correctly).

```
\cs_new_protected_nopar:Npn \fp_exp_integer:
  {
    \if_int_compare:w \l_fp_input_a_integer_int < \c_one_hundred
      \l_fp_exp_integer_int  \c_one
      \l_fp_exp_decimal_int  \c_zero
      \l_fp_exp_extended_int \c_zero
      \l_fp_exp_exponent_int \c_zero
      \exp_after:wN \fp_exp_integer_tens:
    \else:
      \tl_set:Nx \l_fp_internal_tl
        {
          \exp_after:wN \use_i:nnn
            \int_use:N \l_fp_input_a_integer_int
        }
      \l_fp_input_a_integer_int
        \int_eval:w
          \l_fp_input_a_integer_int - \l_fp_internal_tl 00
        \int_eval_end:
      \if_int_compare:w \l_fp_input_a_sign_int < \c_zero
        \if_int_compare:w \l_fp_output_integer_int > 200 \scan_stop:
          \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
            { \c_zero_fp }
        \else:
          \fp_exp_integer_const:n { - \l_fp_internal_tl 00 }
          \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \fp_exp_integer_tens:
        \fi:
      \else:
        \fp_exp_integer_const:n { \l_fp_internal_tl 00 }
        \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \fp_exp_integer_tens:
      \fi:
    \fi:
  }
```

The tens and units parts are handled in a similar way, with a multiplication step to build up the final value. That also includes a correction step to avoid an overflow of the integer part.

```
\cs_new_protected_nopar:Npn \fp_exp_integer_tens:
  {
    \l_fp_output_integer_int  \l_fp_exp_integer_int
    \l_fp_output_decimal_int  \l_fp_exp_decimal_int
    \l_fp_output_extended_int \l_fp_exp_extended_int
    \l_fp_output_exponent_int \l_fp_exp_exponent_int
    \if_int_compare:w \l_fp_input_a_integer_int > \c_nine
      \tl_set:Nx \l_fp_internal_tl
        {
          \exp_after:wN \use_i:nn
            \int_use:N \l_fp_input_a_integer_int
        }
      \l_fp_input_a_integer_int
        \int_eval:w
          \l_fp_input_a_integer_int - \l_fp_internal_tl 0
        \int_eval_end:
      \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
        \fp_exp_integer_const:n { \l_fp_internal_tl 0 }
      \else:
        \fp_exp_integer_const:n { - \l_fp_internal_tl 0 }
      \fi:
      \fp_mul:NNNNNNNNN
        \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
      \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
      \fp_extended_normalise_output:
    \fi:
    \fp_exp_integer_units:
  }
\cs_new_protected_nopar:Npn \fp_exp_integer_units:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \c_zero
      \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
        \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
      \else:
        \fp_exp_integer_const:n
          { - \int_use:N \l_fp_input_a_integer_int }
      \fi:
      \fp_mul:NNNNNNNNN
        \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
```

```
      \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
      \fp_extended_normalise_output:
    \fi:
    \fp_exp_decimal:
  }
```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```
\cs_new_protected:Npn \fp_exp_integer_const:n #1
  {
    \exp_after:wN \exp_after:wN \exp_after:wN
      \fp_exp_integer_const:nnnn
      \cs:w c_fp_exp_ #1 _tl \cs_end:
  }
\cs_new_protected:Npn \fp_exp_integer_const:nnnn #1#2#3#4
  {
    \l_fp_exp_integer_int  #1 \scan_stop:
    \l_fp_exp_decimal_int  #2 \scan_stop:
    \l_fp_exp_extended_int #3 \scan_stop:
    \l_fp_exp_exponent_int #4 \scan_stop:
  }
```

Finding the exponential for the decimal part of the number requires a Taylor series calculation. The set up is done here with the loop itself a separate function. Once the decimal part is available this is multiplied by the integer part already worked out to give the final result.

```
\cs_new_protected_nopar:Npn \fp_exp_decimal:
  {
    \if_int_compare:w \l_fp_input_a_decimal_int > \c_zero
      \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
        \l_fp_exp_integer_int  \c_one
        \l_fp_exp_decimal_int  \l_fp_input_a_decimal_int
        \l_fp_exp_extended_int \l_fp_input_a_extended_int
      \else:
        \l_fp_exp_integer_int \c_zero
        \if_int_compare:w \l_fp_exp_extended_int = \c_zero
          \l_fp_exp_decimal_int
            \int_eval:w
              \c_one_thousand_million - \l_fp_input_a_decimal_int
            \int_eval_end:
          \l_fp_exp_extended_int \c_zero
        \else:
          \l_fp_exp_decimal_int
            \int_eval:w
              999999999 - \l_fp_input_a_decimal_int
            \scan_stop:
          \l_fp_exp_extended_int
            \int_eval:w
              \c_one_thousand_million - \l_fp_input_a_extended_int
            \int_eval_end:
```

```
          \fi:
        \fi:
        \l_fp_input_b_sign_int     \l_fp_input_a_sign_int
        \l_fp_input_b_decimal_int  \l_fp_input_a_decimal_int
        \l_fp_input_b_extended_int \l_fp_input_a_extended_int
        \l_fp_count_int \c_one
        \fp_exp_Taylor:
        \fp_mul:NNNNNNNNN
          \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
          \l_fp_output_integer_int \l_fp_output_decimal_int
            \l_fp_output_extended_int
          \l_fp_output_integer_int \l_fp_output_decimal_int
            \l_fp_output_extended_int
      \fi:
      \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
      \else:
        \tex_advance:D \l_fp_output_decimal_int \c_one
        \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
        \else:
          \l_fp_output_decimal_int \c_zero
          \tex_advance:D \l_fp_output_integer_int \c_one
        \fi:
      \fi:
      \fp_standardise:NNNN
        \l_fp_output_sign_int
        \l_fp_output_integer_int
        \l_fp_output_decimal_int
        \l_fp_output_exponent_int
      \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
        {
          +
          \int_use:N \l_fp_output_integer_int
          .
          \exp_after:wN \use_none:n
            \int_value:w \int_eval:w
               \l_fp_output_decimal_int + \c_one_thousand_million
          e
          \int_use:N \l_fp_output_exponent_int
        }
    }
```

The Taylor series for $\exp(x)$ is

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

which converges for $-1 < x < 1$. The code above sets up the x part, leaving the loop to multiply the running value by x/n and add it onto the sum. The way that this is done is that the running total is stored in the `exp` set of registers, while the current item is stored as `input_b`.

```
\cs_new_protected_nopar:Npn \fp_exp_Taylor:
  {
    \tex_advance:D \l_fp_count_int \c_one
    \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
    \fp_mul:NNNNNN
      \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
      \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
      \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
    \fp_div_integer:NNNNN
      \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
      \l_fp_count_int
      \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
    \if_int_compare:w
      \int_eval:w
        \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
        > \c_zero
      \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
        \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
        \tex_advance:D \l_fp_exp_extended_int
          \l_fp_input_b_extended_int
        \if_int_compare:w \l_fp_exp_extended_int < \c_one_thousand_million
      \else:
          \tex_advance:D \l_fp_exp_decimal_int \c_one
          \tex_advance:D \l_fp_exp_extended_int
            -\c_one_thousand_million
        \fi:
        \if_int_compare:w \l_fp_exp_decimal_int < \c_one_thousand_million
        \else:
          \tex_advance:D \l_fp_exp_integer_int \c_one
          \tex_advance:D \l_fp_exp_decimal_int
            -\c_one_thousand_million
        \fi:
      \else:
        \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
        \tex_advance:D \l_fp_exp_extended_int
          -\l_fp_input_a_extended_int
        \if_int_compare:w \l_fp_exp_extended_int < \c_zero
          \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
          \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
        \fi:
        \if_int_compare:w \l_fp_exp_decimal_int < \c_zero
          \tex_advance:D \l_fp_exp_integer_int \c_minus_one
          \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
        \fi:
      \fi:
      \exp_after:wN \fp_exp_Taylor:
    \fi:
  }
```

This is set up as a function so that the power code can redirect the effect.

```
\cs_new_protected:Npn \fp_exp_const:Nx #1#2
  {
    \tl_new:N #1
    \tl_gset:Nx #1 {#2}
  }
\cs_generate_variant:Nn \fp_exp_const:Nx { c }
```

(*End definition for* `\fp_exp:Nn` *and* `\fp_exp:cn`*. These functions are documented on page* **??**.)

`\c_fp_ln_10_1_tl`
`\c_fp_ln_10_2_tl`
`\c_fp_ln_10_3_tl`
`\c_fp_ln_10_4_tl`
`\c_fp_ln_10_5_tl`
`\c_fp_ln_10_6_tl`
`\c_fp_ln_10_7_tl`
`\c_fp_ln_10_8_tl`
`\c_fp_ln_10_9_tl`

Constants for working out logarithms: first those for the powers of ten.

```
\tl_const:cn { c_fp_ln_10_1_tl } { { 2 } { 302585092 } { 994045684 } { 0 } }
\tl_const:cn { c_fp_ln_10_2_tl } { { 4 } { 605170185 } { 988091368 } { 0 } }
\tl_const:cn { c_fp_ln_10_3_tl } { { 6 } { 907755278 } { 982137052 } { 0 } }
\tl_const:cn { c_fp_ln_10_4_tl } { { 9 } { 210340371 } { 976182736 } { 0 } }
\tl_const:cn { c_fp_ln_10_5_tl } { { 1 } { 151292546 } { 497022842 } { 1 } }
\tl_const:cn { c_fp_ln_10_6_tl } { { 1 } { 381551055 } { 796427410 } { 1 } }
\tl_const:cn { c_fp_ln_10_7_tl } { { 1 } { 611809565 } { 095831979 } { 1 } }
\tl_const:cn { c_fp_ln_10_8_tl } { { 1 } { 842068074 } { 395226547 } { 1 } }
\tl_const:cn { c_fp_ln_10_9_tl } { { 2 } { 072326583 } { 694641116 } { 1 } }
```

(*End definition for* `\c_fp_ln_10_1_tl`*. This function is documented on page* **??**.)

`\c_fp_ln_2_1_tl`
`\c_fp_ln_2_2_tl`
`\c_fp_ln_2_3_tl`

The smaller set for powers of two.

```
\tl_const:cn { c_fp_ln_2_1_tl } { { 0 } { 693147180 } { 559945309 } { 0 } }
\tl_const:cn { c_fp_ln_2_2_tl } { { 1 } { 386294361 } { 119890618 } { 0 } }
\tl_const:cn { c_fp_ln_2_3_tl } { { 2 } { 079441541 } { 679835928 } { 0 } }
```

(*End definition for* `\c_fp_ln_2_1_tl`*. This function is documented on page* **??**.)

`\fp_ln:Nn`
`\fp_ln:cn`
`\fp_gln:Nn`
`\fp_gln:cn`
`\fp_ln_aux:NNn`
`\fp_ln_aux:`
`\fp_ln_exponent:`
`\fp_ln_internal:`
`\fp_ln_exponent_tens:`
`\fp_ln_exponent_units:`
`\fp_ln_normalise:`
`\fp_ln_normalise_aux:NNNNNNNNN`
`\fp_ln_mantissa:`
`\fp_ln_mantissa_aux:`
`\fp_ln_mantissa_divide_two:`
`\fp_ln_integer_const:nn`
`\fp_ln_Taylor:`
`\fp_ln_fixed:`
`\fp_ln_fixed_aux:NNNNNNNNN`
`\fp_ln_Taylor_aux:`

The approach for logarithms is again based on a mix of tables and Taylor series. Here, the initial validation is a bit easier and so it is set up earlier, meaning less need to escape later on.

```
\cs_new_protected_nopar:Npn \fp_ln:Nn  { \fp_ln_aux:NNn \tl_set:Nn  }
\cs_new_protected_nopar:Npn \fp_gln:Nn { \fp_ln_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_ln:Nn  { c }
\cs_generate_variant:Nn \fp_gln:Nn { c }
\cs_new_protected:Npn \fp_ln_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
        \if_int_compare:w
          \int_eval:w
            \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
            > \c_zero
          \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_aux:
        \else:
```

```
        \cs_set_protected:Npx \fp_tmp:w ##1##2
          {
            \group_end:
            ##1 \exp_not:N ##2 { \c_zero_fp }
          }
        \exp_after:wN \exp_after:wN \exp_after:wN \fp_ln_error_msg:
      \fi:
    \else:
      \cs_set_protected:Npx \fp_tmp:w ##1##2
        {
          \group_end:
          ##1 \exp_not:N ##2 { \c_zero_fp }
        }
      \exp_after:wN \fp_ln_error_msg:
    \fi:
  \fp_tmp:w #1 #2
}
```

As the input at this stage meets the validity criteria above, the argument can now be saved for further processing. There is no need to look at the sign of the input as it must be positive. The function here simply sets up to either do the full calculation or recover the stored value, as appropriate.

```
\cs_new_protected_nopar:Npn \fp_ln_aux:
  {
    \tl_set:Nx \l_fp_arg_tl
      {
        +
        \int_use:N \l_fp_input_a_integer_int
        .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
             \l_fp_input_a_decimal_int + \c_one_thousand_million
        e
        \int_use:N \l_fp_input_a_exponent_int
      }
    \if_cs_exist:w c_fp_ln ( \l_fp_arg_tl ) _fp \cs_end:
    \else:
      \exp_after:wN \fp_ln_exponent:
    \fi:
    \cs_set_protected:Npx \fp_tmp:w ##1##2
      {
        \group_end:
        ##1 \exp_not:N ##2
          { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
      }
  }
```

The main algorithm here uses the fact the logarithm can be divided up, first taking out the powers of ten, then powers of two and finally using a Taylor series for the remainder.

$$\ln(10^n \times 2^m \times x) = \ln(10^n) + \ln(2^m) + \ln(x)$$

The second point to remember is that

$$\ln(x^{-1}) = -\ln(x)$$

which means that for the powers of 10 and 2 constants are only needed for positive powers.

The first step is to set up the sign for the output functions and work out the powers of ten in the exponent. First the larger powers are sorted out. The values for the constants are the same as those for the smaller ones, just with a shift in the exponent.

```
\cs_new_protected_nopar:Npn \fp_ln_exponent:
  {
    \fp_ln_internal:
    \if_int_compare:w \l_fp_output_extended_int < \c_five_hundred_million
    \else:
      \tex_advance:D \l_fp_output_decimal_int \c_one
      \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
      \else:
        \l_fp_output_decimal_int \c_zero
        \tex_advance:D \l_fp_output_integer_int \c_one
      \fi:
    \fi:
    \fp_standardise:NNNN
      \l_fp_output_sign_int
      \l_fp_output_integer_int
      \l_fp_output_decimal_int
      \l_fp_output_exponent_int
    \tl_const:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
      {
        \if_int_compare:w \l_fp_output_sign_int > \c_zero
          +
        \else:
          -
        \fi:
        \int_use:N \l_fp_output_integer_int
        .
        \exp_after:wN \use_none:n
          \int_value:w \int_eval:w
             \l_fp_output_decimal_int + \c_one_thousand_million
           \scan_stop:
        e
        \int_use:N \l_fp_output_exponent_int
      }
  }
\cs_new_protected_nopar:Npn \fp_ln_internal:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int < \c_zero
      \l_fp_input_a_exponent_int -\l_fp_input_a_exponent_int
      \l_fp_output_sign_int \c_minus_one
    \else:
```

```
      \l_fp_output_sign_int \c_one
    \fi:
    \if_int_compare:w \l_fp_input_a_exponent_int > \c_nine
      \exp_after:wN \fp_ln_exponent_tens:NN
        \int_use:N \l_fp_input_a_exponent_int
    \else:
      \l_fp_output_integer_int  \c_zero
      \l_fp_output_decimal_int  \c_zero
      \l_fp_output_extended_int \c_zero
      \l_fp_output_exponent_int \c_zero
    \fi:
    \fp_ln_exponent_units:
  }
\cs_new_protected:Npn \fp_ln_exponent_tens:NN #1 #2
  {
    \l_fp_input_a_exponent_int #2 \scan_stop:
    \fp_ln_const:nn { 10 } { #1 }
    \tex_advance:D \l_fp_exp_exponent_int \c_one
    \l_fp_output_integer_int  \l_fp_exp_integer_int
    \l_fp_output_decimal_int  \l_fp_exp_decimal_int
    \l_fp_output_extended_int \l_fp_exp_extended_int
    \l_fp_output_exponent_int \l_fp_exp_exponent_int
  }
```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```
\cs_new_protected_nopar:Npn \fp_ln_exponent_units:
  {
    \if_int_compare:w \l_fp_input_a_exponent_int > \c_zero
      \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
      \fp_ln_normalise:
      \fp_add:NNNNNNNNN
        \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
        \l_fp_output_integer_int \l_fp_output_decimal_int
          \l_fp_output_extended_int
    \fi:
    \fp_ln_mantissa:
  }
```

The smaller table-based parts may need to be exponent shifted so that they stay in line with the larger parts. This is similar to the approach in other places, but here there is a need to watch the extended part of the number. The only case where the new exponent is larger than the old is if there was no previous part. Then simply set the exponent.

```
\cs_new_protected_nopar:Npn \fp_ln_normalise:
  {
    \if_int_compare:w \l_fp_exp_exponent_int < \l_fp_output_exponent_int
      \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
```

```
        \fp_ln_normalise_aux:NNNNNNNNN
        \int_use:N \l_fp_exp_decimal_int
       \exp_after:wN \fp_ln_normalise:
    \else:
      \l_fp_output_exponent_int \l_fp_exp_exponent_int
    \fi:
  }
\cs_new_protected:Npn \fp_ln_normalise_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_int_compare:w \l_fp_exp_integer_int = \c_zero
      \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
    \else:
      \tl_set:Nx \l_fp_internal_tl
        {
          \int_use:N \l_fp_exp_integer_int
          #1#2#3#4#5#6#7#8
        }
      \l_fp_exp_integer_int \c_zero
      \l_fp_exp_decimal_int \l_fp_internal_tl \scan_stop:
    \fi:
    \tex_divide:D \l_fp_exp_extended_int \c_ten
    \tl_set:Nx \l_fp_internal_tl
      {
        #9
        \int_use:N \l_fp_exp_extended_int
      }
    \l_fp_exp_extended_int \l_fp_internal_tl \scan_stop:
    \tex_advance:D \l_fp_exp_exponent_int \c_one
  }
```

The next phase is to decompose the mantissa by division by two to leave a value which is in the range $1 \leq x < 2$. The sum of the two powers needs to take account of the sign of the output: if it is negative then the result gets *smaller* as the mantissa gets *bigger*.

```
\cs_new_protected_nopar:Npn \fp_ln_mantissa:
  {
    \l_fp_count_int \c_zero
    \l_fp_input_a_extended_int \c_zero
    \fp_ln_mantissa_aux:
    \if_int_compare:w \l_fp_count_int > \c_zero
      \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
      \fp_ln_normalise:
      \if_int_compare:w \l_fp_output_sign_int > \c_zero
        \exp_after:wN \fp_add:NNNNNNNNN
      \else:
        \exp_after:wN \fp_sub:NNNNNNNNN
      \fi:
      \l_fp_output_integer_int \l_fp_output_decimal_int
        \l_fp_output_extended_int
      \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
      \l_fp_output_integer_int \l_fp_output_decimal_int
```

```
        \l_fp_output_extended_int
    \fi:
    \if_int_compare:w
      \int_eval:w
        \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
      \exp_after:wN \fp_ln_Taylor:
    \fi:
  }
\cs_new_protected_nopar:Npn \fp_ln_mantissa_aux:
  {
    \if_int_compare:w \l_fp_input_a_integer_int > \c_one
      \tex_advance:D \l_fp_count_int \c_one
      \fp_ln_mantissa_divide_two:
      \exp_after:wN \fp_ln_mantissa_aux:
    \fi:
  }
```

A fast one-shot division by two.

```
\cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two:
  {
    \if_int_odd:w \l_fp_input_a_decimal_int
      \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
    \fi:
    \if_int_odd:w \l_fp_input_a_integer_int
      \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
    \fi:
    \tex_divide:D \l_fp_input_a_integer_int  \c_two
    \tex_divide:D \l_fp_input_a_decimal_int  \c_two
    \tex_divide:D \l_fp_input_a_extended_int \c_two
  }
```

Recovering constants makes use of the same auxiliary code as for exponents.

```
\cs_new_protected:Npn \fp_ln_const:nn #1#2
  {
    \exp_after:wN \exp_after:wN \exp_after:wN
      \fp_exp_integer_const:nnnn
      \cs:w c_fp_ln_ #1 _ #2 _tl \cs_end:
  }
```

The Taylor series for the logarithm function is best implemented using the identity

$$\ln(x) = \ln\left(\frac{y+1}{y-1}\right)$$

with

$$y = \frac{x-1}{x+1}$$

This leads to the series

$$\ln(x) = 2y\left(1 + y^2\left(\frac{1}{3} + y^2\left(\frac{1}{5} + y^2\left(\frac{1}{7} + y^2\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

This expansion has the advantage that a lot of the work can be loaded up early by finding y^2 before the loop itself starts. (In practice, the implementation does the multiplication by two at the end of the loop, and expands out the brackets as this is an overall more efficient approach.)

At the implementation level, the code starts by calculating y and storing that in input `a` (which is no longer needed for other purposes). That is done using the full division system avoiding the parsing step. The value is then switched to a fixed-point representation. There is then some shuffling to get all of the working space set up. At this stage, a lot of registers are in use and so the Taylor series is calculated within a group so that the `output` variables can be used to hold the result. The value of y^2 is held in input `b` (there are a few assignments saved by choosing this over `a`), while input `a` is used for the "loop value".

```
\cs_new_protected_nopar:Npn \fp_ln_Taylor:
  {
    \group_begin:
      \l_fp_input_a_integer_int \c_zero
      \l_fp_input_a_exponent_int \c_zero
      \l_fp_input_b_integer_int \c_two
      \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
      \l_fp_input_b_exponent_int \c_zero
      \fp_div_internal:
      \fp_ln_fixed:
      \l_fp_input_a_integer_int  \l_fp_output_integer_int
      \l_fp_input_a_decimal_int  \l_fp_output_decimal_int
      \l_fp_input_a_extended_int \c_zero
      \l_fp_input_a_exponent_int \l_fp_output_exponent_int
      \l_fp_output_decimal_int \c_zero %^^A Bug?
      \l_fp_output_decimal_int  \l_fp_input_a_decimal_int
      \l_fp_output_extended_int \l_fp_input_a_extended_int
      \fp_mul:NNNNNN
        \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
        \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
        \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
      \l_fp_count_int \c_one
      \fp_ln_Taylor_aux:
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          \l_fp_exp_integer_int \c_zero
          \exp_not:N \l_fp_exp_decimal_int
            \int_use:N \l_fp_output_decimal_int \scan_stop:
          \exp_not:N \l_fp_exp_extended_int
            \int_use:N \l_fp_output_extended_int \scan_stop:
          \exp_not:N \l_fp_exp_exponent_int
            \int_use:N \l_fp_output_exponent_int \scan_stop:
        }
    \fp_tmp:w
```

After the loop part of the Taylor series, the factor of 2 needs to be included. The total for the result can then be constructed.

```
    \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
    \if_int_compare:w \l_fp_exp_extended_int < \c_five_hundred_million
    \else:
      \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
      \tex_advance:D \l_fp_exp_decimal_int \c_one
    \fi:
    \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int
    \fp_ln_normalise:
    \if_int_compare:w \l_fp_output_sign_int > \c_zero
      \exp_after:wN \fp_add:NNNNNNNNN
    \else:
      \exp_after:wN \fp_sub:NNNNNNNNN
    \fi:
    \l_fp_output_integer_int \l_fp_output_decimal_int
      \l_fp_output_extended_int
    \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
    \l_fp_output_integer_int \l_fp_output_decimal_int
      \l_fp_output_extended_int
  }
```

The usual shifts to move to fixed-point working. This is done using the `output` registers as this saves a reassignment here.

```
\cs_new_protected_nopar:Npn \fp_ln_fixed:
  {
    \if_int_compare:w \l_fp_output_exponent_int < \c_zero
      \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
      \exp_after:wN \use_i:nn \exp_after:wN
        \fp_ln_fixed_aux:NNNNNNNNN
        \int_use:N \l_fp_output_decimal_int
       \exp_after:wN \fp_ln_fixed:
     \fi:
  }
\cs_new_protected:Npn \fp_ln_fixed_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_int_compare:w \l_fp_output_integer_int = \c_zero
      \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
    \else:
      \tl_set:Nx \l_fp_internal_tl
        {
          \int_use:N \l_fp_output_integer_int
          #1#2#3#4#5#6#7#8
        }
      \l_fp_output_integer_int \c_zero
      \l_fp_output_decimal_int \l_fp_internal_tl \scan_stop:
    \fi:
    \tex_advance:D \l_fp_output_exponent_int \c_one
  }
```

The main loop for the Taylor series: unlike some of the other similar functions, the result here is not the final value and is therefore subject to further manipulation outside of the loop.

```
\cs_new_protected_nopar:Npn \fp_ln_Taylor_aux:
  {
    \tex_advance:D \l_fp_count_int \c_two
    \fp_mul:NNNNNN
      \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
      \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
      \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
    \if_int_compare:w
      \int_eval:w
        \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
        > \c_zero
      \fp_div_integer:NNNNN
        \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
        \l_fp_count_int
        \l_fp_exp_decimal_int \l_fp_exp_extended_int
        \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
        \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
        \if_int_compare:w \l_fp_output_extended_int < \c_one_thousand_million
        \else:
          \tex_advance:D \l_fp_output_decimal_int \c_one
          \tex_advance:D \l_fp_output_extended_int
            -\c_one_thousand_million
        \fi:
        \if_int_compare:w \l_fp_output_decimal_int < \c_one_thousand_million
        \else:
          \tex_advance:D \l_fp_output_integer_int \c_one
          \tex_advance:D \l_fp_output_decimal_int
            -\c_one_thousand_million
        \fi:
      \exp_after:wN \fp_ln_Taylor_aux:
    \fi:
  }
```

(*End definition for* `\fp_ln:Nn` *and* `\fp_ln:cn`*. These functions are documented on page* **??**.)

\fp_pow:Nn
\fp_pow:cn
\fp_gpow:Nn
\fp_gpow:cn
\fp_pow_aux:NNn
\fp_pow_aux_i:
\fp_pow_positive:
\fp_pow_negative:
\fp_pow_aux_ii:
\fp_pow_aux_iii:
\fp_pow_aux_iv:

The approach used for working out powers is to first filter out the various special cases and then do most of the work using the logarithm and exponent functions. The two storage areas are used in the reverse of the 'natural' logic as this avoids some re-assignment in the sanity checking code.

```
\cs_new_protected_nopar:Npn \fp_pow:Nn  { \fp_pow_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \fp_gpow:Nn { \fp_pow_aux:NNn \tl_gset:Nn }
\cs_generate_variant:Nn \fp_pow:Nn  { c }
\cs_generate_variant:Nn \fp_gpow:Nn { c }
\cs_new_protected:Npn \fp_pow_aux:NNn #1#2#3
  {
    \group_begin:
      \fp_read:N #2
      \l_fp_input_b_sign_int     \l_fp_input_a_sign_int
```

```
      \l_fp_input_b_integer_int  \l_fp_input_a_integer_int
      \l_fp_input_b_decimal_int  \l_fp_input_a_decimal_int
      \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
      \fp_split:Nn a {#3}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \if_int_compare:w
        \int_eval:w
          \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
          = \c_zero
         \if_int_compare:w
           \int_eval:w
             \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
             = \c_zero
             \cs_set_protected:Npx \fp_tmp:w ##1##2
               {
                 \group_end:
                 ##1 ##2 { \c_undefined_fp }
               }
           \else:
             \cs_set_protected:Npx \fp_tmp:w ##1##2
               {
                 \group_end:
                 ##1 ##2 { \c_zero_fp }
               }
          \fi:
       \else:
         \if_int_compare:w
           \int_eval:w
             \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
             = \c_zero
             \cs_set_protected:Npx \fp_tmp:w ##1##2
               {
                 \group_end:
                 ##1 ##2 { \c_one_fp }
               }
           \else:
             \exp_after:wN \exp_after:wN \exp_after:wN
               \fp_pow_aux_i:
          \fi:
       \fi:
    \fp_tmp:w #1 #2
}
```

Simply using the logarithm function directly will fail when negative numbers are raised to integer powers, which is a mathematically valid operation. So there are some more tests to make, after forcing the power into an integer and decimal parts, if necessary.

```
\cs_new_protected_nopar:Npn \fp_pow_aux_i:
  {
    \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
      \tl_set:Nn \l_fp_sign_tl { + }
      \exp_after:wN \fp_pow_aux_ii:
    \else:
      \l_fp_input_a_extended_int \c_zero
      \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
        \group_begin:
        \fp_extended_normalise:
        \if_int_compare:w
          \int_eval:w
            \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
            = \c_zero
           \group_end:
          \tl_set:Nn \l_fp_sign_tl { - }
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \exp_after:wN \exp_after:wN
          \exp_after:wN \fp_pow_aux_ii:
        \else:
          \group_end:
          \cs_set_protected:Npx \fp_tmp:w ##1##2
            {
              \group_end:
              ##1 ##2 { \c_undefined_fp }
           }
        \fi:
      \else:
        \cs_set_protected:Npx \fp_tmp:w ##1##2
          {
            \group_end:
            ##1 ##2 { \c_undefined_fp }
         }
      \fi:
    \fi:
  }
```

The approach used here for powers works well in most cases but gives poorer results for negative integer powers, which often have exact values. So there is some filtering to do. For negative powers where the power is small, an alternative approach is used in which the positive value is worked out and the reciprocal is then taken. The filtering is unfortunately rather long.

```
\cs_new_protected_nopar:Npn \fp_pow_aux_ii:
  {
    \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
      \exp_after:wN \fp_pow_aux_iv:
    \else:
      \if_int_compare:w \l_fp_input_a_exponent_int < \c_ten
        \group_begin:
        \l_fp_input_a_extended_int \c_zero
```

```
        \fp_extended_normalise:
        \if_int_compare:w \l_fp_input_a_decimal_int = \c_zero
          \if_int_compare:w \l_fp_input_a_integer_int > \c_ten
            \group_end:
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
              \fp_pow_aux_iv:
          \else:
            \group_end:
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
              \exp_after:wN \fp_pow_aux_iii:
          \fi:
        \else:
          \group_end:
            \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN
              \exp_after:wN \fp_pow_aux_iv:
        \fi:
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN
          \fp_pow_aux_iv:
      \fi:
    \fi:
    \cs_set_protected:Npx \fp_tmp:w ##1##2
      {
        \group_end:
        ##1 ##2
          {
            \l_fp_sign_tl
            \int_use:N \l_fp_output_integer_int
            .
            \exp_after:wN \use_none:n
              \int_value:w \int_eval:w
                 \l_fp_output_decimal_int + \c_one_thousand_million
            e
            \int_use:N \l_fp_output_exponent_int
          }
      }
  }
```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```
\cs_new_protected_nopar:Npn \fp_pow_aux_iii:
```

```
  {
    \l_fp_input_a_sign_int \c_one
    \fp_pow_aux_iv:
    \l_fp_input_a_integer_int  \c_one
    \l_fp_input_a_decimal_int  \c_zero
    \l_fp_input_a_exponent_int \c_zero
    \l_fp_input_b_integer_int  \l_fp_output_integer_int
    \l_fp_input_b_decimal_int  \l_fp_output_decimal_int
    \l_fp_input_b_exponent_int \l_fp_output_exponent_int
    \fp_div_internal:
  }
```

The business end of the code starts by finding the logarithm of the given base. There is a bit of a shuffle so that this does not have to be re-parsed and so that the output ends up in the correct place. There is also a need to enable using the short-cut for a pre-calculated result. The internal part of the multiplication function can then be used to do the second part of the calculation directly. There is some more set up before doing the exponential: the idea here is to deactivate some internals so that everything works smoothly.

```
\cs_new_protected_nopar:Npn \fp_pow_aux_iv:
  {
    \group_begin:
      \l_fp_input_a_integer_int  \l_fp_input_b_integer_int
      \l_fp_input_a_decimal_int  \l_fp_input_b_decimal_int
      \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
      \fp_ln_internal:
      \cs_set_protected_nopar:Npx \fp_tmp:w
        {
          \group_end:
          \exp_not:N \l_fp_input_b_sign_int
            \int_use:N \l_fp_output_sign_int \scan_stop:
          \exp_not:N \l_fp_input_b_integer_int
            \int_use:N \l_fp_output_integer_int \scan_stop:
          \exp_not:N \l_fp_input_b_decimal_int
            \int_use:N \l_fp_output_decimal_int \scan_stop:
          \exp_not:N \l_fp_input_b_extended_int
            \int_use:N \l_fp_output_extended_int \scan_stop:
          \exp_not:N \l_fp_input_b_exponent_int
            \int_use:N \l_fp_output_exponent_int \scan_stop:
        }
    \fp_tmp:w
    \l_fp_input_a_extended_int  \c_zero
    \fp_mul:NNNNNNNNN
      \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
        \l_fp_input_a_extended_int
      \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
        \l_fp_input_b_extended_int
      \l_fp_output_integer_int \l_fp_output_decimal_int
        \l_fp_output_extended_int
    \l_fp_output_exponent_int
```

```
      \int_eval:w
        \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
      \scan_stop:
    \fp_extended_normalise_output:
    \tex_multiply:D \l_fp_input_a_sign_int \l_fp_input_b_sign_int
    \l_fp_input_a_integer_int  \l_fp_output_integer_int
    \l_fp_input_a_decimal_int  \l_fp_output_decimal_int
    \l_fp_input_a_extended_int \l_fp_output_extended_int
    \l_fp_input_a_exponent_int \l_fp_output_exponent_int
    \l_fp_output_integer_int  \c_zero
    \l_fp_output_decimal_int  \c_zero
    \l_fp_output_extended_int \c_zero
    \l_fp_output_exponent_int \c_zero
    \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
    \fp_exp_internal:
  }
```

(*End definition for* `\fp_pow:Nn` *and* `\fp_pow:cn`*. These functions are documented on page* **??**.)

202.13 Tests for special values

\fp_if_undefined_p:N
\fp_if_undefined:N*TF*

Testing for an undefined value is easy.

```
\prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_undefined_fp
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\fp_if_undefined:N`*. These functions are documented on page 167.*)

\fp_if_zero_p:N
\fp_if_zero:N*TF*

Testing for a zero fixed-point is also easy.

```
\prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_zero_fp
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\fp_if_zero:N`*. These functions are documented on page 167.*)

202.14 Floating-point conditionals

\fp_compare:nNn*TF*
\fp_compare:NNN*TF*
\fp_compare_aux:N
\fp_compare_=:
\fp_compare_<:
\fp_compare_<_aux:
\fp_compare_absolute_a>b:
\fp_compare_absolute_a<b:
\fp_compare_>:

The idea for the comparisons is to provide two versions: slower and faster. The lead off for both is the same: get the two numbers read and then look for a function to handle the comparison.

```
\prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3 { T , F , TF }
```

```
  {
    \group_begin:
      \fp_split:Nn a {#1}
      \fp_standardise:NNNN
        \l_fp_input_a_sign_int
        \l_fp_input_a_integer_int
        \l_fp_input_a_decimal_int
        \l_fp_input_a_exponent_int
      \fp_split:Nn b {#3}
      \fp_standardise:NNNN
        \l_fp_input_b_sign_int
        \l_fp_input_b_integer_int
        \l_fp_input_b_decimal_int
        \l_fp_input_b_exponent_int
      \fp_compare_aux:N #2
  }
\prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3 { T , F , TF }
  {
    \group_begin:
      \fp_read:N #3
      \l_fp_input_b_sign_int     \l_fp_input_a_sign_int
      \l_fp_input_b_integer_int  \l_fp_input_a_integer_int
      \l_fp_input_b_decimal_int  \l_fp_input_a_decimal_int
      \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
      \fp_read:N #1
      \fp_compare_aux:N #2
  }
\cs_new_protected:Npn \fp_compare_aux:N #1
  {
    \cs_if_exist:cTF { fp_compare_#1: }
      { \use:c { fp_compare_#1: } }
      {
        \group_end:
        \prg_return_false:
      }
  }
```

For equality, the test is pretty easy as things are either equal or they are not.

```
\cs_new_protected_nopar:cpn { fp_compare_=: }
  {
    \if_int_compare:w \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
      \if_int_compare:w \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
        \if_int_compare:w \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
          \if_int_compare:w
            \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
            \group_end:
            \prg_return_true:
          \else:
            \group_end:
            \prg_return_false:
```

```
          \fi:
        \else:
          \group_end:
          \prg_return_false:
        \fi:
      \else:
        \group_end:
        \prg_return_false:
      \fi:
    \else:
      \group_end:
      \prg_return_false:
    \fi:
  }
```

Comparing two values is quite complex. First, there is a filter step to check if one or other of the given values is zero. If it is then the result is relatively easy to determine.

```
\cs_new_protected_nopar:cpn { fp_compare_>: }
  {
    \if_int_compare:w \int_eval:w
      \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
      = \c_zero
      \if_int_compare:w \int_eval:w
        \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
        = \c_zero
        \group_end:
        \prg_return_false:
      \else:
        \if_int_compare:w \l_fp_input_b_sign_int > \c_zero
          \group_end:
          \prg_return_false:
        \else:
          \group_end:
          \prg_return_true:
        \fi:
      \fi:
    \else:
      \if_int_compare:w \int_eval:w
        \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
        = \c_zero
        \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
          \group_end:
          \prg_return_true:
        \else:
          \group_end:
          \prg_return_false:
        \fi:
      \else:
        \use:c { fp_compare_>_aux: }
      \fi:
```

```
    \fi:
  }
```

Next, check the sign of the input: this again may give an obvious result. If both signs are the same, then hand off to comparing the absolute values.

```
\cs_new_protected_nopar:cpn { fp_compare_>_aux: }
  {
    \if_int_compare:w \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
      \group_end:
      \prg_return_true:
    \else:
      \if_int_compare:w \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
        \group_end:
        \prg_return_false:
      \else:
        \if_int_compare:w \l_fp_input_a_sign_int > \c_zero
          \use:c { fp_compare_absolute_a>b: }
        \else:
          \use:c { fp_compare_absolute_a<b: }
        \fi:
      \fi:
    \fi:
  }
```

Rather long runs of checks, as there is the need to go through each layer of the input and do the comparison. There is also the need to avoid messing up with equal inputs at each stage.

```
\cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: }
  {
    \if_int_compare:w \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
      \group_end:
      \prg_return_true:
    \else:
      \if_int_compare:w \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
        \group_end:
        \prg_return_false:
      \else:
        \if_int_compare:w \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
          \group_end:
          \prg_return_true:
        \else:
          \if_int_compare:w
            \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
            \group_end:
            \prg_return_false:
          \else:
            \if_int_compare:w
              \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
              \group_end:
              \prg_return_true:
```

```
            \else:
              \group_end:
              \prg_return_false:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
\cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: }
  {
    \if_int_compare:w \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
      \group_end:
      \prg_return_true:
    \else:
      \if_int_compare:w \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
        \group_end:
        \prg_return_false:
      \else:
        \if_int_compare:w \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
          \group_end:
          \prg_return_true:
        \else:
          \if_int_compare:w
            \l_fp_input_b_integer_int < \l_fp_input_a_integer_int
            \group_end:
            \prg_return_false:
          \else:
            \if_int_compare:w
              \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int
              \group_end:
              \prg_return_true:
            \else:
              \group_end:
              \prg_return_false:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
```

This is just a case of reversing the two input values and then running the tests already defined.

```
\cs_new_protected_nopar:cpn { fp_compare_<: }
  {
    \tl_set:Nx \l_fp_internal_tl
      {
        \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
          { \int_use:N \l_fp_input_b_sign_int }
```

```
        \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
          { \int_use:N \l_fp_input_b_integer_int }
        \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
          { \int_use:N \l_fp_input_b_decimal_int }
        \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
          { \int_use:N \l_fp_input_b_exponent_int }
        \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
          { \int_use:N \l_fp_input_a_sign_int }
        \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
          { \int_use:N \l_fp_input_a_integer_int }
        \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
          { \int_use:N \l_fp_input_a_decimal_int }
        \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
          { \int_use:N \l_fp_input_a_exponent_int }
      }
    \l_fp_internal_tl
    \use:c { fp_compare_>: }
  }
```

(*End definition for* `\fp_compare:nNn`. *This function is documented on page* **??**.)

`\fp_compare:nTF`
`\fp_compare_aux_i:w`
`\fp_compare_aux_ii:w`
`\fp_compare_aux_iii:w`
`\fp_compare_aux_iv:w`
`\fp_compare_aux_v:w`
`\fp_compare_aux_vi:w`
`\fp_compare_aux_vii:w`

As TeX cannot help out here, a daisy-chain of delimited functions are used. This is very much a first-generation approach: revision will be needed if these functions are really useful.

```
\prg_new_protected_conditional:Npnn \fp_compare:n #1 { T , F , TF }
  {
    \group_begin:
      \tl_set:Nx \l_fp_internal_tl
        {
          \group_end:
          \fp_compare_aux_i:w #1 \exp_not:n { == \q_nil == \q_stop }
        }
     \l_fp_internal_tl
  }
\cs_new_protected:Npn \fp_compare_aux_i:w #1 == #2 == #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_ii:w #1 != \q_nil != \q_stop }
      { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
  }
\cs_new_protected:Npn \fp_compare_aux_ii:w #1 != #2 != #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_iii:w #1 <= \q_nil <= \q_stop }
      { \fp_compare:nNnTF {#1} = {#2} \prg_return_false: \prg_return_true: }
  }
\cs_new_protected:Npn \fp_compare_aux_iii:w #1 <= #2 <= #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_iv:w #1 >= \q_nil >= \q_stop }
```

```
      { \fp_compare:nNnTF {#1} > {#2} \prg_return_false: \prg_return_true: }
  }
\cs_new_protected:Npn \fp_compare_aux_iv:w #1 >= #2 >= #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_v:w #1 = \q_nil  \q_stop }
      { \fp_compare:nNnTF {#1} < {#2} \prg_return_false: \prg_return_true: }
  }
\cs_new_protected:Npn \fp_compare_aux_v:w #1 = #2 = #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_vi:w #1 < \q_nil < \q_stop }
      { \fp_compare:nNnTF {#1} = {#2} \prg_return_true: \prg_return_false: }
  }
\cs_new_protected:Npn \fp_compare_aux_vi:w #1 < #2 < #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \fp_compare_aux_vii:w #1 > \q_nil > \q_stop }
      { \fp_compare:nNnTF {#1} < {#2} \prg_return_true: \prg_return_false: }
  }
\cs_new_protected:Npn \fp_compare_aux_vii:w #1 > #2 > #3 \q_stop
  {
    \quark_if_nil:nTF {#2}
      { \prg_return_false: }
      { \fp_compare:nNnTF {#1} > {#2} \prg_return_true: \prg_return_false: }
  }
```

(*End definition for* `\fp_compare:n`. *This function is documented on page 167.*)

202.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```
\msg_kernel_new:nnnn { fpu } { overflow }
  { Number~too~big. }
  {
    The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \\
    Further~errors~may~well~occur!
  }
\cs_new_protected_nopar:Npn \fp_overflow_msg:
  { \msg_kernel_error:nn { fpu } { overflow } }
```

(*End definition for* `\fp_overflow_msg:`. *This function is documented on page* **??**.)

`\fp_exp_overflow_msg:` A slightly more helpful message for exponent overflows.

```
\msg_kernel_new:nnnn { fpu } { exponent-overflow }
  { Number~too~big~for~exponent~unit. }
  {
    The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
    unit:~the~maximum~input~value~for~an~exponent~is~230.
  }
\cs_new_protected_nopar:Npn \fp_exp_overflow_msg:
```

```
  { \msg_kernel_error:nn { fpu } { exponent-overflow } }
```

(*End definition for* `\fp_exp_overflow_msg:`. *This function is documented on page* **??**.)

`\fp_ln_error_msg:`

Logarithms are only valid for positive number

```
\msg_kernel_new:nnnn { fpu } { logarithm-input-error }
  { Invalid~input~to~ln~function. }
  { Logarithms~can~only~be~calculated~for~positive~numbers. }
\cs_new_protected_nopar:Npn \fp_ln_error_msg: {
  \msg_kernel_error:nn { fpu } { logarithm-input-error }
}
```

(*End definition for* `\fp_ln_error_msg:`. *This function is documented on page* **??**.)

`\fp_trig_overflow_msg:`

A slightly more helpful message for trigonometric overflows.

```
\msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
  { Number~too~big~for~trigonometry~unit. }
  {
    The~trigonometry~code~can~only~work~with~numbers~smaller~
    than~1000000000.
  }
\cs_new_protected_nopar:Npn \fp_trig_overflow_msg:
  { \msg_kernel_error:nn { fpu } { trigonometric-overflow } }
```

(*End definition for* `\fp_trig_overflow_msg:`. *This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

203 l3luatex implementation

```
⟨*initex | package⟩
```

Announce and ensure that the required packages are loaded.

```
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\package_check_loaded_expl:
⟨/package⟩
```

An error message.

```
\msg_kernel_new:nnnn { luatex } { bad-engine }
  { LuaTeX~engine~not~in~use!~Ignoring~#1. }
  {
    The~feature~you~are~using~is~only~available~
    with~the~LuaTeX~engine.~LaTeX3~ignored~'#1#2'.
  }
```

`\lua_now:n`
`\lua_now:x`
`\lua_shipout_x:n`
`\lua_shipout_x:x`
`\lua_shipout:n`
`\lua_shipout:x`

When LuaTeX is in use, this is all a question of primitives with new names. On the other hand, for pdfTeX and XeTeX the argument should be removed from the input stream before issuing an error. This is expandable, using `\msg_expandable_kernel_error:nnn` as done for V-type expansion in l3expan.

```
\luatex_if_engine:TF
  {
```

```
    \cs_new_eq:NN \lua_now:x       \luatex_directlua:D
    \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
  }
  {
    \cs_new:Npn \lua_now:x #1
      {
        \msg_expandable_kernel_error:nnn
          { luatex } { bad-engine } { \lua_now:x }
      }
    \cs_new_protected:Npn \lua_shipout_x:n #1
      {
        \msg_expandable_kernel_error:nnn
          { luatex } { bad-engine } { \lua_shipout_x:n }
      }
  }
\cs_new:Npn \lua_now:n #1
  { \lua_now:x { \exp_not:n {#1} } }
\cs_generate_variant:Nn \lua_shipout_x:n { x }
\cs_new_protected:Npn \lua_shipout:n #1
  { \lua_shipout_x:n { \exp_not:n {#1} } }
\cs_generate_variant:Nn \lua_shipout:n { x }
```

(*End definition for* `\lua_now:n` *and* `\lua_now:x`*. These functions are documented on page* **??**.)

203.1 Category code tables

`\g_cctab_allocate_int`
`\g_cctab_stack_int`
`\g_cctab_stack_seq`

To allocate category code tables, both the read-only and stack tables need to be followed. There is also a sequence stack for the dynamic tables themselves.

```
\int_new:N  \g_cctab_allocate_int
\int_set:Nn \g_cctab_allocate_int { \c_minus_one }
\int_new:N \g_cctab_stack_int
\seq_new:N \g_cctab_stack_seq
```

(*End definition for* `\g_cctab_allocate_int`*. This function is documented on page* **??**.)

`\cctab_new:N`

Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```
\cs_new_protected:Npn \cctab_new:N #1
  {
    \chk_if_free_cs:N #1
    \int_gadd:Nn \g_cctab_allocate_int { \c_two }
    \int_compare:nNnTF
      \g_cctab_allocate_int <  { \c_max_register_int + \c_one }
       {
         \tex_global:D \tex_chardef:D #1 \g_cctab_allocate_int
         \luatex_initcatcodetable:D #1
       }
```

```
      { \msg_kernel_fatal:nnx { kernel } { out-of-registers } { cctab } }
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_new:N #1
      {
        \msg_kernel_error:nnx { luatex } { bad-engine }
          { \exp_not:N \cctab_new:N }
      }
  }
⟨*package⟩
\luatex_if_engine:T
  {
    \cs_set_protected:Npn \cctab_new:N #1
      {
        \chk_if_free_cs:N #1
        \newcatcodetable #1
        \luatex_initcatcodetable:D #1
      }
  }
⟨/package⟩
```

(End definition for `\cctab_new:N`*. This function is documented on page 172.)*

\cctab_begin:N
\cctab_end:
\l_cctab_internal_tl

The aim here is to ensure that the saved tables are read-only. This is done by using a stack of tables which are not read only, and actually having them as "in use" copies.

```
\cs_new_protected:Npn \cctab_begin:N #1
  {
    \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
    \luatex_catcodetable:D #1
    \int_gadd:Nn \g_cctab_stack_int { \c_two }
    \int_compare:nNnT \g_cctab_stack_int > \c_max_register_int
      { \msg_kernel_fatal:nn { code } { cctab-stack-full } }
    \luatex_savecatcodetable:D \g_cctab_stack_int
    \luatex_catcodetable:D \g_cctab_stack_int
  }
\cs_new_protected_nopar:Npn \cctab_end:
  {
    \int_gsub:Nn \g_cctab_stack_int { \c_two }
    \seq_if_empty:NTF \g_cctab_stack_seq
      { \tl_set:Nn \l_cctab_internal_tl { 0 } }
      { \seq_gpop:NN \g_cctab_stack_seq \l_cctab_internal_tl }
    \luatex_catcodetable:D \l_cctab_internal_tl \scan_stop:
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_begin:N #1
      {
        \msg_kernel_error:nnxx { luatex } { bad-engine }
          { \exp_not:N \cctab_begin:N } {#1}
```

```
      }
    \cs_set_protected_nopar:Npn \cctab_end:
      {
        \msg_kernel_error:nnx { luatex } { bad-engine }
          { \exp_not:N \cctab_end: }
      }
  }
⟨*package⟩
\luatex_if_engine:T
  {
    \cs_set_protected:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
    \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
  }
⟨/package⟩
\tl_new:N \l_cctab_internal_tl
```

(*End definition for* `\cctab_begin:N`. *This function is documented on page* **??**.)

`\cctab_gset:Nn`

Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```
\cs_new_protected:Npn \cctab_gset:Nn #1#2
  {
    \group_begin:
      #2
      \luatex_savecatcodetable:D #1
    \group_end:
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_gset:Nn #1#2
      {
        \msg_kernel_error:nnxx { luatex } { bad-engine }
          { \exp_not:N \cctab_gset:Nn } { #1 {#2} }
      }
  }
```

(*End definition for* `\cctab_gset:Nn`. *This function is documented on page* 172.)

`\c_code_cctab`
`\c_document_cctab`
`\c_initex_cctab`
`\c_other_cctab`
`\c_str_cctab`

Creating category code tables is easy using the function above. The `other` and `string` ones are done by completely ignoring the existing codes as this makes life a lot less complex. The table for expl3 category codes is always needed, whereas when in package mode the rest can be copied from the existing LaTeX 2ε package luatex.

```
\luatex_if_engine:T
  {
    \cctab_new:N \c_code_cctab
    \cctab_gset:Nn \c_code_cctab { }
  }
⟨*package⟩
\luatex_if_engine:T
  {
```

```
    \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
    \cs_new_eq:NN \c_initex_cctab   \CatcodeTableIniTeX
    \cs_new_eq:NN \c_other_cctab    \CatcodeTableOther
    \cs_new_eq:NN \c_str_cctab      \CatcodeTableString
  }
⟨/package⟩
⟨*initex⟩
\luatex_if_engine:T
  {
    \cctab_new:N \c_document_cctab
    \cctab_new:N \c_other_cctab
    \cctab_new:N \c_str_cctab
    \cctab_gset:Nn \c_document_cctab
      {
        \char_set_catcode_space:n          { 9 }
        \char_set_catcode_space:n          { 32 }
        \char_set_catcode_other:n          { 58 }
        \char_set_catcode_math_subscript:n { 95 }
        \char_set_catcode_active:n         { 126 }
      }
    \cctab_gset:Nn \c_other_cctab
      {
        \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
          { \char_set_catcode_other:n {#1} }
      }
    \cctab_gset:Nn \c_str_cctab
      {
        \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
          { \char_set_catcode_other:n {#1} }
        \char_set_catcode_space:n { 32 }
      }
  }
⟨/initex⟩
```

(*End definition for* `\c_code_cctab`. *This function is documented on page* 173.)

203.2 Deprecated functions

Deprecated 2011-12-21, for removal by 2012-03-31.

`\c_string_cctab`

```
\cs_new_eq:NN \c_string_cctab \c_str_cctab
```

(*End definition for* `\c_string_cctab`. *This variable is documented on page* **??**.)

```
⟨/initex | package⟩
```

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

M

Q

R

S

T

V

W

X

Y

Z