The LaTeX3 Interfaces

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\ExplSyntaxOn ⟨code⟩ \ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code régime in which spaces are ignored and in which the colon (`:`) and underscore (`_`) are treated as "letters", thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code 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 2ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as LaTeX 2ε provides in turning on `\makeatletter` within package and class code.)

`\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 2_ε 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    {`⟨$group_1$⟩`}`
`\use:nn   {`⟨$group_1$⟩`} {`⟨$group_2$⟩`}`
`\use:nnn  {`⟨$group_1$⟩`} {`⟨$group_2$⟩`} {`⟨$group_3$⟩`}`
`\use:nnnn {`⟨$group_1$⟩`} {`⟨$group_2$⟩`} {`⟨$group_3$⟩`} {`⟨$group_4$⟩`}`

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 {⟨arg₁⟩} {⟨arg₂⟩}`

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 {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}`

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 {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩}`

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 {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}`

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 {⟨group₁⟩}`

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:N*TF* ⋆
\cs_if_exist_use:c*TF* ⋆

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:NTF ⋆
\bool_if:cTF ⋆

\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:NTF ⋆
\bool_if_exist:cTF ⋆

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_lccode: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 {⟨intexpr_1⟩} {⟨intexpr_2⟩}`

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:nNnTF` ★

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

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

Equal	`=`
Greater than	`>`
Less than	`<`

`\int_compare_p:n` ★
`\int_compare:nTF` ★

```
\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:nTF` ★
`\int_if_odd_p:n` ★
`\int_if_odd:nTF` ★

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

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

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:nn*TF* ⋆

```
\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:n*TF* ⋆

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:NNTF ★
\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:nnTF

`\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:NnTF
\tl_if_in:cnTF

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

Tests if the ⟨*token list*⟩ is found in the content of the ⟨*token list variable*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (assuming the usual TeX category codes apply).

\tl_if_in:nnTF
\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:NTF ★
\tl_if_single:cTF ★

Updated: 2011-08-13

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

Tests if the content of the ⟨*tl var*⟩ consists of a single item, *i.e.* is 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:nTF ★

Updated: 2011-08-13

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

Tests if the token list has exactly one item, *i.e.* is 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:nTF ★

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:nN*TF* ⋆

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:NTF ⋆
\seq_if_exist:cTF ⋆

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:NN*TF*
\seq_get_right:cN*TF*

`\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:NN*TF*
\seq_pop_left:cN*TF*

`\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:NN*TF*
\seq_gpop_left:cN*TF*

`\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:NN*TF*
\seq_pop_right:cN*TF*

`\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:NN*TF*
\seq_gpop_right:cN*TF*

`\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:NTF ⋆
\clist_if_empty:cTF ⋆

\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:NN*TF* ⋆
\clist_if_eq:(Nc|cN|cc)*TF* ⋆

\clist_if_eq_p:NN ⟨*clist*$_1$⟩ ⟨*clist*$_2$⟩
\clist_if_eq:NNTF ⟨*clist*$_1$⟩ ⟨*clist*$_2$⟩ {⟨*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:Nn*TF*
\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`

New: 2011-08-28

⟨*key*⟩ `.bool_set_inverse:N = ⟨boolean⟩`

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to the logical inverse of ⟨*value*⟩ (which must be either `true` or `false`). If the ⟨*boolean*⟩ does not exist, it will be created 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`

New: 2011-08-21

⟨*key*⟩ `.choices:nn ⟨choices⟩ ⟨code⟩`

Sets ⟨*key*⟩ to act as a choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 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`

New: 2011/09/11

⟨*key*⟩ `.clist_set:N = ⟨comma list variable⟩`

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`

New: 2011/09/11

⟨*key*⟩ `.clist_gset:N = ⟨comma list variable⟩`

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:` ⟨*key*⟩ `.multichoice:`

New: 2011-08-21

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` ⟨*key*⟩ `.multichoice:nn` ⟨*choices*⟩ ⟨*code*⟩

New: 2011-08-21

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:N*TF* ⋆

Updated: 2012-02-10

\ior_if_eof_p:N ⟨*stream*⟩
\ior_if_eof:NTF ⟨*stream*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test 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).

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.

B

D

E

Q

R

S

U

V

X