The LaTeX3 Interfaces

The LaTeX3 Project*

November 21, 2016

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 2ε. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2ε 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

I Introduction to expl3 and this document 1

1 Naming functions and variables 1
1.1 Terminological inexactitude . . . 3

2 Documentation conventions 3

3 Formal language conventions which apply generally 5

4 TEX concepts not supported by LATEX3 5

II The l3bootstrap package: Bootstrap code 6

1 Using the LATEX3 modules 6
1.1 Internal functions and variables . . . 7

III The l3names package: Namespace for primitives 8

1 Setting up the LATEX3 programming language 8

IV The l3basics package: Basic definitions 9

1 No operation functions 9

2 Grouping material 9

3 Control sequences and functions 10
3.1 Defining functions . . . 10
3.2 Defining new functions using parameter text . . . 11
3.3 Defining new functions using the signature . . . 12
3.4 Copying control sequences . . . 15
3.5 Deleting control sequences . . . 15
3.6 Showing control sequences . . . 15
3.7 Converting to and from control sequences . . . 16

4 Using or removing tokens and arguments 17
4.1 Selecting tokens from delimited arguments . . . 18

5 Predicates and conditionals 19
5.1 Tests on control sequences . . . 20
5.2 Primitive conditionals . . . 20

6 Internal kernel functions 21

V The l3expan package: Argument expansion 24

1 Defining new variants 24
2 Methods for defining variants 25
3 Introducing the variants 25
4 Manipulating the first argument 26
5 Manipulating two arguments 28
6 Manipulating three arguments 28
7 Unbraced expansion 29
8 Preventing expansion 30
9 Controlled expansion 31
10 Internal functions and variables 32

VI The l3prg package: Control structures 34
1 Defining a set of conditional functions 34
2 The boolean data type 36
3 Boolean expressions 37
4 Logical loops 38
5 Producing multiple copies 39
6 Detecting TEX's mode 39
7 Primitive conditionals 40
8 Internal programming functions 40

VII The l3quark package: Quarks 42
1 Introduction to quarks and scan marks 42
1.1 Quarks 42
2 Defining quarks 42
3 Quark tests 43
4 Recursion 43
5 An example of recursion with quarks 44
6 Internal quark functions 45

7 Scan marks 45

VIII The l3token package: Token manipulation 47

1 Creating character tokens 47

2 Manipulating and interrogating character tokens 49

3 Generic tokens 52

4 Converting tokens 52

5 Token conditionals 53

6 Peeking ahead at the next token 56

7 Decomposing a macro definition 58

8 Description of all possible tokens 59

9 Internal functions 61

IX The l3int package: Integers 62

1 Integer expressions 62

2 Creating and initialising integers 63

3 Setting and incrementing integers 64

4 Using integers 64

5 Integer expression conditionals 65

6 Integer expression loops 66

7 Integer step functions 68

8 Formatting integers 68

9 Converting from other formats to integers 70

10 Viewing integers 71

11 Constant integers 72

12 Scratch integers 72

13 Primitive conditionals 73

14 Internal functions 73

X The l3skip package: Dimensions and skips 75
1 Creating and initialising dim variables 75
2 Setting dim variables 76
3 Utilities for dimension calculations 76
4 Dimension expression conditionals 77
5 Dimension expression loops 79
6 Using dim expressions and variables 80
7 Viewing dim variables 82
8 Constant dimensions 82
9 Scratch dimensions 82
10 Creating and initialising skip variables 82
11 Setting skip variables 83
12 Skip expression conditionals 84
13 Using skip expressions and variables 84
14 Viewing skip variables 84
15 Constant skips 85
16 Scratch skips 85
17 Inserting skips into the output 85
18 Creating and initialising muskip variables 85
19 Setting muskip variables 86
20 Using muskip expressions and variables 87
21 Viewing muskip variables 87
22 Constant muskips 87
23 Scratch muskips 87
24 Primitive conditional 88
25 Internal functions 88

XI The l3tl package: Token lists 89

1 Creating and initialising token list variables 89
2 Adding data to token list variables 90
3 Modifying token list variables 91
4 Reassigning token list category codes 91
5 Token list conditionals 92
6 Mapping to token lists 94
7 Using token lists 96
8 Working with the content of token lists 96
9 The first token from a token list 98
10 Using a single item 100
11 Viewing token lists 100
12 Constant token lists 101
13 Scratch token lists 101
14 Internal functions 101

XII The l3str package:Strings 102
1 Building strings 102
2 Adding data to string variables 103
2.1 String conditionals 103
3 Working with the content of strings 105
4 String manipulation 108
5 Viewing strings 109
6 Constant token lists 110
7 Scratch strings 110
7.1 Internal string functions 110

XIII The l3seq package: Sequences and stacks 112
1 Creating and initialising sequences 112
2 Appending data to sequences 113

3 Recovering items from sequences 113
4 Recovering values from sequences with branching 114
5 Modifying sequences 115
6 Sequence conditionals 116
7 Mapping to sequences 116
8 Using the content of sequences directly 118
9 Sequences as stacks 118
10 Sequences as sets 120
11 Constant and scratch sequences 121
12 Viewing sequences 121
13 Internal sequence functions 122

XIV The l3clist package: Comma separated lists 123
1 Creating and initialising comma lists 123
2 Adding data to comma lists 124
3 Modifying comma lists 125
4 Comma list conditionals 126
5 Mapping to comma lists 126
6 Using the content of comma lists directly 128
7 Comma lists as stacks 129
8 Using a single item 130
9 Viewing comma lists 130
10 Constant and scratch comma lists 130

XV The l3prop package: Property lists 131
1 Creating and initialising property lists 131
2 Adding entries to property lists 132
3 Recovering values from property lists 132

4 Modifying property lists 133
5 Property list conditionals 133
6 Recovering values from property lists with branching 134
7 Mapping to property lists 134
8 Viewing property lists 135
9 Scratch property lists 135
10 Constants 136
11 Internal property list functions 136

XVI The l3box package: Boxes 137
1 Creating and initialising boxes 137
2 Using boxes 138
3 Measuring and setting box dimensions 138
4 Box conditionals 139
5 The last box inserted 139
6 Constant boxes 140
7 Scratch boxes 140
8 Viewing box contents 140
9 Horizontal mode boxes 140
10 Vertical mode boxes 142
11 Primitive box conditionals 143

XVII The l3coffins package: Coffin code layer 145
1 Creating and initialising coffins 145
2 Setting coffin content and poles 145
3 Joining and using coffins 146
4 Measuring coffins 147
5 Coffin diagnostics 147
5.1 Constants and variables 148

XVIII The l3color package: Color support 149

1 Color in boxes 149

XIX The l3msg package: Messages 150

1 Creating new messages 150

2 Contextual information for messages 151

3 Issuing messages 152

4 Redirecting messages 154

5 Low-level message functions 155

6 Kernel-specific functions 156

7 Expandable errors 158

8 Internal l3msg functions 158

XX The l3keys package: Key–value interfaces 160

1 Creating keys 161

2 Sub-dividing keys 165

3 Choice and multiple choice keys 165

4 Setting keys 167

5 Handling of unknown keys 168

6 Selective key setting 169

7 Utility functions for keys 170

8 Low-level interface for parsing key–val lists 170

XXI The l3file package: File and I/O operations 172

1 File operation functions 172
1.1 Input–output stream management 173
1.2 Reading from files 174

2 Writing to files **175**
2.1 Wrapping lines in output . . . 177
2.2 Constant input–output streams . . . 178
2.3 Primitive conditionals . . . 178
2.4 Internal file functions and variables . . . 178
2.5 Internal input–output functions . . . 178

XXII The l3fp package: floating points **180**

1 Creating and initialising floating point variables **181**

2 Setting floating point variables **181**

3 Using floating point numbers **182**

4 Floating point conditionals **183**

5 Floating point expression loops **185**

6 Some useful constants, and scratch variables **186**

7 Floating point exceptions **187**

8 Viewing floating points **188**

9 Floating point expressions **188**
9.1 Input of floating point numbers . . . 188
9.2 Precedence of operators . . . 189
9.3 Operations . . . 190

10 Disclaimer and roadmap **196**

XXIII The l3candidates package: Experimental additions to l3kernel **198**

1 Important notice **198**

2 Additions to l3basics **198**

3 Additions to l3box **199**
3.1 Affine transformations . . . 199
3.2 Viewing part of a box . . . 200

4 Additions to l3clist **201**

5 Additions to l3coffins **201**

6 Additions to l3file **202**

7 Additions to l3fp **203**

8 Additions to l3int 203

9 Additions to l3keys 204

10 Additions to l3msg 204

11 Additions to l3prg 204

12 Additions to l3prop 206

13 Additions to l3seq 206

14 Additions to l3skip 207

15 Additions to l3tl 208

16 Additions to l3tokens 212

XXIV The l3sys package:System/runtime functions 213

1 The name of the job 213

2 Date and time 213
2.1 Engine . . . 213
2.2 Output format . . . 214

XXV The l3luatex package:LuaTeX-specific functions 215

1 Breaking out to Lua 215
1.1 TEX code interfaces . . . 215
1.2 Lua interfaces . . . 216

XXVI The l3drivers package: Drivers 217

1 Box clipping 217

2 Box rotation and scaling 217

3 Color support 218

4 Drawing 218
4.1 Path construction . . . 219
4.2 Stroking and filling . . . 219
4.3 Stroke options . . . 220
4.4 Color . . . 221
4.5 Inserting TEX material . . . 222
4.6 Coordinate system transformations . . . 222

Index 223

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 `\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.

`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;

[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.

`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 them 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`.

Changes to behaviour When new functions are added to expl3, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of expl3 after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

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.

4 TeX concepts not supported by LaTeX3

The TeX concept of an "`\outer`" macro is *not supported* at all by LaTeX3. As such, the functions provided here may break when used on top of LaTeX 2_ε if `\outer` tokens are used in the arguments.

Part II
The l3bootstrap package
Bootstrap code

1 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 régime.

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

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

These functions act broadly in the same way as the corresponding 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.) The ⟨*date*⟩ should be given in the format ⟨*year*⟩/⟨*month*⟩/⟨*day*⟩.

`\GetIdInfo`

Updated: 2012-06-04

`\RequirePackage{l3bootstrap}`
`\GetIdInfo $Id:` ⟨*SVN info field*⟩ `$ {`⟨*description*⟩`}`

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

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

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

1.1 Internal functions and variables

`\l__kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn`/`\ExplSyntaxOff`.

Part III
The l3names package
Namespace for primitives

1 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 régime 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.

1 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.

2 Grouping material

\group_begin: `\group_begin:`
\group_end: `\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.

3 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.

3.1 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` scope, 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` scope, 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` scope, 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 scope 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` restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

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

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

`N` and `n` No manipulation.

`T` and `F` Functionally equivalent to `n` (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 1).

`p` and `w` These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

\cs_new:Npn
\cs_new:cpn
\cs_new:Npx
\cs_new: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
\cs_new_nopar:Npx
\cs_new_nopar: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
\cs_new_protected:Npx
\cs_new_protected: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
\cs_new_protected_nopar:Npx
\cs_new_protected_nopar: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
\cs_set:Npx
\cs_set: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
\cs_set_nopar:Npx
\cs_set_nopar: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
\cs_set_protected:Npx
\cs_set_protected: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
\cs_set_protected_nopar:Npx
\cs_set_protected_nopar: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
\cs_gset:Npx
\cs_gset: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
\cs_gset_nopar:Npx
\cs_gset_nopar: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
\cs_gset_protected:Npx
\cs_gset_protected: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
\cs_gset_protected_nopar:Npx
\cs_gset_protected_nopar: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.

3.3 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_git cogset_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)

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

Updated: 2012-01-14

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.

3.4 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.

3.5 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`

`\cs_undefine:N` ⟨*control sequence*⟩

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

Updated: 2011-09-15

3.6 Showing control sequences

`\cs_meaning:N` ⋆
`\cs_meaning:c` ⋆

`\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.

Updated: 2011-12-22

TEXhackers note: This is TEX's `\meaning` primitive. The `c` variant correctly reports undefined arguments.

\cs_show:N
\cs_show:c

Updated: 2015-08-03

\cs_show:N ⟨*control sequence*⟩

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

TEXhackers note: This is similar to the TEX primitive `\show`, wrapped to a fixed number of characters per line.

3.7 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_if_exist_use:N ⋆
\cs_if_exist_use:c ⋆
\cs_if_exist_use:NTF ⋆
\cs_if_exist_use:cTF ⋆

New: 2012-11-10

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

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type), and if it is inserts the ⟨*control sequence*⟩ into the input stream followed by the ⟨*true code*⟩. Otherwise the ⟨*false code*⟩ is used.

\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.

4 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 situation in force when first function absorbs the token).

`\use:n` ⋆ `\use:n` {⟨*group*$_1$⟩}
`\use:nn` ⋆ `\use:nn` {⟨*group*$_1$⟩} {⟨*group*$_2$⟩}
`\use:nnn` ⋆ `\use:nnn` {⟨*group*$_1$⟩} {⟨*group*$_2$⟩} {⟨*group*$_3$⟩}
`\use:nnnn` ⋆ `\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_i:nn` {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩}
`\use_ii:nn` ⋆

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 ⋆
\use_none:nnn ⋆
\use_none:nnnn ⋆
\use_none:nnnnn ⋆
\use_none:nnnnnn ⋆
\use_none:nnnnnnn ⋆
\use_none:nnnnnnnn ⋆
\use_none: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.

4.1 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.

5 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 2_ε. 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.

5.1 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 they 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`).

5.2 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_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We will prefix primitive conditionals with `\if_`.

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

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

\if_true: always executes ⟨*true code*⟩, while \if_false: always executes ⟨*false code*⟩. \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in l3int and used in case switches.

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.

6 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.

`\__chk_if_exist_var:N` `\__chk_if_exist_var:N ⟨var⟩`

This function checks that ⟨*var*⟩ is defined according to the criteria for `\cs_if_free_p:N`, and if not raises a kernel-level error. This function is only created if the package option `check-declarations` is active.

`\__chk_log:x` `\__chk_log:x {⟨message text⟩}`

If the `log-functions` option is active, this function writes the ⟨*message text*⟩ to the log file using `\iow_log:x`. Otherwise, the ⟨*message text*⟩ is ignored using `\use_none:n`.

`\__chk_suspend_log:` `\__chk_resume_log:` `\__chk_suspend_log: ... \__chk_log:x ... \__chk_resume_log:`

Any `\__chk_log:x` command between `\__chk_suspend_log:` and `\__chk_resume_log:` is suppressed. These commands can be nested.

`\__cs_count_signature:N` ⋆ `\__cs_count_signature:c` ⋆ `\__cs_count_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_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).

`\__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_tmp:w` Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\__kernel_register_show:N` `\__kernel_register_show:c` `\__kernel_register_show:N ⟨register⟩`

Used to show the contents of a TEX register at the terminal, formatted such that internal parts of the mechanism are not visible.

`\__prg_case_end:nw` ⋆ `\__prg_case_end:nw {`⟨*code*⟩`} ⟨`*tokens*`⟩ \q_mark {`⟨*true code*⟩`} \q_mark {`⟨*false code*⟩`} \q_stop`

Used to terminate case statements (`\int_case:nnTF`, *etc.*) by removing trailing ⟨*tokens*⟩ and the end marker `\q_stop`, inserting the ⟨*code*⟩ for the successful case (if one is found) and either the `true code` or `false code` for the over all outcome, as appropriate.

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.

1 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 defined 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_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

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

Providing variants in this way in style files is uncritical as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. 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.

2 Methods for defining variants

\cs_generate_variant:Nn

Updated: 2015-08-06

`\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. Only `n` and `N` arguments can be changed to other types. If the ⟨*parent control sequence*⟩ is protected or if the ⟨*variant*⟩ involves `x` arguments, then the ⟨*variant control sequence*⟩ will also be protected. The ⟨*variant*⟩ is created globally, as is any `\exp_args:N`⟨*variant*⟩ function needed to carry out the expansion.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster then others. Therefore (when speed is important) 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 themselves subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing when more than one argument is being expanded. This special processing 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`, `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.

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. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression $3+4$ and pass the result 7 as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result 7, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

will result in the call `\example:n { 3 , \int_eval:n { 3 + 4 } }` while using `\example:x` instead results in `\example:n { 3 , 7 }` at the cost of being protected. 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!

If is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, `f`-type expansion stops at the emphfirst non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable.

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

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 ⟨*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.

5 Manipulating two arguments

\exp_args:NNo ⋆
\exp_args:NNc ⋆
\exp_args:NNv ⋆
\exp_args:NNV ⋆
\exp_args:NNf ⋆
\exp_args:Nco ⋆
\exp_args:Ncf ⋆
\exp_args:Ncc ⋆
\exp_args:NVV ⋆

\exp_args:NNc ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*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 ⋆
\exp_args:Nnf ⋆
\exp_args:Noo ⋆
\exp_args:Nof ⋆
\exp_args:Noc ⋆
\exp_args:Nff ⋆
\exp_args:Nfo ⋆
\exp_args: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
\exp_args:Ncx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

\exp_args:NNx ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*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.

6 Manipulating three arguments

\exp_args:NNNo ⋆
\exp_args:NNNV ⋆
\exp_args:Nccc ⋆
\exp_args:NcNc ⋆
\exp_args:NcNo ⋆
\exp_args:Ncco ⋆

\exp_args:NNNo ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*token*$_3$⟩ {⟨*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 ⋆
\exp_args:Nnno ⋆
\exp_args:Nnnc ⋆
\exp_args:Nooo ⋆

\exp_args:NNoo ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*token*$_3$⟩} {⟨*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:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox
\exp_args:Ncnx
\exp_args:Nccx

New: 2015-08-12

\exp_args:NNnx ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*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.*

7 Unbraced expansion

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

Updated: 2012-02-12

\exp_last_unbraced:Nno ⟨*token*⟩ ⟨*tokens*$_1$⟩ ⟨*tokens*$_2$⟩

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave 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 \foo_bar: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*⟩ ⟨*tokens*$_1$⟩ {⟨*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` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

Carries out a single expansion of ⟨*token*$_2$⟩ (which may consume arguments) prior to the expansion of ⟨*token*$_1$⟩. If ⟨*token*$_2$⟩ is a TEX primitive, it will be executed rather than expanded, while if ⟨*token*$_2$⟩ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that ⟨*token*$_1$⟩ 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.

8 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. Hence its argument *must* be surrounded by braces.

\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: ⋆ `\foo_bar:f { ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩ }`

Updated: 2011-06-03

This function terminates an `f`-type expansion. Thus if a function `\foo_bar: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 (␣).

9 Controlled expansion

The expl3 language makes all efforts to hide the complexity of TEX expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the "base" functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down TEX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. You will find these commands used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of ⟨*expandable-tokens*⟩ as that will break badly if unexpandable tokens are encountered in that place!

\exp:w ⋆
\exp_end: ⋆ `\exp:w ⟨expandable-tokens⟩ \exp_end:`

New: 2015-08-23

Expands ⟨*expandable-tokens*⟩ until reaching `\exp_end:` at which point expansion stops. The full expansion of ⟨*expandable-tokens*⟩ has to be empty. If any token in ⟨*expandable-tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.[2]

In typical use cases the `\exp_end:` will be hidden somewhere in the replacement text of ⟨*expandable-tokens*⟩ rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

\exp:w ⋆
\exp_end_continue_f:w ⋆

New: 2015-08-23

\exp:w ⟨*expandable-tokens*⟩ \exp_end_continue_f:w ⟨*further-tokens*⟩

Expands ⟨*expandable-tokens*⟩ until reaching \exp_end_continue_f:w at which point expansion continues as an f-type expansion expanding ⟨*further-tokens*⟩ until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by \exp_stop_f:). As with all f-type expansions a space ending the expansion will get removed.

The full expansion of ⟨*expandable-tokens*⟩ has to be empty. If any token in ⟨*expandable-tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result \exp_end_continue_f:w will be misinterpreted later on.[3]

In typical use cases ⟨*expandable-tokens*⟩ contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the \exp_after:wN triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command \exp_after:wN would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

\exp:w ⟨*expandable-tokens*⟩ \exp_end:

can be alternatively achieved through an f-type expansion by using \exp_stop_f:, i.e.

\exp:w \exp_end_continue_f:w ⟨*expandable-tokens*⟩ \exp_stop_f:

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

\exp:w ⋆
\exp_end_continue_f:nw ⋆

New: 2015-08-23

\exp:w ⟨*expandable-tokens*⟩ \exp_end_continue_f:nw ⟨*further-tokens*⟩

The difference to \exp_end_continue_f:w is that we first we pick up an argument which is then returned to the input stream. If ⟨*further-tokens*⟩ starts with a brace group then the braces are removed. If on the other hand it starts with space tokens then these space tokens are removed while searching for the argument. Thus such space tokens will not terminate the f-type expansion.

10 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.

[2] Due to the implementation you might get the character in position 0 in the current font (typically "‘") in the output without any error message!

[3] In this particular case you may get a character into the output as well as an error message.

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

```
\cs_set: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.

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 states returned are ⟨*true*⟩ and ⟨*false*⟩.

LaTeX3 has two forms of conditional flow processing based on these states. The first 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.

TEXhackers note: The arguments are executed after exiting the underlying `\if...\fi:` structure.

1 Defining a set of conditional functions

`\prg_new_conditional:Npnn`
`\prg_set_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\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_set_protected_conditional:Npnn`
`\prg_new_protected_conditional:Nnn`
`\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 cannot be defined 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:Npnn \foo_if_bar:NN #1#2 { 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 erroneous output 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 copy 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_true:`
\prg_return_false: ⋆ `\prg_return_false:`

These 'return' functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an f-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

2 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).

TEXhackers note: The `bool` data type is not implemented using the `\iffalse`/`\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain TEX, LATEX 2ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

\bool_new:N `\bool_new:N` ⟨*boolean*⟩
\bool_new:c

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:N` ⟨*boolean*⟩
\bool_set_false:c
\bool_gset_false:N Sets ⟨*boolean*⟩ logically `false`.
\bool_gset_false:c

\bool_set_true:N `\bool_set_true:N` ⟨*boolean*⟩
\bool_set_true:c
\bool_gset_true:N Sets ⟨*boolean*⟩ logically `true`.
\bool_gset_true:c

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

\bool_set_eq:NN ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩

Sets the content of ⟨*boolean*$_1$⟩ equal to that of ⟨*boolean*$_2$⟩.

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

Updated: 2012-07-08

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

Evaluates the ⟨*boolean expression*⟩ as described for \bool_if:nTF, 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
Updated: 2015-08-01

\bool_show:N ⟨*boolean*⟩

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

\bool_show:n

New: 2012-02-09
Updated: 2015-08-07

\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
\l_tmpb_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
\g_tmpb_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.

3 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 `!` with their usual precedences (namely, `&&` binds more tightly than `||`). 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 } ||
    \str_if_eq_p:nn { abc } { def }
  ) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

At present, the infix operators `&&` and `||` perform lazy evaluation as well, but this will change in a future release.

`\bool_if_p:n` ⋆ `\bool_if:nTF` ⋆

Updated: 2012-07-08

`\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. The logical Not applies to the next predicate or group.

`\bool_not_p:n` ⋆

Updated: 2012-07-08

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

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

`\bool_xor_p:nn` ⋆

Updated: 2012-07-08

`\bool_xor_p:nn {`⟨*boolexpr*$_1$⟩`} {`⟨*boolexpr*$_2$⟩`}`

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

4 Logical loops

Loops using either boolean expressions or stored boolean values.

`\bool_do_until:Nn` ☆
`\bool_do_until:cn` ☆

`\bool_do_until:Nn` ⟨*boolean*⟩ `{`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean*⟩. If it is `false` then the ⟨*code*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean*⟩ is `true`.

`\bool_do_while:Nn` ☆
`\bool_do_while:cn` ☆

`\bool_do_while:Nn` ⟨*boolean*⟩ `{`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean*⟩. If it is `true` then the ⟨*code*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean*⟩ is `false`.

`\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_do_until:nn ☆

Updated: 2012-07-08

\bool_do_until:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean expression*⟩ as described for `\bool_if:nTF`. If it is `false` then the ⟨*code*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean expression*⟩ evaluates to `true`.

\bool_do_while:nn ☆

Updated: 2012-07-08

\bool_do_while:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TEX to process, and then checks the logical value of the ⟨*boolean expression*⟩ as described for `\bool_if:nTF`. If it is `true` then the ⟨*code*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean expression*⟩ evaluates to `false`.

\bool_until_do:nn ☆

Updated: 2012-07-08

\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 ☆

Updated: 2012-07-08

\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`.

5 Producing multiple 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.

6 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 ⋆

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

Detects if TEX is currently in maths mode.

Updated: 2011-09-05

\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.

7 Primitive conditionals

\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.

8 Internal programming functions

\group_align_safe_begin: ⋆
\group_align_safe_end: ⋆

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

Updated: 2011-08-11

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.

__prg_break_point:Nn ⋆

```
\__prg_break_point:Nn \⟨type⟩_map_break: ⟨tokens⟩
```

Used to mark the end of a recursion or mapping: the functions `\⟨type⟩_map_break:` and `\⟨type⟩_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 break functions are *not* applied: `\__prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

__prg_map_break:Nn ⋆

__prg_map_break:Nn \⟨*type*⟩_map_break: {⟨*user code*⟩}
...
__prg_break_point:Nn \⟨*type*⟩_map_break: {⟨*ending code*⟩}

Breaks a recursion in mapping contexts, inserting in the input stream the ⟨*user code*⟩ after the ⟨*ending code*⟩ for the loop. The function breaks loops, inserting their ⟨*ending code*⟩, until reaching a loop with the same ⟨*type*⟩ as its first argument. This \⟨*type*⟩_map_break: argument is simply used as a recognizable marker for the ⟨*type*⟩.

\g__prg_map_int

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions __prg_map_1:w, __prg_map_2:w, *etc.*, labelled by \g__prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.

__prg_break_point: ⋆

This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursions: the function __prg_break:n uses this to break out of the loop.

__prg_break: ⋆
__prg_break:n ⋆

__prg_break:n {⟨*tokens*⟩} ... __prg_break_point:

Breaks a recursion which has no ⟨*ending code*⟩ and which is not a user-breakable mapping (see for instance \prop_get:Nn), and inserts ⟨*tokens*⟩ in the input stream.

Part VII
The l3quark package
Quarks

1 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 for internal use by the kernel: they are not intended for more general use.

1.1 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`.

2 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.

`\q_nil` 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.

3 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).

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 5.

`\q_recursion_tail` This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

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

`\quark_if_recursion_tail_stop:N` `\quark_if_recursion_tail_stop:N` ⟨*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:n {`⟨*token list*⟩`}`
`\quark_if_recursion_tail_stop:o`

Updated: 2011-09-06

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

Updated: 2011-09-06

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.

5 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce "`[-a-b-] [-c-d-] `". Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of `\my_map_dbl:nn`. First of all, define the function that will do the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
 {
   \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
   \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
   \q_recursion_stop
 }
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
 {
   \quark_if_recursion_tail_stop:n {#1}
   \quark_if_recursion_tail_stop:n {#2}
   \__my_map_dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
   \__my_map_dbl:nn
 }
```

Note that contrarily to LaTeX3 built-in mapping functions, this mapping function cannot be nested, since the second map will overwrite the definition of `\__my_map_dbl_fn:nn`.

6 Internal quark functions

`\__quark_if_recursion_tail_break:NN` `\__quark_if_recursion_tail_break:nN {`⟨*token list*⟩`}`
`\__quark_if_recursion_tail_break:nN` `\`⟨*type*⟩`_map_break:`

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

7 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).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

`\__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.

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 functions we will describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its "shape" (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its "meaning", which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of TEX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence 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
```

A list of all possible shapes and a list of all possible meanings are given in section 8.

1 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`
`\char_gset_active_eq:NN`
`\char_gset_active_eq:Nc`

Updated: 2015-11-12

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

Sets the behaviour of the ⟨*char*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc

\char_set_active_eq:nN {⟨*integer expression*⟩} ⟨*function*⟩

New: 2015-11-12

Sets the behaviour of the ⟨*char*⟩ which has character code as given by the ⟨*integer expression*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_generate:nn ⋆ \char_generate:nn {⟨*charcode*⟩} {⟨*catcode*⟩}

New: 2015-09-09

Generates a character token of the given ⟨*charcode*⟩ and ⟨*catcode*⟩ (both of which may be integer expressions). The ⟨*catcode*⟩ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)

and other values will raise an error.

The ⟨*charcode*⟩ may be any one valid for the engine in use. Note however that for XeTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

2 Manipulating and interrogating 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*⟩

Updated: 2015-11-11

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*⟩}

Updated: 2015-11-11

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*$_2$⟩}

Updated: 2015-11-11

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*$_2$⟩}

Updated: 2015-08-06

Sets up the behaviour of the ⟨*character*⟩ when found inside \tl_to_lowercase:n, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. 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 {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Updated: 2015-08-06

Sets up the behaviour of the ⟨*character*⟩ when found inside \tl_to_uppercase:n, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. 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 {⟨intexpr$_1$⟩} {⟨intexpr$_2$⟩}`

Updated: 2015-08-06

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 {⟨intexpr$_1$⟩} {⟨intexpr$_2$⟩}`

Updated: 2015-08-06

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
Updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.

`\l_char_special_seq`

New: 2012-01-23
Updated: 2015-11-11

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.

3 Generic tokens

\token_new:Nn

\token_new:Nn ⟨*token*$_1$⟩ {⟨*token*$_2$⟩}

Defines ⟨*token*$_1$⟩ to globally be a snapshot of ⟨*token*$_2$⟩. This will be an implicit representation of ⟨*token*$_2$⟩.

\c_group_begin_token
\c_group_end_token
\c_math_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.

4 Converting tokens

\token_to_meaning:N ⋆
\token_to_meaning:c ⋆

\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.

5 Token conditionals

\token_if_group_begin_p:N ⋆ \token_if_group_begin_p:N ⟨token⟩
\token_if_group_begin:NTF ⋆ \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_p:N ⟨token⟩
\token_if_group_end:NTF ⋆ \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_p:N ⟨token⟩
\token_if_math_toggle:NTF ⋆ \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_p:N ⟨token⟩
\token_if_alignment:NTF ⋆ \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_p:N ⟨token⟩
\token_if_parameter:NTF ⋆ \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_p:N ⟨token⟩
\token_if_math_superscript:NTF ⋆ \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_p:N ⟨token⟩
\token_if_math_subscript:NTF ⋆ \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_p:N ⟨token⟩
\token_if_space:NTF ⋆ \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_p:N ⟨*token*⟩
\token_if_letter:NTF ⋆ \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_p:N ⟨*token*⟩
\token_if_other:NTF ⋆ \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_p:N ⟨*token*⟩
\token_if_active:NTF ⋆ \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_p:NN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
\token_if_eq_catcode:NNTF ⋆ \token_if_eq_catcode:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_eq_charcode_p:NN ⋆ \token_if_eq_charcode_p:NN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
\token_if_eq_charcode:NNTF ⋆ \token_if_eq_charcode:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_eq_meaning_p:NN ⋆ \token_if_eq_meaning_p:NN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
\token_if_eq_meaning:NNTF ⋆ \token_if_eq_meaning:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

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

Updated: 2011-05-23

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

\token_if_cs_p:N ⋆ \token_if_cs_p:N ⟨*token*⟩
\token_if_cs:NTF ⋆ \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_p:N ⟨*token*⟩
\token_if_expandable:NTF ⋆ \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_p:N ⟨*token*⟩
\token_if_long_macro:NTF ⋆ \token_if_long_macro:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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_p:N ⟨token⟩
\token_if_mathchardef:NTF ⋆ \token_if_mathchardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

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_p:N ⟨token⟩
\token_if_muskip_register:NTF ⋆ \token_if_muskip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

New: 2012-02-15

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

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

Updated: 2012-01-20

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

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

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a toks register (not used by LaTeX3).

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

Updated: 2011-05-23

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

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

Updated: 2012-12-20

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

Updated: 2012-12-20

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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:N*TF* `\peek_catcode_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-12-20

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

Updated: 2012-12-20

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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:N*TF* `\peek_charcode:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-12-20

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

Updated: 2012-12-20

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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:N*TF* `\peek_charcode_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-12-20

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

Updated: 2012-12-20

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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` `\peek_meaning:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2011-07-02

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

Updated: 2012-12-05

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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` `\peek_meaning_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2011-07-02

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

Updated: 2012-12-05

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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).

7 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.

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

`\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

8 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TEX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.

- An active character token, characterized by its character code (between 0 and 1114111 for LuaTEX and XETEX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).[4]

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the`þfont results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A "frozen" `\relax`, which differs from the primitive in both shape and meaning, is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- An `\endtemplate` (which expands to `\outer endtemplate:`) can be enountered when peeking ahead at the next token.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We will call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TEX primitive `\meaning`, together with their LATEX3 names and most common example:

1 begin-group character (`group_begin`, often `{`),

2 end-group character (`group_end`, often `}`),

3 math shift character (`math_toggle`, often `$`),

4 alignment tab character (`alignment`, often `&`),

6 macro parameter character (`parameter`, often `#`),

7 superscript character (`math_superscript`, often `^`),

8 subscript character (`math_subscript`, often `_`),

10 blank space (`space`, often character code 32),

11 the letter (`letter`, such as `A`),

12 the character (`other`, such as `0`).

[4] In LuaTEX, there is also the case of "bytes", which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we will call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in LaTeX3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in LaTeX3 for some functions,
- a register such as `\count123`, used in LaTeX3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros be `\protected` or not, `\long` or not (the opposite of what LaTeX3 calls `nopar`), and `\outer` or not (unused in LaTeX3). Their `\meaning` takes the form

⟨*properties*⟩ `macro:`⟨*parameters*⟩`->`⟨*replacement*⟩

where ⟨*properties*⟩ is among `\protected\long\outer`, ⟨*parameters*⟩ describes parameters that the macro expects, such as `#1#2#3`, and ⟨*replacement*⟩ describes how the parameters are manipulated, such as `#2/#1/#3`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then TeX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens "N-type", as they are suitable to be used as an argument for a function with the signature `:N`.

9 Internal functions

`\__char_generate:nn` ⋆

New: 2016-03-25

`\__char_generate:nn {`⟨*charcode*⟩`} {`⟨*catcode*⟩`}`

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The ⟨*catcode*⟩ must give an explicit integer after a single expansion.

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* ("`intexpr`").

1 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 `)`. Any functions within the expressions should expand to an ⟨*integer denotation*⟩: a sequence of a sign and digits matching the regex `\-?[0-9]+`). After expansion `\int_eval:n` yields an ⟨*integer denotation*⟩ which is left in the input stream.

TEXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result 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⟩}`

Updated: 2012-09-26

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

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

Updated: 2012-09-26

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. 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 an ⟨*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 divides the first value by the second, and rounds the result towards zero. Note that division using / rounds to the closest integer instead. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

\int_max:nn ⋆
\int_min:nn ⋆

Updated: 2012-09-26

\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 ⋆

Updated: 2012-09-26

\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 obtained by subtracting \int_div_truncate:nn {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩} times ⟨*intexpr*$_2$⟩ from ⟨*intexpr*$_1$⟩. Thus, the result has the same sign as ⟨*intexpr*$_1$⟩ and its absolute value is strictly less than that of ⟨*intexpr*$_2$⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

2 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 ⟨*integer*$_1$⟩ ⟨*integer*$_2$⟩

Sets the content of ⟨*integer*$_1$⟩ equal to that of ⟨*integer*$_2$⟩.

\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.

3 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⟩ from the current content of the ⟨integer⟩.

4 Using integers

\int_use:N ⋆
\int_use:c ⋆

Updated: 2011-10-22

\int_use:N ⟨integer⟩

Recovers the content of an ⟨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 an ⟨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.

5 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 ⋆

Updated: 2013-01-13

```
\int_compare_p:n
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+1⟩
  }
\int_compare:nTF
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*integer expressions*⟩ as described for `\int_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields `true` if all comparisons are `true`. Each ⟨*integer expression*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*integer expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

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

\int_case:nn ⋆
\int_case:nnTF ⋆
New: 2013-07-24

```
\int_case:nnTF {⟨test integer expression⟩}
  {
    {⟨intexpr case1⟩} {⟨code case1⟩}
    {⟨intexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨intexpr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

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 any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

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

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

\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.

6 Integer expression loops

\int_do_until:nNnn ☆

```
\int_do_until:nNnn {⟨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: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_do_while:nNnn ☆

```
\int_do_while:nNnn {⟨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: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_until_do:nNnn ☆ `\int_until_do:nNnn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩} {⟨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_while_do:nNnn ☆ `\int_while_do:nNnn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩} {⟨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:nn ☆ `\int_do_until:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ 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_do_while:nn ☆ `\int_do_while:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ 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`.

\int_until_do:nn ☆ `\int_until_do:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*integer relation*⟩ 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_while_do:nn ☆ `\int_while_do:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*integer relation*⟩ 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`.

7 Integer step functions

\int_step_function:nnnN ☆

New: 2012-06-04
Updated: 2014-05-30

`\int_step_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*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

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

\int_step_inline:nnnn

New: 2012-06-04
Updated: 2014-05-30

`\int_step_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. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`).

\int_step_variable:nnnNn

New: 2012-06-04
Updated: 2014-05-30

`\int_step_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. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), 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*⟩.

8 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_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. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\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_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        ...
        { 26 } { z }
      }
  }
```

\int_to_bin:n ⋆

New: 2014-02-11

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

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

\int_to_hex:n ⋆
\int_to_Hex:n ⋆

New: 2014-02-11

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

Calculates the value of the ⟨*integer expression*⟩ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_oct:n ⋆

New: 2014-02-11

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

Calculates the value of the ⟨*integer expression*⟩ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_base:nn ⋆
\int_to_Base:nn ⋆

Updated: 2014-02-11

\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 letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum ⟨*base*⟩ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TEXhackers note: This is a generic version of `\int_to_bin: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).

9 Converting from other formats to integers

\int_from_alph:n ⋆

Updated: 2014-08-25

\int_from_alph:n {⟨*letters*⟩}

Converts the ⟨*letters*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*letters*⟩ are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with "a" equal to 1 through to "z" equal to 26. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_alph:n` and `\int_to_Alph:n`.

\int_from_bin:n ⋆

New: 2014-02-11
Updated: 2014-08-25

\int_from_bin:n {⟨*binary number*⟩}

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*binary number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

`\int_from_hex:n` ⋆ `\int_from_hex:n {⟨hexadecimal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

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. The ⟨*hexadecimal number*⟩ is first converted to a string, with no expansion. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

`\int_from_oct:n` ⋆ `\int_from_oct:n {⟨octal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*octal number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

`\int_from_roman:n` ⋆ `\int_from_roman:n {⟨roman numeral⟩}`

Updated: 2014-08-25

Converts the ⟨*roman numeral*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*roman numeral*⟩ is first converted to a string, with no expansion. The ⟨*roman numeral*⟩ may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value will be -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ⋆ `\int_from_base:nn {⟨number⟩} {⟨base⟩}`

Updated: 2014-08-25

Converts the ⟨*number*⟩ expressed in ⟨*base*⟩ into the appropriate value in base 10. The ⟨*number*⟩ is first converted to a string, with no expansion. The ⟨*number*⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨*base*⟩ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

10 Viewing integers

`\int_show:N` `\int_show:N ⟨integer⟩`
`\int_show:c`

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

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

New: 2011-11-22
Updated: 2015-08-07

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

11 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.

\c_max_char_int

Maximum character code completely supported by the engine.

12 Scratch integers

\l_tmpa_int
\l_tmpb_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.

13 Primitive conditionals

\if_int_compare:w ⋆

```
\if_int_compare: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 (⟨*case*$_0$⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨*case*$_1$⟩) 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`.

\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`.

14 Internal functions

__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.

`\__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`.

`\__prg_compare_error:`
`\__prg_compare_error:Nw`

`\__prg_compare_error:`
`\__prg_compare_error:Nw` ⟨*token*⟩

These are used within `\int_compare:nTF`, `\dim_compare:nTF` and so on to recover correctly if the `n`-type argument does not contain a properly-formed relation.

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.

1 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.

2 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 ⟨*dimension*$_1$⟩ ⟨*dimension*$_2$⟩

Sets the content of ⟨*dimension*$_1$⟩ equal to that of ⟨*dimension*$_2$⟩.

\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*⟩ from the current content of the ⟨*dimension*⟩.

3 Utilities for dimension calculations

\dim_abs:n ⋆

Updated: 2012-09-26

\dim_abs:n {⟨*dimexpr*⟩}

Converts the ⟨*dimexpr*⟩ to its absolute value, leaving the result in the input stream as a ⟨*dimension denotation*⟩.

\dim_max:nn ⋆
\dim_min:nn ⋆

New: 2012-09-09
Updated: 2012-09-26

\dim_max:nn {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}
\dim_min:nn {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Evaluates the two ⟨*dimension expressions*⟩ and leaves either the maximum or minimum value in the input stream as appropriate, as a ⟨*dimension denotation*⟩.

\dim_ratio:nn ☆

Updated: 2011-10-22

\dim_ratio:nn {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

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.

4 Dimension expression conditionals

\dim_compare_p:nNn ⋆
\dim_compare:nNnTF ⋆

\dim_compare_p:nNn {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩}
\dim_compare:nNnTF
  {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩}
  {⟨*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 ⋆

Updated: 2013-01-13

```
\dim_compare_p:n
  {
    ⟨dimexpr1⟩ ⟨relation1⟩
    ...
    ⟨dimexprN⟩ ⟨relationN⟩
    ⟨dimexprN+1⟩
  }
\dim_compare:nTF
  {
    ⟨dimexpr1⟩ ⟨relation1⟩
    ...
    ⟨dimexprN⟩ ⟨relationN⟩
    ⟨dimexprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*dimension expressions*⟩ as described for `\dim_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨$dimexpr_1$⟩ and ⟨$dimexpr_2$⟩ using the ⟨$relation_1$⟩, then ⟨$dimexpr_2$⟩ and ⟨$dimexpr_3$⟩ using the ⟨$relation_2$⟩, until finally comparing ⟨$dimexpr_N$⟩ and ⟨$dimexpr_{N+1}$⟩ using the ⟨$relation_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*dimension expression*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*dimension expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

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

\dim_case:nn ⋆
\dim_case:nnTF ⋆
New: 2013-07-24

```
\dim_case:nnTF {⟨test dimension expression⟩}
  {
    {⟨dimexpr case_1⟩} {⟨code case_1⟩}
    {⟨dimexpr case_2⟩} {⟨code case_2⟩}
    ...
    {⟨dimexpr case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

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 any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  {
    { 5 pt }        { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }     { Negative }
  }
  { No idea! }
```

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

5 Dimension expression loops

\dim_do_until:nNnn ☆

```
\dim_do_until: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_do_while:nNnn ☆

```
\dim_do_while: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_until_do:nNnn ☆

```
\dim_until_do: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_while_do:nNnn ☆ `\dim_while_do:nNnn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩} {⟨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:nn ☆ `\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*dimension relation*⟩ 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_do_while:nn ☆ `\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*dimension relation*⟩ 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`.

\dim_until_do:nn ☆ `\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*dimension relation*⟩ 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_while_do:nn ☆ `\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*dimension relation*⟩ 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`.

6 Using `dim` expressions and variables

\dim_eval:n ⋆ `\dim_eval:n {⟨dimension expression⟩}`

Updated: 2011-10-22

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:N ⟨dimension⟩`

\dim_use:c ⋆

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.

`\dim_to_decimal:n` ⋆ `\dim_to_decimal:n {`⟨*dimexpr*⟩`}`

New: 2014-07-15

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves `1.00374` in the input stream, *i.e.* the magnitude of one "big point" when converted to (TEX) points.

`\dim_to_decimal_in_bp:n` ⋆ `\dim_to_decimal_in_bp:n {`⟨*dimexpr*⟩`}`

New: 2014-07-15

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves `0.99628` in the input stream, *i.e.* the magnitude of one (TEX) point when converted to big points.

`\dim_to_decimal_in_sp:n` ⋆ `\dim_to_decimal_in_sp:n {`⟨*dimexpr*⟩`}`

New: 2015-05-18

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result will necessarily be an integer.

`\dim_to_decimal_in_unit:nn` ⋆ `\dim_to_decimal_in_unit:nn {`⟨*dimexpr*$_1$⟩`} {`⟨*dimexpr*$_2$⟩`}`

New: 2014-07-15

Evaluates the ⟨*dimension expressions*⟩, and leaves the value of ⟨*dimexpr*$_1$⟩, expressed in a unit given by ⟨*dimexpr*$_2$⟩, in the input stream. The result is a decimal number, rounded by TEX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves `0.35277` in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε-TEX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

\dim_to_fp:n ⋆ `\dim_to_fp:n {⟨dimexpr⟩}`

New: 2012-05-08

Expands to an internal floating point number equal to the value of the ⟨*dimexpr*⟩ in `pt`. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision is acceptable.

7 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

`\dim_show:n {⟨dimension expression⟩}`

New: 2011-11-22
Updated: 2015-08-07

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

8 Constant dimensions

\c_max_dim

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim

A zero length as a dimension. This can also be used as a component of a skip.

9 Scratch dimensions

\l_tmpa_dim
\l_tmpb_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.

10 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.

11 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 ⟨$skip_1$⟩ ⟨$skip_2$⟩

Sets the content of ⟨$skip_1$⟩ equal to that of ⟨$skip_2$⟩.

\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*⟩ from the current content of the ⟨*skip*⟩.

12 Skip expression conditionals

\skip_if_eq_p:nn ⋆
\skip_if_eq:nnTF ⋆

```
\skip_if_eq_p:nn {⟨skipexpr1⟩} {⟨skipexpr2⟩}
\dim_compare:nTF
  {⟨skipexpr1⟩} {⟨skipexpr2⟩}
  {⟨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_finite_p:n ⋆
\skip_if_finite:nTF ⋆

```
\skip_if_finite_p:n {⟨skipexpr⟩}
\skip_if_finite:nTF {⟨skipexpr⟩} {⟨true code⟩} {⟨false code⟩}
```

New: 2012-03-05

Evaluates the ⟨*skip expression*⟩ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using `skip` expressions and variables

\skip_eval:n ⋆

```
\skip_eval:n {⟨skip expression⟩}
```

Updated: 2011-10-22

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.

14 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

```
\skip_show:n {⟨skip expression⟩}
```

New: 2011-11-22
Updated: 2015-08-07

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

15 Constant skips

\c_max_skip

Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.

\c_zero_skip

Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

16 Scratch skips

\l_tmpa_skip
\l_tmpb_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.

17 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal: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
\skip_vertical: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.

18 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.

19 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 ⟨*muskip*$_1$⟩ ⟨*muskip*$_2$⟩

Sets the content of ⟨*muskip*$_1$⟩ equal to that of ⟨*muskip*$_2$⟩.

\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*⟩ from the current content of the ⟨*skip*⟩.

20 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.

21 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
Updated: 2015-08-07

\muskip_show:n {⟨*muskip expression*⟩}

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

22 Constant muskips

\c_max_muskip

The maximum value that can be stored as a muskip, with no stretch nor shrink component.

\c_zero_muskip

A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

\l_tmpa_muskip
\l_tmpb_muskip

Scratch muskip 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_muskip
\g_tmpb_muskip

Scratch muskip 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.

24 Primitive conditional

\if_dim:w

```
\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen2⟩
  ⟨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`.

25 Internal functions

__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`.

Part XI
The l3tl package
Token lists

TEX works with tokens, and LATEX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called "token list variable", which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\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 (explicit, or stored in a variable) can be seen either as a list of "items", or a list of "tokens". An item is whatever `\use:n` would grab as its argument: a single non-space 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.

1 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*⟩.

\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 var*$_1$⟩ ⟨*tl var*$_2$⟩

Sets the content of ⟨*tl var*$_1$⟩ equal to that of ⟨*tl var*$_2$⟩.

\tl_concat:NNN
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc

New: 2012-05-18

\tl_concat:NNN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ ⟨*tl var*$_3$⟩

Concatenates the content of ⟨*tl var*$_2$⟩ and ⟨*tl var*$_3$⟩ together and saves the result in ⟨*tl var*$_1$⟩. The ⟨*tl var*$_2$⟩ will be placed at the left side of the new token list.

\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.

2 Adding data to token list variables

\tl_set:Nn
\tl_set:(NV|Nv|No|Nf|Nx|cn|cV|cv|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*⟩.

3 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 `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\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 `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). 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).

\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 `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\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 `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). 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`.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply TEX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes).

\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: 2015-08-11

\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. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ will be those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_rescan:nn`.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

\tl_rescan:nn

Updated: 2015-08-11

\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. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ will be those in force at the point of use of `\tl_rescan:nn`.) The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the ⟨*tokens*⟩ argument of `\tl_rescan:nn`.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

5 Token list conditionals

\tl_if_blank_p:n ⋆
\tl_if_blank_p:(V|o) ⋆
\tl_if_blank:nTF ⋆
\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 space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

\tl_if_empty_p:N ⋆
\tl_if_empty_p:c ⋆
\tl_if_empty:NTF ⋆
\tl_if_empty:cTF ⋆

\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) ⋆
\tl_if_empty:nTF ⋆
\tl_if_empty:(V|o)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).

New: 2012-05-24
Updated: 2012-06-05

\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 var*$_1$⟩ ⟨*tl var*$_2$⟩
\tl_if_eq:NNTF ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ {⟨*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:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields `false`.

\tl_if_eq:nnTF

\tl_if_eq:nnTF {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_1$⟩ and ⟨*token list*$_2$⟩ contain the same list of tokens, 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 ⟨*tl var*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_in:nnTF
\tl_if_in:(Vn|on|no)TF

\tl_if_in:nnTF {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_2$⟩ is found inside ⟨*token list*$_1$⟩. The ⟨*token list*$_2$⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_single_p:N ⋆
\tl_if_single_p:c ⋆
\tl_if_single:NTF ⋆
\tl_if_single:cTF ⋆

\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 a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

Updated: 2011-08-13

\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 a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:n`.

\tl_case:Nn ⋆
\tl_case:cn ⋆
\tl_case:NnTF ⋆
\tl_case:cnTF ⋆
New: 2013-07-24

```
\tl_case:NnTF ⟨test token list variable⟩
  {
    ⟨token list variable case1⟩ {⟨code case1⟩}
    ⟨token list variable case2⟩ {⟨code case2⟩}
    ...
    ⟨token list variable casen⟩ {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

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 any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

6 Mapping to token lists

\tl_map_function:NN ☆
\tl_map_function:cN ☆
Updated: 2012-06-29

```
\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 ☆
Updated: 2012-06-29

```
\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
Updated: 2012-06-29

```
\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
Updated: 2012-06-29

```
\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
Updated: 2012-06-29

\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
Updated: 2012-06-29

\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: ☆
Updated: 2012-06-29

\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:nnT { #1 } { bingo } { \tl_map_break: }
    % Do something useful
  }
```

See also \tl_map_break:n. Use outside of a \tl_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:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\tl_map_break:n ☆
Updated: 2012-06-29

\tl_map_break:n {⟨*tokens*⟩}

Used to terminate a \tl_map_... function before all entries in the ⟨*token list variable*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \tl_map_break:n { <tokens> } }
    % Do something useful
  }
```

Use outside of a \tl_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:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

7 Using token lists

\tl_to_str:n ⋆ `\tl_to_str:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting character tokens in the input stream. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

TEXhackers note: Converting a ⟨*token list*⟩ to a ⟨*string*⟩ yields a concatenation of the string representations of every token in the ⟨*token list*⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not "letter".

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character "backslash", "lower-case a", "space", but it may also produce a single "lower-case a" if the escape character is negative and `a` is currently not a letter.

\tl_to_str:N ⋆ `\tl_to_str:N ⟨tl var⟩`
\tl_to_str:c ⋆

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. For low-level details, see the notes given for `\tl_to_str:n`.

\tl_use:N ⋆ `\tl_use:N ⟨tl var⟩`
\tl_use:c ⋆

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.

8 Working with the content of token lists

\tl_count:n ⋆ `\tl_count:n {⟨tokens⟩}`
\tl_count:(V|o) ⋆

New: 2012-05-13

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_count:N`. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_count:N ⋆
\tl_count:c ⋆

New: 2012-05-13

\tl_count: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 the ⟨*tl var*⟩. See also `\tl_count: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 ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ... ⟨$item_n$⟩ becomes ⟨$item_n$⟩ ... ⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. 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 `\unexpanded`, 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 ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ... ⟨$item_n$⟩ becomes ⟨$item_n$⟩ ... ⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. 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`, and, for improved performance, `\tl_reverse_items: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 the slower function `\tl_reverse:n`.

TEXhackers note: The result is returned within `\unexpanded`, 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: 2012-06-25

\tl_trim_spaces:n {⟨*token list*⟩}

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and leaves the result in the input stream.

TEXhackers note: The result is returned within `\unexpanded`, 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 (explicit tokens with character code 32 and category code 10) from the content of the ⟨*tl var*⟩. Note that this therefore *resets* the content of the variable.

9 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

\tl_head:N ⋆
\tl_head:n ⋆
\tl_head:(V|v|f) ⋆

Updated: 2012-09-09

\tl_head:n {⟨*token list*⟩}

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

will both leave `a` in the input stream. If the "head" is a brace group, rather than a single token, the braces will be removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields ␣`ab`. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) will result in `\tl_head:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\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 ⟨*token list*⟩ { } \q_stop

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank ⟨*token list*⟩ (which consists only of space characters) will result in a low-level TEX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a "blank" argument. 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 ⋆
\tl_tail:(V|v|f) ⋆

Updated: 2012-09-01

\tl_tail:n {⟨*token list*⟩}

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*token list*⟩, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

will both leave ␣{bc}d in the input stream. A blank ⟨*token list*⟩ (see \tl_if_blank:nTF) will result in \tl_tail:n leaving nothing in the input stream.

TEXhackers note: The result is returned within \exp_not:n, which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_if_head_eq_catcode_p:nN ⋆
\tl_if_head_eq_catcode:nNTF ⋆

Updated: 2012-07-09

\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 the ⟨*token list*⟩ is empty, the test will always be false.

\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: 2012-07-09

\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 the ⟨*token list*⟩ is empty, the test will always be false.

\tl_if_head_eq_meaning_p:nN ⋆
\tl_if_head_eq_meaning:nNTF ⋆

Updated: 2012-07-09

\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, the test will always be false.

\tl_if_head_is_group_p:n ⋆
\tl_if_head_is_group:nTF ⋆

New: 2012-07-08

\tl_if_head_is_group_p:n {⟨*token list*⟩}
\tl_if_head_is_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 is empty. This function is useful to implement actions on token lists on a token by token basis.

\tl_if_head_is_N_type_p:n ⋆ \tl_if_head_is_N_type_p:n {⟨token list⟩}
\tl_if_head_is_N_type:nTF ⋆ \tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

New: 2012-07-08

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 (explicit token with character code 32 and category code 10) 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_is_space_p:n ⋆ \tl_if_head_is_space_p:n {⟨token list⟩}
\tl_if_head_is_space:nTF ⋆ \tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Updated: 2012-07-08

Tests if the first ⟨token⟩ in the ⟨token list⟩ is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is `false` if the ⟨token list⟩ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

10 Using a single item

\tl_item:nn ⋆ \tl_item:nn {⟨token list⟩} {⟨integer expression⟩}
\tl_item:Nn ⋆
\tl_item:cn ⋆

New: 2014-07-17

Indexing items in the ⟨token list⟩ from 1 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.

11 Viewing token lists

\tl_show:N \tl_show:N ⟨tl var⟩
\tl_show:c

Updated: 2015-08-01

Displays the content of the ⟨tl var⟩ on the terminal.

TEXhackers note: This is similar to the TEX primitive `\show`, wrapped to a fixed number of characters per line.

\tl_show:n \tl_show:n ⟨token list⟩

Updated: 2015-08-07

Displays the ⟨token list⟩ on the terminal.

TEXhackers note: This is similar to the ε-TEX primitive `\showtokens`, wrapped to a fixed number of characters per line.

12 Constant token lists

`\c_empty_tl` Constant that is always empty.

`\c_space_tl` An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

13 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.

14 Internal functions

`\__tl_trim_spaces:nn` `\__tl_trim_spaces:nn { \q_mark ⟨token list⟩ } {⟨continuation⟩}`

This function removes all leading and trailing explicit space characters from the ⟨*token list*⟩, and expands to the ⟨*continuation*⟩, followed by a brace group containing `\use_none:n \q_mark` ⟨*trimmed token list*⟩. For instance, `\tl_trim_spaces:n` is implemented by taking the ⟨*continuation*⟩ to be `\exp_not:o`, and the `o`-type expansion removes the `\q_mark`. This function is also used in l3clist and l3candidates.

Part XII
The l3str package
Strings

TEX associates each character with a category code: as such, there is no concept of a "string" as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense "ignoring" category codes: this is done by treating token lists as strings in a TEX sense.

A TEX string (and thus an expl3 string) is a series of characters which have category code 12 ("other") with the exception of space characters which have category code 10 ("space"). Thus at a technical level, a TEX string is a token list with the appropriate category codes. In this documentation, these will simply be referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and will not treat a token list or the corresponding string representation differently.

Note that as string variables are a special case of token list variables the coverage of `\str_...:N` functions is somewhat smaller than `\tl_...:N`.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) will generate strings from the appropriate input: these are documented in l3basics, l3tl and l3token, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N`
`\str_new:c`
New: 2015-09-18

`\str_new:N` ⟨*str var*⟩

Creates a new ⟨*str var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*str var*⟩ will initially be empty.

`\str_const:Nn`
`\str_const:(Nx|cn|cx)`
New: 2015-09-18

`\str_const:Nn` ⟨*str var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ will be set globally to the ⟨*token list*⟩, converted to a string.

\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c

New: 2015-09-18

\str_clear:N ⟨*str var*⟩

Clears the content of the ⟨*str var*⟩.

\str_clear_new:N
\str_clear_new:c

New: 2015-09-18

\str_clear_new:N ⟨*str var*⟩

Ensures that the ⟨*str var*⟩ exists globally by applying \str_new:N if necessary, then applies \str_(g)clear:N to leave the ⟨*str var*⟩ empty.

\str_set_eq:NN
\str_set_eq:(cN|Nc|cc)
\str_gset_eq:NN
\str_gset_eq:(cN|Nc|cc)

New: 2015-09-18

\str_set_eq:NN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩

Sets the content of ⟨*str var*$_1$⟩ equal to that of ⟨*str var*$_2$⟩.

2 Adding data to string variables

\str_set:Nn
\str_set:(Nx|cn|cx)
\str_gset:Nn
\str_gset:(Nx|cn|cx)

New: 2015-09-18

\str_set:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

\str_put_left:Nn
\str_put_left:(Nx|cn|cx)
\str_gput_left:Nn
\str_gput_left:(Nx|cn|cx)

New: 2015-09-18

\str_put_left:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_put_right:Nn
\str_put_right:(Nx|cn|cx)
\str_gput_right:Nn
\str_gput_right:(Nx|cn|cx)

New: 2015-09-18

\str_put_right:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

2.1 String conditionals

\str_if_exist_p:N ⋆
\str_if_exist_p:c ⋆
\str_if_exist:NTF ⋆
\str_if_exist:cTF ⋆

New: 2015-09-18

\str_if_exist_p:N ⟨*str var*⟩
\str_if_exist:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ is currently defined. This does not check that the ⟨*str var*⟩ really is a string.

\str_if_empty_p:N ⋆
\str_if_empty_p:c ⋆
\str_if_empty:NTF ⋆
\str_if_empty:cTF ⋆

New: 2015-09-18

```
\sr_if_empty_p:N ⟨str var⟩
\str_if_empty:NTF ⟨str var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*string variable*⟩ is entirely empty (*i.e.* contains no characters at all).

\str_if_eq_p:NN ⋆
\str_if_eq_p:(Nc|cN|cc) ⋆
\str_if_eq:NNTF ⋆
\str_if_eq:(Nc|cN|cc)TF ⋆

New: 2015-09-18

```
\str_if_eq_p:NN ⟨str var1⟩ ⟨str var2⟩
\str_if_eq:NNTF ⟨str var1⟩ ⟨str var2⟩ {⟨true code⟩} {⟨false code⟩}
```

Compares the content of two ⟨*str variables*⟩ and is logically `true` if the two contain the same characters.

\str_if_eq_p:nn ⋆
\str_if_eq_p:(Vn|on|no|nV|VV) ⋆
\str_if_eq:nnTF ⋆
\str_if_eq:(Vn|on|no|nV|VV)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:no { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_if_eq_x_p:nn ⋆
\str_if_eq_x:nnTF ⋆

New: 2012-06-05

```
\str_if_eq_x_p:nn {⟨tl1⟩} {⟨tl2⟩}
\str_if_eq_x:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the full expansion of 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_x_p:nn { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_case:nn ⋆
\str_case:(on|nV|nv) ⋆
\str_case:nnTF ⋆
\str_case:(on|nV|nv)TF ⋆

New: 2013-07-24
Updated: 2015-02-28

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

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 any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

`\str_case_x:nnTF` ⋆

New: 2013-07-24

```
\str_case_x:nnTF {⟨test string⟩}
  {
    {⟨string case_1⟩} {⟨code case_1⟩}
    {⟨string case_2⟩} {⟨code case_2⟩}
    ...
    {⟨string case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string cases*⟩. If the two full expansions are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The ⟨*test string*⟩ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

3 Working with the content of strings

`\str_use:N` ⋆
`\str_use:c` ⋆

New: 2015-09-18

`\str_use:N ⟨str var⟩`

Recovers the content of a ⟨*str 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 ⟨*str*⟩ directly without an accessor function.

`\str_count:N` ⋆
`\str_count:c` ⋆
`\str_count:n` ⋆
`\str_count_ignore_spaces:n` ⋆

New: 2015-09-18

`\str_count:n {⟨token list⟩}`

Leaves in the input stream the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ⋆
`\str_count_spaces:c` ⋆
`\str_count_spaces:n` ⋆

New: 2015-09-18

`\str_count_spaces:n {⟨token list⟩}`

Leaves in the input stream the number of space characters in the string representation of ⟨*token list*⟩, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

\str_head:N ⋆ \str_head:n {⟨*token list*⟩}
\str_head:c ⋆
\str_head:n ⋆
\str_head_ignore_spaces:n ⋆

New: 2015-09-18

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the ⟨*string*⟩ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

\str_tail:N ⋆ \str_tail:n {⟨*token list*⟩}
\str_tail:c ⋆
\str_tail:n ⋆
\str_tail_ignore_spaces:n ⋆

New: 2015-09-18

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` will trim only that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

\str_item:Nn ⋆ \str_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}
\str_item:nn ⋆
\str_item_ignore_spaces:nn ⋆

New: 2015-09-18

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, *etc.*

\str_range:Nnn ⋆
\str_range:cnn ⋆
\str_range:nnn ⋆
\str_range_ignore_spaces:nnn ⋆

New: 2015-09-18

\str_range:nnn {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Positive ⟨*indices*⟩ are counted from the start of the string, 1 being the first character, and negative ⟨*indices*⟩ are counted from the end of the string, −1 being the last character. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is 0, the result is empty. For instance,

```
\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }
```

will print `bcde`, `cdef`, `ef`, and an empty line to the terminal. The ⟨*start index*⟩ must always be smaller than or equal to the ⟨*end index*⟩: if this is not the case then no output is generated. Thus

```
\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

4 String manipulation

\str_lower_case:n ☆
\str_lower_case:f ☆
\str_upper_case:n ☆
\str_upper_case:f ☆

New: 2015-03-01

\str_lower_case:n {⟨*tokens*⟩}
\str_upper_case:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
  {
    \cs_set_protected:cpn
      {
        user
        \str_upper_case:f { \tl_head:n {#1} }
        \str_lower_case:f { \tl_tail:n {#1} }
      }
      { #2 }
  }
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_fold_case:n` for this situation (case folding is district from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

TEXhackers note: As with all expl3 functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XETEX and LuaTEX, subject only to the fact that XETEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

\str_fold_case:n ⋆
\str_fold_case:V ⋆

New: 2014-06-19
Updated: 2016-03-07

\str_fold_case:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for \tl_to_str:n, and then folds the case of the resulting ⟨*string*⟩ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for "text". The folding provided by \str_fold_case:n follows the mappings provided by the Unicode Consortium, who state:

> Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by \str_fold_case:n follows the "full" scheme defined by the Unicode Consortium (*e.g.* SSfolds to ss). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TEXhackers note: As with all expl3 functions, the input supported by \str_fold_case:n is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XETEX and LuaTEX, subject only to the fact that XETEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with \tl_to_str:n.

5 Viewing strings

\str_show:N
\str_show:c
\str_show:n

New: 2015-09-18

\str_show:N ⟨*str var*⟩

Displays the content of the ⟨*str var*⟩ on the terminal.

6 Constant token lists

\c_ampersand_str
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str

Constant strings, containing a single character token, with category code 12.

New: 2015-09-19

7 Scratch strings

\l_tmpa_str
\l_tmpb_str

Scratch strings 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_str
\g_tmpb_str

Scratch strings 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.

7.1 Internal string functions

__str_if_eq_x:nn ⋆ `\__str_if_eq_x:nn {`⟨*tl*$_1$⟩`} {`⟨*tl*$_2$⟩`}`

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Leaves `0` in the input stream if the condition is true, and `+1` or `-1` otherwise.

__str_if_eq_x_return:nn `\__str_if_eq_x_return:nn {`⟨*tl*$_1$⟩`} {`⟨*tl*$_2$⟩`}`

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Either `\prg_return_true:` or `\prg_return_false:` is then left in the input stream. This is a version of `\str_if_eq_x:nnTF` coded for speed.

__str_to_other:n ⋆ `\__str_to_other:n {`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code "other". This function can be `f`-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

__str_count:n ⋆ `\__str_count:n {⟨other string⟩}`

This function expects an argument that is entirely made of characters with category "other", as produced by `\__str_to_other:n`. It leaves in the input stream the number of character tokens in the ⟨*other string*⟩, faster than the analogous `\str_count:n` function.

__str_range:nnn ⋆ `\__str_range:nnn {⟨other string⟩} {⟨start index⟩} {⟨end index⟩}`

Identical to `\str_range:nnn` except that the first argument is expected to be entirely made of characters with category "other", as produced by `\__str_to_other:n`, and the result is also an ⟨*other string*⟩.

Part XIII
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.

1 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` ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩

Sets the content of ⟨*sequence*$_1$⟩ equal to that of ⟨*sequence*$_2$⟩.

`\seq_set_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc)`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc)`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

New: 2014-07-17

`\seq_set_from_clist:NN` ⟨*sequence*⟩ ⟨*comma-list*⟩

Converts the data in the ⟨*comma list*⟩ into a ⟨*sequence*⟩: the original ⟨*comma list*⟩ is unchanged.

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV

New: 2011-08-15
Updated: 2012-07-02

\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 ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ ⟨*sequence*$_3$⟩

Concatenates the content of ⟨*sequence*$_2$⟩ and ⟨*sequence*$_3$⟩ together and saves the result in ⟨*sequence*$_1$⟩. The items in ⟨*sequence*$_2$⟩ 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.

2 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*⟩.

3 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

Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_get_right:NN
\seq_get_right:cN
Updated: 2012-05-19

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_pop_left:NN
\seq_pop_left:cN
Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_gpop_left:NN
\seq_gpop_left:cN
Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_pop_right:NN
\seq_pop_right:cN
Updated: 2012-05-19

\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 the ⟨*token list variable*⟩. Both of the variables are assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_gpop_right:NN
\seq_gpop_right:cN
Updated: 2012-05-19

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_item:Nn ⋆
\seq_item:cn ⋆
New: 2014-07-17

\seq_item:Nn ⟨*sequence*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*sequence*⟩ from 1 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_count: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.

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

\seq_get_left:NNTF
\seq_get_left:cNTF

New: 2012-05-14
Updated: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_get_right:NNTF
\seq_get_right:cNTF

New: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_pop_left:NNTF
\seq_pop_left:cNTF

New: 2012-05-14
Updated: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop_left:NNTF
\seq_gpop_left:cNTF

New: 2012-05-14
Updated: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\seq_pop_right:NNTF
\seq_pop_right:cNTF

New: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

New: 2012-05-19

\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. 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.

5 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:nnTF.

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:nnTF.

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c

New: 2014-07-18

\seq_reverse:N ⟨*sequence*⟩

Reverses the order of the items stored in the ⟨*sequence*⟩.

6 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*⟩.

7 Mapping to sequences

\seq_map_function:NN ☆
\seq_map_function:cN ☆

Updated: 2012-06-29

\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 faster than \seq_map_function:NN for sequences with more than about 10 items. One mapping may be nested inside another.

\seq_map_inline:Nn
\seq_map_inline:cn

Updated: 2012-06-29

\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)

Updated: 2012-06-29

\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: ☆

Updated: 2012-06-29

\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:Nn before further items are taken from the input stream. This will depend on the design of the mapping function.

\seq_map_break:n ☆

Updated: 2012-06-29

\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:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\seq_count:N ⋆
\seq_count:c ⋆

New: 2012-07-13

\seq_count: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.

8 Using the content of sequences directly

\seq_use:Nnnn ⋆
\seq_use:cnnn ⋆

New: 2013-05-26

\seq_use:Nnnn ⟨*seq var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the sequence has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

will insert "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ will not expand further when appearing in an `x`-type argument expansion.

\seq_use:Nn ⋆
\seq_use:cn ⋆

New: 2013-05-26

\seq_use:Nn ⟨*seq var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the sequence has a single item, it is placed in the input stream with no ⟨*separator*⟩, and an empty sequence produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

will insert "`a and b and c and de and f`" in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ will not expand further when appearing in an `x`-type argument expansion.

9 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
Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_pop:NN
\seq_pop:cN
Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_gpop:NN
\seq_gpop:cN
Updated: 2012-05-14

\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 the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_get:NN*TF*
\seq_get:cN*TF*
New: 2012-05-14
Updated: 2012-05-19

\seq_get:NNTF ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the top item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_pop:NN*TF*
\seq_pop:cN*TF*
New: 2012-05-14
Updated: 2012-05-19

\seq_pop:NNTF ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the top item from the ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop:NN*TF*
\seq_gpop:cN*TF*
New: 2012-05-14
Updated: 2012-05-19

\seq_gpop:NNTF ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the top item from the ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\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*⟩.

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurences of a given item. To make sure that a ⟨*sequence variable*⟩ only has distinct items, use `\seq_remove_duplicates:N` ⟨*sequence variable*⟩. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set ⟨*seq var*⟩ are straightforward. For instance, `\seq_count:N` ⟨*seq var*⟩ expands to the number of items, while `\seq_if_in:NnTF` ⟨*seq var*⟩ `{`⟨*item*⟩`}` tests if the ⟨*item*⟩ is in the set.

Adding an ⟨*item*⟩ to a set ⟨*seq var*⟩ can be done by appending it to the ⟨*seq var*⟩ if it is not already in the ⟨*seq var*⟩:

```
\seq_if_in:NnF ⟨seq var⟩ {⟨item⟩}
{ \seq_put_right:Nn ⟨seq var⟩ {⟨item⟩} }
```

Removing an ⟨*item*⟩ from a set ⟨*seq var*⟩ can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}
```

The intersection of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by collecting items of ⟨*seq var*$_1$⟩ which are in ⟨*seq var*$_2$⟩.

```
\seq_clear:N ⟨seq var3⟩
\seq_map_inline:Nn ⟨seq var1⟩
{
\seq_if_in:NnT ⟨seq var2⟩ {#1}
{ \seq_put_right:Nn ⟨seq var3⟩ {#1} }
}
```

The code as written here only works if ⟨*seq var*$_3$⟩ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence `\l__`⟨*pkg*⟩`_internal_seq`, then ⟨*seq var*$_3$⟩ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ through

```
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var1⟩ ⟨seq var2⟩
\seq_remove_duplicates:N ⟨seq var3⟩
```

or by adding items to (a copy of) ⟨*seq var*$_1$⟩ one by one

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
{
\seq_if_in:NnF ⟨seq var3⟩ {#1}
{ \seq_put_right:Nn ⟨seq var3⟩ {#1} }
}
```

The second approach is faster than the first when the ⟨*seq var*$_2$⟩ is short compared to ⟨*seq var*$_1$⟩.

The difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by removing items of the ⟨*seq var*$_2$⟩ from (a copy of) the ⟨*seq var*$_1$⟩ one by one.

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
{ \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
```

The symmetric difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by computing the difference between ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ and storing the result as `\l__`⟨*pkg*⟩`_internal_seq`, then the difference between ⟨*seq var*$_2$⟩ and ⟨*seq var*$_1$⟩, and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l__⟨pkg⟩_internal_seq ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
{ \seq_remove_all:Nn \l__⟨pkg⟩_internal_seq {#1} }
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var2⟩
\seq_map_inline:Nn ⟨seq var1⟩
{ \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var3⟩ \l__⟨pkg⟩_internal_seq
```

11 Constant and scratch sequences

`\c_empty_seq`

New: 2012-07-02

Constant that is always empty.

`\l_tmpa_seq`
`\l_tmpb_seq`

New: 2012-04-26

Scratch sequences 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_seq`
`\g_tmpb_seq`

New: 2012-04-26

Scratch sequences 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.

12 Viewing sequences

`\seq_show:N`
`\seq_show:c`

Updated: 2015-08-01

`\seq_show:N` ⟨*sequence*⟩

Displays the entries in the ⟨*sequence*⟩ in the terminal.

13 Internal sequence functions

\s__seq

This scan mark (equal to \scan_stop:) marks the beginning of a sequence variable.

__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.

Part XIV
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. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual TEX category codes apply).

1 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_const:Nn` `\clist_const:(Nx|cn|cx)`

New: 2014-07-05

`\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_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 `\clist_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 list*$_1$⟩ ⟨*comma list*$_2$⟩

Sets the content of ⟨*comma list*$_1$⟩ equal to that of ⟨*comma list*$_2$⟩.

\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

New: 2014-07-17

`\clist_set_from_seq:NN` ⟨*comma list*⟩ ⟨*sequence*⟩

Converts the data in the ⟨*sequence*⟩ into a ⟨*comma list*⟩: the original ⟨*sequence*⟩ is unchanged. Items which contain either spaces or commas are surrounded by braces.

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

`\clist_concat:NNN` ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩ ⟨*comma list*$_3$⟩

Concatenates the content of ⟨*comma list*$_2$⟩ and ⟨*comma list*$_3$⟩ together and saves the result in ⟨*comma list*$_1$⟩. The items in ⟨*comma list*$_2$⟩ 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.

2 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*⟩ {⟨*item*$_1$⟩,...,⟨*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*⟩ {⟨*item*$_1$⟩,...,⟨*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*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the right of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

3 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).

\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

New: 2014-07-18

\clist_reverse:N ⟨*comma list*⟩

Reverses the order of items stored in the ⟨*comma list*⟩.

\clist_reverse:n

New: 2014-07-18

\clist_reverse:n {⟨*comma list*⟩}

Leaves the items in the ⟨*comma list*⟩ in the input stream in reverse order. Braces and spaces are preserved by this process.

TEXhackers note: The result is returned within `\unexpanded`, which means that the comma list will not expand further when appearing in an `x`-type argument expansion.

4 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_empty_p:n ⋆
\clist_if_empty:nTF ⋆

New: 2014-07-05

\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.

\clist_if_in:NnTF
\clist_if_in:(NV|No|cn|cV|co)TF
\clist_if_in:nnTF
\clist_if_in:(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.

5 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 ☆
\clist_map_function:nN ☆

Updated: 2012-06-29

\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
\clist_map_inline:nn

Updated: 2012-06-29

\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
\clist_map_variable:nNn

Updated: 2012-06-29

\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: ☆

Updated: 2012-06-29

\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:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\clist_map_break:n ☆

Updated: 2012-06-29

\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:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\clist_count:N ⋆
\clist_count:c ⋆
\clist_count:n ⋆

New: 2012-07-13

\clist_count: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.

6 Using the content of comma lists directly

\clist_use:Nnnn ⋆
\clist_use:cnnn ⋆

New: 2013-05-26

\clist_use:Nnnn ⟨*clist var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

will insert "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ will not expand further when appearing in an `x`-type argument expansion.

\clist_use:Nn ⋆
\clist_use:cn ⋆

New: 2013-05-26

\clist_use:Nn ⟨*clist var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

will insert "`a and b and c and de and f`" in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ will not expand further when appearing in an `x`-type argument expansion.

7 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
Updated: 2012-05-14

\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. If the ⟨*comma list*⟩ is empty the ⟨*token list variable*⟩ will contain the marker value `\q_no_value`.

\clist_get:NN*TF*
\clist_get:cN*TF*
New: 2012-05-14

\clist_get:NNTF ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, stores the top item from the ⟨*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_pop:NN*TF*
\clist_pop:cN*TF*
New: 2012-05-14

\clist_pop:NNTF ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, pops the top item from the ⟨*comma list*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*comma list*⟩. Both the ⟨*comma list*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\clist_gpop:NN*TF*
\clist_gpop:cN*TF*
New: 2012-05-14

\clist_gpop:NNTF ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*comma list*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*comma list*⟩ is non-empty, pops the top item from the ⟨*comma list*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from the ⟨*comma list*⟩. The ⟨*comma list*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\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.

8 Using a single item

\clist_item:Nn ⋆
\clist_item:cn ⋆
\clist_item:nn ⋆

New: 2014-07-17

\clist_item:Nn ⟨*comma list*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*comma list*⟩ from 1 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_count: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.

9 Viewing comma lists

\clist_show:N
\clist_show:c

Updated: 2015-08-03

\clist_show:N ⟨*comma list*⟩

Displays the entries in the ⟨*comma list*⟩ in the terminal.

\clist_show:n

Updated: 2013-08-03

\clist_show:n {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

10 Constant and scratch comma lists

\c_empty_clist

New: 2012-07-02

Constant that is always empty.

\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.

Part XV
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.

1 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 list*⟩ 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 list*$_1$⟩ ⟨*property list*$_2$⟩

Sets the content of ⟨*property list*$_1$⟩ equal to that of ⟨*property list*$_2$⟩.

2 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)

Updated: 2012-07-09

\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.

3 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. See also `\prop_pop:NnNTF`.

\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. See also `\prop_gpop:NnNTF`.

\prop_item:Nn ⋆
\prop_item:cn ⋆

New: 2014-07-17

\prop_item: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.

4 Modifying property lists

\prop_remove:Nn
\prop_remove:(NV|cn|cV)
\prop_gremove:Nn
\prop_gremove:(NV|cn|cV)

New: 2012-05-12

\prop_remove:Nn ⟨*property list*⟩ {⟨*key*⟩}

Removes the entry listed under ⟨*key*⟩ from the ⟨*property list*⟩. 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.

5 Property list conditionals

\prop_if_exist_p:N ⋆
\prop_if_exist_p:c ⋆
\prop_if_exist:NTF ⋆
\prop_if_exist:cTF ⋆

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:NTF ⋆
\prop_if_empty:cTF ⋆

\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:NnTF ⋆
\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`.

6 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:NnN*TF*
\prop_get:(NVN|NoN|cnN|cVN|coN)*TF*

Updated: 2012-05-19

```
\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. 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*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*token list variable*⟩ is assigned locally.

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

```
\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. 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.

\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

```
\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. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. 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.

7 Mapping to property lists

\prop_map_function:NN ☆
\prop_map_function:cN ☆

Updated: 2013-01-08

```
\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

Updated: 2013-01-08

```
\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: ☆

Updated: 2012-06-29

\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 to low level TEX errors.

\prop_map_break:n ☆

Updated: 2012-06-29

\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 to low level TEX errors.

8 Viewing property lists

\prop_show:N
\prop_show:c

Updated: 2015-08-01

\prop_show:N ⟨*property list*⟩

Displays the entries in the ⟨*property list*⟩ in the terminal.

9 Scratch property lists

\l_tmpa_prop
\l_tmpb_prop

New: 2012-06-23

Scratch property 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_prop
\g_tmpb_prop

New: 2012-06-23

Scratch property 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.

10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

11 Internal property list functions

\s__prop

The internal token used at the beginning of property lists. This is also used after each ⟨*key*⟩ (see `\__prop_pair:wn`).

__prop_pair:wn

`\__prop_pair:wn` ⟨*key*⟩ `\s__prop {`⟨*item*⟩`}`

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

\l__prop_internal_tl

Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

__prop_split:NnTF

`\__prop_split:NnTF` ⟨*property list*⟩ `{`⟨*key*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Updated: 2013-01-08

Splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three token lists: the ⟨*extract*⟩ of ⟨*property list*⟩ before the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*⟩ of the ⟨*property list*⟩ after the ⟨*value*⟩. Both ⟨*extracts*⟩ retain the internal structure of a property list, and 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, with `#1`, `#2`, and `#3` replaced by the first ⟨*extract*⟩, the ⟨*value*⟩, and the second extract. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the ⟨*false code*⟩ is left in the input stream, with no trailing material. Both ⟨*true code*⟩ and ⟨*false code*⟩ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except `#1`, `#2`, and `#3` which stand in the ⟨*true code*⟩ for the three extracts from the property list. The ⟨*key*⟩ comparison takes place as described for `\str_if_eq:nn`.

Part XVI
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_`.

1 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` ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets the content of ⟨*box*$_1$⟩ equal to that of ⟨*box*$_2$⟩.

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

`\box_set_eq_clear:NN` ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets the content of ⟨*box*$_1$⟩ within the current TEX group equal to that of ⟨*box*$_2$⟩, then clears ⟨*box*$_2$⟩ globally.

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

`\box_gset_eq_clear:NN` ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets the content of ⟨*box*$_1$⟩ equal to that of ⟨*box*$_2$⟩, then clears ⟨*box*$_2$⟩. 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.

2 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 }.

3 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.

4 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.

5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

\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.

6 Constant boxes

\c_empty_box

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

7 Scratch boxes

\l_tmpa_box
\l_tmpb_box

Updated: 2012-11-04

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.

\g_tmpa_box
\g_tmpb_box

Scratch boxes 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.

8 Viewing box contents

\box_show:N
\box_show:c

Updated: 2012-05-11

\box_show:N ⟨*box*⟩

Shows full details of the content of the ⟨*box*⟩ in the terminal.

\box_show:Nnn
\box_show:cnn

New: 2012-05-11

\box_show:Nnn ⟨*box*⟩ ⟨*intexpr*$_1$⟩ ⟨*intexpr*$_2$⟩

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ group levels.

\box_log:N
\box_log:c

New: 2012-05-11

\box_log:N ⟨*box*⟩

Writes full details of the content of the ⟨*box*⟩ to the log.

\box_log:Nnn
\box_log:cnn

New: 2012-05-11

\box_log:Nnn ⟨*box*⟩ ⟨*intexpr*$_1$⟩ ⟨*intexpr*$_2$⟩

Writes the contents of ⟨*box*⟩ to the log, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ group levels.

9 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.

\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.

10 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` `\vbox:n {⟨contents⟩}`

Updated: 2011-12-18

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` `\vbox_top:n {⟨contents⟩}`

Updated: 2011-12-18

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 be equal to that of the *first* item added to the box.

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

`\vbox_to_ht:nn` `\vbox_to_ht:nn {⟨dimexpr⟩} {⟨contents⟩}`

Updated: 2011-12-18

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` `\vbox_to_zero:n {⟨contents⟩}`

Updated: 2011-12-18

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:Nn ⟨box⟩ {⟨contents⟩}`
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

Updated: 2011-12-18

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩.

`\vbox_set_top:Nn` `\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}`
`\vbox_set_top:cn`
`\vbox_gset_top:Nn`
`\vbox_gset_top:cn`

Updated: 2011-12-18

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. The baseline of the box will be equal to that of the *first* item added to the box.

`\vbox_set_to_ht:Nnn` `\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`

Updated: 2011-12-18

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_set: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` ⟨box_1⟩ ⟨box_2⟩ {⟨*dimexpr*⟩}

Sets ⟨box_1⟩ to contain material to the height given by the ⟨*dimexpr*⟩ by removing content from the top of ⟨box_2⟩ (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`.

11 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`.

Part XVII
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).

1 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 ⟨*coffin*$_1$⟩ ⟨*coffin*$_2$⟩

Sets both the content and poles of ⟨*coffin*$_1$⟩ equal to those of ⟨*coffin*$_2$⟩ within the current TEX group level.

\coffin_if_exist_p:N ⋆
\coffin_if_exist_p:c ⋆
\coffin_if_exist:NTF ⋆
\coffin_if_exist:cTF ⋆
New: 2012-06-20

\coffin_if_exist_p:N ⟨*box*⟩
\coffin_if_exist:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*coffin*⟩ is currently defined.

2 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: 2012-05-22

`\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
Updated: 2012-05-22

`\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: 2012-07-20

`\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.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn

New: 2012-07-20

`\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.

3 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)

`\coffin_attach:NnnNnnnn`
⟨*coffin*$_1$⟩ {⟨*coffin*$_1$*-pole*$_1$⟩} {⟨*coffin*$_1$*-pole*$_2$⟩}
⟨*coffin*$_2$⟩ {⟨*coffin*$_2$*-pole*$_1$⟩} {⟨*coffin*$_2$*-pole*$_2$⟩}
{⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function attaches ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ is not altered, *i.e.* ⟨*coffin*$_2$⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ 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
  ⟨*coffin*$_1$⟩ {⟨*coffin*$_1$*-pole*$_1$⟩} {⟨*coffin*$_1$*-pole*$_2$⟩}
  ⟨*coffin*$_2$⟩ {⟨*coffin*$_2$*-pole*$_1$⟩} {⟨*coffin*$_2$*-pole*$_2$⟩}
  {⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function joins ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ 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 ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

Updated: 2012-07-20

\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 ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. The coffin is then typeset in horizontal mode 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.

4 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*⟩.

5 Coffin diagnostics

\coffin_display_handles:Nn
\coffin_display_handles:cn

Updated: 2011-09-02

\coffin_display_handles:Nn ⟨*coffin*⟩ {⟨*color*⟩}

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 ⟨*color*⟩ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
Updated: 2011-09-02

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. 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 ⟨*color*⟩ specified.

\coffin_show_structure:N
\coffin_show_structure:c
Updated: 2015-08-01

\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.

5.1 Constants and variables

\c_empty_coffin

A permanently empty coffin.

\l_tmpa_coffin
\l_tmpb_coffin
New: 2012-06-19

Scratch coffins 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.

Part XVIII
The l3color package
Color support

This module provides support for color in LaTeX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color 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 color group: one used to "trap" color settings.

`\color_ensure_current:`

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use the foreground color 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 XIX
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.

1 Creating new messages

All messages have to be created before they can be used. 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. 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.

\msg_if_exist_p:nn ⋆
\msg_if_exist:nnTF ⋆

\msg_if_exist_p:nn {⟨*module*⟩} {⟨*message*⟩}
\msg_if_exist:nnTF {⟨*module*⟩} {⟨*message*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

New: 2012-03-03

Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

2 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.

\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.

\msg_see_documentation_text:n ⋆ `\msg_see_documentation_text:n {⟨module⟩}`

Produces the standard text

`See the ⟨module⟩ documentation for further information.`

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.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be converted to strings before being added to the message text: the `x`-type variants should be used to expand material.

\msg_fatal:nnnnnn
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn

`\msg_fatal:nnnnnn {⟨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.

Updated: 2012-08-11

\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

`\msg_critical:nnnnnn {⟨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 critical error, TEX will stop reading the current input file. This may halt the TEX run (if the current file is the main file) or may abort reading a sub-file.

TEXhackers note: The TEX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

Updated: 2012-08-11

\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn

Updated: 2012-08-11

\msg_error:nnnnnn {⟨*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 interrupt processing and issue the text at the terminal. After user input, the run will continue.

\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

Updated: 2012-08-11

\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 and the terminal, but the TEX run will not be interrupted.

\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn

Updated: 2012-08-11

\msg_info:nnnnnn {⟨*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:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn

Updated: 2012-08-11

\msg_log:nnnnnn {⟨*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:nnnnnn.

\msg_none:nnnnnn
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_none:nnx
\msg_none:nn

Updated: 2012-08-11

\msg_none:nnnnnn {⟨*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).

4 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.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class will raise errors immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \to B$, $B \to C$ and $C \to A$ in this order, then the $A \to B$ redirection is cancelled.

\msg_redirect_class:nn

Updated: 2012-04-27

\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*⟩.

\msg_redirect_module:nnn

\msg_redirect_module:nnn {⟨*module*⟩} {⟨*class one*⟩} {⟨*class two*⟩}

Updated: 2012-04-27

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

\msg_redirect_name:nnn {⟨*module*⟩} {⟨*message*⟩} {⟨*class*⟩}

Updated: 2012-04-27

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages. No further redirection is performed. This function can be used to make a selected message "silent" without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

5 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_interrupt:nnn

\msg_interrupt:nnn {⟨*first line*⟩} {⟨*text*⟩} {⟨*extra text*⟩}

New: 2012-06-28

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 within the current line length. Wrapping of both ⟨*text*⟩ and ⟨*more text*⟩ takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

\msg_log:n

New: 2012-06-28

`\msg_log:n {⟨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. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

\msg_term:n

New: 2012-06-28

`\msg_term:n {⟨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. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 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 and expanded at the time 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 and expanded at the time the message is used.

__msg_kernel_fatal:nnnnnn
__msg_kernel_fatal:nnxxxx
__msg_kernel_fatal:nnnnn
__msg_kernel_fatal:nnxxx
__msg_kernel_fatal:nnnn
__msg_kernel_fatal:nnxx
__msg_kernel_fatal:nnn
__msg_kernel_fatal:nnx
__msg_kernel_fatal:nn

Updated: 2012-08-11

__msg_kernel_fatal:nnnnnn {⟨*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:nnnnnn
__msg_kernel_error:nnxxxx
__msg_kernel_error:nnnnn
__msg_kernel_error:nnxxx
__msg_kernel_error:nnnn
__msg_kernel_error:nnxx
__msg_kernel_error:nnn
__msg_kernel_error:nnx
__msg_kernel_error:nn

Updated: 2012-08-11

__msg_kernel_error:nnnnnn {⟨*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:nnnnnn
__msg_kernel_warning:nnxxxx
__msg_kernel_warning:nnnnn
__msg_kernel_warning:nnxxx
__msg_kernel_warning:nnnn
__msg_kernel_warning:nnxx
__msg_kernel_warning:nnn
__msg_kernel_warning:nnx
__msg_kernel_warning:nn

Updated: 2012-08-11

__msg_kernel_warning:nnnnnn {⟨*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:nnnnnn
__msg_kernel_info:nnxxxx
__msg_kernel_info:nnnnn
__msg_kernel_info:nnxxx
__msg_kernel_info:nnnn
__msg_kernel_info:nnxx
__msg_kernel_info:nnn
__msg_kernel_info:nnx
__msg_kernel_info:nn

Updated: 2012-08-11

__msg_kernel_info:nnnnnn {⟨*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.

7 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_kernel_expandable_error:nnnnnn` ⋆
`\__msg_kernel_expandable_error:nnnnn` ⋆
`\__msg_kernel_expandable_error:nnnn` ⋆
`\__msg_kernel_expandable_error:nnn` ⋆
`\__msg_kernel_expandable_error:nn` ⋆

New: 2011-11-23

`\__msg_kernel_expandable_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.

8 Internal l3msg functions

The following functions are used in several kernel modules.

`\__msg_log_next:`

New: 2015-08-05

`\__msg_log_next:` ⟨*show-command*⟩

Causes the next ⟨*show-command*⟩ to send its output to the log file instead of the terminal. This allows for instance `\cs_log:N` to be defined as `\__msg_log_next: \cs_show:N`. The effect of this command lasts until the next use of `\__msg_show_wrap:Nn` or `\__msg_show_wrap:n` or `\__msg_show_variable:NNNnn`, in other words until the next time the ε-TeX primitive `\showtokens` would have been used for showing to the terminal or until the next `variable-not-defined` error.

`\__msg_show_pre:nnnnnn`
`\__msg_show_pre:(nnxxxx|nnnnnV)`

New: 2015-08-05

`\__msg_show_pre:nnnnnn {`⟨*module*⟩`} {`⟨*message*⟩`} {`⟨*arg one*⟩`} {`⟨*arg two*⟩`} {`⟨*arg three*⟩`} {`⟨*arg four*⟩`}`

Prints the ⟨*message*⟩ from ⟨*module*⟩ in the terminal (or log file if `\__msg_log_next:` was issued) without formatting. Used in messages which print complex variable contents completely.

`\__msg_show_variable:NNNnn` `\__msg_show_variable:NNNnn ⟨variable⟩ ⟨if-exist⟩ ⟨if-empty⟩ {⟨msg⟩} {⟨formatted content⟩}`

New: 2015-08-04

If the ⟨*variable*⟩ does not exist according to ⟨*if-exist*⟩ (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if ⟨*msg*⟩ is not empty, display the message `LaTeX/kernel/show-`⟨*msg*⟩ with `\token_to_str:N` ⟨*variable*⟩ as a first argument, and a second argument that is `?` or empty depending on the result of ⟨*if-empty*⟩ (typically `\tl_if_empty:NTF`) on the ⟨*variable*⟩. Then display the ⟨*formatted content*⟩ by giving it as an argument to `\__msg_show_wrap:n`.

`\__msg_show_wrap:Nn` `\__msg_show_wrap:Nn ⟨function⟩ {⟨expression⟩}`

New: 2015-08-03
Updated: 2015-08-07

Shows or logs the ⟨*expression*⟩ (turned into a string), an equal sign, and the result of applying the ⟨*function*⟩ to the `{`⟨*expression*⟩`}`. For instance, if the ⟨*function*⟩ is `\int_eval:n` and the ⟨*expression*⟩ is `1+2` then this will log `> 1+2=3.` The case where the ⟨*function*⟩ is `\tl_to_str:n` is special: then the string representation of the ⟨*expression*⟩ is only logged once.

`\__msg_show_wrap:n` `\__msg_show_wrap:n {⟨formatted text⟩}`

New: 2015-08-03

Shows or logs the ⟨*formatted text*⟩. After expansion, unless it is empty, the ⟨*formatted text*⟩ must contain `>`, and the part of ⟨*formatted text*⟩ before the first `>` is removed. Failure to do so causes low-level TEX errors.

`\__msg_show_item:n` `\__msg_show_item:n ⟨item⟩`
`\__msg_show_item:nn` `\__msg_show_item:nn ⟨item-key⟩ ⟨item-value⟩`
`\__msg_show_item_unbraced:nn`

Updated: 2012-09-09

Auxiliary functions used within the last argument of `\__msg_show_variable:NNNnn` or `\__msg_show_wrap:n` to format variable items correctly for display. The `\__msg_show_item:n` version is used for simple lists, the `\__msg_show_item:nn` and `\__msg_show_item_unbraced:nn` versions for key–value like data structures.

`\c__msg_coding_error_text_tl`

The text

```
This is a coding error.
```

used by kernel functions when erroneous programming input is encountered.

Part XX
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. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

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 { mymodule }
  {
    key-one .code:n   = code including parameter #1,
    key-two .tl_set:N = \l_mymodule_store_tl
  }
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
  {
    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 \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 }  }
\DeclareDocumentCommand \MyModuleMacro { o m }
  {
    \group_begin:
      \keys_set:nn { mymodule } { #1 }
      % Main code for \MyModuleMacro
    \group_end:
  }
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`; spaces are *ignored* in key names. As will be discussed in section 2, 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_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  {
    \l_mymodule_tmp_tl .code:n = code
  }
```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn` Updated: 2015-11-07

`\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:n = true
  }
```

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 one or more arguments. 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.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define "actions", such as `.code:n`, `.tl_set:N`, *etc.*, will override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so will replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
  {
    keyname .code:n           = Some~code~using~#1,
    keyname .value_required:n = true
  }
\keys_define:nn { mymodule }
```

```
{
  keyname .value_required:n = true,
  keyname .code:n           = Some~code~using~#1
}
```

Note that with the exception of the special `.undefine:` property, all key properties will define the key within the current TEX scope.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

Updated: 2013-07-08

⟨*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 globally at the point that the key is set up.

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

New: 2011-08-28
Updated: 2013-07-08

⟨*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 globally at the point that the key is set up.

`.choice:`

⟨*key*⟩ `.choice:`

Sets ⟨*key*⟩ to act as a choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 3.

`.choices:nn`
`.choices:(Vn|on|xn)`

New: 2011-08-21
Updated: 2013-07-10

⟨*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 1). Choices are discussed in detail in section 3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

New: 2011-09-11

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

Defines ⟨*key*⟩ to 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 globally at the point that the key is set up.

`.code:n`

Updated: 2013-07-10

⟨*key*⟩ `.code:n = {`⟨*code*⟩`}`

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is used. 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|o|x)`
Updated: 2013-07-09

⟨*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 { mymodule }
  {
    key .code:n    = Hello~#1,
    key .default:n = World
  }
\keys_set:nn { mymodule }
  {
    key = Fred, % Prints 'Hello Fred'
    key,        % Prints 'Hello World'
    key = ,     % Prints 'Hello '
  }
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value will not trigger an error.

`.dim_set:N`
`.dim_set:c`
`.dim_gset:N`
`.dim_gset: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 globally at the point that the key is set up.

`.fp_set:N`
`.fp_set:c`
`.fp_gset:N`
`.fp_gset:c`

⟨*key*⟩ `.fp_set:N =` ⟨*floating point*⟩

Defines ⟨*key*⟩ to set ⟨*floating point*⟩ to ⟨*value*⟩ (which must a floating point expression). If the variable does not exist, it will be created globally at the point that the key is set up.

`.groups:n`
New: 2013-07-14

⟨*key*⟩ `.groups:n =` {⟨*groups*⟩}

Defines ⟨*key*⟩ as belonging to the ⟨*groups*⟩ declared. Groups provide a "secondary axis" for selectively setting keys, and are described in Section 6.

`.initial:n`
`.initial:(V|o|x)`
Updated: 2013-07-09

⟨*key*⟩ `.initial:n =` {⟨*value*⟩}

Initialises the ⟨*key*⟩ with the ⟨*value*⟩, equivalent to

`\keys_set:nn {`⟨*module*⟩`} {` ⟨*key*⟩ = ⟨*value*⟩ }

`.int_set:N`
`.int_set:c`
`.int_gset:N`
`.int_gset: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 globally at the point that the key is set up.

`.meta:n`
Updated: 2013-07-10

⟨*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`).

`.meta:nn` — ⟨*key*⟩ `.meta:nn = {`⟨*path*⟩`} {`⟨*keyval list*⟩`}`

New: 2013-07-10

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go using the ⟨*path*⟩ in place of the current one. 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 3.

`.multichoices:nn` — ⟨*key*⟩ `.multichoices:nn {`⟨*choices*⟩`} {`⟨*code*⟩`}`

`.multichoices:(Vn|on|xn)`

New: 2011-08-21
Updated: 2013-07-10

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 1). Choices are discussed in detail in section 3.

`.skip_set:N` — ⟨*key*⟩ `.skip_set:N =` ⟨*skip*⟩

`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it will be created globally at the point that the key is set up.

`.tl_set:N` — ⟨*key*⟩ `.tl_set:N =` ⟨*token list variable*⟩

`.tl_set:c`
`.tl_gset:N`
`.tl_gset:c`

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩. If the variable does not exist, it will be created globally at the point that the key is set up.

`.tl_set_x:N` — ⟨*key*⟩ `.tl_set_x:N =` ⟨*token list variable*⟩

`.tl_set_x:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

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 globally at the point that the key is set up.

`.undefine:` — ⟨*key*⟩ `.undefine:`

New: 2015-07-14

Removes the definition of the ⟨*key*⟩ within the current scope.

`.value_forbidden:n` — ⟨*key*⟩ `.value_forbidden:n = true|false`

New: 2015-07-14

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued. Setting the property `false` will cancel the restriction.

`.value_required:n` — ⟨*key*⟩ `.value_required:n = true|false`

New: 2015-07-14

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued. Setting the property `false` will cancel the restriction.

2 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 { mymodule }
  { 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.

3 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 { mymodule }
  { 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 all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.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 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

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 { mymodule }
  {
    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 `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
    key / unknown  .code:n =
      \msg_error:nnxxx { mymodule } { unknown-choice }
        { key }                             % Name of choice key
        { choice-a , choice-b ,  choice-c }  % Valid choices
        { \exp_not:n {#1} }                 % Invalid choice given
    %
    %
  }
```

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 { mymodule }
  {
    key .multichoices:nn =
      { choice-a, choice-b, choice-c }
```

```
        {
          You~gave~choice~'\tl_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 { mymodule }
  {
    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.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
  {
    key = { a , b , c } % 'key' defined as a multiple choice
  }
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
  {
    key = a ,
    key = b ,
    key = c ,
  }
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

`\keys_set:nn`
`\keys_set:(nV|nv|no)`

Updated: 2015-11-07

`\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.

\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl

Updated: 2015-07-14

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a "full" description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset  / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_tl`, and will have been processed by `\tl_to_str:n`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_tl`, and will have been processed by `\tl_to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` will look for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_tl'~to~'#1'.
  }
```

\keys_set_known:nnN
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)

New: 2011-08-23
Updated: 2015-11-07

`\keys_set_known:nnN {`⟨*module*⟩`} {`⟨*keyval list*⟩`} `⟨*tl*⟩

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` function 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 ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

the use of `\keys_set:nn` will attempt to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-one   .groups:n = { first }           ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-two   .groups:n = { first }           ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-three .groups:n = { second }          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

will assign `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made "active", or by marking one or more groups to be ignored in key setting.

`\keys_set_filter:nnnN`
`\keys_set_filter:(nnVN|nnvN|nnoN)`
`\keys_set_filter:nnn`
`\keys_set_filter:(nnV|nnv|nno)`

`\keys_set_filter:nnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} ⟨tl⟩`

New: 2013-07-14
Updated: 2015-11-07

Actives key filtering in an "opt-out" sense: keys assigned to any of the ⟨*groups*⟩ specified will be ignored. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key–value pairs for each key which is filtered out will be stored in the ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)

New: 2013-07-14
Updated: 2015-11-07

\keys_set_groups:nnn {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩}

Actives key filtering in an "opt-in" sense: only keys assigned to one or more of the ⟨*groups*⟩ specified will be set. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

\keys_if_exist_p:nn ⋆
\keys_if_exist:nnTF ⋆

Updated: 2015-11-07

\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:nnn ⋆
\keys_if_choice_exist:nnnTF ⋆

New: 2011-08-21
Updated: 2015-11-07

\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

Updated: 2015-08-09

\keys_show:nn {⟨*module*⟩} {⟨*key*⟩}

Shows the information associated to the ⟨*key*⟩ for a ⟨*module*⟩, including the function which is used to actually implement it.

8 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 special 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 (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category "other" (12) so that the parser does not "miss" any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces will have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

\keyval_parse:NNn

Updated: 2011-09-08

`\keyval_parse:NNn` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*key–value list*⟩}

Parses the ⟨*key–value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function*$_1$⟩ should take one argument, while ⟨*function*$_2$⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key–value list*⟩, ⟨*function*$_1$⟩ will be used to process keys given with no value and ⟨*function*$_2$⟩ 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*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing.

Part XXI
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.

For functions which expect a ⟨*file name*⟩ argument, this argument 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 names will be quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1 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_sys_jobname_str` 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` `\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-02-10

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`).

`\file_add_path:nN` `\file_add_path:nN {⟨file name⟩} ⟨tl var⟩`

Updated: 2012-02-10

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`.

`\file_input:n` `\file_input:n {⟨file name⟩}`

Updated: 2012-02-17

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.

`\file_path_include:n` `\file_path_include:n {⟨path⟩}`

Updated: 2012-07-04

Adds ⟨*path*⟩ to the list of those used to search when reading files. The assignment is local. The ⟨*path*⟩ is processed in the same way as a ⟨*file name*⟩, *i.e.*, with `x`-type expansion except active characters.

\file_path_remove:n

Updated: 2012-07-04

`\file_path_remove:n {⟨path⟩}`

Removes ⟨*path*⟩ from the list of those used to search when reading files. The assignment is local. The ⟨*path*⟩ is processed in the same way as a ⟨*file name*⟩, *i.e.*, with `x`-type expansion except active characters.

\file_list:

`\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

1.1 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.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

New: 2011-09-26
Updated: 2011-12-27

`\ior_new:N ⟨stream⟩`
`\iow_new:N ⟨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 is an error, and the ⟨*stream*⟩ will behave as the corresponding `\c_term_...`.

\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 TEX run ends.

\ior_open:NnTF
\ior_open:cnTF

New: 2013-01-12

`\ior_open:NnTF ⟨stream⟩ {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

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 TEX run ends. The ⟨*true code*⟩ is then inserted into the input stream. If the file is not found, no error is raised and the ⟨*false code*⟩ is inserted into the input stream.

\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 TEX run ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

\ior_close:N ⟨*stream*⟩
\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 programmers.

\ior_list_streams:
\iow_list_streams:

Updated: 2015-08-01

\ior_list_streams:
\iow_list_streams:

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

\ior_get:NN

New: 2012-06-24

\ior_get:NN ⟨*stream*⟩ ⟨*token list variable*⟩

Function 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 locally in the ⟨*token list*⟩ variable. 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. Note that any blank lines will be converted to the token `\par`. Therefore, if skipping blank lines is requires a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

may be used. Also notice that if multiple lines are read to match braces then the resulting token list will contain `\par` tokens. As normal TEX tokenization is in force, any lines which do not end in a comment character (usually `%`) will have the line ending converted to a space, so for example input

```
a b  c
```

will result in a token list `a b c` .

TEXhackers note: This protected macro expands to the TEX primitive `\read` along with the `to` keyword.

\ior_get_str:NN

New: 2012-06-24
Updated: 2012-07-31

\ior_get_str:NN ⟨*stream*⟩ ⟨*token list variable*⟩

Function that reads one line from the input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. If the ⟨*stream*⟩ is not open, input is requested from the terminal. 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). Multiple whitespace characters are retained by this process. It will always only read one line and any blank lines in the input will result in the ⟨*token list variable*⟩ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b  c
```

will result in a token list `a b  c` with the letters `a`, `b`, and `c` having category code 12.

TEXhackers note: This protected macro is a wrapper around the ε-TEX primitive `\readline`. However, the end-line character normally added by this primitive is not included in the result of `\ior_get_str:NN`.

\ior_if_eof_p:N ⋆
\ior_if_eof:NTF ⋆

Updated: 2012-02-10

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

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test will also return a `true` value if the ⟨*stream*⟩ is not open.

2 Writing to files

\iow_now:Nn
\iow_now:(Nx|cn|cx)

Updated: 2012-06-05

\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`).

\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|cn|cx)

\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`).

TEXhackers note: When using expl3 with a format other than LATEX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` will not be recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additionnal unwanted line-breaks.

\iow_shipout_x:Nn
\iow_shipout_x:(Nx|cn|cx)

Updated: 2012-09-08

\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: This is a wrapper around the TEX primitive `\write`. When using expl3 with a format other than LATEX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` will not be recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additionnal unwanted line-breaks.

\iow_char:N ⋆

\iow_char:N \⟨*char*⟩

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_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`).

TEXhackers note: When using expl3 with a format other than LATEX, the character inserted by `\iow_newline:` will not be recognized by TEX, which may lead to the insertion of additionnal unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

2.1 Wrapping lines in output

\iow_wrap:nnnN

New: 2012-06-28
Updated: 2015-08-05

`\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨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 character count targeted will be the value of `\l_iow_line_count_int` minus the number of characters in the ⟨*run-on text*⟩ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The ⟨*text*⟩ and ⟨*run-on text*⟩ are exhaustively expanded by the function, with the following substitutions:

- `\\` 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 ⟨*text*⟩ (not 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: this may include overriding the substitutions listed.

Any expandable material in the ⟨*text*⟩ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which will typically be a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the ⟨*function*⟩) will consist of characters of category "other" (category code 12), with the exception of spaces which will have category "space" (category code 10). This means that the output will *not* expand further when written to a file.

TEXhackers note: Internally, `\iow_wrap:nnnN` carries out an `x`-type expansion on the ⟨*text*⟩ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨*text*⟩.

\iow_indent:n

New: 2011-09-21

`\iow_indent:n {⟨text⟩}`

In the first argument of `\iow_wrap:nnnN` (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_count_int

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` 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).

2.2 Constant input–output streams

`\c_term_ior`

Constant input stream for reading from the terminal. Reading from this stream using `\ior_get: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.

2.3 Primitive conditionals

`\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`.

2.4 Internal file functions and variables

`\g__file_internal_ior`

Used to test for the existence of files when opening.

`\l__file_internal_name_tl`

Used to return the full name of a file for internal use. This is set by `\file_if_exist:nTF` and `\__file_if_exist:nT`, and the value may then be used to load a file directly provided no further operations intervene.

`\__file_name_sanitize:nn`

```
\__file_name_sanitize:nn {⟨name⟩} {⟨tokens⟩}
```

New: 2012-02-09

Exhaustively-expands the ⟨*name*⟩ with the exception of any category ⟨*active*⟩ (catcode 13) 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.

2.5 Internal input–output functions

`\__ior_open:Nn`
`\__ior_open:No`

```
\__ior_open:Nn ⟨stream⟩ {⟨file name⟩}
```

New: 2012-01-23

This function has identical syntax to the public version. However, is does not take precautions against active characters in the ⟨*file name*⟩, and it does not attempt to add a ⟨*path*⟩ to the ⟨*file name*⟩: it is 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`,

`\__iow_with:Nnn` `\__iow_with:Nnn` ⟨*integer*⟩ {⟨*value*⟩} {⟨*code*⟩}

New: 2014-08-23

If the ⟨*integer*⟩ is equal to the ⟨*value*⟩ then this function simply runs the ⟨*code*⟩. Otherwise it saves the current value of the ⟨*integer*⟩, sets it to the ⟨*value*⟩, runs the ⟨*code*⟩, and restores the ⟨*integer*⟩ to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is −1 when displaying a message.

Part XXII
The l3fp package: floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x/y, square root $\sqrt{x}$, and parentheses.
- Comparison operators: $x < y$, $x <= y$, $x >? y$, $x != y$ *etc.*
- Boolean logic: negation $!\, x$, conjunction $x \&\& y$, disjunction $x \,||\, y$, ternary operator $x \,?\, y : z$.
- Random numbers: $rand()$, $randint(m, n)$.
- Exponentials: $\exp x$, $\ln x$, x^y.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\sind x$, $\cosd x$, $\tand x$, $\cotd x$, $\secd x$, $\cscd x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\asin x$, $\acos x$, $\atan x$, $\acot x$, $\asec x$, $\acsc x$ giving a result in radians, and $\asind x$, $\acosd x$, $\atand x$, $\acotd x$, $\asecd x$, $\acscd x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\sech x$, csch, and $\asinh x$, $\acosh x$, $\atanh x$, $\acoth x$, $\asech x$, $\acsch x$.

- Extrema: $\max(x, y, \ldots)$, $\min(x, y, \ldots)$, $\abs(x)$.
- Rounding functions ($n = 0$ by default, $t =$ `NaN` by default): $\trunc(x, n)$ rounds towards zero, $\floor(x, n)$ rounds towards $-\infty$, $\ceil(x, n)$ rounds towards $+\infty$, $\round(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$. And *(not yet)* modulo, and "quantize".
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\`⟨*type*⟩`_use:N`) of integer, dimension, and skip variables to floating points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {sin 3.5 /2 + 2e-3} $.
```

But in all fairness, this module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnum { 2 pi * sin ( 2.3 ^ 5 ) }
```

1 Creating and initialising floating point variables

\fp_new:N
\fp_new:c

Updated: 2012-05-08

\fp_new:N ⟨*fp var*⟩

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

\fp_const:Nn
\fp_const:cn

Updated: 2012-05-08

\fp_const:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Creates a new constant ⟨*fp var*⟩ or raises an error if the name is already taken. The ⟨*fp var*⟩ will be set globally equal to the result of evaluating the ⟨*floating point expression*⟩.

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Updated: 2012-05-08

\fp_zero:N ⟨*fp var*⟩

Sets the ⟨*fp var*⟩ to +0.

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Updated: 2012-05-08

\fp_zero_new:N ⟨*fp var*⟩

Ensures that the ⟨*fp var*⟩ exists globally by applying \fp_new:N if necessary, then applies \fp_(g)zero:N to leave the ⟨*fp var*⟩ set to +0.

2 Setting floating point variables

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn

Updated: 2012-05-08

\fp_set:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Sets ⟨*fp var*⟩ equal to the result of computing the ⟨*floating point expression*⟩.

\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)

Updated: 2012-05-08

\fp_set_eq:NN ⟨*fp var*$_1$⟩ ⟨*fp var*$_2$⟩

Sets the floating point variable ⟨*fp var*$_1$⟩ equal to the current value of ⟨*fp var*$_2$⟩.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

Updated: 2012-05-08

\fp_add:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Adds the result of computing the ⟨*floating point expression*⟩ to the ⟨*fp var*⟩.

\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn

Updated: 2012-05-08

\fp_sub:Nn ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩.

3 Using floating point numbers

\fp_eval:n ⋆

New: 2012-05-08
Updated: 2012-07-08

\fp_eval:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an "invalid operation" exception. This function is identical to \fp_to_decimal:n.

\fp_to_decimal:N ⋆
\fp_to_decimal:c ⋆
\fp_to_decimal:n ⋆

New: 2012-05-08
Updated: 2012-07-08

\fp_to_decimal:N ⟨*fp var*⟩
\fp_to_decimal:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an "invalid operation" exception.

\fp_to_dim:N ⋆
\fp_to_dim:c ⋆
\fp_to_dim:n ⋆

Updated: 2016-03-22

\fp_to_dim:N ⟨*fp var*⟩
\fp_to_dim:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_to_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14}+2^{-17}, 2^{14}-2^{-17}]$ of valid TEX dimensions, leading to overflow errors if used as a dimension. The values $\pm\infty$ and NaN trigger an "invalid operation" exception.

\fp_to_int:N ⋆
\fp_to_int:c ⋆
\fp_to_int:n ⋆

Updated: 2012-07-08

\fp_to_int:N ⟨*fp var*⟩
\fp_to_int:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31}+1, 2^{31}-1]$ of valid TEX integers, leading to overflow errors if used in an integer expression. The values $\pm\infty$ and `NaN` trigger an "invalid operation" exception.

\fp_to_scientific:N ⋆
\fp_to_scientific:c ⋆
\fp_to_scientific:n ⋆

New: 2012-05-08
Updated: 2016-03-22

\fp_to_scientific:N ⟨*fp var*⟩
\fp_to_scientific:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result in scientific notation:

⟨*optional* `-`⟩⟨*digit*⟩`.`⟨*15 digits*⟩`e`⟨*optional sign*⟩⟨*exponent*⟩

The leading ⟨*digit*⟩ is non-zero except in the case of ± 0. The values $\pm\infty$ and `NaN` trigger an "invalid operation" exception. Normal category codes apply: thus the `e` is category code 11 (a letter).

\fp_to_tl:N ⋆
\fp_to_tl:c ⋆
\fp_to_tl:n ⋆

Updated: 2016-03-22

\fp_to_tl:N ⟨*fp var*⟩
\fp_to_tl:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (see `\fp_to_scientific:n`). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`. Negative numbers start with `-`. The special values ± 0, $\pm\infty$ and `NaN` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, will be made up of letters.

\fp_use:N ⋆
\fp_use:c ⋆

Updated: 2012-07-08

\fp_use:N ⟨*fp var*⟩

Inserts the value of the ⟨*fp var*⟩ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and `NaN` trigger an "invalid operation" exception. This function is identical to `\fp_to_decimal:N`.

4 Floating point conditionals

\fp_if_exist_p:N ⋆
\fp_if_exist_p:c ⋆
\fp_if_exist:NTF ⋆
\fp_if_exist:cTF ⋆

Updated: 2012-05-08

\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.

\fp_compare_p:nNn ⋆
\fp_compare:nNnTF ⋆

Updated: 2012-05-08

\fp_compare_p:nNn {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩}
\fp_compare:nNnTF {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the ⟨*fpexpr*$_1$⟩ and the ⟨*fpexpr*$_2$⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x\langle y{,}x{=}y{,}x\rangle y$, or x and y are not ordered. The latter case occurs exactly when either operand is `NaN`, and this relation is denoted by the symbol `?`. Note that a `NaN` is distinct from any value, even another `NaN`, hence $x = x$ is not true for a `NaN`. To test if a value is `NaN`, compare it to an arbitrary number with the "not ordered" relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
  { } % <value> is nan
  { } % <value> is not nan
```

\fp_compare_p:n ⋆
\fp_compare:nTF ⋆

Updated: 2012-12-14

```
\fp_compare_p:n
  {
    ⟨fpexpr1⟩ ⟨relation1⟩
    ...
    ⟨fpexprN⟩ ⟨relationN⟩
    ⟨fpexprN+1⟩
  }
\fp_compare:nTF
  {
    ⟨fpexpr1⟩ ⟨relation1⟩
    ...
    ⟨fpexprN⟩ ⟨relationN⟩
    ⟨fpexprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*floating point expressions*⟩ as described for `\fp_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*intexpr*$_1$⟩ and ⟨*intexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*intexpr*$_2$⟩ and ⟨*intexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*intexpr*$_N$⟩ and ⟨*intexpr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*floating point expression*⟩ is evaluated only once. Contrarily to `\int_compare:nTF`, all ⟨*floating point expressions*⟩ are computed, even if one comparison is `false`. Two floating point numbers x and y may obey four mutually exclusive relations: $x\langle y{,}x{=}y{,}x\rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is `NaN`, and this relation is denoted by the symbol `?`. Each ⟨*relation*⟩ can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the ⟨*relation*⟩), with the restriction that the ⟨*relation*⟩ may not start with `?`, as this symbol has a different meaning (in combination with `:`) within floatin point expressions. The comparison x ⟨*relation*⟩ y is then `true` if the ⟨*relation*⟩ does not start with `!` and the actual relation (`<`, `=`, `>`, or `?`) between x and y appears within the ⟨*relation*⟩, or on the contrary if the ⟨*relation*⟩ starts with `!` and the relation between x and y does not appear within the ⟨*relation*⟩. Common choices of ⟨*relation*⟩ include `>=` (greater or equal), `!=` (not equal), `!?` or `<=>` (comparable).

5 Floating point expression loops

\fp_do_until:nNnn ☆ `\fp_do_until:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}`

New: 2012-08-16

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_do_while:nNnn ☆ `\fp_do_while:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}`

New: 2012-08-16

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_until_do:nNnn ☆ `\fp_until_do:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}`

New: 2012-08-16

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_while_do:nNnn ☆ `\fp_while_do:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}`

New: 2012-08-16

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_do_until:nn ☆ `\fp_do_until:nn { ⟨fpexpr1⟩ ⟨relation⟩ ⟨fpexpr2⟩ } {⟨code⟩}`

New: 2012-08-16

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_do_while:nn ☆ `\fp_do_while:nn { ⟨fpexpr1⟩ ⟨relation⟩ ⟨fpexpr2⟩ } {⟨code⟩}`

New: 2012-08-16

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_until_do:nn ☆ `\fp_until_do:nn { ⟨fpexpr1⟩ ⟨relation⟩ ⟨fpexpr2⟩ } {⟨code⟩}`

New: 2012-08-16

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_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`.

\fp_while_do:nn ☆

New: 2012-08-16

\fp_while_do:nn { ⟨*fpexpr*$_1$⟩ ⟨*relation*⟩ ⟨*fpexpr*$_2$⟩ } {⟨*code*⟩}

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_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.

\fp_step_function:nnnN ☆

New: 2016-11-21

\fp_step_function:nnnN {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*function*⟩

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point 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*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]

\fp_step_inline:nnnn

New: 2016-11-21

\fp_step_inline:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

6 Some useful constants, and scratch variables

\c_zero_fp
\c_minus_zero_fp

New: 2012-05-08

Zero, with either sign.

\c_one_fp

New: 2012-05-08

One as an fp: useful for comparisons in some places.

\c_inf_fp
\c_minus_inf_fp

New: 2012-05-08

Infinity, with either sign. These can be input directly in a floating point expression as inf and -inf.

\c_e_fp

Updated: 2012-05-08

The value of the base of the natural logarithm, e = exp(1).

\c_pi_fp

Updated: 2013-11-17

The value of π. This can be input directly in a floating point expression as `pi`.

\c_one_degree_fp

New: 2012-05-08

Updated: 2013-11-17

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

\l_tmpa_fp
\l_tmpb_fp

Scratch floating points 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_fp
\g_tmpb_fp

Scratch floating points 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.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

"Exceptions" may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The IEEE standard defines 5 types of exceptions.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0.
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$, or $\sin(\infty)$, and almost any operation involving a `NaN`. This normally results in a `NaN`, except for conversion functions whose target type does not have a notion of `NaN` (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.
- *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in LaTeX3.

To each exception is associated a "flag", which can be either *on* or *off*. By default, the "invalid operation" exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

\fp_if_flag_on_p:n ⋆
\fp_if_flag_on:nTF ⋆

New: 2012-08-08

\fp_if_flag_on_p:n {⟨exception⟩}
\fp_if_flag_on:nTF {⟨exception⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the flag for the ⟨exception⟩ is on, which normally means the given ⟨exception⟩ has occurred. *This function is experimental, and may be altered or removed.*

\fp_flag_off:n

New: 2012-08-08

\fp_flag_off:n {⟨exception⟩}

Locally turns off the flag which indicates whether the ⟨exception⟩ has occurred. *This function is experimental, and may be altered or removed.*

\fp_flag_on:n ⋆

New: 2012-08-08

\fp_flag_on:n {⟨exception⟩}

Locally turns on the flag to indicate (or pretend) that the ⟨exception⟩ has occurred. Note that this function is expandable: it is used internally by l3fp to signal when exceptions do occur. *This function is experimental, and may be altered or removed.*

\fp_trap:nn

New: 2012-07-19
Updated: 2012-08-08

\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}

All occurrences of the ⟨exception⟩ (`invalid_operation`, `division_by_zero`, `overflow`, or `underflow`) within the current group are treated as ⟨trap type⟩, which can be

- `none`: the ⟨exception⟩ will be entirely ignored, and leave no trace;
- `flag`: the ⟨exception⟩ will turn the corresponding flag on when it occurs;
- `error`: additionally, the ⟨exception⟩ will halt the TEX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

8 Viewing floating points

\fp_show:N
\fp_show:c
\fp_show:n

New: 2012-05-08
Updated: 2015-08-07

\fp_show:N ⟨fp var⟩
\fp_show:n {⟨floating point expression⟩}

Evaluates the ⟨floating point expression⟩ and displays the result in the terminal.

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1 d_2 \ldots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0, 9]$, $d_1 \neq 0$, and $|n| \leq 10000$;
- ± 0, zero, with a given sign;
- $\pm\infty$, infinity, with a given sign;

- `NaN`, is "not a number", and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

(not yet) subnormal numbers $\pm 0.d_1 d_2 \dots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 16 significant figures.

On input, a normal floating point number consists of:

- ⟨*sign*⟩: a possibly empty string of `+` and `-` characters;
- ⟨*significand*⟩: a non-empty string of digits together with zero or one dot;
- ⟨*exponent*⟩ optionally: the character `e`, followed by a possibly empty string of `+` and `-` tokens, and a non-empty string of digits.

The sign of the resulting number is `+` if ⟨*sign*⟩ contains an even number of `-`, and `-` otherwise, hence, an empty ⟨*sign*⟩ denotes a non-negative input. The stored significand is obtained from ⟨*significand*⟩ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input ⟨*significand*⟩ has at most 16 digits. The stored ⟨*exponent*⟩ is obtained by combining the input ⟨*exponent*⟩ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting ⟨*exponent*⟩ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if ⟨*significand*⟩ contains no non-zero digit (*i.e.*, consists only in `0` characters, and an optional `.` character), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to $+0$, but that is not guaranteed to remain true.

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any ⟨*sign*⟩, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that will be ignored.
- Any unrecognizable string triggers an error, and produces a `NaN`.

Note that `e-1` is not a representation of 10^{-1}, because it could be mistaken with the difference of "`e`" and 1. This is consistent with several other programming languages. However, in order to avoid confusions, `e-1` is not considered to be this difference either. To input the base of natural logarithms, use `exp(1)` or `\c_e_fp`.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.

- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc.*
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\texttt{sin2pi} = \sin(2\pi) = 0,$$
$$\texttt{2\^{}2max(3,4)} = 2^{2\max(3,4)} = 256.$$

Functions are called on the value of their argument, contrarily to TEX macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0, and `true` otherwise, including when it is `NaN`.

`?:` `\fp_eval:n { ⟨operand1⟩ ? ⟨operand2⟩ : ⟨operand3⟩ }`

The ternary operator `?:` results in ⟨*operand*$_2$⟩ if ⟨*operand*$_1$⟩ is true, and ⟨*operand*$_3$⟩ if it is false (equal to ± 0). All three ⟨*operands*⟩ are evaluated in all cases. The operator is right associative, hence

```
\fp_eval:n
  {
    1 + 3 > 4 ? 1 :
    2 + 4 > 5 ? 2 :
    3 + 5 > 6 ? 3 : 4
  }
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2+4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

`||` `\fp_eval:n { ⟨operand1⟩ ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is true (non-zero), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

`&&` `\fp_eval:n { ⟨operand1⟩ && ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is false (equal to ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

<
=
>
?

Updated: 2013-12-14

```
\fp_eval:n
  {
    ⟨operand_1⟩ ⟨relation_1⟩
    ...
    ⟨operand_N⟩ ⟨relation_N⟩
    ⟨operand_{N+1}⟩
  }
```

Each ⟨*relation*⟩ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to +1 if all comparisons ⟨*operand*$_i$⟩ ⟨*relation*$_j$⟩ ⟨*operand*$_{i+1}$⟩ are true, and +0 otherwise. All ⟨*operands*⟩ are evaluated in all cases. See `\fp_compare:nTF` for details.

\+ `\fp_eval:n { ⟨operand_1⟩ + ⟨operand_2⟩ }`
\- `\fp_eval:n { ⟨operand_1⟩ - ⟨operand_2⟩ }`

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate.

* `\fp_eval:n { ⟨operand_1⟩ * ⟨operand_2⟩ }`
/ `\fp_eval:n { ⟨operand_1⟩ / ⟨operand_2⟩ }`

Computes the product or the ratio of its two ⟨*operands*⟩. The "invalid operation" exception occurs for ∞/∞, $0/0$, or $0 * \infty$. "Division by zero" occurs when dividing a finite non-zero number by ± 0. "Underflow" and "overflow" occur when appropriate.

\+ `\fp_eval:n { + ⟨operand⟩ }`
\- `\fp_eval:n { - ⟨operand⟩ }`
! `\fp_eval:n { ! ⟨operand⟩ }`

The unary `+` does nothing, the unary `-` changes the sign of the ⟨*operand*⟩, and `!` ⟨*operand*⟩ evaluates to 1 if ⟨*operand*⟩ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

** `\fp_eval:n { ⟨operand_1⟩ ** ⟨operand_2⟩ }`
^ `\fp_eval:n { ⟨operand_1⟩ ^ ⟨operand_2⟩ }`

Raises ⟨*operand*$_1$⟩ to the power ⟨*operand*$_2$⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. The "invalid operation" exception occurs if ⟨*operand*$_1$⟩ is negative or -0, and ⟨*operand*$_2$⟩ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be +0). "Division by zero" occurs when raising ± 0 to a strictly negative power. "Underflow" and "overflow" occur when appropriate.

abs `\fp_eval:n { abs( ⟨fpexpr⟩ ) }`

Computes the absolute value of the ⟨*fpexpr*⟩. This function does not raise any exception beyond those raised when computing its operand ⟨*fpexpr*⟩. See also `\fp_abs:n`.

exp `\fp_eval:n { exp( ⟨fpexpr⟩ ) }`

Computes the exponential of the ⟨*fpexpr*⟩. "Underflow" and "overflow" occur when appropriate.

ln `\fp_eval:n { ln( ⟨fpexpr⟩ ) }`

Computes the natural logarithm of the ⟨*fpexpr*⟩. Negative numbers have no (real) logarithm, hence the "invalid operation" is raised in that case, including for $\ln(-0)$. "Division by zero" occurs when evaluating $\ln(+0) = -\infty$. "Underflow" and "overflow" occur when appropriate.

max `\fp_eval:n { max( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ... ) }`
min `\fp_eval:n { min( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ... ) }`

Evaluates each ⟨*fpexpr*⟩ and computes the largest (smallest) of those. If any of the ⟨*fpexpr*⟩ is a `NaN`, the result is `NaN`. Those operations do not raise exceptions.

round
trunc
ceil
floor

`\fp_eval:n { round ( ⟨fpexpr⟩ ) }`
`\fp_eval:n { round ( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }`
`\fp_eval:n { round ( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ⟨fpexpr3⟩ ) }`

New: 2013-12-14
Updated: 2015-08-08

Only `round` accepts a third argument. Evaluates $\langle fpexpr_1 \rangle = x$ and $\langle fpexpr_2 \rangle = n$ and $\langle fpexpr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n}; if $n = +\infty$, this always yields x; if $n = -\infty$, this yields one of ± 0, $\pm\infty$, or `NaN`; if n is neither $\pm\infty$ nor an integer, then an "invalid operation" exception is raised. When $\langle fpexpr_2 \rangle$ is omitted, $n = 0$, *i.e.*, $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- `round` yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is `nan` or not given the even multiple is chosen ("ties to even"), if $t = \pm 0$ the multiple closest to 0 is chosen ("ties to zero"), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen ("ties towards positive/negative infinity").
- `floor`, or the deprecated `round-`, yields the largest multiple of 10^{-n} smaller or equal to x ("round towards negative infinity");
- `ceil`, or the deprecated `round+`, yields the smallest multiple of 10^{-n} greater or equal to x ("round towards positive infinity");
- `trunc`, or the deprecated `round0`, yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less that that of x ("round towards zero").

"Overflow" occurs if x is finite and the result is infinite (this can only happen if $\langle fpexpr_2 \rangle < -9984$).

sin
cos
tan
cot
csc
sec

```
\fp_eval:n { sin( ⟨fpexpr⟩ ) }
\fp_eval:n { cos( ⟨fpexpr⟩ ) }
\fp_eval:n { tan( ⟨fpexpr⟩ ) }
\fp_eval:n { cot( ⟨fpexpr⟩ ) }
\fp_eval:n { csc( ⟨fpexpr⟩ ) }
\fp_eval:n { sec( ⟨fpexpr⟩ ) }
```

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fpexpr*⟩ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, sin(8*pi*) is not quite zero, while its analog sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate.

sind
cosd
tand
cotd
cscd
secd

```
\fp_eval:n { sind( ⟨fpexpr⟩ ) }
\fp_eval:n { cosd( ⟨fpexpr⟩ ) }
\fp_eval:n { tand( ⟨fpexpr⟩ ) }
\fp_eval:n { cotd( ⟨fpexpr⟩ ) }
\fp_eval:n { cscd( ⟨fpexpr⟩ ) }
\fp_eval:n { secd( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fpexpr*⟩ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, sin(8*pi*) is not quite zero, while its analog sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate.

asin
acos
acsc
asec

```
\fp_eval:n { asin( ⟨fpexpr⟩ ) }
\fp_eval:n { acos( ⟨fpexpr⟩ ) }
\fp_eval:n { acsc( ⟨fpexpr⟩ ) }
\fp_eval:n { asec( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fpexpr*⟩ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate.

asind
acosd
acscd
asecd

```
\fp_eval:n { asind( ⟨fpexpr⟩ ) }
\fp_eval:n { acosd( ⟨fpexpr⟩ ) }
\fp_eval:n { acscd( ⟨fpexpr⟩ ) }
\fp_eval:n { asecd( ⟨fpexpr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fpexpr*⟩ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate.

atan
acot

New: 2013-11-02

```
\fp_eval:n { atan( ⟨fpexpr⟩ ) }
\fp_eval:n { atan( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
\fp_eval:n { acot( ⟨fpexpr⟩ ) }
\fp_eval:n { acot( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
```

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the ⟨*fpexpr*⟩: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2\rangle, \langle fpexpr_1\rangle)$: this is the arctangent of $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1\rangle$ and $\langle fpexpr_2\rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1\rangle, \langle fpexpr_2\rangle)$, equal to the arccotangent of $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. Only the "underflow" exception can occur.

atand
acotd

New: 2013-11-02

```
\fp_eval:n { atand( ⟨fpexpr⟩ ) }
\fp_eval:n { atand( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
\fp_eval:n { acotd( ⟨fpexpr⟩ ) }
\fp_eval:n { acotd( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
```

Those functions yield an angle in degrees: `atand` and `acotd` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the ⟨*fpexpr*⟩: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2\rangle, \langle fpexpr_1\rangle)$: this is the arctangent of $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1\rangle$ and $\langle fpexpr_2\rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1\rangle, \langle fpexpr_2\rangle)$, equal to the arccotangent of $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1\rangle/\langle fpexpr_2\rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. Only the "underflow" exception can occur.

sqrt

New: 2013-12-14

```
\fp_eval:n { sqrt( ⟨fpexpr⟩ ) }
```

Computes the square root of the ⟨*fpexpr*⟩. The "invalid operation" is raised when the ⟨*fpexpr*⟩ is negative; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\texttt{NaN}} = \texttt{NaN}$.

rand

New: 2016-11-20

```
\fp_eval:n { rand() }
```

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded.

TEXhackers note: This is a wrapper around `\pdfuniformdeviate`. The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed`.

randint

New: 2016-11-20

`\fp_eval:n { randint( ⟨fpexpr⟩ ) }`
`\fp_eval:n { randint( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }`

Produces a pseudo-random integer between 1 and ⟨*fpexpr*⟩ or between ⟨*fpexpr*$_1$⟩ and ⟨*fpexpr*$_2$⟩ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second.

TEXhackers note: This is a wrapper around `\pdfuniformdeviate`. The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed`.

inf
nan

The special values $+\infty$, $-\infty$, and `NaN` are represented as `inf`, `-inf` and `nan` (see `\c_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi

The value of π (see `\c_pi_fp`).

deg

The value of 1° in radians (see `\c_one_degree_fp`).

em
ex
in
pt
pc
cm
mm
dd
cc
nd
nc
bp
sp

Those units of measurement are equal to their values in `pt`, namely

$$\begin{aligned}
1\texttt{in} &= 72.27\texttt{pt}\\
1\texttt{pt} &= 1\texttt{pt}\\
1\texttt{pc} &= 12\texttt{pt}\\
1\texttt{cm} &= \frac{1}{2.54}\texttt{in} = 28.45275590551181\texttt{pt}\\
1\texttt{mm} &= \frac{1}{25.4}\texttt{in} = 2.845275590551181\texttt{pt}\\
1\texttt{dd} &= 0.376065\texttt{mm} = 1.07000856496063\texttt{pt}\\
1\texttt{cc} &= 12\texttt{dd} = 12.84010277952756\texttt{pt}\\
1\texttt{nd} &= 0.375\texttt{mm} = 1.066978346456693\texttt{pt}\\
1\texttt{nc} &= 12\texttt{nd} = 12.80374015748031\texttt{pt}\\
1\texttt{bp} &= \frac{1}{72}\texttt{in} = 1.00375\texttt{pt}\\
1\texttt{sp} &= 2^{-16}\texttt{pt} = 1.52587890625e-5\texttt{pt}.
\end{aligned}$$

The values of the (font-dependent) units `em` and `ex` are gathered from TEX when the surrounding floating point expression is evaluated.

true
false

Other names for 1 and +0.

\fp_abs:n ⋆

New: 2012-05-14
Updated: 2012-07-08

`\fp_abs:n {⟨floating point expression⟩}`

Evaluates the ⟨*floating point expression*⟩ as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `abs()` can be used.

\fp_max:nn ⋆
\fp_min:nn ⋆

New: 2012-09-26

\fp_max:nn {⟨*fp expression 1*⟩} {⟨*fp expression 2*⟩}

Evaluates the ⟨*floating point expressions*⟩ as described for \fp_eval:n and leaves the resulting larger (max) or smaller (min) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, max() and min() can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789_+; if it receives a TEX primitive conditional affected by \exp_not:N.

The following need to be done. I'll try to time-order the items.

- Decide what exponent range to consider.
- Support signalling nan.
- Modulo and remainder, and rounding functions quantize, quantize0, quantize+, quantize-, quantize=, round=. Should the modulo also be provided as (catcode 12) %?
- \fp_format:nn {⟨*fpexpr*⟩} {⟨*format*⟩}, but what should ⟨*format*⟩ be? More general pretty printing?
- Add and, or, xor? Perhaps under the names all, any, and xor?
- Add $\log(x, b)$ for logarithm of x in base b.
- hypot (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions cosh, sinh, tanh.
- Inverse hyperbolics.
- Base conversion, input such as 0xAB.CDEF.
- Random numbers (pgfmath provides rnd, rand, random), with seed reset at every \fp_set:Nn.
- Factorial (not with !), gamma function.
- Improve coefficients of the sin and tan series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an array(1,2,3) and i=complex(0,1).
- Provide an experimental map function? Perhaps easier to implement if it is a single character, @sin(1,2)?
- Provide \fp_if_nan:nTF, and an isnan function?
- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0, not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a TEX "number too large" error.
- Subnormals are not implemented.
- The overflow trap receives the wrong argument in l3fp-expo (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Document that l3trial/l3fp-types introduces tools for adding new types.
- In subsection 9.1, write a grammar.
- Fix the `TWO BARS` business with the index.
- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in l3fp-parse.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (l3fp-basics).
- Understand and document `\__fp_basics_pack_weird_low:NNNNw` and `\__fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to l3fp-aux under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TEX fp packages, the international standards,...
- Also take into account the "inexact" exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)? Perhaps for including comments inside the computation itself??

Part XXIII
The l3candidates package Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

> **As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.**

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to l3basics

\cs_log:N
\cs_log:c

New: 2014-08-22
Updated: 2015-08-03

\cs_log:N ⟨*control sequence*⟩

Writes the definition of the ⟨*control sequence*⟩ in the log file. See also \cs_show:N which displays the result in the terminal.

__kernel_register_log:N
__kernel_register_log:c

Updated: 2015-08-03

__kernel_register_log:N ⟨*register*⟩

Used to write the contents of a TEX register to the log file in a form similar to __kernel_register_show:N.

3 Additions to l3box

3.1 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

\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. Thus negative y-sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TEX group level.

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn

\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. Thus negative y-sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current TEX group level.

\box_resize_to_ht:Nn
\box_resize_to_ht:cn

\box_resize_to_ht: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 height only, not including 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. Thus negative y-sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current TEX group level.

\box_resize_to_wd:Nn
\box_resize_to_wd:cn

\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. Thus negative y-sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TEX group level.

\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn

New: 2014-07-03

\box_resize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resize the ⟨box⟩ to a *height* of ⟨*x-size*⟩ horizontally and ⟨*y-size*⟩ vertically (both of the sizes are dimension expressions). The ⟨*y-size*⟩ is the *height* of the box, ignoring any depth. 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.

\box_rotate:Nn
\box_rotate:cn

\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.

\box_scale:Nnn
\box_scale:cnn

\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. Thus negative *y*-scales will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TEX group level.

3.2 Viewing part of a box

\box_clip:N
\box_clip:c

\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.

These functions require the LATEX3 native drivers: they will not work with the LATEX 2ε graphics drivers!

TEXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed 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

\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 trim operation is applied. The adjustment applies within the current TEX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

\box_viewport:Nnnnn
\box_viewport:cnnnn

\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 adjustment applies within the current TEX group level.

4 Additions to l3clist

\clist_log:N
\clist_log:c

New: 2014-08-22

\clist_log:N ⟨*comma list*⟩

Writes the entries in the ⟨*comma list*⟩ in the log file. See also `\clist_show:N` which displays the result in the terminal.

\clist_log:n

New: 2014-08-22

\clist_log:n {⟨*tokens*⟩}

Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

5 Additions to l3coffins

\coffin_resize:Nnn
\coffin_resize:cnn

\coffin_resize:Nnn ⟨*coffin*⟩ {⟨*width*⟩} {⟨*total-height*⟩}

Resized the ⟨*coffin*⟩ to ⟨*width*⟩ and ⟨*total-height*⟩, both of which should be given as dimension expressions.

\coffin_rotate:Nn
\coffin_rotate:cn

\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

\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.

\coffin_log_structure:N
\coffin_log_structure:c

New: 2014-08-22

\coffin_log_structure:N ⟨*coffin*⟩

This function writes the structural information about the ⟨*coffin*⟩ in the log file. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin. See also `\coffin_show_structure:N` which displays the result in the terminal.

6 Additions to l3file

\file_if_exist_input:nTF

New: 2014-07-02

\file_if_exist_input:n {⟨*file name*⟩}
\file_if_exist_input: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). If found, inserts the ⟨*true code*⟩ then reads in the file as additional LATEX source as described for \file_input:n. Note that \file_if_exist_input:n does not raise an error if the file is not found, in contrast to \file_input:n.

\ior_map_inline:Nn

New: 2012-02-11

\ior_map_inline:Nn ⟨*stream*⟩ {⟨*inline function*⟩}

Applies the ⟨*inline function*⟩ to ⟨*lines*⟩ 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. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads.

\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 ⟨*stream*⟩. 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. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads.

\ior_map_break:

New: 2012-06-29

\ior_map_break:

Used to terminate a \ior_map_... function before all lines from the ⟨*stream*⟩ have been processed. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \ior_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:Nn before further items are taken from the input stream. This will depend on the design of the mapping function.

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n {⟨*tokens*⟩}

Used to terminate a \ior_map_... function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \ior_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:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\ior_log_streams:
\iow_log_streams:

New: 2014-08-22

\ior_log_streams:
\iow_log_streams:

Writes in the log file a list of the file names associated with each open stream: intended for tracking down problems.

7 Additions to l3fp

\fp_log:N
\fp_log:c
\fp_log:n

New: 2014-08-22
Updated: 2015-08-07

\fp_log:N ⟨*fp var*⟩
\fp_log:n {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and writes the result in the log file.

8 Additions to l3int

\int_log:N
\int_log:c

New: 2014-08-22
Updated: 2015-08-03

\int_log:N ⟨*integer*⟩

Writes the value of the ⟨*integer*⟩ in the log file.

\int_log:n

New: 2014-08-22
Updated: 2015-08-07

\int_log:n {⟨*integer expression*⟩}

Writes the result of evaluating the ⟨*integer expression*⟩ in the log file.

9 Additions to l3keys

\keys_log:nn

New: 2014-08-22

\keys_log:nn {⟨*module*⟩} {⟨*key*⟩}

Writes in the log file the function which is used to actually implement a ⟨*key*⟩ for a ⟨*module*⟩.

10 Additions to l3msg

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using \msg_error:nnnnnn or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors 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. As a result, the message text and arguments are not expanded, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn ⋆
\msg_expandable_error:nnfffff ⋆
\msg_expandable_error:nnnnn ⋆
\msg_expandable_error:nnfff ⋆
\msg_expandable_error:nnnn ⋆
\msg_expandable_error:nnff ⋆
\msg_expandable_error:nnn ⋆
\msg_expandable_error:nnf ⋆
\msg_expandable_error:nn ⋆

New: 2015-08-06

\msg_expandable_error:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues an "Undefined error" message from TEX itself using the undefined control sequence \::error then prints "! ⟨*module*⟩: "⟨*error message*⟩, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11 Additions to l3prg

Minimal (lazy) evaluation can be obtained using the conditionals \bool_lazy_all:nTF, \bool_lazy_and:nnTF, \bool_lazy_any:nTF, or \bool_lazy_or:nnTF, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
  {
    \bool_lazy_any_p:n
      {
        { \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 } }
```

the line marked with `is skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

`\bool_lazy_all_p:n` ⋆
`\bool_lazy_all:nTF` ⋆

New: 2015-11-15

`\bool_lazy_all_p:n { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ⋯ {⟨boolexpr$_N$⟩} }`
`\bool_lazy_all:nTF { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ⋯ {⟨boolexpr$_N$⟩} } {⟨true code⟩} {⟨false code⟩}`

Implements the "And" operation on the ⟨*boolean expressions*⟩, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the ⟨*boolean expressions*⟩ which are needed to determine the result of `\bool_lazy_all:nTF` will be evaluated. See also `\bool_lazy_and:nnTF` when there are only two ⟨*boolean expressions*⟩.

`\bool_lazy_and_p:nn` ⋆
`\bool_lazy_and:nnTF` ⋆

New: 2015-11-15

`\bool_lazy_and_p:nn {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩}`
`\bool_lazy_and:nnTF {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} {⟨true code⟩} {⟨false code⟩}`

Implements the "And" operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the ⟨*boolexpr$_2$*⟩ will only be evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two ⟨*boolean expressions*⟩.

`\bool_lazy_any_p:n` ⋆
`\bool_lazy_any:nTF` ⋆

New: 2015-11-15

`\bool_lazy_any_p:n { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ⋯ {⟨boolexpr$_N$⟩} }`
`\bool_lazy_any:nTF { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ⋯ {⟨boolexpr$_N$⟩} } {⟨true code⟩} {⟨false code⟩}`

Implements the "Or" operation on the ⟨*boolean expressions*⟩, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the ⟨*boolean expressions*⟩ which are needed to determine the result of `\bool_lazy_any:nTF` will be evaluated. See also `\bool_lazy_or:nnTF` when there are only two ⟨*boolean expressions*⟩.

`\bool_lazy_or_p:nn` ⋆
`\bool_lazy_or:nnTF` ⋆

New: 2015-11-15

`\bool_lazy_or_p:nn {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩}`
`\bool_lazy_or:nnTF {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} {⟨true code⟩} {⟨false code⟩}`

Implements the "Or" operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the ⟨*boolexpr$_2$*⟩ will only be evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two ⟨*boolean expressions*⟩.

`\bool_log:N`
`\bool_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\bool_log:N ⟨boolean⟩`

Writes the logical truth of the ⟨*boolean*⟩ in the log file.

`\bool_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\bool_log:n {⟨boolean expression⟩}`

Writes the logical truth of the ⟨*boolean expression*⟩ in the log file.

12 Additions to l3prop

\prop_count:N ⋆
\prop_count:c ⋆

\prop_count:N ⟨*property list*⟩

Leaves the number of key–value pairs in the ⟨*property list*⟩ in the input stream as an ⟨*integer denotation*⟩.

\prop_map_tokens:Nn ☆
\prop_map_tokens:cn ☆

\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_item:Nn is faster.

\prop_log:N
\prop_log:c

New: 2014-08-12

\prop_log:N ⟨*property list*⟩

Writes the entries in the ⟨*property list*⟩ in the log file.

13 Additions to l3seq

\seq_mapthread_function:NNN ☆
\seq_mapthread_function:(NcN|cNN|ccN) ☆

\seq_mapthread_function:NNN ⟨*seq*$_1$⟩ ⟨*seq*$_2$⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every pair of items ⟨*seq*$_1$*-item*⟩–⟨*seq*$_2$*-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_filter:NNn
\seq_gset_filter:NNn

\seq_set_filter:NNn ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ {⟨*inline boolexpr*⟩}

Evaluates the ⟨*inline boolexpr*⟩ for every ⟨*item*⟩ stored within the ⟨*sequence*$_2$⟩. 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 ⟨*sequence*$_1$⟩.

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 ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence*$_2$⟩. 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 ⟨*sequence*$_1$⟩. 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.

\seq_log:N
\seq_log:c

New: 2014-08-12

\seq_log:N ⟨*sequence*⟩

Writes the entries in the ⟨*sequence*⟩ in the log file.

14 Additions to l3skip

\skip_split_finite_else_action:nnNN

\skip_split_finite_else_action:nnNN {⟨*skipexpr*⟩} {⟨*action*⟩} ⟨*dimen*$_1$⟩ ⟨*dimen*$_2$⟩

Checks if the ⟨*skipexpr*⟩ contains finite glue. If it does then it assigns ⟨*dimen*$_1$⟩ the stretch component and ⟨*dimen*$_2$⟩ the shrink component. If it contains infinite glue set ⟨*dimen*$_1$⟩ and ⟨*dimen*$_2$⟩ to 0 pt and place `#2` into the input stream: this is usually an error or warning message of some sort.

\dim_log:N
\dim_log:c

New: 2014-08-22
Updated: 2015-08-03

\dim_log:N ⟨*dimension*⟩

Writes the value of the ⟨*dimension*⟩ in the log file.

\dim_log:n

New: 2014-08-22
Updated: 2015-08-07

\dim_log:n {⟨*dimension expression*⟩}

Writes the result of evaluating the ⟨*dimension expression*⟩ in the log file.

\skip_log:N
\skip_log:c

New: 2014-08-22
Updated: 2015-08-03

\skip_log:N ⟨*skip*⟩

Writes the value of the ⟨*skip*⟩ in the log file.

\skip_log:n

New: 2014-08-22
Updated: 2015-08-07

\skip_log:n {⟨*skip expression*⟩}

Writes the result of evaluating the ⟨*skip expression*⟩ in the log file.

\muskip_log:N
\muskip_log:c

New: 2014-08-22
Updated: 2015-08-03

\muskip_log:N ⟨*muskip*⟩

Writes the value of the ⟨*muskip*⟩ in the log file.

\muskip_log:n

New: 2014-08-22
Updated: 2015-08-07

\muskip_log:n {⟨*muskip expression*⟩}

Writes the result of evaluating the ⟨*muskip expression*⟩ in the log file.

15 Additions to l3tl

\tl_if_single_token_p:n ⋆
\tl_if_single_token:nTF ⋆

\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.

\tl_reverse_tokens:n ⋆

\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_count_tokens:n ⋆

\tl_count_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 token count of `a~{bc}` is 6. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_lower_case:n ⋆
\tl_upper_case:n ⋆
\tl_mixed_case:n ⋆
\tl_lower_case:nn ⋆
\tl_upper_case:nn ⋆
\tl_mixed_case:nn ⋆

New: 2014-06-30
Updated: 2016-01-12

\tl_upper_case:n {⟨*tokens*⟩}
\tl_upper_case:nn {⟨*language*⟩} {⟨*tokens*⟩}

These functions are intended to be applied to input which may be regarded broadly as "text". They traverse the ⟨*tokens*⟩ and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters will have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the ⟨*tokens*⟩ are normalized and become `{` and `}`, respectively.

Importantly, notice that these functions are intended for working with user text for typesetting. For case changing programmatic data see the l3str module and discussion there of `\str_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions will be expanded *unless* it falls within one of the special handling classes described below. This expansion approach means that in general the result of case changing will match the "natural" outcome expected from a "functional" approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

will produce

```
HELLO WORLD
```

The expansion approach taken means that in package mode any LaTeX 2_ε "robust" commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the etoolbox package.

`\l_tl_case_change_math_tl`

Case changing will not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

will become

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_tl_case_change_math_tl`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \( \)
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be "hidden" inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of "text" the case changing functions are intended to apply to this should not be an issue.

`\l_tl_case_change_exclude_tl`

Case changing can be prevented by using any command on the list `\l_tl_case_change_exclude_tl`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_tl_case_change_exclude_tl { \NoChangeCase }
```

the input

```
\tl_upper_case:n
  { Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with LaTeX 2_ε the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

\l_tl_case_change_accents_tl

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\tl_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of \".

The standard contents of this variable is \", \', \., \^, \‘, \~, \c, \H, \k, \r, \t, \u and \v.

"Mixed" case conversion may be regarded informally as converting the first character of the ⟨*tokens*⟩ to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example ij in Dutch which becomes IJ. As such, \tl_mixed_case:n(n) implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the ⟨*tokens*⟩ are ignored when finding the first "letter" for conversion.

```
\tl_mixed_case:n { hello~WORLD }   % => "Hello world"
\tl_mixed_case:n { ~hello~WORLD }  % => " Hello world"
\tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first "letter" for this process, any content in math mode or covered by \l_tl_case_change_exclude_tl is ignored.

(Note that the Unicode Consortium describe this as "title case", but that in English title case applies on a word-by-word basis. The "mixed" case implemented here is a lower level concept needed for both "title" and "sentence" casing of text.)

\l_tl_mixed_case_ignore_tl

The list of characters to ignore when searching for the first "letter" in mixed-casing is determined by \l_tl_mixed_change_ignore_tl. This has the standard setting

```
( [ { ‘ -
```

where comparisons are made on a character basis.

As is generally true for expl3, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with XeTeX or LuaTeX a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the Unicode Consortium in UnicodeData.txt and SpecialCasing.txt. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1 font encoding. Thus for example Ãđ can be case-changed using pdfTeX. For pTeX only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection will expand input but treats any unexpandable control sequences as "failures" to match a context.

Language-sensitive conversions are enabled using the ⟨*language*⟩ argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lower casing I-dot and introduced when upper casing i-dotless.
- German (`de-alt`). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (`lt`). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdfTEX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

`\tl_set_from_file:Nnn`
`\tl_set_from_file:cnn`
`\tl_gset_from_file:Nnn`
`\tl_gset_from_file:cnn`

New: 2014-06-25

`\tl_set_from_file:Nnn` ⟨*tl*⟩ {⟨*setup*⟩} {⟨*filename*⟩}

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument.

`\tl_set_from_file_x:Nnn`
`\tl_set_from_file_x:cnn`
`\tl_gset_from_file_x:Nnn`
`\tl_gset_from_file_x:cnn`

New: 2014-06-25

`\tl_set_from_file_x:Nnn` ⟨*tl*⟩ {⟨*setup*⟩} {⟨*filename*⟩}

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the ⟨*setup*⟩ argument.

`\tl_log:N`
`\tl_log:c`

New: 2014-08-22
Updated: 2015-08-01

`\tl_log:N` ⟨*tl var*⟩

Writes the content of the ⟨*tl var*⟩ in the log file. See also `\tl_show:N` which displays the result in the terminal.

`\tl_log:n`

New: 2014-08-22

`\tl_log:n` ⟨*token list*⟩

Writes the ⟨*token list*⟩ in the log file. See also `\tl_show:n` which displays the result in the terminal.

16 Additions to l3tokens

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

Updated: 2012-12-20

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), or an outer token (never used in LaTeX3) 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 XXIV
The l3sys package
System/runtime functions

1 The name of the job

\c_sys_jobname_str

New: 2015-09-19

Constant that gets the "job name" assigned when TEX starts.

TEXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by TEX and should not be overwritten by the package.

2 Date and time

\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

TEXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the "real" values.

2.1 Engine

\sys_if_engine_luatex_p: ⋆
\sys_if_engine_luatex:*TF* ⋆
\sys_if_engine_pdftex_p: ⋆
\sys_if_engine_pdftex:*TF* ⋆
\sys_if_engine_ptex_p: ⋆
\sys_if_engine_ptex:*TF* ⋆
\sys_if_engine_uptex_p: ⋆
\sys_if_engine_uptex:*TF* ⋆
\sys_if_engine_xetex_p: ⋆
\sys_if_engine_xetex:*TF* ⋆

New: 2015-09-07

`\sys_if_engine_pdftex:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the `(u)ptex` tests are for ε-pTEX and ε-upTEX as expl3 requires the ε-TEX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for ε-pTEX but false for ε-upTEX.

\c_sys_engine_str

New: 2015-09-19

The current engine given as a lower case string: will be one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

2.2 Output format

\sys_if_output_dvi_p: ⋆
\sys_if_output_dvi:TF ⋆
\sys_if_output_pdf_p: ⋆
\sys_if_output_pdf:TF ⋆

New: 2015-09-19

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

Conditionals which give the current output mode the TEX run is operating in. This will always be one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

\c_sys_output_str

New: 2015-09-19

The current output mode given as a lower case string: will be one of `dvi` or `pdf`.

Part XXV
The l3luatex package LuaTeX-specific functions

1 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 `\sys_if_engine_luatex:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

1.1 TeX code interfaces

`\lua_now_x:n` ⋆
`\lua_now:n` ⋆

New: 2015-06-29

`\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.

In the case of the `\lua_now_x:n` version the input is fully expanded by TeX in an `x`-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions will be required to yield the result of the Lua code.

`\lua_shipout_x:n`
`\lua_shipout:n`

New: 2015-06-30

`\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: no TeX expansion of the ⟨*Lua input*⟩ will occur at this stage.

In the case of the `\lua_shipout_x:n` version the input is fully expanded by TeX in an `x`-type manner during the shipout operation.

TeXhackers note: At a TeX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

\lua_escape_x:n ⋆ `\lua_escape:n {⟨token list⟩}`
\lua_escape:n ⋆

New: 2015-06-29

Converts the ⟨*token list*⟩ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

In the case of the `\lua_escape_x:n` version the input is fully expanded by TEX in an `x`-type manner *but* the function remains fully expandable.

TEXhackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when LuaTEX is in use two expansions will be required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for TEX, there are a small number of Lua functions provided here. Currently these are intended for internal use only.

l3kernel.strcmp `\l3kernel.strcmp(⟨str one⟩, ⟨str two⟩)`

Compares the two strings and returns `0` to TEX if the two are identical.

l3kernel.charcat `\l3kernel.charcat(⟨charcode⟩, ⟨catcode⟩)`

Constructs a character of ⟨*charcode*⟩ and ⟨*catcode*⟩ and returns the result to TEX.

Part XXVI
The l3drivers package Drivers

TEX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, LATEX3 is aware of the following drivers:

- `pdfmode`: The "driver" for direct PDF output by *both* pdfTEX and LuaTEX (no separate driver is used in this case: the engine deals with PDF creation itself).
- `dvips`: The `dvips` program, which works in conjugation with pdfTEX or LuaTEX in DVI mode.
- `dvipdfmx`: The `dvipdfmx` program, which works in conjugation with pdfTEX or LuaTEX in DVI mode.
- `dvisvgm`: The `dvisvgm` program, which works in conjugation with pdfTEX or LuaTEX in DVI mode to create SVG output.
- `xdvipdfmx`: The driver used by XETEX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level "up", and they must be used in the correct contexts.d

1 Box clipping

`\__driver_box_use_clip:N` `\__driver_box_use_clip:N` ⟨*box*⟩

New: 2011-11-11

Inserts the content of the ⟨*box*⟩ at the current insertion point such that any material outside of the bounding box will not be displayed by the driver. The material in the ⟨*box*⟩ is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

`\__driver_box_use_rotate:Nn` `\__driver_box_use_rotate:Nn` ⟨*box*⟩ {⟨*angle*⟩}

New: 2016-05-12

Inserts the content of the ⟨*box*⟩ at the current insertion point rotated by the ⟨*angle*⟩ (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the TEX reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the TEX side.

This function should only be used within a surrounding horizontal box construct.

__driver_box_use_scale:Nnn __driver_box_use_scale:Nnn ⟨*box*⟩ {⟨*x-scale*⟩} {⟨*y-scale*⟩}

New: 2016-05-12

Inserts the content of the ⟨*box*⟩ at the current insertion point scale by the ⟨*x-scale*⟩ and ⟨*y-scale*⟩. The material is inserted with no apparent height or width. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the TEX side.

This function should only be used within a surrounding horizontal box construct.

3 Color support

__driver_color_ensure_current: __driver_color_ensure_current:

New: 2011-09-03
Updated: 2012-05-18

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any "color safe" box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of pgf (most have the same interface as the same functions in the latter's \pgfsys@... namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by pgf and Ti*k*z.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

__driver_draw_begin:
__driver_draw_end:

__driver_draw_begin:
⟨*content*⟩
__driver_draw_end:

Defines a drawing environment. This will be a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the ⟨*content*⟩ should be zero from the TEX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

__driver_draw_scope_begin:
__driver_draw_scope_end:

__driver_draw_scope_begin:
⟨*content*⟩
__driver_draw_scope_end:

Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form TEX groups and may not be aligned with them.

4.1 Path construction

`\__driver_draw_moveto:nn` `\__driver_draw_move:nn {⟨x⟩} {⟨y⟩}`

Moves the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix will apply.

`\__driver_draw_lineto:nn` `\__driver_draw_lineto:nn {⟨x⟩} {⟨y⟩}`

Adds a path from the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_curveto:nnnnnn` `\__driver_draw_curveto:nnnnnn {⟨x1⟩} {⟨y1⟩} {⟨x2⟩} {⟨y2⟩} {⟨x3⟩} {⟨y3⟩}`

Adds a Bezier curve path from the current drawing reference point to (⟨x_3⟩, ⟨y_3⟩), using (⟨x_1⟩, ⟨y_1⟩) and (⟨x_2⟩, ⟨y_2⟩) as control points; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_rectangle:nnnn` `\__driver_draw_rectangle:nnnn {⟨x⟩} {⟨y⟩} {⟨width⟩} {⟨height⟩}`

Adds rectangular path from (⟨x_1⟩, ⟨y_1⟩) of ⟨*height*⟩ and ⟨*width*⟩; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

`\__driver_draw_closepath:` `\__driver_draw_closepath:`

Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

`\__driver_draw_stroke:` ⟨*path construction*⟩
`\__driver_draw_closestroke:` `\__driver_draw_stroke:`

Draws a line alone the current path, which will also be closed when `\__driver_draw_closestroke:` is used. The nature of the line drawn is influenced by settings for

- Line thickness
- Stroke color (or the current color if no specific stroke color is set)
- Line capping (how non-closed line ends should look)
- Join style (how a bend in the path should be rendered)
- Dash pattern

The path may also be used for clipping.

`\__driver_draw_fill:` ⟨*path construction*⟩
`\__driver_draw_fillstroke:` `\__driver_draw_fill:`

Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version will also stroke the path as described for `\__driver_draw_stroke:`. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

`\__driver_draw_nonzero_rule:` `\__driver_draw_nonzero_rule:`
`\__driver_draw_evenodd_rule:`

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

`\__driver_draw_clip:` ⟨*path construction*⟩
`\__driver_draw_clip:`

Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

`\__driver_draw_discardpath:` ⟨*path construction*⟩
`\__driver_draw_discardpath:`

Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

4.3 Stroke options

`\__driver_draw_linewidth:n` `\__driver_draw_linewidth:n {`⟨*dimexpr*⟩`}`

Sets the width to be used for stroking to ⟨*dimexpr*⟩.

`\__driver_draw_dash:nn` `\__driver_draw_dash:nn {`⟨*dash pattern*⟩`} {`⟨*phase*⟩`}`

Sets the pattern of dashing to be used when stroking a line. The ⟨*dash pattern*⟩ should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example `3pt, 4pt` means that 3 pt on, 4 pt off, 3 pt on, and so on. A more complex pattern will also repeat: `3pt, 4pt, 1pt, 2pt` results in 3 pt on, 4 pt off, 1 pt on, 2 pt off, 3 pt on, and so on. An odd number of entries means that the last is repeated, for example `3pt` is equal to `3pt, 3pt`. An empty pattern yields a solid line.

The ⟨*phase*⟩ specifies an offset at the start of the cycle. For example, with a pattern `3pt` a phase of `1pt` will mean that the output is 2 pt on, 3 pt off, 3 pt on, 3 pt on, *etc.*

__driver_draw_cap_butt:
__driver_draw_cap_rectangle:
__driver_draw_cap_round:

__driver_draw_cap_butt:

Sets the style of terminal stroke position to one of butt, rectangle or round.

__driver_draw_join_bevel:
__driver_draw_join_miter:
__driver_draw_join_round:

__driver_draw_cap_butt:

Sets the style of stroke joins to one of bevel, miter or round.

__driver_draw_miterlimit:n

__driver_draw_miterlimit:n {⟨*dimexpr*⟩}

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

__driver_draw_color_cmyk:nnnn
__driver_draw_color_cmyk_fill:nnnn
__driver_draw_color_cmyk_stroke:nnnn

__driver_draw_color_cmyk:nnnn {⟨*cyan*⟩} {⟨*magneta*⟩} {⟨*yellow*⟩} {⟨*black*⟩}

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

__driver_draw_color_gray:n
__driver_draw_color_gray_fill:n
__driver_draw_color_gray_stroke:n

__driver_draw_color_gray:n {⟨*gray*⟩}

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

__driver_draw_color_rgb:nnn
__driver_draw_color_rgb_fill:nnn
__driver_draw_color_rgb_stroke:nnn

__driver_draw_color_gray:n {⟨*red*⟩} {⟨*green*⟩} {⟨*blue*⟩}

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting TEX material

__driver_draw_hbox:nnnnnn

__driver_draw_hbox:Nnnnnnn ⟨*box*⟩
{⟨a⟩} {⟨b⟩} {⟨c⟩} {⟨d⟩} {⟨x⟩} {⟨y⟩}

Inserts the ⟨*box*⟩ as an hbox with the box reference point placed at (x, y). The transformation matrix [*abcd*] will be applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that TEX material should not be inserted directly into a drawing as it will not be in the correct location. Also note that as for other drawing elements the box here will have no size from a TEX perspective.

4.6 Coordinate system transformations

__driver_draw_transformcm:nnnnnn

__driver_draw_transformcm:nnnnnn {⟨a⟩} {⟨b⟩} {⟨c⟩} {⟨d⟩}
{⟨x⟩} {⟨y⟩}

Applies the transformation matrix [*abcd*] and offset vector (x, y) to the current graphic state. This will affect any subsequent items in the same scope but not those already given.

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.

Symbols

&& *190*
* *191*
** *191*
+ *191*, *191*
- *191*, *191*
/ *191*
\::: *33*
\::N *33*
\::V *33*
\::c *33*
\::error 204
\::f *33*
\::n *33*
\::o *33*
\::p *33*
\::v *33*
\::x *33*
< *191*
= *191*
? *191*
?: *190*
^ *191*

A

abs *191*
acos *193*
acosd *193*
acot *194*
acotd *194*
acsc *193*
acscd *193*
asec *193*
asecd *193*
asin *193*
asind *193*
atan *194*
atand *194*

B

bool commands:
\bool_do_until:Nn *38*, 38
\bool_do_until:nn *39*, 39
\bool_do_while:Nn *38*, 38
\bool_do_while:nn *39*, 39
.bool_gset:N *162*
\bool_gset:Nn *37*
\bool_gset_eq:NN *37*
\bool_gset_false:N *36*
.bool_gset_inverse:N *162*
\bool_gset_true:N *36*
\bool_if:NTF *37*, 37
\bool_if:nTF . 37, *38*, 38, 39, 39, 39, 39
\bool_if_exist:NTF *37*, 37
\bool_if_exist_p:N *37*, 37
\bool_if_p:N *37*, 37
\bool_if_p:n *38*, 38
\bool_lazy_all:nTF 204, *205*, 205, 205, 205
\bool_lazy_all_p:n *205*, 205
\bool_lazy_and:nnTF 204, 205, *205*, 205, 205
\bool_lazy_and_p:nn . . . 205, *205*, 205
\bool_lazy_any:nTF 204, *205*, 205, 205, 205
\bool_lazy_any_p:n 205, *205*, 205
\bool_lazy_or:nnTF 204, 205, *205*, 205, 205
\bool_lazy_or_p:nn *205*, 205
\bool_log:N *205*, 205
\bool_log:n *205*, 205
\bool_new:N *36*, 36
\bool_not_p:n *38*, 38
.bool_set:N *162*
\bool_set:Nn *37*, 37
\bool_set_eq:NN *37*, 37
\bool_set_false:N *36*, 36
.bool_set_inverse:N *162*
\bool_set_true:N *36*, 36
\bool_show:N *37*, 37
\bool_show:n *37*, 37
\bool_until_do:Nn *38*, 38
\bool_until_do:nn *39*, 39
\bool_while_do:Nn *39*, 39
\bool_while_do:nn *39*, 39
\bool_xor_p:nn *38*, 38
\c_false_bool *20*, 36
\g_tmpa_bool *37*
\l_tmpa_bool *37*
\g_tmpb_bool *37*
\l_tmpb_bool *37*
\c_true_bool *20*, 36
box commands:
\box_clear:N *137*, 137
box_clear:N 137
\box_clear_new:N *137*, 137

\box_clip:N *200*, 200, 200, 201
\box_dp:N *138*, 138
\box_gclear:N *137*
\box_gclear_new:N *137*
\box_gset_eq:NN *137*
\box_gset_eq_clear:NN *137*, 137
\box_gset_to_last:N *139*
\box_ht:N *138*, 138
\box_if_empty:NTF *139*, 139
\box_if_empty_p:N *139*, 139
\box_if_exist:NTF *137*, 137
\box_if_exist_p:N *137*, 137
\box_if_horizontal:NTF *139*, 139
\box_if_horizontal_p:N *139*, 139
\box_if_vertical:NTF *139*, 139
\box_if_vertical_p:N *139*, 139
\box_log:N *140*, 140
\box_log:Nnn *140*, 140
\box_move_down:nn *138*
\box_move_left:nn *138*
\box_move_right:nn *138*, 138
\box_move_up:nn *138*, 138
\box_new:N *137*, 137, 137
\box_resize:Nnn *199*, 199
\box_resize_to_ht:Nn *199*, 199
\box_resize_to_ht_plus_dp:Nn *199*, 199
\box_resize_to_wd:Nn *199*, 199
\box_resize_to_wd_and_ht:Nnn *200*, 200
\box_rotate:Nn *200*, 200
\box_scale:Nnn *200*, 200
\box_set_dp:Nn *139*, 139
\box_set_eq:NN *137*, 137
\box_set_eq_clear:NN *137*, 137
\box_set_ht:Nn *139*, 139
\box_set_to_last:N *139*, 139
\box_set_wd:Nn *139*, 139
\box_show:N *140*, 140
\box_show:Nnn *140*, 140
\box_trim:Nnnnn *200*, 200
\box_use:N *138*, 138
\box_use_clear:N *138*, 138
\box_viewport:Nnnnn *201*, 201
\box_wd:N *139*, 139
\c_empty_box 139, *140*
\g_tmpa_box *140*
\l_tmpa_box *140*
\g_tmpb_box *140*
\l_tmpb_box *140*
\c_void_box 137
bp *195*

C

catcode commands:
\c_catcode_active_tl *52*
\c_catcode_letter_token *52*
\c_catcode_other_space_tl *177*
\c_catcode_other_token *52*
cc *195*
ceil *192*
char commands:
\l_char_active_seq *51*, 172, 178
\char_generate:nn *48*, 48, 61
\char_gset_active_eq:NN *47*
\char_gset_active_eq:nN *48*
\char_set_active_eq:NN *47*, 47
\char_set_active_eq:nN *48*, 48
\char_set_active_eq:nN\char_gset_active_eq:nN *48*
\char_set_catcode:nn *50*, 50
\char_set_catcode_⟨*type*⟩ 50
\char_set_catcode_active:N *49*
\char_set_catcode_active:n *49*
\char_set_catcode_alignment:N ... *49*
\char_set_catcode_alignment:n ... *49*
\char_set_catcode_comment:N *49*
\char_set_catcode_comment:n *49*
\char_set_catcode_end_line:N ... *49*
\char_set_catcode_end_line:n ... *49*
\char_set_catcode_escape:N *49*
\char_set_catcode_escape:n *49*
\char_set_catcode_group_begin:N . *49*
\char_set_catcode_group_begin:n . *49*
\char_set_catcode_group_end:N ... *49*
\char_set_catcode_group_end:n ... *49*
\char_set_catcode_ignore:N *49*
\char_set_catcode_ignore:n *49*
\char_set_catcode_invalid:N *49*
\char_set_catcode_invalid:n *49*
\char_set_catcode_letter:N .. *49*, 49
\char_set_catcode_letter:n .. *49*, 49
\char_set_catcode_math_subscript:N *49*
\char_set_catcode_math_subscript:n *49*
\char_set_catcode_math_superscript:N *49*
\char_set_catcode_math_superscript:n *49*
\char_set_catcode_math_toggle:N . *49*
\char_set_catcode_math_toggle:n . *49*
\char_set_catcode_other:N *49*
\char_set_catcode_other:n *49*
\char_set_catcode_parameter:N ... *49*
\char_set_catcode_parameter:n ... *49*
\char_set_catcode_space:N *49*
\char_set_catcode_space:n *49*
\char_set_lccode:nn *50*, 50
\char_set_mathcode:nn *51*, 51

\char_set_sfcode:nn 51, 51
\char_set_uccode:nn 50, 50
\char_show_value_catcode:n .. 50, 50
\char_show_value_lccode:n ... 50, 50
\char_show_value_mathcode:n . 51, 51
\char_show_value_sfcode:n ... 51, 51
\char_show_value_uccode:n ... 51, 51
\l_char_special_seq 51
\char_value_catcode:n 50, 50
\char_value_lccode:n 50, 50
\char_value_mathcode:n 51, 51
\char_value_sfcode:n 51, 51
\char_value_uccode:n 51, 51
char internal commands:
__char_generate:nn 61, 61
chk internal commands:
__chk_if_exist_cs:N 21, 21
__chk_if_exist_var:N 22, 22
__chk_if_free_cs:N 21, 21
__chk_log:n 22, 22, 22, 22
__chk_resume_log: 22, 22, 22
__chk_suspend_log: 22, 22, 22
choice commands:
.choice: 162
choices commands:
.choices:nn 162
clist commands:
\clist_clear:N 123, 123
clist_clear:N 123
\clist_clear_new:N 123, 123
\clist_concat:NNN 124, 124
\clist_const:Nn 123, 123
\clist_count:N 128, 128, 130
\clist_count:n 128
\clist_gclear:N 123
\clist_gclear_new:N 123
\clist_gconcat:NNN 124
\clist_get:NN 129, 129
\clist_get:NNTF 129, 129
\clist_gpop:NN 129, 129
\clist_gpop:NNTF 129, 129
\clist_gpush:Nn 130
\clist_gput_left:Nn 124
\clist_gput_right:Nn 125
\clist_gremove_all:Nn 125
\clist_gremove_duplicates:N ... 125
\clist_greverse:N 125
.clist_gset:N 162
\clist_gset:Nn 124
\clist_gset_eq:NN 124
\clist_gset_from_seq:NN 124
\clist_if_empty:NTF 126, 126
\clist_if_empty:nTF 126, 126
\clist_if_empty_p:N 126, 126
\clist_if_empty_p:n 126, 126
\clist_if_exist:NTF 124, 124
\clist_if_exist_p:N 124, 124
\clist_if_in:NnTF 126, 126
\clist_if_in:nnTF 126
\clist_item:Nn 130, 130
\clist_item:nn 130
\clist_log:N 201, 201
\clist_log:n 201, 201
\clist_map_break: 127, 127
\clist_map_break:n 127, 127
\clist_map_function:NN .....
..... 42, 123, 126, 126, 126
\clist_map_function:nN 126
\clist_map_inline:Nn .. 126, 127, 127
\clist_map_inline:nn 127
\clist_map_variable:NNn 127, 127
\clist_map_variable:nNn 127
\clist_new:N 123, 123, 123
\clist_pop:NN 129, 129
\clist_pop:NNTF 129, 129
\clist_push:Nn 130, 130
\clist_put_left:Nn 124, 124
\clist_put_right:Nn 125, 125
\clist_remove_all:Nn 125, 125
\clist_remove_duplicates:N . 125, 125
\clist_reverse:N 125, 125
\clist_reverse:n 125, 125
.clist_set:N 162
\clist_set:Nn 124, 124
\clist_set_eq:NN 124, 124
\clist_set_from_seq:NN 124, 124
\clist_show:N 130, 130, 201
\clist_show:n 130, 130, 201
\clist_use:Nn 128, 128
\clist_use:Nnnn 128, 128
\c_empty_clist 130
\g_tmpa_clist 130
\l_tmpa_clist 130
\g_tmpb_clist 130
\l_tmpb_clist 130
cm 195
code commands:
.code:n 162
coffin commands:
\coffin_attach:NnnNnnnn 146, 146
\coffin_clear:N 145, 145
\coffin_display_handles:Nn . 147, 147
\coffin_dp:N 147, 147
\coffin_ht:N 147, 147
\coffin_if_exist:NTF 145, 145
\coffin_if_exist_p:N 145, 145
\coffin_join:NnnNnnnn 147, 147
\coffin_log_structure:N 201, 201

\coffin_mark_handle:Nnnn ... *148*, 148
\coffin_new:N *145*, 145
\coffin_resize:Nnn *201*, 201
\coffin_rotate:Nn *201*, 201
\coffin_scale:Nnn *201*, 201
\coffin_set_eq:NN *145*, 145
\coffin_set_horizontal_pole:Nnn *146*, 146
\coffin_set_vertical_pole:Nnn *146*, 146
\coffin_show_structure:N *148*, 148, 201
\coffin_typeset:Nnnnn *147*, 147
\coffin_wd:N *147*, 147
\c_empty_coffin *148*
\l_tmpa_coffin *148*
\l_tmpb_coffin *148*
color commands:
\color_ensure_current: *149*, 149
\color_group_begin: ... *149*, 149, 149
\color_group_end: *149*, 149, 149
cos *193*
cosd *193*
cot *193*
cotd *193*
cs commands:
\cs:w *16*, 16, 16, 17
\cs_end: *16*, 16, 16
\cs_generate_from_arg_count:NNnn *14*, 14
\cs_generate_variant:Nn 10, 24, 24, *25*, 25, 25, 26
\cs_git␣cogset_nopar:Nn 14
\cs_gset:Nn *14*, 14
\cs_gset:Npn 10, *12*, 12
\cs_gset:Npx *12*
\cs_gset_eq:NN *15*, 15, 15
\cs_gset_nopar:Nn *14*
\cs_gset_nopar:Npn *12*, 12
\cs_gset_nopar:Npx *12*
\cs_gset_protected:Nn *14*, 14
\cs_gset_protected:Npn *12*, 12
\cs_gset_protected:Npx *12*
\cs_gset_protected_nopar:Nn . *14*, 14
\cs_gset_protected_nopar:Npn *12*, 12
\cs_gset_protected_nopar:Npx ... *12*
\cs_if_eq:NNTF *20*, 20
\cs_if_eq_p:NN *20*, 20
\cs_if_exist:N 20
\cs_if_exist:NTF *20*, 20, 159
\cs_if_exist_p:N *20*, 20, 21
\cs_if_exist_use:N *16*, 16
\cs_if_exist_use:NTF *16*, 16
\cs_if_free:NTF *20*, 20, 34
\cs_if_free_p:N . 19, *20*, 20, 21, 22, 34
\cs_log:N 158, *198*, 198
\cs_meaning:N *15*, 15
\cs_new:Nn *12*, 12, 35
\cs_new:Npn .. 10, *11*, 11, 14, 34, 34, 35
\cs_new:Npx *11*
\cs_new_eq:NN *15*, 15, 15
\cs_new_nopar:Nn *13*, 13
\cs_new_nopar:Npn *11*, 11
\cs_new_nopar:Npx *11*
\cs_new_protected:Nn *13*, 13
\cs_new_protected:Npn *11*, 11
\cs_new_protected:Npx *11*
\cs_new_protected_nopar:Nn .. *13*, 13
\cs_new_protected_nopar:Npn . *11*, 11
\cs_new_protected_nopar:Npx *11*
\cs_set:Nn *13*, 13
\cs_set:Npn 10, *11*, 11, 34, 34, 35
\cs_set:Npx *11*
\cs_set_eq:NN *15*, 15, 15
\cs_set_nopar:Nn *13*, 13
\cs_set_nopar:Npn 10, *11*, 11, 52
\cs_set_nopar:Npx *11*
\cs_set_protected:Nn *13*, 13
\cs_set_protected:Npn 10, *11*, 11
\cs_set_protected:Npx *11*
\cs_set_protected_nopar:Nn .. *14*, 14
\cs_set_protected_nopar:Npn . *12*, 12
\cs_set_protected_nopar:Npx *12*
\cs_show:N *16*, 16, 20, 158, 198
\cs_to_str:N *4*, *17*, 17, 96, 102
\cs_undefine:N *15*, 15
cs internal commands:
__cs_count_signature:N *22*, 22
__cs_get_function_name:N ... *22*, 22
__cs_get_function_signature:N *22*, 22
__cs_split_function:NN *22*, 22
__cs_tmp:w *22*
csc *193*
cscd *193*

D

dd *195*
default commands:
.default:n *163*
deg *195*
dim commands:
\dim_abs:n *76*, 76
\dim_add:Nn *76*, 76
\dim_case:nn *79*, 79
\dim_case:nnTF *79*, 79
\dim_compare:nNnTF *77*, 77, 79, 79, 79, 80
\dim_compare:nTF 74, *78*, 78, 80, 80, 80, 80, 84

\dim_compare_p:n *78*, 78
\dim_compare_p:nNn *77*, 77
\dim_const:Nn *75*, 75
\dim_do_until:nn *80*, 80
\dim_do_until:nNnn *79*, 79
\dim_do_while:nn *80*, 80
\dim_do_while:nNnn *79*, 79
\dim_eval:n 77, 78, *80*, 80, 80, 88
\dim_gadd:Nn *76*
.dim_gset:N *163*
\dim_gset:Nn *76*
\dim_gset_eq:NN *76*
\dim_gsub:Nn *76*
\dim_gzero:N *75*
\dim_gzero_new:N *75*
\dim_if_exist:NTF *75*, 75
\dim_if_exist_p:N *75*, 75
\dim_log:N *207*, 207
\dim_log:n *207*, 207
\dim_max:nn *76*, 76
\dim_min:nn *76*, 76
\dim_new:N 75, 75, *75*
\dim_ratio:nn *77*, 77, 77
.dim_set:N *163*
\dim_set:Nn *76*, 76
\dim_set_eq:NN *76*, 76
\dim_show:N *82*, 82
\dim_show:n *82*, 82
\dim_sub:Nn *76*, 76
\dim_to_decimal:n *81*, 81
\dim_to_decimal_in_bp:n ... *81*, 81, 81
\dim_to_decimal_in_sp:n ... *81*, 81, 81
\dim_to_decimal_in_unit:nn *81*, 81, 81
\dim_to_fp:n *82*, 82, 82
\dim_until_do:nn *80*, 80
\dim_until_do:nNnn *79*, 79
\dim_use:N 80, *80*, 80, 80
\dim_while_do:nn *80*, 80
\dim_while_do:nNnn *80*, 80
\dim_zero:N *75*, 75
dim_zero:N 75
\dim_zero_new:N *75*, 75
\c_max_dim *82*, 85
\g_tmpa_dim *82*
\l_tmpa_dim *82*
\g_tmpb_dim *82*
\l_tmpb_dim *82*
\c_zero_dim *82*
dim internal commands:
__dim_eval:w *88*, 88
__dim_eval_end: *88*, 88, 88, 88
driver internal commands:
__driver_box_use_clip:N ... *217*, 217
__driver_box_use_rotate:Nn *217*, 217
__driver_box_use_scale:Nnn *218*, 218
__driver_color_ensure_current: *218*, 218
__driver_draw_begin: *218*, 218
__driver_draw_cap_butt: *221*, 221, 221
__driver_draw_cap_rectangle: .. *221*
__driver_draw_cap_round: *221*
__driver_draw_clip: *220*, 220
__driver_draw_closepath: .. *219*, 219
__driver_draw_closestroke: *219*, 219
__driver_draw_color_cmyk:nnnn *221*, 221
__driver_draw_color_cmyk_- fill:nnnn *221*
__driver_draw_color_cmyk_- stroke:nnnn *221*
__driver_draw_color_gray:n *221*, 221, 221
__driver_draw_color_gray_fill:n *221*
__driver_draw_color_gray_- stroke:n *221*
__driver_draw_color_rgb:nnn .. *221*
__driver_draw_color_rgb_- fill:nnn *221*
__driver_draw_color_rgb_- stroke:nnn *221*
__driver_draw_curveto:nnnnnn *219*, 219
__driver_draw_dash:nn *220*, 220
__driver_draw_discardpath: *220*, 220
__driver_draw_end: *218*, 218
__driver_draw_evenodd_rule: .. *220*
__driver_draw_fill: *220*, 220
__driver_draw_fillstroke: *220*
__driver_draw_hbox:nnnnnn *222*
__driver_draw_hbox:Nnnnnnn ... 222
__driver_draw_join_bevel: *221*
__driver_draw_join_miter: *221*
__driver_draw_join_round: *221*
__driver_draw_lineto:nn ... *219*, 219
__driver_draw_linewidth:n . *220*, 220
__driver_draw_miterlimit:n *221*, 221
__driver_draw_move:nn 219
__driver_draw_moveto:nn *219*
__driver_draw_nonzero_rule: *220*, 220
__driver_draw_rectangle:nnnn *219*, 219
__driver_draw_scope_begin: *218*, 218
__driver_draw_scope_end: .. *218*, 218
__driver_draw_stroke: . *219*, 219, 220
__driver_draw_transformcm:nnnnnn *222*, 222

E

\edef 4
else commands:
\else: *21*, 35, 40, 40, 73, 73, 73, 73, 73, 88, 143, 143, 144, 178, 178
em *195*
etex commands:
\etex_... 8
ex *195*
exp *191*
exp commands:
\exp:w *31*, 31, 31, *32*, 32, 32, 32, *32*, 32
\exp_after:wN *30*, 30, 31, 32, 32
\exp_arg:N 30
\exp_args:cc *27*
\exp_args:N⟨*variant*⟩ 25
\exp_args:Nc *27*, 27
\exp_args:Ncc *28*
\exp_args:Nccc *28*
\exp_args:Ncco *28*
\exp_args:Nccx *29*
\exp_args:Ncf *28*
\exp_args:NcNc *28*
\exp_args:NcNo *28*
\exp_args:Ncnx *29*
\exp_args:Nco *28*
\exp_args:Ncx *28*
\exp_args:Nf *27*, 27
\exp_args:Nff *28*
\exp_args:Nfo *28*
\exp_args:NNc *28*, 28
\exp_args:Nnc *28*
\exp_args:NNf *28*
\exp_args:Nnf *28*
\exp_args:Nnnc *28*
\exp_args:NNNo *28*, 28
\exp_args:NNno *28*
\exp_args:Nnno *28*
\exp_args:NNNV *28*
\exp_args:NNNx *29*
\exp_args:NNnx *29*, 29
\exp_args:Nnnx *29*
\exp_args:NNo 24, 24, 24, *28*
\exp_args:Nno *28*
\exp_args:NNoo *28*, 28
\exp_args:NNox *29*
\exp_args:Nnox *29*
\exp_args:NNV *28*
\exp_args:NNv *28*
\exp_args:NnV *28*
\exp_args:NNx *28*, 28
\exp_args:Nnx *28*
\exp_args:No *27*, 27
\exp_args:Noc *28*
\exp_args:Nof *28*
\exp_args:Noo *28*, 28
\exp_args:Nooo *28*
\exp_args:Noox *29*
\exp_args:Nox *28*
\exp_args:NV *27*, 27
\exp_args:Nv *27*, 27
\exp_args:NVV *28*
\exp_args:Nx *27*, 27
\exp_args:Nxo *28*
\exp_args:Nxx *28*
\exp_end: *31*, 31, 31, 31, 31, 31, 32
\exp_end_continue_f:nw *32*, 32
\exp_end_continue_f:w *32*, 32, 32, 32, 32, 32
\exp_last_two_unbraced:Nnn . . *29*, 29
\exp_last_unbraced:Nn . *29*, 29, *29*, 29
\exp_last_unbraced:NNn *29*
\exp_last_unbraced:Nnn *29*, 29
\exp_last_unbraced:NNNn *29*
\exp_last_unbraced:NnNn *29*
\exp_not:N *30*, 30, *30*, 30, 177, 196
\exp_not:n *30*, 30, *30*, 30, *30*, 30, *31*, 31, *31*, 31, 98, 99, 100, 101, 114, 118, 118, 128, 128, 130, 133, 177, 208, 209
\exp_stop_f: *31*, 31, 31, 32, 32, 32
exp internal commands:
\l__exp_internal_tl *32*
\ExplFileDate 6
\ExplFileDescription 6
\ExplFileName 6
\ExplFileVersion 6
\ExplSyntaxOff *3*, *6*, 6, 6, 6, 7
\ExplSyntaxOn *3*, *6*, 6, 6, 6, 7

F

false *195*
fi commands:
\fi: *21*, 35, 40, 40, 73, 73, 73, 88, 143, 143, 144, 178
file commands:
\file_add_path:nN *172*, 172, 178
\g_file_current_name_tl *172*
\file_if_exist:nTF *172*, 172, 172, 172, 178
\file_if_exist_input:n 202, 202
\file_if_exist_input:nTF . . . *202*, 202
\file_input:n 172, *172*, 172, 173, 202, 202
\file_list: *173*, 173
\file_path_include:n 172, *172*, 172, 202
\file_path_remove:n *173*, 173

file internal commands:
__file_if_exist:nTF 178
\g__file_internal_ior *178*
\l__file_internal_name_tl *178*
__file_name_sanitize:nn ... *178*, 178
floor *192*
fp commands:
\c_e_fp *186*, 189
\fp_abs:n 191, *195*, 195
\fp_add:Nn *182*, 182
\fp_compare:nNnTF *184*, 184, 185, 185, 185, 185
\fp_compare:nTF *184*, 184, 185, 185, 185, 186, 191
\fp_compare_p:n *184*, 184
\fp_compare_p:nNn *184*, 184
\fp_const:Nn *181*, 181
\fp_do_until:nn *185*, 185
\fp_do_until:nNnn *185*, 185
\fp_do_while:nn *185*, 185
\fp_do_while:nNnn *185*, 185
\fp_eval:n *182*, 182, 184, 190, 190, 190, 191, 191, 191, 191, 191, 191, 191, 191, 191, 191, 192, 192, 192, 192, 192, 192, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 193, 194, 194, 194, 194, 194, 194, 194, 194, 194, 194, 195, 195, 195, 196
\fp_flag_off:n *188*, 188
\fp_flag_on:n *188*, 188
\fp_format:nn 196
\fp_gadd:Nn *182*
.fp_gset:N *163*
\fp_gset:Nn *181*
\fp_gset_eq:NN *182*
\fp_gsub:Nn *182*
\fp_gzero:N *181*
\fp_gzero_new:N *181*
\fp_if_exist:NTF *183*, 183
\fp_if_exist_p:N *183*, 183
\fp_if_flag_on:nTF *188*, 188
\fp_if_flag_on_p:n *188*, 188
\fp_if_nan:nTF 196
\fp_log:N *203*, 203
\fp_log:n *203*, 203
\fp_max:nn *196*, 196
\fp_min:nn *196*
\fp_new:N *181*, 181, 181
.fp_set:N *163*
\fp_set:Nn *181*, 181, 196
\fp_set_eq:NN *182*, 182
\fp_show:N *188*, 188
\fp_show:n *188*, 188
\fp_step_function:nnnN *186*, 186
\fp_step_inline:nnnn *186*, 186
\fp_sub:Nn *182*, 182
\fp_to_decimal:N *182*, 182, 183
\fp_to_decimal:n 182, *182*, 182, 182, 183
\fp_to_dim:N *182*, 182
\fp_to_dim:n *182*, 182, 187
\fp_to_int:N *183*, 183
\fp_to_int:n *183*, 183
\fp_to_scientific:N *183*, 183
\fp_to_scientific:n ... *183*, 183, 183
\fp_to_tl:N *183*, 183
\fp_to_tl:n *183*, 183
\fp_trap:nn 187, *188*, 188
\fp_until_do:nn *185*, 185
\fp_until_do:nNnn *185*, 185
\fp_use:N *183*, 183
\fp_while_do:nn *186*, 186
\fp_while_do:nNnn *185*, 185
\fp_zero:N *181*, 181
fp_zero:N 181
\fp_zero_new:N *181*, 181
\c_inf_fp *186*, 195
\c_nan_fp 195
\c_one_fp *186*
\c_pi_fp *187*, 195
\g_tmpa_fp *187*
\l_tmpa_fp *187*
\g_tmpb_fp *187*
\l_tmpb_fp *187*
\c_zero_fp *186*
fp internal commands:
__fp_basics_pack_weird_high:NNNNNNNNw 197
__fp_basics_pack_weird_low:NNNNw 197

G

\GetIdInfo *6*, 6, 6
group commands:
\group_align_safe_begin: .. *40*, 40, 40
\group_align_safe_end: *40*, 40, 40
\group_begin: *9*, 9, 9
\c_group_begin_token *52*, 99
\group_end: *9*, 9, 9, 9
\c_group_end_token *52*
\group_insert_after:N *9*, 9, 9
groups commands:
.groups:n *163*

H

hbox commands:
\hbox:n *140*, 140

\hbox_gset:Nn 141
\hbox_gset:Nw 141
\hbox_gset_end: 141
\hbox_gset_to_wd:Nnn 141
\hbox_overlap_left:n 141, 141
\hbox_overlap_right:n 141, 141
\hbox_set:Nn 141, 141, 141
\hbox_set:Nw 141, 141
\hbox_set_end: 141, 141
\hbox_set_to_wd:Nnn 141, 141
\hbox_to_wd:nn 141, 141
\hbox_to_zero:n 141, 141
\hbox_unpack:N 141, 141
\hbox_unpack_clear:N 141, 141
hcoffin commands:
\hcoffin_set:Nn 145, 145
\hcoffin_set:Nw 145, 145
\hcoffin_set_end: 145, 145
hundred commands:
\c_one_hundred 72

I

if commands:
\if:w 21, 47, 47, 47
\if_bool:N 40, 40, 40
\if_box_empty:N 144, 144
\if_case:w 73, 73
\if_catcode:w 21
\if_charcode:w 21, 47
\if_cs_exist:N 21
\if_cs_exist:w 21
\if_dim:w 88, 88
\if_eof:w 178, 178
\if_false: 21, 36
\if_hbox:N 143, 143
\if_int_compare:w 20, 73, 73
\if_int_odd:w 73, 73
\if_meaning:w 21
\if_mode_horizontal: 21
\if_mode_inner: 21
\if_mode_math: 21
\if_mode_vertical: 21
\if_predicate:w 34, 36, 40, 40
\if_true: 21, 36
\if_vbox:N 143, 143
in 195
inf 195
initial commands:
.initial:n 163
int commands:
\c_eight 72
\c_eleven 72
\c_fifteen 72
\c_five 72
\c_four 72
\c_fourteen 72
\int_abs:n 62, 62
\int_add:Nn 64, 64
\int_case:nn 66, 66
\int_case:nnTF 23, 66, 66
\int_compare:nNnTF 64, 65, 65, 66, 66, 67, 67
\int_compare:nTF 65, 65, 67, 67, 67, 67, 74, 184
\int_compare_p:n 65, 65
\int_compare_p:nNn 20, 65, 65
\int_const:Nn 63, 63
\int_decr:N 64, 64
\int_div_round:nn 62, 62
\int_div_truncate:nn 63, 63, 63
\int_do_until:nn 67, 67
\int_do_until:nNnn 66, 66
\int_do_while:nn 67, 67
\int_do_while:nNnn 66, 66
\int_eval:n . 14, 26, 26, 62, 62, 62, 62, 62, 63, 64, 65, 65, 66, 73, 74, 159
\int_from_alph:n 70, 70
\int_from_base:nn 71, 71
\int_from_bin:n 70, 70
\int_from_hex:n 71, 71
\int_from_oct:n 71, 71
\int_from_roman:n 71, 71
\int_gadd:Nn 64
\int_gdecr:N 64
\int_gincr:N 64
.int_gset:N 163
\int_gset:Nn 64
\int_gset_eq:NN 63
\int_gsub:Nn 64
\int_gzero:N 63
\int_gzero_new:N 63
\int_if_even:nTF 66
\int_if_even_p:n 66
\int_if_exist:NTF 64, 64
\int_if_exist_p:N 64, 64
\int_if_odd:nTF 66, 66
\int_if_odd_p:n 66, 66
\int_incr:N 64, 64
\int_log:N 203, 203
\int_log:n 203, 203
\int_max:nn 63, 63
\int_min:nn 63, 63
\int_mod:nn 63, 63
\int_new:N 63, 63, 63
.int_set:N 163
\int_set:Nn 64, 64
\int_set_eq:NN 63, 63
\int_show:N 71, 71

\int_show:n *71*, 71
\int_step_function:nnnN *68*, 68
\int_step_inline:nnnn *68*, 68
\int_step_variable:nnnNn *68*, 68
\int_sub:Nn *64*, 64
\int_to_Alph:n *69*, 69, 70
\int_to_alph:n *69*, 69, 69, 69, 70
\int_to_arabic:n *68*, 68
\int_to_Base:n 70
\int_to_base:n 70
\int_to_Base:nn *70*, 71
\int_to_base:nn *70*, 70, 71
\int_to_bin:n *69*, 69, 70, 70
\int_to_Hex:n *70*, 70, 71
\int_to_hex:n *70*, 70, 70, 71
\int_to_oct:n *70*, 70, 71
\int_to_Roman:n *70*, 70, 71
\int_to_roman:n *70*, 70, 70, 71
\int_to_symbols:nnn 69, *69*, 69
\int_until_do:nn *67*, 67
\int_until_do:nNnn *67*, 67
\int_use:N 62, *64*, 64, 64
\int_while_do:nn *67*, 67
\int_while_do:nNnn *67*, 67
\int_zero:N *63*, 63
int_zero:N 63
\int_zero_new:N *63*, 63
\c_max_int *72*
\c_nine *72*
\c_one *72*
\c_seven *72*
\c_six *72*
\c_sixteen *72*
\c_ten *72*
\c_thirteen *72*
\c_three *72*
\g_tmpa_int *72*
\l_tmpa_int 2, *72*
\g_tmpb_int *72*
\l_tmpb_int 2, *72*
\c_twelve *72*
\c_two *72*
\c_zero *72*
int internal commands:
__int_eval:w *74*, 74
__int_eval_end: *74*, 74, 74, 74
__int_to_roman:w *73*, 73
__int_value:w *74*, 74, 74
ior commands:
\ior_close:N 173, 173, *174*, 174
\ior_get:NN *174*, 174, 175, 178
\ior_get_str:NN *175*, 175, 175
\ior_if_eof:NTF *175*, 175
\ior_if_eof_p:N *175*, 175
\ior_list_streams: *174*, 174
\ior_log_streams: *203*, 203
\ior_map_break: *202*, 202
\ior_map_break:n *203*, 203
\ior_map_inline:Nn *202*, 202
\ior_new:N *173*, 173
\ior_open:Nn *173*, 173
\ior_open:NnTF *173*, 173
\ior_str_map_inline:Nn *202*, 202
\c_term_ior *178*
ior internal commands:
__ior_open:Nn *178*, 178
iow commands:
\iow_char:N *176*, 176
\iow_close:N 173, *174*, 174
\iow_indent:n 177, *177*, 177
\l_iow_line_count_int . 177, 177, *177*
\iow_list_streams: *174*, 174
\iow_log:n 22, *175*, 175
\iow_log_streams: *203*, 203
\iow_new:N *173*, 173
\iow_newline: .....
..... 175, 176, *176*, 176, 176, 179
\iow_now:Nn .....
.... *175*, 175, 175, 175, 175, 176, 176
\iow_open:Nn *173*, 173
\iow_shipout:Nn *175*, 175, 175, 176, 176
\iow_shipout_x:Nn .. 175, *176*, 176, 176
\iow_term:n *175*, 175
\iow_wrap:nnnN 155, 156, 156,
175, 176, *177*, 177, 177, 177, 177, 177
\c_log_iow *178*
\c_term_iow *178*
iow internal commands:
__iow_with:Nnn *179*, 179

K

kernel internal commands:
\l__kernel_expl_bool *7*
__kernel_register_log:N ... *198*, 198
__kernel_register_show:N *22*, 22, 198
keys commands:
\l_keys_choice_int .....
.... 162, 164, 166, *166*, 166, 166, 167
\l_keys_choice_tl .....
..... 162, 164, *166*, 166, 166, 167
\keys_define:nn *161*, 161, 161
\keys_if_choice_exist:nnnTF *170*, 170
\keys_if_choice_exist_p:nnn *170*, 170
\keys_if_exist:nnTF *170*, 170
\keys_if_exist_p:nn *170*, 170
\l_keys_key_tl *168*, 168
\keys_log:nn *204*, 204
\l_keys_path_tl *168*, 168

\keys_set:nn 160, 163, *167*, 167, 168, 168, 169
\keys_set_filter:nnn *169*, 169
\keys_set_filter:nnnN . *169*, 169, 169
\keys_set_groups:nnn *170*, 170
\keys_set_known:nn *168*, 168
\keys_set_known:nnN *168*, 168, 168, 168
\keys_show:nn *170*, 170
\l_keys_value_tl *168*, 168
keyval commands:
\keyval_parse:NNn *171*, 171, 171

L

\l3kernel.charcat 216
l3kernel.charcat *216*
\l3kernel.strcmp 216
l3kernel.strcmp *216*
left commands:
\c_left_brace_str *110*
\let 1
ln *192*
lua commands:
\lua_escape:n *216*, 216
\lua_escape_x:n *216*, 216, 216
\lua_now:n *215*, 215, 215
\lua_now_x:n *215*, 215, 215
\lua_shipout:n *215*, 215, 215
\lua_shipout_x:n *215*, 215
luatex commands:
\luatex_... 8

M

math commands:
\c_math_subscript_token *52*
\c_math_superscript_token *52*
\c_math_toggle_token *52*
max *192*
max commands:
\c_max_char_int *72*
\c_max_register_int *72*
meta commands:
.meta:n *163*
.meta:nn *164*
min *192*
minus commands:
\c_minus_inf_fp *186*, 195
\c_minus_zero_fp *186*
mm *195*
mode commands:
\mode_if_horizontal:TF *39*, 39
\mode_if_horizontal_p: *39*, 39
\mode_if_inner:TF *40*, 40
\mode_if_inner_p: *40*, 40
\mode_if_math:TF *40*, 40
\mode_if_math_p: *40*
\mode_if_vertical:TF *40*, 40
\mode_if_vertical_p: *40*, 40
msg commands:
\msg_critical:nn *152*
\msg_critical:nnn *152*
\msg_critical:nnnn *152*
\msg_critical:nnnnn *152*
\msg_critical:nnnnnn *152*, 152
\msg_critical_text:n *151*, 151
\msg_error:nn *153*
\msg_error:nnn *153*
\msg_error:nnnn *153*
\msg_error:nnnnn *153*
\msg_error:nnnnnn *153*, 153, 204
\msg_error_text:n *151*, 151
\msg_expandable_error:nn *204*
\msg_expandable_error:nnn *204*
\msg_expandable_error:nnnn *204*
\msg_expandable_error:nnnnn ... *204*
\msg_expandable_error:nnnnnn *204*, 204
\msg_fatal:nn *152*
\msg_fatal:nnn *152*
\msg_fatal:nnnn *152*
\msg_fatal:nnnnn *152*
\msg_fatal:nnnnnn *152*, 152
\msg_fatal_text:n *151*, 151
\msg_gset:nnn *150*
\msg_gset:nnnn *150*
\msg_if_exist:nnTF *151*, 151
\msg_if_exist_p:nn *151*, 151
\msg_info:nn *153*
\msg_info:nnn *153*
\msg_info:nnnn *153*
\msg_info:nnnnn *153*
\msg_info:nnnnnn *153*, 153, 153
\msg_info_text:n *152*, 152
\msg_interrupt:nnn *155*, 155
\msg_line_context: *151*, 151
\msg_line_number: *151*, 151
\msg_log:n *156*, 156
\msg_log:nn *153*
\msg_log:nnn *153*
\msg_log:nnnn *153*
\msg_log:nnnnn *153*
\msg_log:nnnnnn *153*, 153
\msg_new:nnn *150*
\msg_new:nnnn *150*, 150
\msg_none:nn *154*
\msg_none:nnn *154*
\msg_none:nnnn *154*
\msg_none:nnnnn *154*
\msg_none:nnnnnn *154*, 154
\msg_redirect_class:nn *154*, 154

\msg_redirect_module:nnn ... *155*, 155
\msg_redirect_name:nnn *155*, 155
\msg_see_documentation_text:n *152*, 152
\msg_set:nnn *150*
\msg_set:nnnn *150*, 150
\msg_term:n *156*, 156
\msg_warning:nn *153*
\msg_warning:nnn *153*
\msg_warning:nnnn *153*
\msg_warning:nnnnn *153*
\msg_warning:nnnnnn *153*, 153
\msg_warning_text:n *151*, 151
msg internal commands:
\c__msg_coding_error_text_tl .. *159*
__msg_expandable_error:n .. *158*, 158
__msg_kernel_error:nn *157*
__msg_kernel_error:nnn *157*
__msg_kernel_error:nnnn *157*
__msg_kernel_error:nnnnn *157*
__msg_kernel_error:nnnnnn . *157*, 157
__msg_kernel_expandable_error:nn *158*
__msg_kernel_expandable_error:nnn *158*
__msg_kernel_expandable_error:nnnn *158*
__msg_kernel_expandable_error:nnnnn *158*
__msg_kernel_expandable_error:nnnnnn *158*, 158
__msg_kernel_fatal:nn *157*
__msg_kernel_fatal:nnn *157*
__msg_kernel_fatal:nnnn *157*
__msg_kernel_fatal:nnnnn *157*
__msg_kernel_fatal:nnnnnn . *157*, 157
__msg_kernel_info:nn *157*
__msg_kernel_info:nnn *157*
__msg_kernel_info:nnnn *157*
__msg_kernel_info:nnnnn *157*
__msg_kernel_info:nnnnnn .. *157*, 157
__msg_kernel_new:nnn *156*
__msg_kernel_new:nnnn *156*, 156
__msg_kernel_set:nnn *156*
__msg_kernel_set:nnnn *156*, 156
__msg_kernel_warning:nn *157*
__msg_kernel_warning:nnn *157*
__msg_kernel_warning:nnnn *157*
__msg_kernel_warning:nnnnn ... *157*
__msg_kernel_warning:nnnnnn *157*, 157
__msg_log_next: ... *158*, 158, 158, 158
__msg_show_item:n *159*, 159, 159
__msg_show_item:nn ... *159*, 159, 159
__msg_show_item_unbraced:nn *159*, 159
__msg_show_pre:nnnnnn *158*, 158
__msg_show_variable:NNNnn 158, *159*, 159, 159
__msg_show_wrap:n 158, 159, *159*, 159, 159
__msg_show_wrap:Nn ... 158, *159*, 159
multichoice commands:
.multichoice: *164*
multichoices commands:
.multichoices:nn *164*
muskip commands:
\c_max_muskip *87*
\muskip_add:Nn *86*, 86
\muskip_const:Nn *86*, 86
\muskip_eval:n *87*, 87, 87
\muskip_gadd:Nn *86*
\muskip_gset:Nn *86*
\muskip_gset_eq:NN *86*
\muskip_gsub:Nn *86*
\muskip_gzero:N *86*
\muskip_gzero_new:N *86*
\muskip_if_exist:NTF *86*, 86
\muskip_if_exist_p:N *86*, 86
\muskip_log:N *207*, 207
\muskip_log:n *208*, 208
\muskip_new:N *85*, 85, 86
\muskip_set:Nn *86*, 86
\muskip_set_eq:NN *86*, 86
\muskip_show:N *87*, 87
\muskip_show:n *87*, 87
\muskip_sub:Nn *86*, 86
\muskip_use:N 87, *87*, 87, 87
\muskip_zero:N *86*
muskip_zero:N 86
\muskip_zero_new:N *86*, 86
\g_tmpa_muskip *88*
\l_tmpa_muskip *87*
\g_tmpb_muskip *88*
\l_tmpb_muskip *87*
\c_zero_muskip *87*

N

nan *195*
nc *195*
nd *195*
\next 59, 59, 59

O

one commands:
\c_minus_one *72*
\c_one_degree_fp *187*, 195
or commands:
\or: *73*, 73, 73
\outer 5, 5

P

\par 10, 10, 11, 11, 11, 12, 12, 12, 13, 13, 13, 14, 14, 14, 174, 174
pc *195*
pdftex commands:
\pdftex_... 8
peek commands:
\peek_after:Nw 40, 56, *56*, 56, 56
\peek_catcode:NTF *56*, 56
\peek_catcode_ignore_spaces:NTF *57*, 57
\peek_catcode_remove:NTF *57*, 57
\peek_catcode_remove_ignore_spaces:NTF *57*, 57
\peek_charcode:NTF *57*, 57
\peek_charcode_ignore_spaces:NTF *57*, 57
\peek_charcode_remove:NTF . . . *57*, 57
\peek_charcode_remove_ignore_spaces:NTF *58*, 58
\peek_gafter:Nw *56*, 56, 56
\peek_meaning:NTF *58*, 58
\peek_meaning_ignore_spaces:NTF *58*, 58
\peek_meaning_remove:NTF *58*, 58
\peek_meaning_remove_ignore_spaces:NTF *58*, 58
\peek_N_type:TF *212*, 212
pi *195*
prg commands:
\prg_break_point:Nn 45
\prg_do_nothing: *9*, 9, 41
\prg_new_conditional:Nnn *34*, 34
\prg_new_conditional:Npnn . *34*, 34, 34
\prg_new_eq_conditional:NNn . *35*, 35
\prg_new_protected_conditional:Nnn *34*, 34
\prg_new_protected_conditional:Npnn *34*, 34
\prg_replicate:nn *39*, 39
\prg_return_false: 35, *36*, 36, 36, 110
\prg_return_true: . 35, *36*, 36, 36, 110
\prg_set_conditional:Nnn *34*
\prg_set_conditional:Npnn . *34*, 35, 36
\prg_set_eq_conditional:NNn *35*
\prg_set_protected_conditional:Nnn *34*
\prg_set_protected_conditional:Npnn *34*
prg internal commands:
__prg_break: *41*
__prg_break:n 41, *41*, 41
__prg_break_point: *41*, 41
__prg_break_point:Nn *40*, 40, 40, 41, 95, 95, 117, 117, 127, 127, 202, 203
__prg_case_end:nw *23*, 23
__prg_compare_error: *74*, 74
__prg_compare_error:Nw *74*, 74
__prg_map_1:w 41
__prg_map_2:w 41
__prg_map_break:Nn *41*, 41
\g__prg_map_int *41*, 41
prop commands:
\c_empty_prop *136*
\prop_clear:N *131*, 131
prop_clear:N 131
\prop_clear_new:N *131*, 131
\prop_count:N *206*, 206
\prop_gclear:N *131*
\prop_gclear_new:N *131*
\prop_get:Nn 41
\prop_get:NnN . . . 42, 43, *132*, 132, 133
\prop_get:NnNTF . . . 132, 133, *134*, 134
\prop_gpop:NnN *132*, 132
\prop_gpop:NnNTF 132, *134*, 134
\prop_gput:Nnn *132*
\prop_gput_if_new:Nnn *132*
\prop_gremove:Nn *133*
\prop_gset_eq:NN *131*
\prop_if_empty:NTF *133*, 133
\prop_if_empty_p:N *133*, 133
\prop_if_exist:NTF *133*, 133
\prop_if_exist_p:N *133*, 133
\prop_if_in:NnTF *133*, 133
\prop_if_in_p:Nn *133*
\prop_item:Nn *133*, 133, 206
\prop_log:N *206*, 206
\prop_map_break: *135*, 135
\prop_map_break:n *135*, 135
\prop_map_function:NN . *134*, 134, 206
\prop_map_inline:Nn *134*, 134
\prop_map_tokens:Nn *206*, 206
\prop_new:N *131*, 131, 131
\prop_pop:NnN *132*, 132
\prop_pop:NnNTF 132, *134*, 134
\prop_put:Nnn *132*, 132, 136
\prop_put_if_new:Nnn *132*, 132
\prop_remove:Nn *133*, 133
\prop_set_eq:NN *131*, 131
\prop_show:N *135*, 135
\g_tmpa_prop *136*
\l_tmpa_prop *135*
\g_tmpb_prop *136*
\l_tmpb_prop *135*
prop internal commands:
\l__prop_internal_tl *136*
__prop_pair:wn 136, *136*, 136

__prop_split:NnTF *136*, 136
\ProvidesExplClass *6*
\ProvidesExplFile *6*
\ProvidesExplPackage *6*, 6
pt *195*

Q

quark commands:
\q_mark 23, 23, *43*, 101, 101
\q_nil 19, 19, *43*, 43, 43, 43
\q_no_value 42, *43*, 43, 43, 43, 113, 114, 114, 114, 114, 114, 119, 119, 119, 129, 132, 132, 132, 172
\quark_if_nil:NTF *43*, 43
\quark_if_nil:nTF *43*, 43
\quark_if_nil_p:N *43*, 43
\quark_if_nil_p:n *43*, 43
\quark_if_no_value:NTF *43*, 43
\quark_if_no_value:nTF *43*, 43
\quark_if_no_value_p:N *43*, 43
\quark_if_no_value_p:n *43*, 43
\quark_if_recursion_tail_stop:N *44*, 44
\quark_if_recursion_tail_stop:n *44*, 44
\quark_if_recursion_tail_stop_do:Nn *44*, 44
\quark_if_recursion_tail_stop_do:nn *44*, 44
\quark_new:N *42*, 42
\q_recursion_stop 19, 19, *44*, 44, 44, 44, 44, 45
\q_recursion_tail *44*, 44, 44, 44, 44, 44, 44, 44, 44, 45, 45, 45
\q_stop 19, 19, 23, 23, 29, 42, *43*, 43, 43, 98
quark internal commands:
\s__prop *136*, 136
__quark_if_recursion_tail_break:NN *45*
__quark_if_recursion_tail_break:nN *45*, 45
\s__seq *122*
\s__stop *46*, 46, 46, 46

R

rand *194*
randint *195*
reverse commands:
\reverse_if:N *21*, 21
right commands:
\c_right_brace_str *110*
round *192*

S

scan commands:
\scan_stop: *9*, 9, 45, 46, 58, 59, 59, 59, 122
scan internal commands:
__scan_new:N *46*, 46
sec *193*
secd *193*
seq commands:
\c_empty_seq *121*
\seq_clear:N *112*, 112, 120
seq_clear:N 112
\seq_clear_new:N *112*, 112
\seq_concat:NNN . . . *113*, 113, 120, 121
\seq_count:N 114, *118*, 118, 120
\seq_gclear:N *112*
\seq_gclear_new:N *112*
\seq_gconcat:NNN *113*
\seq_get:NN *119*, 119
\seq_get:NNTF *119*, 119
\seq_get_left:NN *113*, 113
\seq_get_left:NNTF *115*, 115
\seq_get_right:NN *114*, 114
\seq_get_right:NNTF *115*, 115
\seq_gpop:NN *119*, 119
\seq_gpop:NNTF *119*, 119
\seq_gpop_left:NN *114*, 114
\seq_gpop_left:NNTF *115*, 115
\seq_gpop_right:NN *114*, 114
\seq_gpop_right:NNTF *115*, 115
\seq_gpush:Nn 24, *119*
\seq_gput_left:Nn *113*
\seq_gput_right:Nn *113*
\seq_gremove_all:Nn *116*
\seq_gremove_duplicates:N *116*
\seq_greverse:N *116*
\seq_gset_eq:NN *112*
\seq_gset_filter:NNn *206*
\seq_gset_from_clist:NN *112*
\seq_gset_from_clist:Nn *112*
\seq_gset_map:NNn *207*
\seq_gset_split:Nnn *113*
\seq_if_empty:NTF *116*, 116
\seq_if_empty_p:N *116*, 116
\seq_if_exist:NTF *113*, 113
\seq_if_exist_p:N *113*, 113
\seq_if_in:NnTF *116*, 116, 120, 120, 120, 120
\seq_item:Nn *114*, 114
\seq_log:N *207*, 207
\seq_map_break: . . . *117*, 117, 206, 207
\seq_map_break:n *117*, 117
\seq_map_function:NN *4*, *116*, 116, 116

\seq_map_inline:Nn 116, *116*, 116, 120, 120, 121, 121, 121
\seq_map_variable:NNn *117*, 117
\seq_mapthread_function:NNN *206*, 206
\seq_new:N *4*, *112*, 112, 112
\seq_pop:NN *119*, 119
\seq_pop:NNTF *119*, 119
\seq_pop_left:NN *114*, 114
\seq_pop_left:NNTF *115*, 115
\seq_pop_right:NN *114*, 114
\seq_pop_right:NNTF *115*, 115
\seq_push:Nn *119*, 119
\seq_put_left:Nn *113*, 113
\seq_put_right:Nn *113*, 113, 120, 120, 120
\seq_remove_all:Nn 113, *116*, 116, 120, 120, 121, 121, 121
\seq_remove_duplicates:N *116*, 116, 120, 120
\seq_reverse:N *116*, 116
\seq_set_eq:NN *112*, 112, 120, 121, 121, 121
\seq_set_filter:NNn *206*, 206
\seq_set_from_clist:NN *112*, 112
\seq_set_from_clist:Nn *112*
\seq_set_map:NNn *207*, 207
\seq_set_split:Nnn *113*, 113, 113
\seq_show:N *121*, 121
\seq_use:Nn *118*, 118
\seq_use:Nnnn *118*, 118
\g_tmpa_seq *121*
\l_tmpa_seq *121*
\g_tmpb_seq *121*
\l_tmpb_seq *121*
seq internal commands:
__seq_item:n *122*, 122, 122, 122
__seq_pop_item_def: . . 122, *122*, 122
__seq_push_item_def:n *122*, 122, 122, 122
sin *193*
sind *193*
skip commands:
\c_max_skip *85*
\skip_add:Nn *83*, 83
\skip_const:Nn *83*, 83
\skip_eval:n 84, 84, *84*, 84, 84
\skip_gadd:Nn *83*
.skip_gset:N *164*
\skip_gset:Nn *83*
\skip_gset_eq:NN *83*
\skip_gsub:Nn *83*
\skip_gzero:N *83*
\skip_gzero_new:N *83*
\skip_horizontal:N *85*, 85, 85
\skip_horizontal:n *85*, 85
\skip_if_eq:nnTF *84*
\skip_if_eq_p:nn *84*, 84
\skip_if_exist:NTF *83*, 83
\skip_if_exist_p:N *83*, 83
\skip_if_finite:nTF *84*, 84
\skip_if_finite_p:n *84*, 84
\skip_log:N *207*, 207
\skip_log:n *207*, 207
\skip_new:N *82*, 82, 83
.skip_set:N *164*
\skip_set:Nn *83*, 83
\skip_set_eq:NN *83*, 83
\skip_show:N *84*, 84
\skip_show:n *84*, 84
\skip_split_finite_else_action:nnNN *207*, 207
\skip_sub:Nn *83*, 83
\skip_use:N 84, *84*, 84, 84
\skip_vertical:N *85*, 85, 85
\skip_vertical:n *85*, 85
\skip_zero:N *83*, 83, 86
skip_zero:N 83
\skip_zero_new:N *83*, 83
\g_tmpa_skip *85*
\l_tmpa_skip *85*
\g_tmpb_skip *85*
\l_tmpb_skip *85*
\c_zero_skip *85*
sp *195*
sqrt *194*
sr commands:
\sr_if_empty_p:N 104
str commands:
\c_ampersand_str *110*
\c_atsign_str *110*
\c_backslash_str *110*
\c_circumflex_str *110*
\c_colon_str *110*
\c_dollar_str *110*
\c_hash_str *110*
\c_percent_str *110*
\str_case:nn *104*, 104
\str_case:nnTF *104*, 104
\str_case_x:nn 105
\str_case_x:nnTF *105*, 105
\str_clear:N *103*, 103
str_clear:N 103
\str_clear_new:N *103*, 103
\str_const:Nn *102*, 102
\str_count:N *105*, 105
\str_count:n *105*, 105, 105, 111
\str_count_ignore_spaces:n . *105*, 105
\str_count_spaces:N *105*

\str_count_spaces:n *105*, 105
\str_fold_case:n .....
108, 108, *109*, 109, 109, 109, 109, 208
\str_gclear:N *103*
\str_gput_left:Nn *103*
\str_gput_right:Nn *103*
\str_gset:Nn *103*
\str_gset_eq:NN *103*
\str_head:N *106*, 106
\str_head:n *106*, 106, 106
\str_head_ignore_spaces:n .. *106*, 106
\str_if_empty:NTF *104*, 104
\str_if_empty_p:N *104*
\str_if_eq:nn 131, 136
\str_if_eq:NNTF *104*, 104
\str_if_eq:nnTF .....
..... *104*, 104, 104, 105, 133, 206
\str_if_eq_p:NN *104*, 104
\str_if_eq_p:nn *104*, 104
\str_if_eq_x:nnTF *104*, 104, 110
\str_if_eq_x_p:nn *104*, 104
\str_if_exist:NTF *103*, 103
\str_if_exist_p:N *103*, 103
\str_item:Nn *106*, 106
\str_item:nn *106*, 106, 106
\str_item_ignore_spaces:nn . *106*, 106
\str_lower_case:n *108*, 108, 208
\str_new:N *102*, 102, 103
\str_put_left:Nn *103*, 103
\str_put_right:Nn *103*, 103
\str_range:Nnn *107*
\str_range:nnn *107*, 107, 111
\str_range_ignore_spaces:nnn .. *107*
\str_set:Nn *103*, 103
\str_set_eq:NN *103*, 103
\str_show:N *109*, 109
\str_show:n *109*
\str_tail:N *106*, 106
\str_tail:n *106*, 106, 106
\str_tail_ignore_spaces:n .. *106*, 106
\str_upper_case:n *108*, 108, 208
\str_use:N *105*, 105
\c_tilde_str *110*
\g_tmpa_str *110*
\l_tmpa_str *110*
\g_tmpb_str *110*
\l_tmpb_str *110*
\c_underscore_str *110*
str internal commands:
__str_count:n *111*, 111
__str_if_eq_x:nn *110*, 110
__str_if_eq_x_return:nn ... *110*, 110
__str_range:nnn *111*, 111
__str_to_other:n .. *110*, 110, 111, 111
sys commands:
\c_sys_day_int *213*
\c_sys_engine_str *213*
\c_sys_hour_int *213*
\sys_if_engine_luatex:TF ... *213*, 215
\sys_if_engine_luatex_p: *213*
\sys_if_engine_pdftex:TF ... *213*, 213
\sys_if_engine_pdftex_p: *213*
\sys_if_engine_ptex:TF *213*
\sys_if_engine_ptex_p: *213*
\sys_if_engine_uptex:TF *213*
\sys_if_engine_uptex_p: *213*
\sys_if_engine_xetex:TF *213*
\sys_if_engine_xetex_p: *213*
\sys_if_output_dvi:TF *214*, 214
\sys_if_output_dvi_p: *214*
\sys_if_output_pdf:TF *214*
\sys_if_output_pdf_p: *214*
\c_sys_jobname_str 172, *213*
\c_sys_minute_int *213*
\c_sys_month_int *213*
\c_sys_output_str *214*
\c_sys_year_int *213*

T

tan *193*
tand *193*
TEX and LATEX 2ε commands:
\box 138
\char 61
\copy 138
\count 61
\csname 16
\def 61
\dimexpr 88
\directlua 215
\dp 138
\edef 1
\endcsname 16
\endinput 152
\endlinechar 92, 92
\endtemplate 40
\errorcontextlines 179
\escapechar 96, 96, 96
\expandafter 30, 31
\fi 60
\halign 40
\hskip 85
\ht 138
\ifcase 73
\ifdim 88
\ifeof 178
\iffalse 36
\ifhbox 143

\ifnum 73
\ifodd 73
\iftrue 36
\ifvbox 143
\ifvoid 144
\ifx 21
\jobname 213
\long 61, 61
\luaescapestring 216
\makeatletter 6
\mathchar 61
\meaning 15, 52, 60, 61
\newif 36
\newlinechar 92, 92, 179
\noexpand 30
\number 74
\numexpr 74
\or 73
\outer 61, 61
\pdfrandomseed 194, 195
\pdfsetrandomseed 194, 195
\pdfuniformdeviate 194, 195
\pgfsys@... 218
\protected 61, 61
\ProvidesClass 6
\ProvidesFile 6
\ProvidesPackage 6
\read 174
\readline 175
\RequirePackage 6
\robustify 209
\romannumeral 73
\show 16, 100
\showtokens 100, 158
\string 52
\tenrm 60
\the 60, 64, 80, 84, 87
\topmark 61
\unexpanded 30, 97, 97, 97, 100, 114, 118, 118, 125, 128, 128, 130, 133, 208
\unhbox 141
\unhcopy 141
\unless 21
\unvbox 143
\unvcopy 143
\vbox 142
\vskip 85
\vsplit 143
\vtop 142
\wd 139
\write 176

tex commands:

\tex_... 8
\tex_if:D 47

thousand commands:

\c_one_thousand *72*
\c_ten_thousand *72*

tl commands:

\c_empty_tl *101*
\c_space_tl *101*
\tl_case:Nn *94*, 94
\tl_case:NnTF *94*, 94
\l_tl_case_change_accents_tl . . *210*
\l_tl_case_change_exclude_tl *209*, 209, 210
\l_tl_case_change_math_tl . . *209*, 209
\tl_clear:N *89*, 89
tl_clear:N 90
\tl_clear_new:N *90*, 90
\tl_concat:NNN *90*, 90
\tl_const:Nn *89*, 89
\tl_count:N 93, 96, *97*, 97
\tl_count:n 94, *96*, 96, 97
\tl_count_tokens:n *208*, 208
\tl_gclear:N *89*
\tl_gclear_new:N *90*
\tl_gconcat:NNN *90*
\tl_gput_left:Nn *90*
\tl_gput_right:Nn *90*
\tl_gremove_all:Nn *91*
\tl_gremove_once:Nn *91*
\tl_greplace_all:Nnn *91*
\tl_greplace_once:Nnn *91*
\tl_greverse:N *97*
.tl_gset:N *164*
\tl_gset:Nn *90*, 113
\tl_gset_eq:NN *90*
\tl_gset_from_file:Nnn *211*
\tl_gset_from_file_x:Nnn *211*
\tl_gset_rescan:Nnn *92*
.tl_gset_x:N *164*
\tl_gtrim_spaces:N *97*
\tl_head:N *98*
\tl_head:n *98*, 98, 98, 98
\tl_head:w *98*, 98
\tl_if_blank:nTF . . . *92*, 92, 98, 98, 99
\tl_if_blank_p:n *92*, 92
\tl_if_empty:NTF *93*, 93, 159
\tl_if_empty:nTF *93*, 93
\tl_if_empty_p:N *93*, 93
\tl_if_empty_p:n *93*, 93
\tl_if_eq:nn(TF) 125, 125
\tl_if_eq:NNTF 42, *93*, 93, 94
\tl_if_eq:nnTF *93*, 93, 116, 116
\tl_if_eq_p:NN *93*, 93
\tl_if_exist:NTF *90*, 90
\tl_if_exist_p:N *90*, 90
\tl_if_head_eq_catcode:nNTF . *99*, 99

\tl_if_head_eq_catcode_p:nN . *99*, 99
\tl_if_head_eq_charcode:nNTF *99*, 99
\tl_if_head_eq_charcode_p:nN *99*, 99
\tl_if_head_eq_meaning:nNTF . *99*, 99
\tl_if_head_eq_meaning_p:nN . *99*, 99
\tl_if_head_is_group:nTF *99*, 99
\tl_if_head_is_group_p:n *99*, 99
\tl_if_head_is_N_type:nTF .. *100*, 100
\tl_if_head_is_N_type_p:n .. *100*, 100
\tl_if_head_is_space:nTF ... *100*, 100
\tl_if_head_is_space_p:n ... *100*, 100
\tl_if_in:NnTF *93*, 93
\tl_if_in:nnTF *93*, 93
\tl_if_single:NTF *93*, 93
\tl_if_single:nTF *94*, 94
\tl_if_single_p:N *93*, 93
\tl_if_single_p:n *94*, 94
\tl_if_single_token:nTF *208*, 208
\tl_if_single_token_p:n *208*, 208
\tl_item:Nn *100*
\tl_item:nn *100*, 100
\tl_log:N *211*, 211
\tl_log:n *211*, 211
\tl_lower_case:n *208*
tl_lower_case:n 108
\tl_lower_case:nn *208*
\tl_map_break: *95*, 95
\tl_map_break:n 95, *95*, 95
\tl_map_function:NN ... *94*, 94, 94, 94
\tl_map_function:nN ... 94, *94*, 94, 94
\tl_map_inline:Nn *94*, 94, 95
\tl_map_inline:nn 44, *94*, 94, 95
\tl_map_variable:NNn *95*, 95
\tl_map_variable:nNn *95*, 95
\tl_mixed_case:n *208*
tl_mixed_case:n 108, 210
\tl_mixed_case:nn *208*
\l_tl_mixed_case_ignore_tl *210*
\l_tl_mixed_change_ignore_tl .. 210
\tl_new:N 52, *89*, 89, 90
\tl_put_left:Nn *90*, 90
\tl_put_right:Nn *90*, 90
\tl_remove_all:Nn 91, *91*, 91, 91
\tl_remove_once:Nn *91*, 91
\tl_replace_all:Nnn *91*, 91
\tl_replace_once:Nnn *91*, 91
\tl_rescan:nn 92, *92*, 92, 92, 92
\tl_reverse:N 97, *97*, 97
\tl_reverse:n *97*, 97, 97, 97
\tl_reverse_items:n ... 97, 97, *97*, 97
\tl_reverse_tokens:n .. *208*, 208, 208
.tl_set:N *164*
\tl_set:Nn *90*, 90, 91, 92, 113, 164
\tl_set_eq:NN *90*, 90
\tl_set_from_file:Nnn *211*, 211
\tl_set_from_file_x:Nnn *211*, 211
\tl_set_rescan:Nnn *92*, 92, 92, 92
.tl_set_x:N *164*
\tl_show:N *100*, 100, 211
\tl_show:n *100*, 100, 211
\tl_tail:N *99*
\tl_tail:n *99*, 99, 99
\tl_to_lowercase:n 50
\tl_to_str:N *96*, 96, 102, 177
\tl_to_str:n 92, 92, *96*, 96, 96, 96, 102, 102, 108, 108, 109, 109, 132, 132, 159, 161, 168, 168, 177
\tl_to_uppercase:n 50
\tl_trim_spaces:N *97*, 97
\tl_trim_spaces:n *97*, 97, 101
\tl_upper_case:n *208*, 208
tl_upper_case:n 108
\tl_upper_case:nn *208*, 208
\tl_use:N 62, 80, 84, 87, *96*, 96
\g_tmpa_tl *101*
\l_tmpa_tl *4*, 91, 91, 91, *101*
\g_tmpb_tl *101*
\l_tmpb_tl *101*

tl internal commands:

__tl_trim_spaces:nn *101*, 101

token commands:

\c_alignment_token *52*
\c_parameter_token *52*
\g_peek_token 56, *56*
\l_peek_token 56, *56*
\c_space_token *52*, 100, 101, 212
\token_get_arg_spec:N *59*, 59
\token_get_prefix_spec:N *59*, 59
\token_get_replacement_spec:N *59*, 59
\token_if_active:NTF *54*, 54
\token_if_active_p:N *54*, 54
\token_if_alignment:NTF ... *53*, 53, 53
\token_if_alignment_p:N *53*, 53
\token_if_chardef:NTF *55*, 55
\token_if_chardef_p:N *55*, 55
\token_if_cs:NTF *54*, 54
\token_if_cs_p:N *54*, 54
\token_if_dim_register:NTF .. *55*, 55
\token_if_dim_register_p:N .. *55*, 55
\token_if_eq_catcode:NNTF *54*, 54, 56, 57, 57, 57
\token_if_eq_catcode_p:NN ... *54*, 54
\token_if_eq_charcode:NNTF *54*, 54, 57, 57, 57, 58
\token_if_eq_charcode_p:NN .. *54*, 54
\token_if_eq_meaning:NNTF *54*, 54, 58, 58, 58, 58
\token_if_eq_meaning_p:NN ... *54*, 54

\token_if_expandable:NTF *54*, 54
\token_if_expandable_p:N *54*, 54
\token_if_group_begin:NTF ... *53*, 53
\token_if_group_begin_p:N ... *53*, 53
\token_if_group_end:NTF *53*, 53
\token_if_group_end_p:N *53*, 53
\token_if_int_register:NTF .. *55*, 55
\token_if_int_register_p:N .. *55*, 55
\token_if_letter:NTF *54*, 54
\token_if_letter_p:N *54*, 54
\token_if_long_macro:NTF *54*, 54
\token_if_long_macro_p:N *54*, 54
\token_if_macro:NTF *54*, 54
\token_if_macro_p:N *54*, 54
\token_if_math_subscript:NTF *53*, 53
\token_if_math_subscript_p:N *53*, 53
\token_if_math_superscript:NTF *53*, 53
\token_if_math_superscript_p:N *53*, 53
\token_if_math_toggle:NTF ... *53*, 53
\token_if_math_toggle_p:N ... *53*, 53
\token_if_mathchardef:NTF ... *55*, 55
\token_if_mathchardef_p:N ... *55*, 55
\token_if_muskip_register:NTF *55*, 55
\token_if_muskip_register_p:N *55*, 55
\token_if_other:NTF *54*, 54
\token_if_other_p:N *54*, 54
\token_if_parameter:NTF *53*
\token_if_parameter_p:N *53*, 53
\token_if_primitive:NTF *56*, 56
\token_if_primitive_p:N *56*, 56
\token_if_protected_long_macro:NTF *55*, 55
\token_if_protected_long_macro_p:N *55*, 55
\token_if_protected_macro:NTF *54*, 54
\token_if_protected_macro_p:N *54*, 54
\token_if_skip_register:NTF . *55*, 55
\token_if_skip_register_p:N . *55*, 55
\token_if_space:NTF *53*, 53
\token_if_space_p:N *53*, 53
\token_if_toks_register:NTF . *56*, 56
\token_if_toks_register_p:N . *56*, 56
\token_new:Nn *52*, 52
\token_to_meaning:N *52*, 52
\token_to_str:N *4*, 17, *52*, 52, 52, 102, 159, 177
true *195*
trunc *192*
two commands:
\c_thirty_two *72*
\c_two_hundred_fifty_five *72*
\c_two_hundred_fifty_six *72*

U

undefine commands:
.undefine: *164*
use commands:
\use:N *16*, 16, 16, 16
\use:n *17*, 17, *18*, 18, 59, 89, 209
\use:nn *17*, 17
\use:nnn *17*, 17
\use:nnnn *17*, 17
\use_i:nn *17*, 17, 17
\use_i:nnn *18*, 18, 18
\use_i:nnnn *18*, 18, 18
\use_i_delimit_by_q_nil:nw .. *19*, 19
\use_i_delimit_by_q_recursion_stop:nw *19*, 19
\use_i_delimit_by_q_stop:nw . *19*, 19
\use_i_ii:nnn *18*, 18
\use_ii:nn *17*, 17, 40
\use_ii:nnn *18*, 18
\use_ii:nnnn *18*, 18
\use_iii:nnn *18*, 18
\use_iii:nnnn *18*, 18
\use_iv:nnnn *18*, 18
\use_none:n *18*, 18, 22, 101
\use_none:nn *18*
\use_none:nnn *18*
\use_none:nnnn *18*
\use_none:nnnnn *18*
\use_none:nnnnnn *18*
\use_none:nnnnnnn *18*
\use_none:nnnnnnnn *18*
\use_none:nnnnnnnnn *18*
\use_none_delimit_by_q_nil:w *19*, 19
\use_none_delimit_by_q_recursion_stop:w *19*, 19, 44, 44, 44, 44
\use_none_delimit_by_q_stop:w *19*, 19
use internal commands:
__use_none_delimit_by_s__stop:w 46, *46*, 46

V

value commands:
.value_forbidden:n *164*
.value_required:n *164*
vbox commands:
\vbox:n *142*, 142
\vbox_gset:Nn *142*
\vbox_gset:Nw *143*
\vbox_gset_end: *143*
\vbox_gset_to_ht:Nnn *142*
\vbox_gset_top:Nn *142*
\vbox_set:Nn *142*, 142, 143
\vbox_set:Nw *143*, 143
\vbox_set_end: *143*, 143

\vbox_set_split_to_ht:NNn . . *143*, 143
\vbox_set_to_ht:Nnn *142*, 142
\vbox_set_top:Nn *142*, 142
\vbox_to_ht:nn *142*, 142
\vbox_to_zero:n *142*, 142
\vbox_top:n *142*, 142
\vbox_unpack:N *143*, 143, 143
\vbox_unpack_clear:N *143*
vcoffin commands:
\vcoffin_set:Nnn *146*, 146
\vcoffin_set:Nnw *146*, 146
\vcoffin_set_end: *146*, 146

X

xetex commands:
\xetex_... 8
\xetex_if_engine:TF *4*