The \LaTeX3 Interfaces

The \LaTeX3 Project*

Released 2020-02-14

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 \e\-\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 \LaTeX2\e. In time, a \LaTeX3 format will be produced based on this code. This allows the code to be used in \LaTeX2\e\ 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. Naming functions and variables .......................... 1
   1.1 Terminological inexactitude .......................... 3

2. Documentation conventions .................................. 4

3. Formal language conventions which apply generally ............... 5

4. \TeX{} concepts not supported by \LaTeX{} 3 ..................... 6

### II The l3bootstrap package: Bootstrap code

1. Using the \LaTeX{} modules .................................. 7

### III The l3names package: Namespace for primitives

1. Setting up the \LaTeX{} programming language ................. 8

### IV The l3basics package: Basic definitions

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. Analysing control sequences ................................ 17

5. Using or removing tokens and arguments ....................... 18
   5.1 Selecting tokens from delimited arguments .............. 20

6. Predicates and conditionals ................................ 21
   6.1 Tests on control sequences .............................. 22
   6.2 Primitive conditionals ................................ 22

7. Starting a paragraph ........................................ 24
   7.1 Debugging support ....................................... 24
V  The \texttt{i3expan} package: Argument expansion 25
1  Defining new variants 25
2  Methods for defining variants 26
3  Introducing the variants 27
4  Manipulating the first argument 29
5  Manipulating two arguments 31
6  Manipulating three arguments 32
7  Unbraced expansion 33
8  Preventing expansion 33
9  Controlled expansion 35
10 Internal functions 37

VI  The \texttt{i3tl} package: Token lists 38
1  Creating and initialising token list variables 38
2  Adding data to token list variables 39
3  Modifying token list variables 40
4  Reassigning token list category codes 40
5  Token list conditionals 41
6  Mapping to token lists 43
7  Using token lists 46
8  Working with the content of token lists 46
9  The first token from a token list 48
10 Using a single item 51
11 Viewing token lists 53
12 Constant token lists 53
13 Scratch token lists 54

VII  The \texttt{i3str} package: Strings 55
<table>
<thead>
<tr>
<th></th>
<th>Building strings</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Adding data to string variables</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Modifying string variables</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>String conditionals</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>Mapping to strings</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>Working with the content of strings</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>String manipulation</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>Viewing strings</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>Constant token lists</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>Scratch strings</td>
<td>66</td>
</tr>
</tbody>
</table>

VIII  The l3str-convert package: string encoding conversions  67
<table>
<thead>
<tr>
<th></th>
<th>Encoding and escaping schemes</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Conversion functions</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>Creating 8-bit mappings</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Possibilities, and things to do</td>
<td>69</td>
</tr>
</tbody>
</table>

IX  The l3quark package: Quarks  70
<table>
<thead>
<tr>
<th></th>
<th>Quarks</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Defining quarks</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>Quark tests</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>Recursion</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>An example of recursion with quarks</td>
<td>72</td>
</tr>
<tr>
<td>6</td>
<td>Scan marks</td>
<td>73</td>
</tr>
</tbody>
</table>

X  The l3seq package: Sequences and stacks  75
<table>
<thead>
<tr>
<th></th>
<th>Creating and initialising sequences</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Appending data to sequences</td>
<td>76</td>
</tr>
<tr>
<td>3</td>
<td>Recovering items from sequences</td>
<td>76</td>
</tr>
</tbody>
</table>
4 Recovering values from sequences with branching 78
5 Modifying sequences 79
6 Sequence conditionals 80
7 Mapping to sequences 80
8 Using the content of sequences directly 82
9 Sequences as stacks 83
10 Sequences as sets 84
11 Constant and scratch sequences 85
12 Viewing sequences 86

XI The l3int package: Integers 87
1 Integer expressions 88
2 Creating and initialising integers 89
3 Setting and incrementing integers 90
4 Using integers 91
5 Integer expression conditionals 91
6 Integer expression loops 93
7 Integer step functions 95
8 Formatting integers 96
9 Converting from other formats to integers 97
10 Random integers 98
11 Viewing integers 99
12 Constant integers 99
13 Scratch integers 99
   13.1 Direct number expansion 100
14 Primitive conditionals 100

XII The l3flag package: Expandable flags 102
1 Setting up flags 102
2 Expandable flag commands 103

XIII The \texttt{\LaTeX{}rg} package: Control structures 104
1 Defining a set of conditional functions 104
2 The boolean data type 106
3 Boolean expressions 108
4 Logical loops 110
5 Producing multiple copies 111
6 Detecting \TeX{}’s mode 111
7 Primitive conditionals 112
8 Nestable recursions and mappings 112
8.1 Simple mappings 112
9 Internal programming functions 113

XIV The \texttt{\LaTeX{}ys} package: System/runtime functions 114
1 The name of the job 114
2 Date and time 114
3 Engine 114
4 Output format 115
5 Platform 115
6 Random numbers 115
7 Access to the shell 116
7.1 Loading configuration data 117
7.2 Final settings 117

XV The \texttt{\LaTeX{}clist} package: Comma separated lists 118
1 Creating and initialising comma lists 118
2 Adding data to comma lists 120
3 Modifying comma lists 120
4 Comma list conditionals 122
1 Creating new messages 150
2 Contextual information for messages 151
3 Issuing messages 152
4 Redirecting messages 154

XIX The \texttt{l3file} package: File and I/O operations 156
1 Input–output stream management 156
  1.1 Reading from files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
  1.2 Writing to files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
  1.3 Wrapping lines in output . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
  1.4 Constant input–output streams, and variables . . . . . . . . . . . . . . 163
  1.5 Primitive conditionals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
2 File operation functions 163

XX The \texttt{l3skip} package: Dimensions and skips 168
1 Creating and initialising \texttt{dim} variables 168
2 Setting \texttt{dim} variables 169
3 Utilities for dimension calculations 169
4 Dimension expression conditionals 170
5 Dimension expression loops 172
6 Dimension step functions 173
7 Using \texttt{dim} expressions and variables 174
8 Viewing \texttt{dim} variables 175
9 Constant dimensions 176
10 Scratch dimensions 176
11 Creating and initialising \texttt{skip} variables 176
12 Setting \texttt{skip} variables 177
13 Skip expression conditionals 178
14 Using \texttt{skip} expressions and variables 178
15 Viewing \texttt{skip} variables 178
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Constant skips</td>
<td>179</td>
</tr>
<tr>
<td>17</td>
<td>Scratch skips</td>
<td>179</td>
</tr>
<tr>
<td>18</td>
<td>Inserting skips into the output</td>
<td>179</td>
</tr>
<tr>
<td>19</td>
<td>Creating and initialising muskip variables</td>
<td>180</td>
</tr>
<tr>
<td>20</td>
<td>Setting muskip variables</td>
<td>180</td>
</tr>
<tr>
<td>21</td>
<td>Using muskip expressions and variables</td>
<td>181</td>
</tr>
<tr>
<td>22</td>
<td>Viewing muskip variables</td>
<td>181</td>
</tr>
<tr>
<td>23</td>
<td>Constant muskips</td>
<td>182</td>
</tr>
<tr>
<td>24</td>
<td>Scratch muskips</td>
<td>182</td>
</tr>
<tr>
<td>25</td>
<td>Primitive conditional</td>
<td>182</td>
</tr>
</tbody>
</table>

XXI The l3keys package: Key–value interfaces

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Creating keys</td>
<td>184</td>
</tr>
<tr>
<td>2</td>
<td>Sub-dividing keys</td>
<td>188</td>
</tr>
<tr>
<td>3</td>
<td>Choice and multiple choice keys</td>
<td>189</td>
</tr>
<tr>
<td>4</td>
<td>Setting keys</td>
<td>191</td>
</tr>
<tr>
<td>5</td>
<td>Handling of unknown keys</td>
<td>191</td>
</tr>
<tr>
<td>6</td>
<td>Selective key setting</td>
<td>192</td>
</tr>
<tr>
<td>7</td>
<td>Utility functions for keys</td>
<td>193</td>
</tr>
<tr>
<td>8</td>
<td>Low-level interface for parsing key–val lists</td>
<td>194</td>
</tr>
</tbody>
</table>

XXII The l3intarray package: fast global integer arrays

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>l3intarray documentation</td>
<td>196</td>
</tr>
<tr>
<td>1.1</td>
<td>Implementation notes</td>
<td>197</td>
</tr>
</tbody>
</table>

XXIII The l3fp package: Floating points

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Creating and initialising floating point variables</td>
<td>199</td>
</tr>
<tr>
<td>2</td>
<td>Setting floating point variables</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>Using floating points</td>
<td>200</td>
</tr>
</tbody>
</table>
1 Creating and initialising boxes 235
2 Using boxes 235
3 Measuring and setting box dimensions 236
4 Box conditionals 237
5 The last box inserted 237
6 Constant boxes 237
7 Scratch boxes 238
8 Viewing box contents 238
9 Boxes and color 238
10 Horizontal mode boxes 238
11 Vertical mode boxes 240
12 Using boxes efficiently 241
13 Affine transformations 242
14 Primitive box conditionals 245

XXIX The l3coffins package: Coffin code layer 246
1 Creating and initialising coffins 246
2 Setting coffin content and poles 246
3 Coffin affine transformations 248
4 Joining and using coffins 248
5 Measuring coffins 249
6 Coffin diagnostics 249
7 Constants and variables 250

XXX The l3color-base package: Color support 251
1 Color in boxes 251

XXXI The l3luatex package: LuaTEX-specific functions 252
1 Breaking out to Lua 252
2 Lua interfaces 253

XXXII The l3unicode package: Unicode support functions 254

XXXIII The l3text package: text processing 255

1 l3text documentation 255
   1.1 Expanding text .............................................. 255
   1.2 Case changing .............................................. 257
   1.3 Control variables ........................................... 258

XXXIV The l3legacy package: Interfaces to legacy concepts 259

XXXV The l3candidates package: Experimental additions to l3kernel 260

1 Important notice 260

2 Additions to l3box 260
   2.1 Viewing part of a box ....................................... 260

3 Additions to l3exp

4 Additions to l3fp 261

5 Additions to l3file 261

6 Additions to l3flag 262

7 Additions to l3intarray 262
   7.1 Working with contents of integer arrays ................. 262

8 Additions to l3msg 263

9 Additions to l3prg 264

10 Additions to l3prop 265

11 Additions to l3seq 265

12 Additions to l3sys 267

13 Additions to l3tl 268

14 Additions to l3token 269

Index 271
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 L\TeX3 programming language is found in expl3.pdf.

1 Naming functions and variables

L\TeX3 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:

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 not expandable.

e The e specifier is in many respects identical to x, but with a very different implementation. Functions which feature an e-type argument may be expandable. The drawback is that e is extremely slow (often more than 200 times slower) in older engines, more precisely in non-Lua\TeX engines older than 2019.
The \texttt{f} specifier stands for \textit{full expansion}, and in contrast to \texttt{x} stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a \texttt{⟨space token⟩}, it is gobbled, and thus won’t be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

\begin{verbatim}
\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 }
\end{verbatim}

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

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

\textbf{p} The letter \texttt{p} indicates \TeX\ parameters. Normally this will be used for delimited functions as \texttt{expl3} provides better methods for creating simple sequential arguments.

\textbf{w} Finally, there is the \texttt{w} specifier for \textit{weird} arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).

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

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

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

\textbf{c} Constant: global parameters whose value should not be changed.

\textbf{g} Parameters whose value should only be set globally.

\textbf{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\textsuperscript{1} name and then a descriptive part. Variables end with a short identifier to show the variable type:

\textbf{clist} Comma separated list.

\textbf{dim} “Rigid” lengths.

\textbf{fp} Floating-point values;

\textbf{int} Integer-valued count register.

\footnote{The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the \texttt{int} module contains some scratch variables called \texttt{\l_tmpa_int}, \texttt{\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 \texttt{\l_int_tmpa_int} would be very unreadable.}
muskip “Rubber” lengths for use in mathematics.

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

skip “Rubber” lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

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

flag Integer that can be incremented expandably.

fparray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

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 almost the same.\(^2\) 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.

\(^2\)\TeX\nically, functions with no arguments are \texttt{\textbackslash long} while token list variables are not.
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

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

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 or e-type argument (in plain TEX terms, inside an \edef or \expanded), 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:
The underlining and italic of \texttt{TF} indicates that three functions are available:

- \texttt{sys_if_engine_xetex:T}
- \texttt{sys_if_engine_xetex:F}
- \texttt{sys_if_engine_xetex:TF}

Usually, the illustration will use the \texttt{TF} variant, and so both \texttt{(true code)} and \texttt{(false code)} will be shown. The two variant forms \texttt{T} and \texttt{F} take only \texttt{(true code)} and \texttt{(false code)}, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, \textit{all} conditional functions in the \texttt{expl3} modules should be defined in this way.

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

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 \LaTeX2ε or plain \TeX. In these cases, the text will include an extra \texttt{TEXhackers note} section:

The normal description text.

\texttt{TEXhackers note}: Detail for the experienced \TeX or \LaTeX2ε programmer. In this case, it would point out that this function is the \TeX primitive \texttt{\string}.

Changes to behaviour When new functions are added to \texttt{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 \texttt{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 \texttt{TF} argument specification, the test if evaluated to give a logically \texttt{TRUE} or \texttt{FALSE} result. Depending on this result, either the \texttt{(true code)} or the \texttt{(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 \text{\TeX} concepts not supported by \LaTeX3

The \TeX\ concept of an "\texttt{\textbackslash outer}" macro is \textit{not supported} at all by \LaTeXe. As such, the functions provided here may break when used on top of \LaTeXe\ if \texttt{\textbackslash outer} tokens are used in the arguments.
Part II

The \texttt{l3bootstrap} package

Bootstrap code

1 Using the \LaTeX3 modules

The modules documented in \texttt{source3} are designed to be used on top of \LaTeX2ε and are loaded all as one with the usual \texttt{\usepackage{expl3}} or \texttt{\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 \LaTeX2ε it provides a few functions for setting it up.

\begin{verbatim}
\ExplSyntaxOn
\ExplSyntaxOff
\end{verbatim}

The \texttt{\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, \texttt{-} is used to input a space. The \texttt{\ExplSyntaxOff} reverts to the document category code régime.

\texttt{\RequiresPackage{expl3}}

\texttt{\ProvidesExplPackage{⟨package⟩}{⟨date⟩}{⟨version⟩}{⟨description⟩}}

These functions act broadly in the same way as the corresponding \LaTeX2ε kernel functions \texttt{\ProvidesPackage}, \texttt{\ProvidesClass} and \texttt{\ProvidesFile}. However, they also implicitly switch \texttt{\ExplSyntaxOn} for the remainder of the code with the file. At the end of the file, \texttt{\ExplSyntaxOff} will be called to reverse this. (This is the same concept as \LaTeX2ε provides in turning on \texttt{\makeatletter} within package and class code.) The \texttt{⟨date⟩} should be given in the format \texttt{⟨year⟩/⟨month⟩/⟨day⟩}. If the \texttt{⟨version⟩} is given then it will be prefixed with \texttt{v} in the package identifier line.

\texttt{\GetIdInfo}$\verb+\$Id:\langle SVN info field \rangle+\$ {⟨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 \texttt{\ExplFileName} for the part of the file name leading up to the period, \texttt{\ExplFileDate} for date, \texttt{\ExplFileVersion} for version and \texttt{\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 \texttt{\RequiresPackage} or similar are loaded with usual \LaTeX2ε category codes and the \LaTeX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the \texttt{\GetIdInfo} command you can use the information when loading a package with

\begin{verbatim}
\ProvidesExplPackage{\ExplFileName}
{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\end{verbatim}
Part III
The \texttt{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: \emph{The TeXbook}, \emph{\TeX\ by Topic} and the manuals for pdf\TeX, Xe\TeX, \texttt{luatex}, \texttt{mte}X and \texttt{uplatex} should be consulted for details of the primitives. These are named \texttt{\textbackslash tex\_⟨name⟩}; typically based on the primitive’s ⟨name⟩ in pdf\TeX and omitting a leading \texttt{pdf} when the primitive is not related to pdf output.
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

\preamble{\texttt{\textbackslash prg\_do\_nothing}: \ast }\prg_do_nothing:
An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

\preamble{\texttt{\textbackslash scan\_stop}: \textbackslash scan\_stop:}
A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

2 Grouping material

\preamble{\texttt{\textbackslash group\_begin}: \textbackslash group\_begin:}
\preamble{\texttt{\textbackslash group\_end}: \textbackslash 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 \texttt{\textbackslash group\_begin:} must be matched by a \texttt{\textbackslash 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.

\preamble{\texttt{\textbackslash group\_insert\_after:}\textsubscript{\texttt{\textbackslash N}} (\texttt{\textbackslash token})}
\texttt{\textbackslash group\_insert\_after:}\textsubscript{\texttt{\textbackslash N}} \texttt{(\texttt{\textbackslash token})}
Adds \texttt{(token)} to the list of \texttt{(tokens)} to be inserted when the current group level ends. The list of \texttt{(tokens)} to be inserted is empty at the beginning of a group: multiple applications of \texttt{\textbackslash group\_insert\_after:}\textsubscript{\texttt{\textbackslash N}} may be used to build the inserted list one \texttt{(token)} at a time. The current group level may be closed by a \texttt{\textbackslash group\_end:} function or by a token with category code 2 (close-group), namely a \texttt{\textbackslash} 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” are 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 is 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 results 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 does 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 does 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 or e-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

<table>
<thead>
<tr>
<th>\texttt{\textbackslash cs_new\textbackslash Npn}</th>
<th>\texttt{\textbackslash cs_new\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_new\textbackslash cpn}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash cs_new\textbackslash Npn (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_new\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_new\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
</tr>
<tr>
<td>Creates \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the \texttt{function} is already defined.</td>
<td>Creates \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. When the \texttt{function} is used the \texttt{parameters} absorbed cannot contain \texttt{par} tokens. The definition is global and an error results if the \texttt{function} is already defined.</td>
<td>Creates \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The \texttt{function} will not expand within an \texttt{x}-type argument. The definition is global and an error results if the \texttt{function} is already defined.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash Npn}</th>
<th>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash cpn}</th>
<th>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash cpn}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash Npn (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_new\textbackslash nopar\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
</tr>
<tr>
<td>Creates \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The \texttt{function} will not expand within an \texttt{x}-type argument or \texttt{e}-type argument. The definition is global and an error results if the \texttt{function} is already defined.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{\textbackslash cs_set\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_set\textbackslash cpn}</th>
<th>\texttt{\textbackslash cs_set\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_set\textbackslash cpn}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash cs_set\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
</tr>
<tr>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash cpn}</th>
<th>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash Npx}</th>
<th>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash cpn}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash Npx (function) \langle parameters \rangle {\langle code \rangle}}</td>
<td>\texttt{\textbackslash cs_set\textbackslash protected\textbackslash cpn (function) \langle parameters \rangle {\langle code \rangle}}</td>
</tr>
<tr>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
<td>Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.</td>
</tr>
</tbody>
</table>
3.3 Defining new functions using the signature

\texttt{\textbackslash csnew:Nn (function) \{\textbackslash code\}}

Creates \texttt{(function)} to expand to \texttt{\{code\}} as replacement text. Within the \texttt{\{code\}}, the number of \texttt{(parameters)} is detected automatically from the function signature. These \texttt{(parameters)} \texttt{(#1, #2, etc.)} will be replaced by those absorbed by the function. The definition is global and an error results if the \texttt{(function)} is already defined.
\texttt{\textbackslash cs\_new\_nopar:Nn} \hspace{1cm} \texttt{\textbackslash cs\_new\_nopar:(cn Nx|cx)}

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

\texttt{\textbackslash cs\_new\_protected:Nn} \hspace{1cm} \texttt{\textbackslash cs\_new\_protected:(cn Nx|cx)}

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

\texttt{\textbackslash cs\_new\_protected\_nopar:Nn} \hspace{1cm} \texttt{\textbackslash cs\_new\_protected\_nopar:(cn Nx|cx)}

Creates \emph{(function)} to expand to \emph{(code)} as replacement text. Within the \emph{(code)}, the number of \emph{(parameters)} is detected automatically from the function signature. These \emph{(parameters)} \((#1, #2, \text{etc.})\) will be replaced by those absorbed by the function. The \emph{(function)} will not expand within an \texttt{x}-type or \texttt{e}-type argument. The definition is global and an error results if the \emph{(function)} is already defined.

\texttt{\textbackslash cs\_set:Nn} \hspace{1cm} \texttt{\textbackslash cs\_set:(cn Nx|cx)}

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

\texttt{\textbackslash cs\_set\_nopar:Nn} \hspace{1cm} \texttt{\textbackslash cs\_set\_nopar:(cn Nx|cx)}

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

\texttt{\textbackslash cs\_set\_protected:Nn} \hspace{1cm} \texttt{\textbackslash cs\_set\_protected:(cn Nx|cx)}

Sets \emph{(function)} to expand to \emph{(code)} as replacement text. Within the \emph{(code)}, the number of \emph{(parameters)} is detected automatically from the function signature. These \emph{(parameters)} \((#1, #2, \text{etc.})\) will be replaced by those absorbed by the function. The \emph{(function)} will not expand within an \texttt{x}-type argument. The assignment of a meaning to the \emph{(function)} is restricted to the current \TeX{} group level.
\cs_gset_protected_nopar:Nn \cs_gset_protected_nopar:(cn|Nx|cx) \cs_gset_protected_nopar:Nn \cs_gset_protected_nopar:(cn|Nx|cx)

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

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

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

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

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

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

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

\cs_generate_from_arg_count:NNN \cs_generate_from_arg_count:NNN \cs_generate_from_arg_count:NNnn

Uses the \texttt{\texttt{creator}} function (which should have signature \texttt{\texttt{Npn}}, for example \texttt{\texttt{cs_new:Npn}}) to define a \texttt{\texttt{function}} which takes \texttt{\texttt{number}} arguments and has \texttt{\texttt{code}} as replacement text. The \texttt{\texttt{number}} of arguments is an integer expression, evaluated as detailed for \texttt{\texttt{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\langle cs_1\rangle \langle cs_2\rangle
\cs_new_eq:NN\langle cs_1\rangle \langle token\rangle

Globally creates \langle control sequence_1\rangle and sets it to have the same meaning as \langle control sequence_2\rangle or \langle token\rangle. The second control sequence may subsequently be altered without affecting the copy.

\cs_set_eq:NN\langle cs_1\rangle \langle cs_2\rangle
\cs_set_eq:NN\langle cs_1\rangle \langle token\rangle

Sets \langle control sequence_1\rangle to have the same meaning as \langle control sequence_2\rangle (or \langle token\rangle). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \langle control sequence_1\rangle is restricted to the current \TeX group level.

\cs_gset_eq:NN\langle cs_1\rangle \langle cs_2\rangle
\cs_gset_eq:NN\langle cs_1\rangle \langle token\rangle

Globally sets \langle control sequence_1\rangle to have the same meaning as \langle control sequence_2\rangle (or \langle token\rangle). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \langle control sequence_1\rangle 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\langle control sequence\rangle
\cs_undefine:c\langle control sequence\rangle

Sets \langle control sequence\rangle to be globally undefined.

### 3.6 Showing control sequences

\cs_meaning:N \langle control sequence\rangle
\cs_meaning:c \langle control sequence\rangle

This function expands to the *meaning* of the \langle control sequence\rangle control sequence. For a macro, this includes the \langle replacement text\rangle.

\textbf{\TeX hackers note}: This is \TeX’s \texttt{\textbackslash meaning} primitive. For tokens that are not control sequences, it is more logical to use \texttt{\textbackslash token_to_meaning:N}. The \texttt{c} variant correctly reports undefined arguments.
\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.

\cs_log:N (control sequence)
\cs_log:c
Write the definition of the (control sequence) in the log file. See also `\cs_show:N` which displays the result in the terminal.

New: 2014-08-22
Updated: 2017-02-14

3.7 Converting to and from control sequences

\use:c \{⟨control sequence name⟩\}
Expands the (control sequence name) until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other c-type arguments the (control sequence name) must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

**\TeXhackers note:** Protected macros that appear in a c-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

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
Tests whether the (control sequence) is currently defined according to the conditional `\cs_if_exist:NTF` (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.

New: 2012-11-10
\cs:w * \cs:w \langle control sequence name \rangle \cs_end:
\cs_end: *
Converts the given \langle control sequence name \rangle into a single control sequence token. This process requires one expansion. The content for \langle control sequence name \rangle may be literal material or from other expandable functions. The \langle control sequence name \rangle must, when fully expanded, consist of character tokens which are not active: typically 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 \langle control sequence \rangle

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

4 Analysing control sequences

\cs_split_function:N * \cs_split_function:N \langle function \rangle

Splits the \langle function \rangle into the \langle name \rangle (i.e. the part before the colon) and the \langle signature \rangle (i.e. after the colon). This information is then placed in the input stream in three parts: the \langle name \rangle, the \langle signature \rangle and a logic token indicating if a colon was found (to differentiate variables from function names). The \langle name \rangle does not include the escape character, and both the \langle name \rangle and \langle signature \rangle are made up of tokens with category code 12 (other).

The next three functions decompose \TeX macros into their constituent parts: if the \langle token \rangle passed is not a macro then no decomposition can occur. In the latter case, all three functions leave \scan_stop: in the input stream.
\texttt{\cs_prefix_spec:N} \langle \texttt{token} \rangle

If the \langle \texttt{token} \rangle is a macro, this function leaves the applicable \TeX{} prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

\begin{verbatim}
\cs_set:Npn \next:nn #1#2 \{ x #1-y #2 \}
\cs_prefix_spec:N \next:nn
\end{verbatim}

leaves \texttt{\long} in the input stream. If the \langle \texttt{token} \rangle is not a macro then \texttt{\scan_stop:} is left in the input stream.

\textbf{\TeX{}hackers note:} The prefix can be empty, \texttt{\long}, \texttt{\protected} or \texttt{\protected\long} with backslash replaced by the current escape character.

\texttt{\cs_argument_spec:N} \langle \texttt{token} \rangle

If the \langle \texttt{token} \rangle is a macro, this function leaves the primitive \TeX{} argument specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

\begin{verbatim}
\cs_set:Npn \next:nn #1#2 \{ x #1 y #2 \}
\cs_argument_spec:N \next:nn
\end{verbatim}

leaves \#1\#2 in the input stream. If the \langle \texttt{token} \rangle is not a macro then \texttt{\scan_stop:} is left in the input stream.

\textbf{\TeX{}hackers note:} If the argument specification contains the string \texttt{->}, then the function produces incorrect results.

\texttt{\cs_replacement_spec:N} \langle \texttt{token} \rangle

If the \langle \texttt{token} \rangle is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

\begin{verbatim}
\cs_set:Npn \next:nn #1#2 \{ x #1-y #2 \}
\cs_replacement_spec:N \next:nn
\end{verbatim}

leaves \texttt{x\#1_\texttt{uni2423}y\#2} in the input stream. If the \langle \texttt{token} \rangle is not a macro then \texttt{\scan_stop:} is left in the input stream.

\textbf{\TeX{}hackers note:} If the argument specification contains the string \texttt{->}, then the function produces incorrect results.

5 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 when absorbing them the outer set is 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 absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also 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 \} \}

results in the input stream containing

abc \{ def \}

i.e. only the outer braces are removed.

**\TeXhackers note:** The \use:n function is equivalent to \LaTeX's \@firstofone.

\use_i:nn * \use_i:nn \{(arg_1)\} \{(arg_2)\}
\use_ii:nn * \use_ii:nn \{(arg_1)\}

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 is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

**\TeXhackers note:** These are equivalent to \LaTeX's \@firstoftwo and \@secondoftwo.

\use_i:nnn * \use_i:nnn \{(arg_1)\} \{(arg_2)\} \{(arg_3)\}
\use_ii:nnn * \use_ii:nnn \{(arg_2)\} \{(arg_3)\}
\use_iii:nnn * \use_iii:nnn \{(arg_3)\}

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 is also 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_i:nnnn \{(arg_1)\} \{(arg_2)\} \{(arg_3)\} \{(arg_4)\}
\use_ii:nnnn * \use_ii:nnnn \{(arg_2)\} \{(arg_3)\} \{(arg_4)\}
\use_iii:nnnn * \use_iii:nnnn \{(arg_3)\} \{(arg_4)\}
\use_iv:nnnn * \use_iv:nnnn \{(arg_4)\}

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 is also 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 \{\langle \text{arg}_1 \rangle \} \{\langle \text{arg}_2 \rangle \} \{\langle \text{arg}_3 \rangle \}

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

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

results in the input stream containing

\begin{verbatim}
abc \{ def \}
\end{verbatim}

\textit{i.e.} the outer braces are removed and the third group is removed.

\use_ii_i:nn \{\langle \text{arg}_1 \rangle \} \{\langle \text{arg}_2 \rangle \}

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

\use_none:n \{\langle \text{group}_1 \rangle \}

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

\textbf{\LaTeXhackers note:} These are equivalent to \LaTeX\'s \texttt{\@gobble}, \texttt{\@gobbletwo}, etc.

\use:e \{\langle \text{expandable tokens} \rangle \}

Fully expands the \( \langle \text{token list} \rangle \) in an \( x\)-type manner, \textit{but} the function remains fully expandable, and parameter character (usually \#) need not be doubled.

\textbf{\LaTeXhackers note:} \texttt{\use:e} is a wrapper around the primitive \texttt{\expanded} where it is available: it requires two expansions to complete its action. When \texttt{\expanded} is not available this function is very slow.

\use:x \{\langle \text{expandable tokens} \rangle \}

Fully expands the \( \langle \text{expandable tokens} \rangle \) and inserts the result into the input stream at the current location. Any hash characters (\#) in the argument must be doubled.

5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.
Absorb the ⟨balanced text⟩ form the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

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.

6 Predicates and conditionals

\textbf{B\TeX 3} has three concepts for conditional flow processing:

\textbf{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 \texttt{T} and \texttt{F}, respectively. An example would be

\begin{verbatim}
cs_if_free:cTF {abc} {⟨true code⟩} {⟨false code⟩}
\end{verbatim}

a function that turns the first argument into a control sequence (since it’s marked as \texttt{c}) then checks whether this control sequence is still free and then depending on the result carries 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 \texttt{TF} function is defined it is usually accompanied by \texttt{T} and \texttt{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 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 are accompanied by a “predicate” for the same test as described below.

\textbf{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 \texttt{p} in the description part. For example,

\begin{verbatim}
cs_if_free_p:N
\end{verbatim}

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 \texttt{N}) is still free for definition. It would be used in constructions like
For each predicate defined, a “branching conditional” also exists 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\epsilon. Their use is discouraged in \texttt{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.

\begin{verbatim}
\c_true_bool
\c_false_bool
\end{verbatim}

Constants that represent \texttt{true} and \texttt{false}, respectively. Used to implement predicates.

### 6.1 Tests on control sequences

\begin{verbatim}
\cs_if_eq_p:NN \cs_if_eq_p:NNN \cs_if_eq:NNTF \cs_if_eq:NNTF \cs_if_eq:NNTT \cs_if_eq:NNTTT
\end{verbatim}

Compares the definition of two \texttt{control sequences} and is logically \texttt{true} if they are the same, \textit{i.e.} if they have exactly the same definition when examined with \texttt{\cs_show:N}.

\begin{verbatim}
\cs_if_exist_p:N \cs_if_exist_p:c \cs_if_exist:N \cs_if_exist:c
\end{verbatim}

Tests whether the \texttt{control sequence} is currently defined (whether as a function or another control sequence type). Any definition of \texttt{control sequence} other than \texttt{\relax} evaluates as \texttt{true}.

\begin{verbatim}
\cs_if_free_p:N \cs_if_free_p:c \cs_if_free:N \cs_if_free:c
\end{verbatim}

Tests whether the \texttt{control sequence} is currently free to be defined. This test is \texttt{false} if the \texttt{control sequence} currently exists (as defined by \texttt{\cs_if_exist:N}).

### 6.2 Primitive conditionals

The \texttt{\\epsilon-\TeX} engine itself provides many different conditionals. Some expand whatever comes after them and others don’t. Hence the names for these underlying functions often contains a \texttt{\\w} part but higher level functions are often available. See for instance \texttt{\int_compare_p:nNn} which is a wrapper for \texttt{\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 prefix primitive conditionals with \texttt{\if_}.
\if_true: \else: \fi: always executes \textit{true code}, while \if_false: always executes \textit{false code}. \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \texttt{or}: is documented in \texttt{l3int} and used in case switches.

\TeXhackersnote: These are equivalent to their corresponding \TeX primitive conditionals; \reverse_if:N is \TeX’s \texttt{unless}.

\if_meaning:w \else: \fi: \if_meaning:w \textit{true code} when \texttt{arg1} and \texttt{arg2} are the same, otherwise it executes \textit{false code}. \texttt{arg1} and \texttt{arg2} could be functions, variables, tokens; in all cases the \texttt{unexpanded} definitions are compared.

\TeXhackersnote: This is \TeX’s \texttt{ifx}.

\if:w \else: \fi: \if:w \textit{true code} if currently in horizontal mode, otherwise execute \textit{false code}. Similar for the other functions.
7 Starting a paragraph

\texttt{\mode\leave\vertical:}

Ensures that \TeX{} is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width \texttt{\parindent}, followed by the \texttt{\everypar} token list.

\textbf{\TeX{}hackers note:} This results in the contents of the \texttt{\everypar} token register being inserted, after \texttt{\mode\leave\vertical:} is complete. Notice that in contrast to the \LaTeX{}2ε \texttt{\leavevmode} approach, no box is used by the method implemented here.

7.1 Debugging support

\texttt{\debug\on:n} \texttt{\debug\off:n}

\texttt{\debug\on:n \{ (comma-separated list) \}}
\texttt{\debug\off:n \{ (comma-separated list) \}}

Turn on and off within a group various debugging code, some of which is also available as expl3 load-time options. The items that can be used in the \langle list \rangle are

- \texttt{check-declarations} that checks all expl3 variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- \texttt{check-expressions} that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- \texttt{deprecation} that makes soon-to-be-deprecated commands produce errors;
- \texttt{log-functions} that logs function definitions;
- \texttt{all} that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call \texttt{\debug\on:n}, and load the code that one is interested in testing. These functions can only be used in \LaTeX{}2ε package mode loaded with \texttt{\enable\-\debug} or another option implying it.

\texttt{\debug\suspend: ... \debug\resume:}

Suppress (locally) errors and logging from \texttt{debug} commands, except for the \texttt{deprecation} errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance \texttt{l3coffins}.

24
Part V
The \texttt{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 \texttt{exp}.

Not all possible variations are implemented for every base function. Instead only those that are used within the \LaTeX 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 of the form \texttt{\exp_{-}...}. They all look alike, an example would be \texttt{\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 \texttt{\exp_args:NNo} expands the content of third argument once before any expansion of the first and second arguments. If \texttt{\seq_gpush:No} was not defined it could be coded in the following way:

\begin{verbatim}
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  \{ \l_tmpa_tl \}
\end{verbatim}

In other words, the first argument to \texttt{\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 \texttt{\exp_{}} function followed by the desired base function, \textit{e.g.}

\begin{verbatim}
\cs_generate_variant:Nn \seq_gpush:Nn { No }
\end{verbatim}

results in the definition of \texttt{\seq_gpush:No}

\begin{verbatim}
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
\end{verbatim}

Providing variants in this way in style files is safe as the \texttt{\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 \texttt{\cs_generate_variant:Nn}, described next.
2 Methods for defining variants

We recall the set of available argument specifiers.

- \( N \) is used for single-token arguments while \( c \) constructs a control sequence from its name and passes it to a parent function as an \( N \)-type argument.

- Many argument types extract or expand some tokens and provide it as an \( n \)-type argument, namely a braced multiple-token argument: \( V \) extracts the value of a variable, \( v \) extracts the value from the name of a variable, \( n \) uses the argument as it is, \( o \) expands once, \( f \) expands fully the front of the token list, \( e \) and \( x \) expand fully all tokens (differences are explained later).

- A few odd argument types remain: \( T \) and \( F \) for conditional processing, otherwise identical to \( n \)-type arguments, \( p \) for the parameter text in definitions, \( w \) for arguments with a specific syntax, and \( D \) to denote primitives that should not be used directly.
This function is used to define argument-specifier variants of the (parent control sequence) for \TeX\textsuperscript{3} 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) if these are not already defined. For each (variant) given, a function is created that expands its arguments as detailed and passes them to the (parent control sequence). So for example

\begin{verbatim}
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
\end{verbatim}

creates a new function \texttt{\foo:cn} which expands its first argument into a control sequence name and passes the result to \texttt{\foo:Nn}. Similarly

\begin{verbatim}
\cs_generate_variant:Nn \foo:Nn \foo:Nn { NV , cV }
\end{verbatim}

generates the functions \texttt{\foo:NV} and \texttt{\foo:cV} in the same way. The \texttt{\cs_generate_variant:Nn} function can only be applied if the (parent control sequence) is already defined. If the (parent control sequence) is protected or if the (variant) involves any x argument, then the (variant control sequence) is also protected. The (variant) is created globally, as is any \texttt{\exp_args:N(variant)} function needed to carry out the expansion.

Only n and N arguments can be changed to other types. The only allowed changes are

- c variant of an N parent;
- o, V, v, f, e, or x variant of an n parent;
- N, n, T, F, or p argument unchanged.

This means the (parent) of a (variant) form is always unambiguous, even in cases where both an n-type parent and an N-type parent exist, such as for \texttt{\tl_count:n} and \texttt{\tl_count:N}.

For backward compatibility it is currently possible to make n, o, V, v, f, e, or x-type variants of an N-type argument or N or c-type variants of an n-type argument. Both are deprecated. The first because passing more than one token to an N-type argument will typically break the parent function’s code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a V-type or v-type variant instead of c-type. In those cases, using the lower-level \texttt{\exp_args:No} or \texttt{\exp_args:Nc} functions explicitly is preferred to defining confusing variants.

### 3 Introducing the variants

The V type returns the value of a register, which can be one of \texttt{tl}, \texttt{clist}, \texttt{int}, \texttt{skip}, \texttt{dim}, \texttt{muskip}, or built-in \TeX\textsuperscript{X} 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 e type expands all tokens fully, starting from the first. More precisely the
expansion is identical to that of \TeX{}\'s \texttt{message} (in particular # needs not be doubled).
It was added in May 2018. In recent enough engines (starting around 2019) it relies
on the primitive \texttt{expanded} hence is fast. In older engines it is very much slower. As
a result it should only be used in performance critical code if typical users will have a
recent installation of the \TeX{} ecosystem.

The x type expands all tokens fully, starting from the first. In contrast to e, all macro
parameter characters # must be doubled, and omitting this leads to low-level errors. In
addition this type of expansion is not expandable, namely functions that have x in their
signature do not themselves expand when appearing inside x or e expansion.

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 the front of the token list 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 \texttt{example:n}. For
this, one should define a variant using \texttt{cs\_generate\_variant:Nn \example:n \{ f \}}, then do

\begin{verbatim}
\example:f { \int_eval:n \{ 3 + 4 \} }
\end{verbatim}

Note that x-expansion would also expand \texttt{int\_eval:n} fully to its result 7, but the
variant \texttt{example:x} cannot be expandable. Note also that o-expansion would not expand
\texttt{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

\begin{verbatim}
\example:f { \int_eval:n \{ 1 + 2 \} , \int_eval:n \{ 3 + 4 \} }
\end{verbatim}

results in the call

\begin{verbatim}
\example:n \{ 3 , \int_eval:n \{ 3 + 4 \} }
\end{verbatim}

while using \texttt{example:x} or \texttt{example:e} instead results in

\begin{verbatim}
\example:n \{ 3 , 7 \}
\end{verbatim}

at the cost of being protected (for x type) or very much slower in old engines (for e type).
If you use f type expansion in conditional processing then you should stick to using TF
type functions only as the expansion does not finish any \texttt{if... fi:} itself!

It 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

\begin{verbatim}
\exp_after:wN \texttt{<base function>} \exp_after:wN \{ \texttt{<argument>} \}
\end{verbatim}

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

\begin{verbatim}
\tl_set:No \texttt{l\_tmpa\_tl} \{ \{ \texttt{g\_tmpb\_tl} \} \}
\end{verbatim}
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.

It is usually best to keep the following in mind when using variant forms.

- Variants with x-type arguments (that are fully expanded before being passed to the n-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using f or e expansion.

- In contrast, e expansion (full expansion, almost like x except for the treatment of #) does not prevent variants from being expandable (if the base function is). The drawback is that e expansion is very much slower in old engines (before 2019). Consider using f expansion if that type of expansion is sufficient to perform the required expansion, or x expansion if the variant will not itself need to be expandable.

- Finally f expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.

- Arguments that should consist of single tokens N, c, V, or v should come first among these.

- Arguments that appear after the first multi-token argument n, f, e, or o require slightly slower special processing to be expanded. 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, e, with possible trailing N or n or T or F, which are not expanded. Any x-type argument causes slightly slower processing.

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:Nc \exp_args:cc

This function absorbs two arguments (the \texttt{function} name and the \texttt{tokens}). The \texttt{tokens} are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream after reinsertion of the \texttt{function}. Thus the \texttt{function} may take more than one argument: all others are left unchanged.

The :cc variant constructs the \texttt{function} name in the same manner as described for the \texttt{tokens}.

\textbf{\TeX\ hackers note:} Protected macros that appear in a c-type argument are expanded despite being protected; \exp_not:n also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.
This function absorbs two arguments (the \texttt{function} name and the \texttt{tokens}). The \texttt{tokens} are expanded once, and the result is inserted in braces into the input stream after reinsertion of the \texttt{function}. Thus the \texttt{function} may take more than one argument: all others are left unchanged.

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

This function absorbs two arguments (the \texttt{function} name and the \texttt{tokens}). The \texttt{tokens} are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a \texttt{variable}. The content of the \texttt{variable} are recovered and placed inside braces into the input stream after reinsertion of the \texttt{function}. Thus the \texttt{function} may take more than one argument: all others are left unchanged.

\textbf{\TeXhackers\!note:} Protected macros that appear in a v-type argument are expanded despite being protected; \texttt{\exp\!not:n} also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

This function absorbs two arguments (the \texttt{function} name and the \texttt{tokens}) and exhaustively expands the \texttt{tokens}. The result is inserted in braces into the input stream after reinsertion of the \texttt{function}. Thus the \texttt{function} may take more than one argument: all others are left unchanged.

\textbf{\TeXhackers\!note:} This relies on the \texttt{\expanded} primitive when available (in \LaTeX\! and starting around 2019 in other engines). Otherwise it uses some fall-back code that is very much slower. As a result it should only be used in performance-critical code if typical users have a recent installation of the \TeX\! ecosystem.

This function absorbs two arguments (the \texttt{function} name and the \texttt{tokens}). The \texttt{tokens} are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream after reinsertion of the \texttt{function}. Thus the \texttt{function} may take more than one argument: all others are left unchanged.
\exp_args:Nx \exp_args:Nx \{\langle \text{function} \rangle \} \{\langle \text{tokens} \rangle \}

This function absorbs two arguments (the \textit{function} name and the \textit{tokens}) and exhaustively expands the \textit{tokens}. The result is inserted in braces into the input stream after reinsertion of the \textit{function}. Thus the \textit{function} may take more than one argument: all others are left unchanged.

5 \hspace{1cm} \textbf{Manipulating two arguments}

\exp_args:NNo \exp_args:NNo \{\langle \text{token} \rangle \} \{\langle \text{token} \rangle \} \{\langle \text{tokens} \rangle \}

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:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe
\exp_args:NNo \exp_args:NNo \exp_args:NNo
\exp_args:NnV \exp_args:NnV
\exp_args:Nnv \exp_args:Nnv
\exp_args:Nne \exp_args:Nne
\exp_args:Nnf \exp_args:Nnf
\exp_args:Noc \exp_args:Noc
\exp_args:Noo \exp_args:Noo
\exp_args:Nof \exp_args:Nof
\exp_args:NVo \exp_args:NVo
\exp_args:Nfo \exp_args:Nfo
\exp_args:Nff \exp_args:Nff
\exp_args:Nee \exp_args:Nee
\exp_args:Nxx \exp_args:Nxx
\exp_args:Nv \exp_args:Nv
\exp_args:Nx \exp_args:Nx
\exp_args:Nx \exp_args:Nx
\exp_args:Nx \exp_args:Nx
\exp_args:Nxx

Updated: 2018-05-15

\exp_args:NNc \exp_args:NNc {\langle \text{token} \rangle} {\langle \text{token} \rangle} {\langle \text{tokens} \rangle}

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 slower processing.

\exp_args:Nnc \exp_args:Nnc \exp_args:Nnc \exp_args:Nnc \exp_args:Nnc
\exp_args:Nno \exp_args:Nno \exp_args:Nno
\exp_args:NnV \exp_args:NnV
\exp_args:Nnv \exp_args:Nnv
\exp_args:Nne \exp_args:Nne
\exp_args:Nnf \exp_args:Nnf
\exp_args:Noc \exp_args:Noc
\exp_args:Noo \exp_args:Noo
\exp_args:Nof \exp_args:Nof
\exp_args:NVo \exp_args:NVo
\exp_args:Nfo \exp_args:Nfo
\exp_args:Nff \exp_args:Nff
\exp_args:Nee \exp_args:Nee
\exp_args:Nxx \exp_args:Nxx

Updated: 2018-05-15

\exp_args:NNx \exp_args:NNx {\langle \text{token} \rangle} {\langle \text{token} \rangle} {\langle \text{tokens} \rangle}

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 due to their x-type argument.
6 Manipulating three arguments

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.

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 slower processing.

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

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, :Nfo and :NnNo variants need slower processing.

\textbf{\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, \texttt{\exp_last_unbraced:Nf \foo_bar:w \{ \} q_stop} leads to an infinite loop, as the quark is f-expanded.

This function fully expands the \langle tokens\rangle and leaves the result in the input stream after reinsertion of the \langle function\rangle. This function is not expandable.

\textbf{\TeXhackers note:} This is the \TeX primitive \texttt{\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 ‘expand-
able’ since they themselves disappear after the expansion has completed.

\texttt{\exp_not:N} \langle\text{token}\rangle

Prevents expansion of the \langle\text{token}\rangle in a context where it would otherwise be expanded, for example an \texttt{x}-type argument or the first token in an \texttt{c} or \texttt{e} or \texttt{f} argument.

\textbf{\TeXhackers note:} This is the \TeX\ \texttt{noexpand} primitive. It only prevents expansion. At the beginning of an \texttt{f}-type argument, a space \langle\text{token}\rangle is removed even if it appears as \texttt{\exp_not:N \c_space_token}. In an \texttt{x}-expanding definition (\texttt{\cs_new:Npx}), a macro parameter introduces an argument even if it appears as \texttt{\exp_not:N \# 1}. This differs from \texttt{\exp_not:n}.

\texttt{\exp_not:c \{\langle\text{tokens}\rangle\}}

Expands the \langle\text{tokens}\rangle until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using \texttt{\exp_not:N}.

\textbf{\TeXhackers note:} Protected macros that appear in a \texttt{c}-type argument are expanded despite being protected; \texttt{\exp_not:n} also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

\texttt{\exp_not:n \{\langle\text{tokens}\rangle\}}

Prevents expansion of the \langle\text{tokens}\rangle in an \texttt{e} or \texttt{x}-type argument. In all other cases the \langle\text{tokens}\rangle continue to be expanded, for example in the input stream or in other types of arguments such as \texttt{c}, \texttt{f}, \texttt{v}. The argument of \texttt{\exp_not:n} must be surrounded by braces.

\textbf{\TeXhackers note:} This is the e-\TeX\ \texttt{unexpanded} primitive. In an \texttt{x}-expanding definition (\texttt{\cs_new:Npx}), \texttt{\exp_not:n \{\#\}} is equivalent to \texttt{\#\#1} rather than to \texttt{\#1}, namely it inserts the two characters \# and 1. In an \texttt{e}-type argument \texttt{\exp_not:n \{\#\}} is equivalent to \texttt{\#}, namely it inserts the character \#.

\texttt{\exp_not:o \{\langle\text{tokens}\rangle\}}

Expands the \langle\text{tokens}\rangle once, then prevents any further expansion in \texttt{x}-type or \texttt{e}-type arguments using \texttt{\exp_not:n}.

\texttt{\exp_not:V \langle\text{variable}\rangle}

Recover the content of the \langle\text{variable}\rangle, then prevents expansion of this material in \texttt{x}-type or \texttt{e}-type arguments using \texttt{\exp_not:n}.
\texttt{\textbackslash exp\_not:v} ⋆ \texttt{\textbackslash exp\_not:n} \{\textbackslash tokens\}\}

Expands the \{\textbackslash tokens\} until only characters remains, and then converts this into a control sequence which should be a \textit{variable} name. The content of the \textit{variable} is recovered, and further expansion in \texttt{x}-type or \texttt{e}-type arguments is prevented using \texttt{\exp_not:n}.

\textbf{\TeXhackers note:} Protected macros that appear in a \texttt{v}-type argument are expanded despite being protected; \texttt{\exp_not:n} also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

\texttt{\textbackslash exp\_not:e} ⋆ \texttt{\textbackslash exp\_not:n} \{\textbackslash tokens\}\}

Expands \{\textbackslash tokens\} exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in \texttt{e} or \texttt{x}-type arguments using \texttt{\exp_not:n}. This is very rarely useful but is provided for consistency.

\texttt{\textbackslash exp\_not:f} ⋆ \texttt{\textbackslash exp\_not:n} \{\textbackslash tokens\}\}

Expands \{\textbackslash tokens\} fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in \texttt{x}-type or \texttt{e}-type arguments using \texttt{\exp_not:n}.

\texttt{\textbackslash exp\_stop\_f:} ⋆ \texttt{\textbackslash foo\_bar:f} \{\texttt{\textbackslash more\_tokens}\}\}

This function terminates an \texttt{f}-type expansion. Thus if a function \texttt{\foo_bar:f} starts an \texttt{f}-type expansion and all of \{\textbackslash tokens\} are expandable \texttt{\exp\_stop\_f:} terminates the expansion of tokens even if \{\texttt{\textbackslash more\_tokens}\} are also expandable. The function itself is an implicit space token. Inside an \texttt{x}-type expansion, it retains its form, but when typeset it produces the underlying space (\textbackslash uni2423).

\section{Controlled expansion}

The \texttt{exp3} 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 \texttt{\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. These commands are 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 \texttt{\exp\_after:w\!} 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 \{\texttt{expandable\_tokens}\} as that will break badly if unexpandable tokens are encountered in that place!
\exp:w \langle \text{expandable tokens} \rangle \exp_end:

Expands \langle \text{expandable-tokens} \rangle until reaching \exp_end: at which point expansion stops. The full expansion of \langle \text{expandable tokens} \rangle has to be empty. If any token in \langle \text{expandable tokens} \rangle 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.\footnote{Due to the implementation you might get the character in position 0 in the current font (typically "") in the output without any error message!}

In typical use cases the \exp_end: is hidden somewhere in the replacement text of \langle \text{expandable-tokens} \rangle 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.

\TeXhackers note: The current implementation uses \romannumeral hence ignores space tokens and explicit signs + and - in the expansion of the \langle \text{expandable tokens} \rangle, but this should not be relied upon.

\exp:w \langle \text{expandable-tokens} \rangle \exp_end_continue_f:w \langle \text{further-tokens} \rangle

Expands \langle \text{expandable-tokens} \rangle until reaching \exp_end_continue_f:w at which point expansion continues as an f-type expansion expanding \langle \text{further-tokens} \rangle 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 gets removed.

The full expansion of \langle \text{expandable-tokens} \rangle has to be empty. If any token in \langle \text{expandable-tokens} \rangle 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.\footnote{In this particular case you may get a character into the output as well as an error message.}

In typical use cases \langle \text{expandable-tokens} \rangle 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 \langle \text{expandable-tokens} \rangle \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 \langle \text{expandable-tokens} \rangle \exp_stop_f:
The difference to \texttt{\exp_end_continue_f:w} is that we first we pick up an argument which is then returned to the input stream. If \texttt{(further-tokens)} starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

\section{Internal functions}

\begin{itemize}
  \item \texttt{\cs_new:Npn \exp_args:Ncof} \{ c o f \}
  \item \texttt{\cs_new:Npn \exp_last_unbraced:Nno} \{ o_unbraced \}
\end{itemize}

Internal forms for the base expansion types. These names do not conform to the general \LaTeX{} approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

\begin{itemize}
  \item \texttt{\cs_new:Npn \exp_args:Ncof} \{ c o f \}
  \item \texttt{\cs_new:Npn \exp_last_unbraced:Nno} \{ o_unbraced \}
\end{itemize}

Internal forms for the expansion types which leave the terminal argument unbraced. These names do not conform to the general \LaTeX{} approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.
Part VI
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 and manipulate the lists of tokens, and these have the module prefix tl. In many cases, functions 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 \} \rightarrow \text{world}

contains six items (Hello, w, o, r, l and d), but thirteen tokens (\{, H, e, l, o, \}, \or, 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

\begin{verbatim}
\tl_new:N \tl_new:c
\tl_const:Nn \tl_const:c
\tl_clear:N \tl_clear:c
\tl_gclear:N \tl_gclear:c
\end{verbatim}

\begin{itemize}
  \item \tl_new:N \tl_new:c \langle tl var \rangle
  \begin{itemize}
    \item Creates a new \langle tl var \rangle or raises an error if the name is already taken. The declaration is global. The \langle tl var \rangle is initially empty.
  \end{itemize}

  \item \tl_const:Nn \tl_const:c \langle tl var \rangle \{ \langle token list \rangle \}
  \begin{itemize}
    \item Creates a new constant \langle tl var \rangle or raises an error if the name is already taken. The value of the \langle tl var \rangle is set globally to the \langle token list \rangle.
  \end{itemize}

  \item \tl_clear:N \tl_clear:c \langle tl var \rangle
  \begin{itemize}
    \item Clears all entries from the \langle tl var \rangle.
  \end{itemize}
\end{itemize}
\tl_clear_new:N \tl_clear_new:c \tl_gclear_new:N \tl_gclear_new:c

\tl_set_eq:NN \tl_set_eq:NN \tl_gset_eq:NN \tl_gset_eq:NN

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

\tl_if_exist_p:N \tl_if_exist_p:N \tl_if_exist_p:N \tl_if_exist_p:N

\tl_if_exist:NTF \tl_if_exist:NTF \tl_if_exist:NTF \tl_if_exist:NTF

\tl_set:Nn \tl_set:Nn \tl_gset:Nn \tl_gset:Nn

\tl_put_left:Nn \tl_put_left:Nn \tl_gput_left:Nn \tl_gput_left:Nn

\tl_put_right:Nn \tl_put_right:Nn \tl_gput_right:Nn \tl_gput_right:Nn

\tl_set:NN \tl_set:NN \tl_gset:NN \tl_gset:NN

\tl_put_left:NN \tl_put_left:NN \tl_gput_left:NN \tl_gput_left:NN

\tl_put_right:NN \tl_put_right:NN \tl_gput_right:NN \tl_gput_right:NN

-------------------

\tl_clear_new:N \tl_clear_new:N \tl_gclear_new:N \tl_gclear_new:N

Ensures that the \langle tl var \rangle exists globally by applying \tl_new:N if necessary, then applies \tl_(g)clear:N to leave the \langle tl var \rangle empty.

\tl_set_eq:NN \tl_set_eq:NN \tl_gset_eq:NN \tl_gset_eq:NN

Sets the content of \langle tl var \rangle equal to that of \langle tl var \rangle.

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

Concatenates the content of \langle tl var \rangle and \langle tl var \rangle together and saves the result in \langle tl var \rangle. The \langle tl var \rangle is placed at the left side of the new token list.

\tl_if_exist_p:N \tl_if_exist_p:N \tl_if_exist:NTF \tl_if_exist:NTF

Tests whether the \langle tl var \rangle is currently defined. This does not check that the \langle tl var \rangle really is a token list variable.

2 Adding data to token list variables

\tl_set:Nn \tl_set:NN \tl_gset:Nn \tl_gset:NN

Sets \langle tl var \rangle to contain \langle tokens \rangle, removing any previous content from the variable.

\tl_put_left:Nn \tl_put_left:NN \tl_gput_left:Nn \tl_gput_left:NN

Appends \langle tokens \rangle to the left side of the current content of \langle tl var \rangle.

\tl_put_right:Nn \tl_put_right:NN \tl_gput_right:Nn \tl_gput_right:NN

Appends \langle tokens \rangle to the right side of the current content of \langle tl var \rangle.
3 Modifying token list variables

\tlreplace_once:Nnn \tlgreplace_once:Nnn

Replaces the first (leftmost) occurrence of \langle old tokens\rangle in the \langle tl var\rangle with \langle new tokens\rangle. \langle Old tokens\rangle cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tlreplace_all:Nnn \tlgreplace_all:Nnn

Replaces all occurrences of \langle old tokens\rangle in the \langle tl var\rangle with \langle new tokens\rangle. \langle Old tokens\rangle 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 \langle old tokens\rangle may remain after the replacement (see \tlremove_all:Nn for an example).

\tlremove_once:Nn \tlgremove_once:Nn

Removes the first (leftmost) occurrence of \langle tokens\rangle from the \langle tl var\rangle. \langle Tokens\rangle cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tlremove_all:Nn \tlgremove_all:Nn

Removes all occurrences of \langle tokens\rangle from the \langle tl var\rangle. \langle Tokens\rangle 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 \langle tokens\rangle may remain after the removal, for instance,

\tlset:Nn \l_tmpa_tl {abbccd} \tlremove_all:Nn \l_tmpa_tl \{bc\}

results 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 or using \char_generate:nn).
Sets \texttt{\{tl var\}} to contain \texttt{\{tokens\}}, applying the category code régime specified in the \texttt{\{setup\}} before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the \texttt{\{setup\}} are those in force at the point of use of \texttt{\tl_set_rescan:Nnn}.) This allows the \texttt{\{tl var\}} to contain material with category codes other than those that apply when \texttt{\{tokens\}} are absorbed. The \texttt{\{setup\}} is run within a group and may contain any valid input, although only changes in category codes are relevant. See also \texttt{\tl_rescan:nn}.

\textbf{\TeXhackers note:} The \texttt{\{tokens\}} are first turned into a string (using \texttt{\tl_to_str:n}). If the string contains one or more characters with character code \texttt{\newlinechar} (set equal to \texttt{\endlinechar} unless that is equal to \texttt{32}, before the user \texttt{\{setup\}}), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode \texttt{10}) at the beginning and spaces and tabs (character code \texttt{32} or \texttt{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.

\texttt{\tl_rescan:nn} \\ \texttt{\{setup\}} \texttt{\{tokens\}}

Rescans \texttt{\{tokens\}} applying the category code régime specified in the \texttt{\{setup\}}, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the \texttt{\{setup\}} are those in force at the point of use of \texttt{\tl_rescan:nn}.) The \texttt{\{setup\}} is run within a group and may contain any valid input, although only changes in category codes are relevant. See also \texttt{\tl_set_rescan:Nnn}, which is more robust than using \texttt{\tl_set:Nn} in the \texttt{\{tokens\}} argument of \texttt{\tl_rescan:nn}.

\textbf{\TeXhackers note:} The \texttt{\{tokens\}} are first turned into a string (using \texttt{\tl_to_str:n}). If the string contains one or more characters with character code \texttt{\newlinechar} (set equal to \texttt{\endlinechar} unless that is equal to \texttt{32}, before the user \texttt{\{setup\}}), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode \texttt{10}) at the beginning and spaces and tabs (character code \texttt{32} or \texttt{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.

\section{Token list conditionals}

\texttt{\tl_if_blank_p:n} \texttt{\{token list\}} \texttt{\{\false code\}} \texttt{\{\true code\}}

Tests if the \texttt{\{token list\}} consists only of blank spaces (i.e. contains no item). The test is \texttt{true} if \texttt{\{token list\}} is zero or more explicit space characters (explicit tokens with character code \texttt{32} and category code \texttt{10}), and is \texttt{false} otherwise.
Tests if the \langle token list variable \rangle is entirely empty (i.e. contains no tokens at all).

\texttt{\textbackslash tl\_if\_empty\_p:N} \star \texttt{\textbackslash tl\_if\_empty\_p:c} \star \texttt{\textbackslash tl\_if\_empty:NTF} \star \texttt{\textbackslash tl\_if\_empty:cTF} \\
New: 2012-05-24 \\
Updated: 2012-06-05

\texttt{\textbackslash tl\_if\_empty:p:N} \langle \texttt{tl var} \rangle \texttt{\textbackslash tl\_if\_empty:NTF} \langle \texttt{tl var} \rangle \{(\texttt{true code})\} \{(\texttt{false code})\}

Tests if the \langle token list \rangle is entirely empty (i.e. contains no tokens at all).

\texttt{\textbackslash tl\_if\_empty:p:n} \star \texttt{\textbackslash tl\_if\_empty:p:(V|o)} \star \texttt{\textbackslash tl\_if\_empty:nTF} \star \texttt{\textbackslash tl\_if\_empty:n} \\
New: 2017-11-14 \\
Updated: 2017-06-05

\texttt{\textbackslash tl\_if\_eq:p:NN} \star \texttt{\textbackslash tl\_if\_eq:p:(Nc|cN|cc) \star \texttt{\textbackslash tl\_if\_eq:NNTF}} \langle \texttt{tl var} \rangle \langle \texttt{tl var} \rangle \{\langle \texttt{true code} \rangle \} \{\langle \texttt{false code} \rangle \}

Compares the content of two \langle token list variables \rangle 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

\texttt{\textbackslash tl\_set:Nn \_l\_tmpa\_tl} \{ \texttt{abc} \}
\texttt{\textbackslash tl\_set:Nx \_l\_tmpb\_tl} \{ \texttt{\_l\_to\_str:n} \{ \texttt{abc} \} \}
\texttt{\textbackslash tl\_if\_eq:NNTF \_l\_tmpa\_tl \_l\_tmpb\_tl} \{ \texttt{true} \} \{ \texttt{false} \}

yields false.

\texttt{\textbackslash tl\_if\_eq:NN} \star \texttt{\textbackslash tl\_if\_eq:nnTF} \langle \texttt{token list1} \rangle \langle \texttt{token list2} \rangle \{\langle \texttt{true code} \rangle \} \{\langle \texttt{false code} \rangle \}

Tests if \langle token list1 \rangle and \langle token list2 \rangle contain the same list of tokens, both in respect of character codes and category codes.

\texttt{\textbackslash tl\_if\_in:NnTF} \star \texttt{\textbackslash tl\_if\_in:cnTF} \\
\texttt{\textbackslash tl\_if\_in:NnTF} \langle \texttt{tl var} \rangle \{\langle \texttt{token list} \rangle\} \{\langle \texttt{true code} \rangle\} \{\langle \texttt{false code} \rangle\}

Tests if the \langle token list \rangle is found in the content of the \langle tl var \rangle. The \langle token list \rangle 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).

\texttt{\textbackslash tl\_if\_in:nnTF} \\
\texttt{\textbackslash tl\_if\_in:(Vn|on|no)TF} \\
\texttt{\textbackslash tl\_if\_in:nnTF} \langle \texttt{token list1} \rangle \{\langle \texttt{token list2} \rangle\} \{\langle \texttt{true code} \rangle\} \{\langle \texttt{false code} \rangle\}

Tests if \langle token list2 \rangle is found inside \langle token list1 \rangle. The \langle token list2 \rangle 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).

\texttt{\textbackslash tl\_if\_novalue_p:n} \star \texttt{\textbackslash tl\_if\_novalue:p:n} \star \texttt{\textbackslash tl\_if\_novalue:NTF} \star \texttt{\textbackslash tl\_if\_novalue:nTF} \\
New: 2017-11-14

Tests if the \langle token list \rangle is exactly equal to the special \c_novalue_tl marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.
Tests if the content of the \texttt{tl var} consists of a single item, \textit{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 \texttt{tl_count:N}.

Tests if the \texttt{token list} has exactly one item, \textit{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 \texttt{tl_count:n}.

Tests if the token list consists of exactly one token, \textit{i.e.} is either a single space character or a single “normal” token. Token groups (\{...\}) are not single tokens.

This function compares the \texttt{test token list variable} in turn with each of the \texttt{token list variable cases}. If the two are equal (as described for \texttt{tl_if_eq:NNTF}) then the associated \texttt{code} is left in the input stream and other cases are discarded. If any of the cases are matched, the \texttt{true code} is also inserted into the input stream (after the code for the appropriate case), while if none match then the \texttt{false code} is inserted. The function \texttt{tl_case:Nn}, which does nothing if there is no match, is also available.

### 6 Mapping to token lists

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \langle function \rangle or \langle code \rangle discussed below remain in effect after the loop.
\tl_map_function:nN \{\langle token list\rangle\} \langle function\rangle

Applies \langle function\rangle to every \langle item\rangle in the \langle token list\rangle. The \langle function\rangle receives one argument for each iteration. This may be a number of tokens if the \langle item\rangle was stored within braces. Hence the \langle function\rangle should anticipate receiving n-type arguments. See also \tl_map_function:NN.

\tl_map_inline:Nn \\langle tl var\rangle \{\langle inline function\rangle\}

Applies the \langle inline function\rangle to every \langle item\rangle stored within the \langle tl var\rangle. The \langle inline function\rangle should consist of code which receives the \langle item\rangle as #1. See also \tl_map_function:NN.

\tl_map_tokens:Nn \\langle tokens\rangle \{\langle code\rangle\}

Analogue of \tl_map_function:NN which maps several tokens instead of a single function. The \langle code\rangle receives each item in the \langle tokens\rangle as two trailing brace groups. For instance,

\tl_map_tokens:Nn \l_my_tl \{ \prg_replicate:nn { 2 } \}

expands to twice each item in the \langle sequence\rangle: for each item in \l_my_tl the function \prg_replicate:nn receives 2 and \langle item\rangle as its two arguments. The function \tl_map_inline:Nn is typically faster but is not expandable.

\tl_map_variable:NNn \\langle tl var\rangle \\langle variable\rangle \{\langle code\rangle\}

Stores each \langle item\rangle of the \langle tl var\rangle in turn in the \langle token list\rangle \langle variable\rangle and applies the \langle code\rangle. The \langle code\rangle will usually make use of the \langle variable\rangle, but this is not enforced. The assignments to the \langle variable\rangle are local. Its value after the loop is the last \langle item\rangle in the \langle tl var\rangle, or its original value if the \langle tl var\rangle is blank. See also \tl_map_inline:Nn.
\tl_map_break: \tl_map_break:

Used to terminate a \tl_map... function before all entries in the \langle token list variable \rangle have been processed. This normally takes 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 leads to low level \TeX{} errors.

\TeXhackersnote: When the mapping is broken, additional tokens may be inserted before the \langle tokens \rangle are inserted into the input stream. This depends on the design of the mapping function.

\tl_map_break:n \tl_map_break:n \{\langle code\rangle\}

Used to terminate a \tl_map... function before all entries in the \langle token list variable \rangle have been processed, inserting the \langle code\rangle after the mapping has ended. This normally takes place within a conditional statement, for example

\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  \{ \tl_map_break:n \{ \langle code\rangle \} \}
  \% Do something useful
}

Use outside of a \tl_map... scenario leads to low level \TeX{} errors.

\TeXhackersnote: When the mapping is broken, additional tokens may be inserted before the \langle code\rangle is inserted into the input stream. This depends on the design of the mapping function.
7 Using token lists

\tl_to_str:n \{\langle token list\rangle\}

Converts the \langle token list\rangle to a \langle string\rangle, leaving the resulting character tokens in the input stream. A \langle string\rangle is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument must be braced.

\textbf{\texttt{\textbackslash tlpXhackers note:}} This is the \texttt{\textbackslash detokenize} \texttt{\textbackslash e-\texttt{\textbackslash tex} primitive.} Converting a \langle token list\rangle to a \langle string\rangle yields a concatenation of the string representations of every token in the \langle token list\rangle. The string representation of a control sequence is

\begin{itemize}
  \item an escape character, whose character code is given by the internal parameter \texttt{\textbackslash escapechar}, absent if the \texttt{\textbackslash escapechar} is negative or greater than the largest character code;
  \item the control sequence name, as defined by \texttt{\textbackslash cs_to_str:n};
  \item a space, unless the control sequence name is a single character whose category at the time of expansion of \texttt{\textbackslash tl_to_str:n} is not “letter”.
\end{itemize}

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 \texttt{\textbackslash escapechar}: for instance \texttt{\textbackslash 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 \texttt{a} is currently not a letter.

\tl_to_str:n \langle tl var\rangle

Converts the content of the \langle tl var\rangle into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This \langle string\rangle is then left in the input stream. For low-level details, see the notes given for \texttt{\textbackslash tl_to_str:n}.

\tl_use:N \langle tl var\rangle

Recovers the content of a \langle tl var\rangle and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a \langle tl var\rangle directly without an accessor function.

8 Working with the content of token lists

\tl_count:n \{\langle tokens\rangle\}

Counts the number of \langle items\rangle in \langle tokens\rangle and leaves this information in the input stream. Unbraced tokens count as one element as do each token group \{\ldots\}. This process ignores any unprotected spaces within \langle tokens\rangle. See also \texttt{\textbackslash tl_count:N}. This function requires three expansions, giving an \langle integer denotation\rangle.
\tl_count:N \tl_count:c

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 \{\ldots\}. This process ignores any unprotected spaces within the \tl var\. See also \tl_count:n. This function requires three expansions, giving an \textit{integer denotation}.

\tl_count_tokens:n \tl_count_tokens:o

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 \texttt{a\{bc\}} is 6.

\tl_reverse:n \tl_reverse:v

Reverses the order of the \items\ in the \token list\, so that \{item\}_1\{item\}_2\{item\}_3 \ldots \{item\}_n becomes \{item\}_n\ldots\{item\}_3\{item\}_2\{item\}_1. This process preserves 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 \texttt{\unexpanded}, which means that the token list does not expand further when appearing in an \texttt{x-type} argument expansion.

\tl_reverse_items:n

Reverses the order of the \items\ stored in \tl var, so that \{item\}_1\{item\}_2\{item\}_3 \ldots \{item\}_n becomes \{item\}_n\ldots\{item\}_3\{item\}_2\{item\}_1. This process preserves 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. \TeXhackers note: The result is returned within \texttt{\unexpanded}, which means that the token list does not expand further when appearing in an \texttt{x-type} argument expansion.

\tl_trim_spaces:n \tl_trim_spaces:o

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 \texttt{\unexpanded}, which means that the token list does not expand further when appearing in an \texttt{x-type} argument expansion.
\tl_trim_spaces_apply:nN \tl_trim_spaces_apply:oN

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the \langle token list \rangle and passes the result to the \langle function \rangle as an n-type argument.

\tl_trim_spaces:N \tl_gtrim_spaces:N

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the \langle tl var \rangle. Note that this therefore resets the content of the variable.

\tl_sort:Nn \tl_sort:cn \tl_gsort:Nn \tl_gsort:cn

Sorts the items in the \langle tl var \rangle according to the \langle comparison code \rangle, and assigns the result to \langle tl var \rangle. The details of sorting comparison are described in Section 1.

\tl_sort:nN {\langle token list \rangle} \langle conditional \rangle

Sorts the items in the \langle token list \rangle, using the \langle conditional \rangle to compare items, and leaves the result in the input stream. The \langle conditional \rangle should have signature :nnTF, and return true if the two items being compared should be left in the same order, and false if the items should be swapped. The details of sorting comparison are described in Section 1.

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

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 \{ \langle token list \rangle \}
Leaves in the input stream the first \langle item \rangle in the \langle token list \rangle, discarding the rest of the \langle token list \rangle. 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 \}

both leave a in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

\tl_head:n \{ - \{ - ab \} c \}

yields \{ab\}. A blank \langle token list \rangle (see \tl_if_blank:nTF) results in \tl_head:n leaving nothing in the input stream.

\textit{\LaTeX}hackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an \texttt{x}-type argument expansion.

\tl_tail:n \{ \langle token list \rangle \}
Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first \langle item \rangle in the \langle token list \rangle, 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 \}

both leave \{bc\}d in the input stream. A blank \langle token list \rangle (see \tl_if_blank:nTF) results in \tl_tail:n leaving nothing in the input stream.

\textit{\LaTeX}hackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an \texttt{x}-type argument expansion.
Tests if the first \textit{token} in the \textit{token list} has the same category code as the \textit{test token}. In the case where the \textit{token list} is empty, the test is always \texttt{false}.

Tests if the first \textit{token} in the \textit{token list} has the same character code as the \textit{test token}. In the case where the \textit{token list} is empty, the test is always \texttt{false}.

Tests if the first \textit{token} in the \textit{token list} has the same meaning as the \textit{test token}. In the case where \textit{token list} is empty, the test is always \texttt{false}.

Tests if the first \textit{token} in the \textit{token list} is an explicit begin-group character (with category code 1 and any character code), in other words, if the \textit{token list} starts with a \texttt{bracegroup}. In particular, the test is \texttt{false} if the \textit{token list} starts with an implicit token such as \texttt{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.

Tests if the first \textit{token} in the \textit{token list} is a normal \texttt{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 \texttt{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.

Tests if the first \textit{token} in the \textit{token list} is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is \texttt{false} if the \textit{token list} starts with an implicit token such as \texttt{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

\texttt{\textbackslash tl\_item:nn} * \texttt{\textbackslash tl\_item:Hn} * \texttt{\textbackslash tl\_item:cn} *
\texttt{\textbackslash tl\_item:nn \{ token list \} \{ integer expression \}}

Indexing items in the \texttt{token list} from 1 on the left, this function evaluates the \texttt{integer expression} and leaves the appropriate item from the \texttt{token list} in the input stream. If the \texttt{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 the function expands to nothing.

\textbf{\TeXhackers note:} The result is returned within the \texttt{\unexpanded primitive (\exp_not:n)}, which means that the \texttt{item} does not expand further when appearing in an \texttt{x}-type argument expansion.

\texttt{\textbackslash tl\_rand\_item:N} * \texttt{\textbackslash tl\_rand\_item:c} * \texttt{\textbackslash tl\_rand\_item:n} *
\texttt{\textbackslash tl\_rand\_item:N \{ tl\ var \}} \texttt{\textbackslash tl\_rand\_item:n \{ \{ token list \} \}}

Selects a pseudo-random item of the \texttt{token list}. If the \texttt{token list} is blank, the result is empty. This is not available in older versions of \TeX.

\textbf{\TeXhackers note:} The result is returned within the \texttt{\unexpanded primitive (\exp_not:n)}, which means that the \texttt{item} does not expand further when appearing in an \texttt{x}-type argument expansion.
Leaves in the input stream the items from the \textit{start index} to the \textit{end index} inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here \textit{start index} and \textit{end index} should be integer denotations. For describing in detail the functions’ behavior, let \textit{m} and \textit{n} be the start and end index respectively. If either is 0, the result is empty. A positive index means ‘start counting from the left end’, and a negative index means ‘from the right end’. Let \textit{l} be the count of the token list.

The \textit{actual start point} is determined as \(M = m\) if \(m > 0\) and as \(M = l + m + 1\) if \(m < 0\). Similarly the \textit{actual end point} is \(N = n\) if \(n > 0\) and \(N = l + n + 1\) if \(n < 0\). If \(M > N\), the result is empty. Otherwise it consists of all items from position \textit{M} to position \textit{N} inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions \textit{s} for \(s \leq 0\) or \(s > l\).

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to \texttt{\tl_trim_spaces:n} to begin with.

Thus, with \textit{l} = 7 as in the examples below, all of the following are equivalent and result in the whole token list

\begin{verbatim}
\tl_range:nnn { abcd-{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd-{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd-{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd-{e{}}fg } { -12 } { 7 }
\end{verbatim}

Here are some more interesting examples. The calls

\begin{verbatim}
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
\end{verbatim}

are all equivalent and will print \texttt{bcd{e{}}} on the terminal; similarly

\begin{verbatim}
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
\end{verbatim}

are all equivalent and will print \texttt{bcd \{e{}}} on the terminal (note the space in the middle).

To the contrary,

\begin{verbatim}
\tl_range:nnn { abcd-{e{}}f } { 2 } { 4 }
\end{verbatim}

will discard the space after ‘d’.

If we want to get the items from, say, the third to the last in a token list \texttt{<tl>}, the call is \texttt{\tl_range:nnn { <tl> } { 3 } { -1 }}. Similarly, for discarding the last item, we can do \texttt{\tl_range:nnn { <tl> } { 1 } { -2 } }.

For better performance, see \texttt{\tl_range_braced:nnn} and \texttt{\tl_range_unbraced:nnn}.

\textbf{\TeX{}hackers note}: The result is returned within the \texttt{\unexpanded} primitive \texttt{\exp_not:n}, which means that the \texttt{\item} does not expand further when appearing in an \texttt{x}-type argument expansion.
11 Viewing token lists

\tl_show:N \tl_show:c
Displays the content of the \langle tl var \rangle 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:c
Updated: 2015-08-01
Displays the \langle token list \rangle on the terminal.

**TeXhackers note:** This is similar to the \TeX primitive \shoutokens, wrapped to a fixed number of characters per line.

\tl_log:N \tl_log:c
New: 2014-08-22
Updated: 2015-08-01
Writes the content of the \langle tl var \rangle in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n \tl_log:c
New: 2014-08-22
Updated: 2015-08-07
Writes the \langle token list \rangle in the log file. See also \tl_show:n which displays the result in the terminal.

12 Constant token lists

\c_empty_tl
Constant that is always empty.

\c_novalue_tl
New: 2017-11-14
A marker for the absence of an argument. This constant \tl can safely be typeset (\cf \q_-nil), with the result being \texttt{-NoValue-}. It is important to note that \c_novalue_tl is constructed such that it will not match the simple text input \texttt{-NoValue-}, \ie that

\tl_if_eq:VNfT \c_novalue_tl \{ -NoValue- \}

is logically \texttt{false}. The \c_novalue_tl marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty \tl.

\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 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.

\l_tmpb_tl

\g_tmpa_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.

\g_tmpb_tl
Part VII

The l3str package: Strings

\[\text{TEX}\] 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 \text{T\textsc{e}X} sense.

A \text{T\textsc{e}X} string (and thus an \texttt{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 \text{T\textsc{e}X} string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix \ldots\texttt{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 \texttt{\tl_to_\_str:n} for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions \texttt{\cs_to_\_str:N}, \texttt{\tl_to_\_str:n}, \texttt{\tl_to_\_str:N} and \texttt{\token_to_\_str:N} (and variants) generate strings from the appropriate input: these are documented in \texttt{l3basics}, \texttt{l3tl} and \texttt{l3token}, respectively.

Most expandable functions in this module come in three flavours:

- \texttt{\str_\ldots\_N}, which expect a token list or string variable as their argument;
- \texttt{\str_\ldots\_n}, taking any token list (or string) as an argument;
- \texttt{\str_\ldots\_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

\begin{verbatim}
\texttt{\str_new:N \str_new:c}
\end{verbatim}

\texttt{\str_new:N \str_var}

Creates a new \langle \texttt{str var} \rangle or raises an error if the name is already taken. The declaration is global. The \langle \texttt{str var} \rangle is initially empty.

\begin{verbatim}
\texttt{\str_const:Nn \str_const:(NV|Nx|cn|cV|cx)}
\end{verbatim}

\texttt{\str_const:Nn \str_var \{\texttt{token list}\}}

Creates a new constant \langle \texttt{str var} \rangle or raises an error if the name is already taken. The value of the \langle \texttt{str var} \rangle is set globally to the \langle \texttt{token list} \rangle, converted to a string.
\str_clear:N \str_clear:c \str_gclear:N \str_gclear:c
New: 2015-09-18

\str_clear_new:N \str_clear_new:c
New: 2015-09-18

\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_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc
New: 2017-10-08

\str_set:Nn \str_set:(NV|Nx|cn|cV|cx) \str_gset:Nn \str_gset:(NV|Nx|cn|cV|cx)
Updated: 2018-07-28

\str_put_left:Nn \str_put_left:(NV|Nx|cn|cV|cx) \str_gput_left:Nn \str_gput_left:(NV|Nx|cn|cV|cx)
Updated: 2018-07-28

2 Adding data to string variables

\str_set:Nn \str_set:(token list) \str_gset:Nn \str_gset:(token list)
Converts the (token list) to a (string), and stores the result in (str var).

\str_put_left:Nn \str_put_left:(token list) \str_gput_left:Nn \str_gput_left:(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.
Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle}, and appends the result to \texttt{\langle str var\rangle}. The current contents of the \texttt{\langle str var\rangle} are not automatically converted to a string.

### 3 Modifying string variables

\begin{verbatim}
\str_replace_once:Nnn \str_replace_once:cnn \str_greplace_once:Nnn \str_greplace_once:cnn
\end{verbatim}

Converts the \texttt{\langle old\rangle} and \texttt{\langle new\rangle} token lists to strings, then replaces the first (leftmost) occurrence of \texttt{\langle old string\rangle} in the \texttt{\langle str var\rangle} with \texttt{\langle new string\rangle}.

\begin{verbatim}
\str_replace_all:Nnn \str_replace_all:cnn \str_greplace_all:Nnn \str_greplace_all:cnn
\end{verbatim}

Converts the \texttt{\langle old\rangle} and \texttt{\langle new\rangle} token lists to strings, then replaces all occurrences of \texttt{\langle old string\rangle} in the \texttt{\langle str var\rangle} with \texttt{\langle new string\rangle}. As this function operates from left to right, the pattern \texttt{\langle old string\rangle} may remain after the replacement (see \texttt{\str_remove_all:Nn} for an example).

\begin{verbatim}
\str_remove_once:Nn \str_remove_once:cn \str_gremove_once:Nn \str_gremove_once:cn
\end{verbatim}

Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle} then removes the first (leftmost) occurrence of \texttt{\langle string\rangle} from the \texttt{\langle str var\rangle}.

\begin{verbatim}
\str_remove_all:Nn \str_remove_all:cn \str_gremove_all:Nn \str_gremove_all:cn
\end{verbatim}

Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle} then removes all occurrences of \texttt{\langle string\rangle} from the \texttt{\langle str var\rangle}. As this function operates from left to right, the pattern \texttt{\langle string\rangle} may remain after the removal, for instance,

\begin{verbatim}
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str {bc}
\end{verbatim}

results in \texttt{\l_tmpa_str} containing \texttt{abcd}.

57
4 String conditionals

\str_if_exist_p:N \str_if_exist:NTF \str_if_exist_p:c \str_if_exist:TF

Tests whether the \textit{\str var} is currently defined. This does not check that the \textit{\str var} really is a string.

\str_if_empty_p:N \str_if_empty:NTF \str_if_empty_p:c \str_if_empty:TF

Tests if the \textit{string variable} is entirely empty (i.e. contains no characters at all).

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

Compares the content of two \textit{str variables} and is logically \textit{true} if the two contain the same characters in the same order.

\str_if_eq_p:nn \str_if_eq:nnTF \str_if_eq_p:(Vn|on|no|nV|vn|nv|ee) \str_if_eq:nn TF \str_if_eq:(Vn|on|no|nV|vn|nv|ee)TF

Compares the two \textit{token lists} on a character by character basis (namely after converting them to strings), and is \textit{true} if the two \textit{strings} contain the same characters in the same order. Thus for example

\str_if_eq_p:no \{ abc \} \tl_to_str:n \{ abc \}

is logically \textit{true}.

\str_if_in:NnTF \str_if_in:cnTF \str_if_in:nnTF

Converts the \textit{token list} to a \textit{string} and tests if that \textit{string} is found in the content of the \textit{str var}.

\str_if_in:nnTF

Converts both \textit{token lists} to \textit{strings} and tests whether \textit{string2} is found inside \textit{string1}. 
Compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$ (all token lists are converted to strings). If the two are equal (as described for $\str_if_eq:nnTF$) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\str_case:nn$, which does nothing if there is no match, is also available.

Compares the full expansion of the $\langle test\ string\rangle$ in turn with the full expansion of the $\langle string\ cases \rangle$ (all token lists are converted to strings). If the two full expansions are equal (as described for $\str_if_eq:nnTF$) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\str_case_e:nn$, which does nothing if there is no match, is also available. The $\langle test\ string\rangle$ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

5 Mapping to strings

All mappings are done at the current group level, i.e. any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

Applies $\langle function \rangle$ to every $\langle character \rangle$ in the $\langle str\ var \rangle$ including spaces. See also $\str_map_function:NN$.

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then applies $\langle function \rangle$ to every $\langle character \rangle$ in the $\langle string \rangle$ including spaces. See also $\str_map_function:NN$. 
\str_map_inline:Nn \str_map_inline:cn
\textit{New: 2017-11-14}
Applies the \textit{inline function} to every \textit{character} in the \textit{str var} including spaces. The \textit{inline function} should consist of code which receives the \textit{character} as \#1. See also \str_map_function:NN.

\str_map_inline:nn
\textit{New: 2017-11-14}
Converts the \textit{token list} to a \textit{string} then applies the \textit{inline function} to every \textit{character} in the \textit{string} including spaces. The \textit{inline function} should consist of code which receives the \textit{character} as \#1. See also \str_map_function:NN.

\str_map_variable:Nn \str_map_variable:cNn
\textit{New: 2017-11-14}
Stores each \textit{character} of the \textit{string} (including spaces) in turn in the (string or token list) \textit{variable} and applies the \textit{code}. The \textit{code} will usually make use of the \textit{variable}, but this is not enforced. The assignments to the \textit{variable} are local. Its value after the loop is the last \textit{character} in the \textit{string}, or its original value if the \textit{string} is empty. See also \str_map_inline:Nn.

\str_map_variable:nNn
\textit{New: 2017-11-14}
Converts the \textit{token list} to a \textit{string} then stores each \textit{character} in the \textit{string} (including spaces) in turn in the (string or token list) \textit{variable} and applies the \textit{code}. The \textit{code} will usually make use of the \textit{variable}, but this is not enforced. The assignments to the \textit{variable} are local. Its value after the loop is the last \textit{character} in the \textit{string}, or its original value if the \textit{string} is empty. See also \str_map_inline:Nn.

\str_map_break: ☆ \str_map_break:
\textit{New: 2017-10-08}
Used to terminate a \str_map___ function before all characters in the \textit{string} have been processed. This normally takes place within a conditional statement, for example
\begin{verbatim}
\str_map_inline:Nn \l_my_str
{ \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
% Do something useful
}
\end{verbatim}
See also \str_map_break:n. Use outside of a \str_map___ scenario leads to low level \TeX errors.

\textbf{\TeXhackers note:} When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.
\str_map_break:n \{\texttt{\langle code\rangle}\}

Used to terminate a \str_map... function before all characters in the \langle string\rangle have been processed, inserting the \langle code\rangle after the mapping has ended. This normally takes place within a conditional statement, for example

\str_map_inline:Nn \l_my_str
{\str_if_eq:nnT \{ #1 \} \{ bingo \}
{ \str_map_break:n \{ <\texttt{code}> \} }
% Do something useful
}\n
Use outside of a \str_map... scenario leads to low level \TeX{} errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the \langle code\rangle is inserted into the input stream. This depends on the design of the mapping function.

6 Working with the content of strings

\str_use:N \str_var
Recovers the content of a \langle str var\rangle and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a \langle str\rangle directly without an accessor function.

\str_count:n \{\langle token list\rangle\}
\str_count:N \str_count:c
\str_count:n
\str_count_ignore_spaces:n

Leaves in the input stream the number of characters in the string representation of \langle token list\rangle, 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

Leaves in the input stream the number of space characters in the string representation of \langle token list\rangle, as an integer denotation. Of course, this function has no \_ignore_spaces variant.
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: \texttt{\textbackslash str\_head:N} and \texttt{\textbackslash str\_head:n} return a space token with category code 10 (blank space), while the \texttt{\textbackslash 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.

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: \texttt{\textbackslash str\_tail:N} and \texttt{\textbackslash str\_tail:n} only trim that space, while \texttt{\textbackslash 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.

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 \texttt{\textbackslash str\_item:Nn} and \texttt{\textbackslash str\_item:nn}, all characters including spaces are taken into account. The \texttt{\textbackslash 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, \textit{etc}.
\texttt{\textbackslash str\_range:Nnn} \* \texttt{\textbackslash str\_range:nnn \{\textit{token list}\} \{\textit{start index}\} \{\textit{end index}\}}
\texttt{\textbackslash str\_range:cnn} \* \texttt{\textbackslash str\_range:nnn} \* \texttt{\textbackslash str\_range\_ignore\_spaces:nnn} \*

Converting the \texttt{\{token list\}} to a \texttt{\{string\}}, and leaves in the input stream the characters from the \texttt{\{start index\}} to the \texttt{\{end index\}} inclusive. Spaces are preserved and counted as items (contrast this with \texttt{\textbackslash tl\_range:nnn} where spaces are not counted as items and are possibly discarded from the output).

Here \texttt{\{start index\}} and \texttt{\{end index\}} should be integer denotations. For describing in detail the functions’ behavior, let \(m\) and \(n\) be the start and end index respectively. If either is 0, the result is empty. A positive index means ‘start counting from the left end’, a negative index means ‘start counting from the right end’. Let \(l\) be the count of the token list.

The actual start point is determined as \(M = m\) if \(m > 0\) and as \(M = l + m + 1\) if \(m < 0\). Similarly the actual end point is \(N = n\) if \(n > 0\) and \(N = l + n + 1\) if \(n < 0\). If \(M > N\), the result is empty. Otherwise it consists of all items from position \(M\) to position \(N\) inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions \(s\) for \(s \leq 0\) or \(s > l\). For instance,

\begin{verbatim}
\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 \} }
\end{verbatim}

prints \texttt{bcde}, \texttt{cdef}, \texttt{ef}, and an empty line to the terminal. The \texttt{\{start index\}} must always be smaller than or equal to the \texttt{\{end index\}}: if this is not the case then no output is generated. Thus

\begin{verbatim}
\iow_term:x { \str_range:nnn { abcdef } \{ 5 \} \{ 2 \} }
\iow_term:x { \str_range:nnn { abcdef } \{ -1 \} \{ -4 \} }
\end{verbatim}

both yield empty strings.

The behavior of \texttt{\str\_range\_ignore\_spaces:nnn} is similar, but spaces are removed before starting the job. The input

\begin{verbatim}
\iow_term:x { \str_range:nnn { abcdefg } \{ 2 \} \{ 5 \} }
\iow_term:x { \str_range:nnn { abcdefg } \{ 2 \} \{ -3 \} }
\iow_term:x { \str_range:nnn { abcdefg } \{ -6 \} \{ 5 \} }
\iow_term:x { \str_range:nnn { abcdefg } \{ -6 \} \{ -3 \} }
\iow_term:x { \str_range:nnn { abc-efg } \{ 2 \} \{ 5 \} }
\iow_term:x { \str_range:nnn { abc-efg } \{ 2 \} \{ -3 \} }
\iow_term:x { \str_range:nnn { abc-efg } \{ -6 \} \{ 5 \} }
\iow_term:x { \str_range:nnn { abc-efg } \{ -6 \} \{ -3 \} }
\iow_term:x { \str_range\_ignore\_spaces:nnn { abcdefg } \{ 2 \} \{ 5 \} }
\iow_term:x { \str_range\_ignore\_spaces:nnn { abcdefg } \{ 2 \} \{ -3 \} }
\iow_term:x { \str_range\_ignore\_spaces:nnn { abcdefg } \{ -6 \} \{ 5 \} }
\iow_term:x { \str_range\_ignore\_spaces:nnn { abcdefg } \{ -6 \} \{ -3 \} }
\end{verbatim}

63
will print four instances of \texttt{bcde}, four instances of \texttt{bc e} and eight instances of \texttt{bcde}.

\section*{7 String manipulation}

\begin{itemize}
\item \texttt{\str_lowercase:n} \langle \texttt{tokens} \rangle
\item \texttt{\str_uppercase:n} \langle \texttt{tokens} \rangle
\end{itemize}

Converts the input \langle \texttt{tokens} \rangle to their string representation, as described for \texttt{\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 \texttt{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

\begin{verbatim}
\cs_new_protected:Npn \myfunc:nn #1#2
{ \cs_set_protected:cpn{ user \str_uppercase:f { \tl_head:n {#1} } \str_lowercase:f { \tl_tail:n {#1} } }{ #2 } }
\end{verbatim}

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 \textit{not} be used for

\begin{itemize}
\item Caseless comparisons: use \texttt{\str_foldcase:n} for this situation (case folding is distinct from lower casing).
\item Case changing text for typesetting: see the \texttt{\text_lowercase:n(n)}, \texttt{\text_uppercase:n(n)} and \texttt{\text_titlecase:n(n)} functions which correctly deal with context-dependence and other factors appropriate to text case changing.
\end{itemize}

\texttt{\TeX} hackers note: As with all \texttt{expl3} functions, the input supported by \texttt{\str_foldcase:n} is \textit{engine-native} characters which are or interoperate with \texttt{UTF-8}. As such, when used with pdf\TeX\ only the Latin alphabet characters A–Z are case-folded \textit{(i.e. the ASCII range which coincides with UTF-8)}. Full UTF-8 support is available with both Xpdf\TeX\ and Lua\TeX.
\str_foldcase:n \{\{tokens\}\}

Converts the input (\textit{tokens}) to their string representation, as described for \texttt{\tl_to_str:n}, and then folds the case of the resulting (\textit{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 \texttt{\str_foldcase: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 \texttt{\str_foldcase:n} follows the “full” scheme defined by the Unicode Consortium (\textit{e.g.} SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (\textit{i.e.} I always folds to î and not to ı).

**\TeX hacking note:** As with all expl3 functions, the input supported by \texttt{\str_foldcase:n} is \texttt{engine-native} characters which are or interoperate with UTF-8. As such, when used with \texttt{pdf\TeX} \textit{only} the Latin alphabet characters A–Z are case-folded (\textit{i.e.} the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both \texttt{Xe\TeX} and \texttt{Lua\TeX}, subject only to the fact that \texttt{Xe\TeX} in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with \texttt{\tl_to_str:n}.

## 8 Viewing strings

\str_show:N \{\str var\}

Displays the content of the (\textit{str var}) on the terminal.

\str_log:N \{\str var\}

Writes the content of the (\textit{str var}) in the log file.
9 Constant token lists

\c_\text{ampersand_str} \c_\text{at_sign_str} \c_\text{backslash_str} \c_\text{left brace_str} \c_\text{right brace_str} \c_\text{circumflex_str} \c_\text{colon_str} \c_\text{dollar_str} \c_\text{hash_str} \c_\text{percent_str} \c_\text{tilde_str} \c_\text{underscore_str}

New: 2015-09-19

10 Scratch strings

\l_t\text{mpa_str} \l_t\text{mpb_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_t\text{mpa_str} \g_t\text{mpb_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.
Part VIII
The l3str-convert package: string encoding conversions

1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (e.g., PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be utf-16, ISO 8859-1, etc. See Table 1 for a list of supported encodings.\(^5\)
- Bytes are translated to \TeX tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.\(^6\)

2 Conversion functions

\begin{verbatim}
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
\end{verbatim}

This function converts the \langle string \rangle from the encoding given by \langle name 1 \rangle to the encoding given by \langle name 2 \rangle, and stores the result in the \langle str var \rangle. Each \langle name \rangle can have the form \langle encoding \rangle or \langle encoding \rangle/\langle escaping \rangle, where the possible values of \langle encoding \rangle and \langle escaping \rangle are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty \langle name \rangle indicates the use of “native” strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,\(^7\)

\begin{verbatim}
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
\end{verbatim}

results in the variable \l_foo_str holding the string FEFF00480065006C006C006F0021. This is obtained by converting each character in the (native) string Hello! to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark "FEFF, which can be avoided by specifying the encoding utf16be/hex.

An error is raised if the \langle string \rangle is not valid according to the \langle escaping 1 \rangle and \langle encoding 1 \rangle, or if it cannot be reencoded in the \langle encoding 2 \rangle and \langle escaping 2 \rangle (for instance, if a character does not exist in the \langle encoding 2 \rangle). Erroneous input is replaced by the Unicode replacement character "FFFD, and characters which cannot be reencoded are replaced by either the replacement character "FFFD if it exists in the \langle encoding 2 \rangle, or an encoding-specific replacement character, or the question mark character.

\(^5\)Encodings and escapings will be added as they are requested.
Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utf8</td>
<td>UTF-8</td>
</tr>
<tr>
<td>utf16</td>
<td>UTF-16, with byte-order mark</td>
</tr>
<tr>
<td>utf16be</td>
<td>UTF-16, big-endian</td>
</tr>
<tr>
<td>utf16le</td>
<td>UTF-16, little-endian</td>
</tr>
<tr>
<td>utf32</td>
<td>UTF-32, with byte-order mark</td>
</tr>
<tr>
<td>utf32be</td>
<td>UTF-32, big-endian</td>
</tr>
<tr>
<td>utf32le</td>
<td>UTF-32, little-endian</td>
</tr>
<tr>
<td>iso88591, latin1</td>
<td>ISO 8859-1</td>
</tr>
<tr>
<td>iso88592, latin2</td>
<td>ISO 8859-2</td>
</tr>
<tr>
<td>iso88593, latin3</td>
<td>ISO 8859-3</td>
</tr>
<tr>
<td>iso88594, latin4</td>
<td>ISO 8859-4</td>
</tr>
<tr>
<td>iso88595</td>
<td>ISO 8859-5</td>
</tr>
<tr>
<td>iso88596</td>
<td>ISO 8859-6</td>
</tr>
<tr>
<td>iso88597</td>
<td>ISO 8859-7</td>
</tr>
<tr>
<td>iso88598</td>
<td>ISO 8859-8</td>
</tr>
<tr>
<td>iso88599, latin5</td>
<td>ISO 8859-9</td>
</tr>
<tr>
<td>iso885910, latin6</td>
<td>ISO 8859-10</td>
</tr>
<tr>
<td>iso885911</td>
<td>ISO 8859-11</td>
</tr>
<tr>
<td>iso885913, latin7</td>
<td>ISO 8859-13</td>
</tr>
<tr>
<td>iso885914, latin8</td>
<td>ISO 8859-14</td>
</tr>
<tr>
<td>iso885915, latin9</td>
<td>ISO 8859-15</td>
</tr>
<tr>
<td>iso885916, latin10</td>
<td>ISO 8859-16</td>
</tr>
<tr>
<td>clist</td>
<td>comma-list of integers</td>
</tr>
<tr>
<td>⟨empty⟩</td>
<td>native (Unicode) string</td>
</tr>
</tbody>
</table>

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<table>
<thead>
<tr>
<th>Escaping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes, or empty</td>
<td>arbitrary bytes</td>
</tr>
<tr>
<td>hex, hexadecimal</td>
<td>byte = two hexadecimal digits</td>
</tr>
<tr>
<td>name</td>
<td>see \pdfescapename</td>
</tr>
<tr>
<td>string</td>
<td>see \pdfescapestring</td>
</tr>
<tr>
<td>url</td>
<td>encoding used in URLs</td>
</tr>
</tbody>
</table>
\texttt{\textbackslash str\_set\_convert:NNNNTF} \texttt{\textbackslash str\_gset\_convert:NNNNTF}\texttt{\{str var\}} \texttt{\{\langle string\rangle\}} \texttt{\{(name 1)\}} \texttt{\{(name 2)\}} \texttt{\{(true code)\}} \texttt{\{(false code)\}}

As \texttt{\textbackslash str\_set\_convert:NNNN}, converts the \texttt{\langle string\rangle} from the encoding given by \texttt{\langle name 1\rangle} to the encoding given by \texttt{\langle name 2\rangle}, and assigns the result to \texttt{\langle str var\rangle}. Contrarily to \texttt{\textbackslash str\_set\_convert:NNNN}, the conditional variant does not raise errors in case the \texttt{\langle string\rangle} is not valid according to the \texttt{\langle name 1\rangle} encoding, or cannot be expressed in the \texttt{\langle name 2\rangle} encoding. Instead, the \texttt{\langle false code\rangle} is performed.

3 Creating 8-bit mappings

\texttt{\textbackslash str\_declare\_eight\_bit\_encoding:nnn} \texttt{\textbackslash str\_declare\_eight\_bit\_encoding:nnn} \texttt{\{\langle name\rangle\}} \texttt{\{(mapping)\}} \texttt{\{\langle missing\rangle\}}

Declares the encoding \texttt{\langle name\rangle} to map bytes to Unicode characters according to the \texttt{\langle mapping\rangle}, and map those bytes which are not mentioned in the \texttt{\langle mapping\rangle} either to the replacement character (if they appear in \texttt{\langle missing\rangle}), or to themselves.

4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In \LaTeX, would it be better to use the \ldots approach to build a string from a given list of character codes? Namely, within a group, assign 0–9a-f and all characters we want to category “other”, then assign ~ the category superscript, and use \texttt{\scantokens}.

- Change \texttt{\textbackslash str\_set\_convert:NNNN} to expand its last two arguments.

- Describe the internal format in the code comments. Refuse code points in [*D800,*DFFF] in the internal representation?

- Add documentation about each encoding and escaping method, and add examples.

- The \texttt{hex} unescaping should raise an error for odd-token count strings.

- Decide what bytes should be escaped in the \texttt{url} escaping. Perhaps the characters ! ' * +-. / \texttt{0123456789} are safe, and all other characters should be escaped?

- Automate generation of 8-bit mapping files.

- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.

- More encodings (see Heiko’s \texttt{stringenc}). CESU?

- More escapings: \texttt{ASCII85}, shell escapes, lua escapes, etc.?
Part IX
The \texttt{l3quark} package
Quarks

Two special types of constants in \LaTeXe are “quarks” and “scan marks”. By convention all constants of type quark start out with $\q$, and scan marks start with $\s$.

1 Quarks

Quarks are control sequences that expand to themselves and should therefore \emph{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, the most common use case being the ‘stop token’ (\emph{i.e.} $\q\texttt{stop}$). For example, when writing a macro to parse a user-defined date

$\texttt{\date\_parse:n \{19/June/1981\}}$

one might write a command such as

\begin{verbatim}
\cs_new:Npn \date\_parse:n \ #1 \ { \date\_parse\_aux:w \ #1 \ \q\texttt{stop} \ }
\cs_new:Npn \date\_parse\_aux:w \ #1 / #2 / #3 \ \q\texttt{stop}
{ <do something with the date> }
\end{verbatim}

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function $\texttt{\prop\_get:NnN}$ 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 $\q\texttt{no\_value}$. 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 $\texttt{\tl\_if\_eq:NNTF}$. 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 $\texttt{\clist\_map\_function:NN}$.

2 Defining quarks

$\texttt{\quark\_new:N} \langle \texttt{quark} \rangle$

Creates a new $\langle \texttt{quark} \rangle$ which expands only to $\langle \texttt{quark} \rangle$. The $\langle \texttt{quark} \rangle$ is defined globally, and an error message is raised if the name was already taken.

$\texttt{\q\texttt{stop}}$

Used as a marker for delimited arguments, such as

\begin{verbatim}
\cs_set:Npn \tmp:w \ #1 \ #2 \ \q\texttt{stop} \ \{ \ #1 \}
\end{verbatim}
\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 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 latter 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

Tests if the \langle token \rangle is equal to \q_nil.

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

Tests if the \langle token list \rangle contains only \q_nil (distinct from \langle token list \rangle 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:p:n
\quark_if_no_value:nTF
\quark_if_no_value:nTF
\quark_if_no_value:p:c
\quark_if_no_value:NTF

Tests if the \langle token \rangle is equal to \q_no_value.

\quark_if_no_value_p:n
\quark_if_no_value:nTF
\quark_if_no_value:p:(o|V)
\quark_if_no_value:nTF
\quark_if_no_value:p:(o|V)TF

Tests if the \langle token list \rangle contains only \q_no_value (distinct from \langle token list \rangle 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.
This quark is added after the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

Tests if \(\text{token}\) contains only the marker \(\texttt{q\_recursion\_tail}\), and if so uses \texttt{use-\_none_delimit_by_q\_recursion\_stop:w} to terminate the recursion that this belongs to. The recursion input must include the marker tokens \(\texttt{q\_recursion\_tail}\) and \(\texttt{q\_\text{-recursion\_stop}}\) as the last two items.

Tests if the \(\text{token list}\) contains only \(\texttt{q\_recursion\_tail}\), and if so uses \texttt{use-\_i_delimit_by_q\_recursion\_stop:w} to terminate the recursion that this belongs to. The recursion input must include the marker tokens \(\texttt{q\_recursion\_tail}\) and \(\texttt{q\_\text{-recursion\_stop}}\) as the last two items. The \(\langle\text{insertion}\rangle\) code is then added to the input stream after the recursion has ended.

Tests if \(\text{token list}\) contains only \(\texttt{q\_recursion\_tail}\), and if so terminates the recursion using \(\langle\text{type}\rangle\_\text{map\_break}\). The recursion end should be marked by \texttt{prg\_break-point:Nn} \(\langle\text{type}\rangle\_\text{map\_break}\).

5 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as \texttt{\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 \texttt{\my_map_dbl:nn} which takes a token list and applies an operation to every pair of tokens. For example, \texttt{\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 \texttt{\my_map_dbl:nn}. First of all, define the function that does the processing based on the inline function argument \#2. Then initiate the recursion using an internal function. The token list \#1 is terminated using \texttt{\q_recursion_tail}, with delimiters according to the type of recursion (here a pair of \texttt{\q_recursion_tail}), concluding with \texttt{\q_recursion_stop}. These quarks are used to mark the end of the token list being operated upon.

\begin{verbatim}
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
}
\end{verbatim}

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.

\begin{verbatim}
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}
}\__my_map_dbl:nn
}\end{verbatim}

Finally, recurse:

\begin{verbatim}
\__my_map_dbl:nn
}
\end{verbatim}

Note that contrarily to \LaTeX{}3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of \texttt{\__my_map_dbl_fn:nn}.

\section{Scan marks}

Scan marks are control sequences set equal to \texttt{\scan_stop}; hence never expand in an expansion context and are (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 \texttt{l3regex}).

\begin{verbatim}
\scan_new:N \scan_stop
\end{verbatim}

\begin{verbatim}
\scan_new:N \scan_stop: \scan_stop
\end{verbatim}

Creates a new \texttt{scan mark} which is set equal to \texttt{\scan_stop;}. The \texttt{scan mark} is defined globally, and an error message is 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 \textit{(tokens)} and \s_stop from the input stream. This leads to a low-level \TeX
error if \s_stop is absent.
Part X

The \texttt{l3seq} package

Sequences and stacks

\LaTeX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any \textit{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

\begin{itemize}
\item \texttt{\seq_new:N} \langle sequence \rangle
\end{itemize}

Creates a new \langle sequence \rangle or raises an error if the name is already taken. The declaration is global. The \langle sequence \rangle initially contains no items.

\begin{itemize}
\item \texttt{\seq_clear:N} \langle sequence \rangle
\end{itemize}

Clears all items from the \langle sequence \rangle.

\begin{itemize}
\item \texttt{\seq_new:N} \langle sequence \rangle
\end{itemize}

Ensures that the \langle sequence \rangle exists globally by applying \texttt{\seq_new:N} if necessary, then applies \texttt{\seq_(g)clear:N} to leave the \langle sequence \rangle empty.

\begin{itemize}
\item \texttt{\seq_set_eq:NN} \langle sequence1 \rangle \langle sequence2 \rangle
\end{itemize}

Sets the content of \langle sequence1 \rangle equal to that of \langle sequence2 \rangle.

\begin{itemize}
\item \texttt{\seq_set_from_clist:NN} \langle sequence \rangle \langle comma list \rangle
\end{itemize}

Converts the data in the \langle comma list \rangle into a \langle sequence \rangle: the original \langle comma list \rangle is unchanged.
\seq_const_from_clist:Nn \seq_const_from_clist:cn
\seq_set_split:Nnn \seq_set_split:NN \seq_gset_split:Nnn \seq_gset_split:NN
\seq_concat:NNN \seq_concat:ccc \seq_gconcat:NNN \seq_gconcat:ccc
\seq_if_exist_p:N \seq_if_exist_p:c \seq_if_exist:NTF \seq_if_exist:cTF
\seq_put_left:Nn \seq_put_left:NV \seq_gput_left:Nn \seq_gput_left:NV
\seq_put_right:Nn \seq_put_right:NV \seq_gput_right:Nn \seq_gput_right:NV
\seq_put_left:NN \seq_put_left:NV \seq_gput_left:NN \seq_gput_left:NV
\seq_put_right:NN \seq_put_right:NV \seq_gput_right:NN \seq_gput_right:NV

\seq_const_from_clist:Nn \seq_const_from_clist:cn

Creates a new constant \seq_var{} or raises an error if the name is already taken. The \seq_var{} is set globally to contain the items in the comma list.

\seq_set_split:Nnn \seq_set_split:NN \seq_gset_split:Nnn \seq_gset_split:NN

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{} \token_list{}. 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

Concatenates the content of \sequence_2{} and \sequence_3{} together and saves the result in \sequence_1{}. The items in \sequence_2{} are 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

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 \seq_gput_left:Nn \seq_gput_left:NV

Appends the \item{} to the left of the \sequence{}.

\seq_put_right:Nn \seq_put_right:NV \seq_gput_right:Nn \seq_gput_right:NV

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 (sequence) (token list variable)
Stores the left-most item from a \langle sequence \rangle in the \langle token list variable \rangle without removing it from the \langle sequence \rangle. The \langle token list variable \rangle is assigned locally. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_get_right:NN (sequence) (token list variable)
Stores the right-most item from a \langle sequence \rangle in the \langle token list variable \rangle without removing it from the \langle sequence \rangle. The \langle token list variable \rangle is assigned locally. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_pop_left:NN (sequence) (token list variable)
Pops the left-most item from a \langle sequence \rangle into the \langle token list variable \rangle, i.e. removes the item from the sequence and stores it in the \langle token list variable \rangle. Both of the variables are assigned locally. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_gpop_left:NN (sequence) (token list variable)
Pops the left-most item from a \langle sequence \rangle into the \langle token list variable \rangle, i.e. removes the item from the sequence and stores it in the \langle token list variable \rangle. The \langle sequence \rangle is modified globally, while the assignment of the \langle token list variable \rangle is local. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_pop_right:NN (sequence) (token list variable)
Pops the right-most item from a \langle sequence \rangle into the \langle token list variable \rangle, i.e. removes the item from the sequence and stores it in the \langle token list variable \rangle. Both of the variables are assigned locally. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_gpop_right:NN (sequence) (token list variable)
Pops the right-most item from a \langle sequence \rangle into the \langle token list variable \rangle, i.e. removes the item from the sequence and stores it in the \langle token list variable \rangle. The \langle sequence \rangle is modified globally, while the assignment of the \langle token list variable \rangle is local. If \langle sequence \rangle is empty the \langle token list variable \rangle is set to the special marker \q_no_value.

\seq_item:Nn \langle sequence \rangle \{ \langle integer expression \rangle \}
Indexing items in the \langle sequence \rangle from 1 at the top (left), this function evaluates the \langle integer expression \rangle and leaves the appropriate item from the sequence in the input stream. If the \langle integer expression \rangle is negative, indexing occurs from the bottom (right) of the sequence. If the \langle integer expression \rangle is larger than the number of items in the \langle sequence \rangle (as calculated by \seq_count:N) then the function expands to nothing.

\TeXHackersNote{The result is returned within the \unexpanded primitive \exp_not:n, which means that the \langle item \rangle does not expand further when appearing in an x-type argument expansion.}
\seq_rand_item:N \seq_rand_item:C \* \new: 2016-12-06

\seq_rand_item:N \seq_rand_item:C \* \new: 2016-12-06

Selects a pseudo-random item of the \textit{sequence}. If the \textit{sequence} is empty the result is empty. This is not available in older versions of Xe\TeX.

\TeXhackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \textit{item} does 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

If the \textit{sequence} is empty, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{sequence} is non-empty, stores the left-most item from the \textit{sequence} in the \textit{token list variable} without removing it from the \textit{sequence}, then leaves the \textit{true code} in the input stream. The \textit{token list variable} is assigned locally.

\seq_get_right:NNTF \seq_get_right:cNTF \new: 2012-05-19

If the \textit{sequence} is empty, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{sequence} is non-empty, stores the right-most item from the \textit{sequence} in the \textit{token list variable} without removing it from the \textit{sequence}, then leaves the \textit{true code} in the input stream. The \textit{token list variable} is assigned locally.

\seq_pop_left:NNTF \seq_pop_left:cNTF \new: 2012-05-14 \updated: 2012-05-19

If the \textit{sequence} is empty, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{sequence} is non-empty, pops the left-most item from the \textit{sequence} in the \textit{token list variable}, i.e. removes the item from the \textit{sequence}, then leaves the \textit{true code} in the input stream. Both the \textit{sequence} and the \textit{token list variable} are assigned locally.

\seq_gpop_left:NNTF \seq_gpop_left:cNTF \new: 2012-05-14 \updated: 2012-05-19

If the \textit{sequence} is empty, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{sequence} is non-empty, pops the left-most item from the \textit{sequence} in the \textit{token list variable}, i.e. removes the item from the \textit{sequence}, then leaves the \textit{true code} in the input stream. The \textit{sequence} is modified globally, while the \textit{token list variable} is assigned locally.
If the \(\langle\text{sequence}\rangle\) is empty, leaves the \(\langle\text{false code}\rangle\) in the input stream. The value of the \(\langle\text{token list variable}\rangle\) is not defined in this case and should not be relied upon. If the \(\langle\text{sequence}\rangle\) is non-empty, pops the right-most item from the \(\langle\text{sequence}\rangle\) in the \(\langle\text{token list variable}\rangle\), i.e. removes the item from the \(\langle\text{sequence}\rangle\), then leaves the \(\langle\text{true code}\rangle\) in the input stream. Both the \(\langle\text{sequence}\rangle\) and the \(\langle\text{token list variable}\rangle\) are 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.

Removes duplicate items from the \(\langle\text{sequence}\rangle\), leaving the left most copy of each item in the \(\langle\text{sequence}\rangle\). The \(\langle\text{item}\rangle\) comparison takes place on a token basis, as for \(\texttt{tl_if_eq:nnTF}\).

\textit{\TeX}hackers note: This function iterates through every item in the \(\langle\text{sequence}\rangle\) and does a comparison with the \(\langle\text{items}\rangle\) already checked. It is therefore relatively slow with large sequences.

Removes every occurrence of \(\langle\text{item}\rangle\) from the \(\langle\text{sequence}\rangle\). The \(\langle\text{item}\rangle\) comparison takes place on a token basis, as for \(\texttt{tl_if_eq:nnTF}\).

Reverses the order of the items stored in the \(\langle\text{sequence}\rangle\).

Sorts the items in the \(\langle\text{sequence}\rangle\) according to the \(\langle\text{comparison code}\rangle\), and assigns the result to \(\langle\text{sequence}\rangle\). The details of sorting comparison are described in Section 1.
\seq_shuffle:N \seq_shuffle:c \seq_gshuffle:N \seq_gshuffle:c

Sets the \seq{seq var} to the result of placing the items of the \seq{seq var} in a random order. Each item is (roughly) as likely to end up in any given position.

\textbf{\TeX} hackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed \sys_rand_seed: only has 28-bits. The use of \toks internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

## 6 Sequence conditionals

\seq_if_empty_p:N \seq_if_empty_p:c \seq_if_empty:NTF \seq_if_empty:cTF

Tests if the \seq{sequence} is empty (containing no items).

\seq_if_in:NnTF \seq_if_in:NV \seq_if_in:Nv \seq_if_in:No \seq_if_in:Nx \seq_if_in:cn \seq_if_in:cV \seq_if_in:cv \seq_if_in:co \seq_if_in:cx

Tests if the \seq{item} is present in the \seq{sequence}.

## 7 Mapping to sequences

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \seq{function} or \seq{code} discussed below remain in effect after the loop.

\seq_map_function:NN \seq_map_function:CN

Applies \seq{function} to every \seq{item} stored in the \seq{sequence}. The \seq{function} will receive one argument for each iteration. The \seq{items} are returned from left to right. To pass further arguments to the \seq{function}, see \seq_map_tokens:Nn. The function \seq_map_inline:Nn is faster than \seq_map_function:NN for sequences with more than about 10 items.

\seq_map_inline:Nn \seq_map_inline:cn

Applies \seq{inline function} to every \seq{item} stored within the \seq{sequence}. The \seq{inline function} should consist of code which will receive the \seq{item} as \#1. The \seq{items} are returned from left to right.
\seq_map_tokens:Nn \seq_map_tokens:cn

Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The ⟨code⟩ receives each item in the ⟨sequence⟩ as two trailing brace groups. For instance,

\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }

expands to twice each item in the ⟨sequence⟩: for each item in \l_my_seq the function \prg_replicate:nn receives 2 and ⟨item⟩ as its two arguments. The function \seq_map_inline:Nn is typically faster but is not expandable.

\seq_map_variable:NNn \seq_map_variable:Ncn

Stores each ⟨item⟩ of the ⟨sequence⟩ in turn in the (token list) ⟨variable⟩ and applies the ⟨code⟩. The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last ⟨item⟩ in the ⟨sequence⟩, or its original value if the ⟨sequence⟩ is empty. The ⟨items⟩ are returned from left to right.

\seq_map_break:

Used to terminate a \seq_map_... function before all entries in the ⟨sequence⟩ have been processed. This normally takes place within a conditional statement, for example

\seq_map_inline:Nn \l_my_seq
{ \str_if_eq:nTF { #1 } { bingo } { \seq_map_break: } { % Do something useful } }

Use outside of a \seq_map_... scenario leads to low level \TeX errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.
\seq_map_break:n \seq_map_break:n \langle code \rangle

Used to terminate a \seq_map_inline:Nn \l_my_seq

\str_if_eq:nnTF { #1 } { bingo } \seq_map_break:n \langle code \rangle

\%

Do something useful

Use outside of a \seq_map_inline:Nn scenario leads to low level \TeX errors.

\textbf{\TeX hackers note}: When the mapping is broken, additional tokens may be inserted before the \langle code \rangle is inserted into the input stream. This depends on the design of the mapping function.

\seq_count:N \seq_count:N

Leaves the number of items in the \langle sequence \rangle in the input stream as an \langle integer denotation \rangle. The total number of items in a \langle sequence \rangle includes those which are empty and duplicates, i.e. every item in a \langle sequence \rangle is unique.

8 Using the content of sequences directly

\seq_use:Nnnn \seq_use:Nnnn \seq_use:Nnnn

Places the contents of the \langle seq var \rangle in the input stream, with the appropriate \langle separator \rangle between the items. Namely, if the sequence has more than two items, the \langle separator between more than two \rangle is placed between each pair of items except the last, for which the \langle separator between final two \rangle is used. If the sequence has exactly two items, then they are placed in the input stream separated by the \langle separator between two \rangle. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is 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- }

inserts “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.

\textbf{\TeX hackers note}: The result is returned within the \texttt{\unexpanded} primitive (\exp_not:n), which means that the \langle items \rangle do not expand further when appearing in an x-type argument expansion.
\seq_use:Nn \seq_use:cn *

\texttt{\seq_use:Nn \seq_use:cn \{\textit{separator}\}}

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

For example,

\begin{verbatim}
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | \{de\} | f }
\seq_use:Nn \l_tmpa_seq { \textit{\texttt{-and-}} }
\end{verbatim}

inserts “a and b and c and de and f” in the input stream.

\textit{\texttt{\TeX}hackers note}: The result is returned within the \texttt{\unexpanded} primitive (\exp_not:n), which means that the \textit{\texttt{items}} do not expand further when appearing in an \texttt{x-type} argument expansion.

\section{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.

\head{read}{\texttt{\seq_get:NN} \texttt{\seq_get:cN}}

Reads the top item from a \textit{sequence} into the \textit{token list variable} without removing it from the \textit{sequence}. The \textit{token list variable} is assigned locally. If \textit{sequence} is empty the \textit{token list variable} is set to the special marker \texttt{\q_no_value}.

\head{pop}{\texttt{\seq_pop:NN} \texttt{\seq_pop:cN}}

Pops the top item from a \textit{sequence} into the \textit{token list variable}. Both of the variables are assigned locally. If \textit{sequence} is empty the \textit{token list variable} is set to the special marker \texttt{\q_no_value}.

\head{global pop}{\texttt{\seq_gpop:NN} \texttt{\seq_gpop:cN}}

Pops the top item from a \textit{sequence} into the \textit{token list variable}. The \textit{sequence} is modified globally, while the \textit{token list variable} is assigned locally. If \textit{sequence} is empty the \textit{token list variable} is set to the special marker \texttt{\q_no_value}.

\head{get with true false}{\texttt{\seq_get:NNTF} \texttt{\seq_get:CNTF}}

If the \textit{sequence} is empty, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{sequence} is non-empty, stores the top item from a \textit{sequence} in the \textit{token list variable} without removing it from the \textit{sequence}. The \textit{token list variable} is assigned locally.
If the \textlangle sequence \rangle is empty, leaves the \textlangle false code \rangle in the input stream. The value of the \textlangle token list variable \rangle is not defined in this case and should not be relied upon. If the \textlangle sequence \rangle is non-empty, pops the top item from the \textlangle sequence \rangle in the \textlangle token list variable \rangle, i.e. removes the item from the \textlangle sequence \rangle. Both the \textlangle sequence \rangle and the \textlangle token list variable \rangle are assigned locally.

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

Adds the \{item\} to the top of the \textlangle sequence \rangle.

## 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 occurrences of a given item. To make sure that a \textlangle sequence variable \rangle only has distinct items, use $\text\textbackslash seq\text\textunderscore remove\text\textunderscore duplicates:N \textlangle sequence variable \rangle$. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set \textlangle seq var \rangle are straightforward. For instance, $\text\textbackslash seq\text\textunderscore count:N \textlangle seq var \rangle$ expands to the number of items, while $\text\textbackslash seq\text\textunderscore if\text\textunderscore in:NnTF \textlangle seq var \rangle \{\textitem\}$ tests if the \textlangle item \rangle is in the set.

Adding an \textlangle item \rangle to a set \textlangle seq var \rangle can be done by appending it to the \textlangle seq var \rangle if it is not already in the \textlangle seq var \rangle:

\begin{verbatim}
\seq_if_in:NnF \textlangle seq var \rangle \{\textitem\}
{ \seq_put_right:Nn \textlangle seq var \rangle \{\textitem\} }
\end{verbatim}

Removing an \textlangle item \rangle from a set \textlangle seq var \rangle can be done using $\text\textbackslash seq\text\textunderscore remove\text\textunderscore all:Nn$, $\text\textbackslash seq\text\textunderscore remove\text\textunderscore all:Nn \textlangle seq var \rangle \{\textitem\}$

The intersection of two sets \textlangle seq var_1 \rangle and \textlangle seq var_2 \rangle can be stored into \textlangle seq var_3 \rangle by collecting items of \textlangle seq var_1 \rangle which are in \textlangle seq var_2 \rangle.
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__\seq var, 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 var 3 \seq var 1 \seq var 2
or by adding items to (a copy of) \seq var 1 one by one
\seq_set_eq:NN \seq var 3 \seq var 1
\seq_map_inline:Nn \seq var 2
{ \seq_remove_all:Nn \seq var 3 {#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 var 3 \seq var 1
\seq_map_inline:Nn \seq var 2
{ \seq_remove_all:Nn \seq var 3 {#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__\seq var, 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__\seq \seq var 1
\seq_map_inline:Nn \seq var 2
{ \seq_remove_all:Nn \l__\seq \seq var 1 } }
\seq_set_eq:NN \seq var 1 \seq var 1
\seq_map_inline:Nn \seq var 2
{ \seq_remove_all:Nn \seq var 3 \seq var 1 } }
\seq_concat:NNN \seq var 3 \seq var 3 \l__\seq

11 Constant and scratch sequences

\c_empty_seq Constant that is always empty.

See: 2012-07-02
Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any \TeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\l_tmpa_seq
\l_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 \TeX3-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

12 Viewing sequences

\seq_show:N \seq_show:N \langle sequence \rangle
Displays the entries in the \langle sequence \rangle in the terminal.

\seq_log:N \seq_log:N \langle sequence \rangle
Writes the entries in the \langle sequence \rangle in the log file.
Part XI
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

\texttt{\textbackslash int\_eval:n \{ (integer expression) \}}

Evaluates the (integer expression) and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results \texttt{-} followed by such a sequence, and \texttt{0} for zero. The (integer expression) should consist, after expansion, of \texttt{+, \ -, \ *, \ /, (, )} and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- \texttt{/} denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds \texttt{2^{31} - 1}, except in the case of scaling operations \texttt{a*b/c}, for which \texttt{a*b} may be arbitrarily large;
- parentheses may not appear after unary \texttt{+} or \texttt{-}, namely placing \texttt{+} ( or \texttt{-} ( at the start of an expression or after \texttt{+, \ -, \ *, \ /} or \texttt{)} leads to an error.

Each integer operand can be either an integer variable (with no need for \texttt{\textbackslash int\_use:N}) or an integer denotation. For example both

\texttt{\textbackslash int\_eval:n \{ 5 + 4 * 3 - ( 3 + 4 * 5 ) \}}

and

\texttt{\tl\_new:N \l\_my\_tl}
\texttt{\tl\_set:Nn \l\_my\_tl \{ 5 \}}
\texttt{\int\_new:N \l\_my\_int}
\texttt{\int\_set:Nn \l\_my\_int \{ 4 \}}
\texttt{\int\_eval:n \{ \l\_my\_tl + \l\_my\_int * 3 - ( 3 + 4 * 5 ) \}}

evaluate to \texttt{-6} because \texttt{\l\_my\_tl} expands to the integer denotation \texttt{5}. As the (integer expression) is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and \texttt{\exp\_not:n} and its variants have no effect while \texttt{\exp\_not:N} may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate \texttt{\int\_eval:n}. The result is not an (internal integer), and therefore requires suitable termination if used in a \TeX{}-style integer assignment.

As all \TeX{} integers, integer operands can also be dimension or skip variables, converted to integers in \texttt{sp}, or octal numbers given as \texttt{\textasciitilde} followed by digits other than 8 and 9, or hexadecimal numbers given as \texttt{\textasciitilde} followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as \texttt{\langle\textit{char}\rangle}.
\int_eval:w \int_eval:w (integer expression)
Evaluates the (integer expression) as described for \int_eval:n. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is \scan_stop: it is removed, otherwise not. Spaces do not terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, \int_eval:w 1\_i+/uni24231/uni24239 expands to 29 since the digit 9 is not part of the expression.

\int_sign:n \int_sign:n {(intexpr)}
Evaluates the (integer expression) then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\int_abs:n \int_abs:n {(integer expression)}
Evaluates the (integer expression) as described for \int_eval:n and leaves the absolute value of the result in the input stream as an (integer denotation) after two expansions.

\int_div_round:nn \int_div_round:nn {(intexpr)} {(intexpr)}
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 \int_div_truncate:nn {(intexpr)} {(intexpr)}
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_max:nn {(intexpr)} {(intexpr)} \int_min:nn \int_min:nn {(intexpr)} {(intexpr)}
Evaluates the (integer expressions) as described for \int_eval:n and leaves either the larger or smaller value in the input stream as an (integer denotation) after two expansions.

\int_mod:nn \int_mod:nn {(intexpr)} {(intexpr)}
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)} {(intexpr)} times \int_div_truncate:nn {(intexpr)} \int_div_truncate:nn {(intexpr)} from \int_div_truncate:nn {(intexpr)}. Thus, the result has the same sign as \int_div_truncate:nn and its absolute value is strictly less than that of \int_div_truncate:nn. 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:N (integer)
Create a new (integer) or raises an error if the name is already taken. The declaration is global. The (integer) is initially equal to 0.
3 Setting and incrementing integers

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

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

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

\int_set_eq:NN \int_set_eq:cn\int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

\int_const:Nn \int_const:cn
\int_zero:N \int_zero:c
\int_gzero:N \int_gzero:c
\int_zero_new:N \int_zero_new:cn \int_gzero_new:N \int_gzero_new:cn

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_zero:N \int_zero:c
Sets (integer) to 0.

\int_zero_new:N \int_zero_new:cn \int_gzero_new:N \int_gzero_new:cn
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 \int_gset_eq:NN \int_gset_eq:cn
Sets the content of (integer)\textsubscript{1} equal to that of (integer)\textsubscript{2}.

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF
Tests whether the (int) is currently defined. This does not check that the (int) really is an integer variable.

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

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

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

\int_const:Nn \int_const:cn
Creates a new constant (integer) or raises an error if the name is already taken. The value of the (integer) is set globally to the (integer expression).

\int_new:N \int_new:cn
Updated: 2011-10-22

\int_zero:N \int_zero:c
\int_gzero:N \int_gzero:c
\int_zero_new:N \int_zero_new:cn \int_gzero_new:N \int_gzero_new:cn

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF

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

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

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

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

\int_if_exist_p:N \int_if_exist_p:cn \int_if_exist:NTF \int_if_exist:cntF
\int_set:Nn \int_set:cn \int_gset:Nn \int_gset:cn

Sets \textit{(integer)} to the value of \textit{(integer expression)}, which must evaluate to an integer (as described for \texttt{\int_eval:n}).

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

Subtracts the result of the \textit{(integer expression)} from the current content of the \textit{(integer)}.

\section{Using integers}

\int_use:N \int_use:c

Recovers the content of an \textit{(integer)} and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an \textit{(integer)} is required (such as in the first and third arguments of \texttt{\int_compare:nNnTF}).

\textbf{\texttt{\TeX} hackers note:} \texttt{\int_use:N} is the \texttt{\TeX} primitive \texttt{\the}: this is one of several I\TeX3 names for this primitive.

\section{Integer expression conditionals}

\int_compare_p:nNn \int_compare:nNnTF

This function first evaluates each of the \textit{(integer expressions)} as described for \texttt{\int_eval:n}. The two results are then compared using the \textit{(relation)}:

\begin{itemize}
  \item Equal =
  \item Greater than >
  \item Less than <
\end{itemize}

This function is less flexible than \texttt{\int_compare:nTF} but around 5 times faster.

91
This function evaluates the \emph{integer expressions} as described for \code{\int_eval:n} and compares consecutive result using the corresponding \emph{relation}, namely it compares \code{\intexpr_1} and \code{\intexpr_2} using the \emph{relation}_1, then \code{\intexpr_2} and \code{\intexpr_3} using the \emph{relation}_2, until finally comparing \code{\intexpr_N} and \code{\intexpr_{N+1}} using the \emph{relation}_N. The test yields \true{} if all comparisons are \true{}. Each \emph{integer expression} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \false{}, then no other \emph{integer expression} is evaluated and no other comparison is performed. The \emph{relations} can be any of the following:

\begin{itemize}
  \item Equal \quad = \text{ or } ==
  \item Greater than or equal to \quad \geq
  \item Greater than \quad >
  \item Less than or equal to \quad \leq
  \item Less than \quad <
  \item Not equal \quad !=
\end{itemize}

This function is more flexible than \code{\int_compare:nNnTF} but around 5 times slower.
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 \(\langle code\rangle\) is left in the input stream and other cases are discarded. If any of the cases are matched, the \(\langle true\ code\rangle\) is also inserted into the input stream (after the code for the appropriate case), while if none match then the \(\langle false\ code\rangle\) is inserted. The function \(\texttt{\int_case:nn}\), which does nothing if there is no match, is also available. For example

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

leaves “Medium” in the input stream.

This function first evaluates the \((integer\ expression)\) as described for \(\texttt{\int_eval:n}\). It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

\[
\texttt{\int_do_until:nNnn}\{expr1\} \{relation\} \{expr2\} \{code\}
\]

Places the \(\langle code\rangle\) in the input stream for \TeX{} to process, and then evaluates the relationship between the two \(\langle integer\ expressions\rangle\) as described for \(\texttt{\int_compare:nNnTF}\). If the test is \texttt{false} then the \(\langle code\rangle\) is inserted into the input stream again and a loop occurs until the \(\langle relation\rangle\) is \texttt{true}.

\[
\texttt{\int_do_while:nNnn}\{expr1\} \{relation\} \{expr2\} \{code\}
\]

Places the \(\langle code\rangle\) in the input stream for \TeX{} to process, and then evaluates the relationship between the two \(\langle integer\ expressions\rangle\) as described for \(\texttt{\int_compare:nNnTF}\). If the test is \texttt{true} then the \(\langle code\rangle\) is inserted into the input stream again and a loop occurs until the \(\langle relation\rangle\) is \texttt{false}.
\texttt{\textbackslash int\_until\_do:nNnn} \texttt{\textbackslash int\_until\_do:nNnn \{(intexpr\_1\}\) \{relation\} \{(intexpr\_2\}\) \{(code\}\)\texttt{\textbackslash int\_until\_do:nNnn} \texttt{\textbackslash int\_until\_do:nNnn}\text{ Evaluates the relationship between the two \textit{integer expressions} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is false. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is true.}\texttt{\textbackslash int\_while\_do:nNnn} \texttt{\textbackslash int\_while\_do:nNnn \{(intexpr\_1\}\) \{relation\} \{(intexpr\_2\}\) \{(code\}\)\texttt{\textbackslash int\_while\_do:nNnn} \texttt{\textbackslash int\_while\_do:nNnn}\text{ Evaluates the relationship between the two \textit{integer expressions} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is true. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is false.}\texttt{\textbackslash int\_do\_until:nn} \texttt{\textbackslash int\_do\_until:nn \{(integer relation\)} \{(code\)} \texttt{\textbackslash int\_do\_until:nn} \texttt{\textbackslash int\_do\_until:nn}\text{ Places the \texttt{(code)} in the stream for \TeX{} to process, and then evaluates the \texttt{(integer relation)} as described for \texttt{\textbackslash int\_compare:nNnTF}. If the test is false then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is true.}\texttt{\textbackslash int\_do\_while:nn} \texttt{\textbackslash int\_do\_while:nn \{(integer relation\)} \{(code\)} \texttt{\textbackslash int\_do\_while:nn} \texttt{\textbackslash int\_do\_while:nn}\text{ Places the \texttt{(code)} in the stream for \TeX{} to process, and then evaluates the \texttt{(integer relation)} as described for \texttt{\textbackslash int\_compare:nNnTF}. If the test is true then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is false.}\texttt{\textbackslash int\_until\_do:nn} \texttt{\textbackslash int\_until\_do:nn \{(integer relation\)} \{(code\)} \texttt{\textbackslash int\_until\_do:nn} \texttt{\textbackslash int\_until\_do:nn}\text{ Evaluates the \texttt{(integer relation)} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is false. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is true.}\texttt{\textbackslash int\_while\_do:nn} \texttt{\textbackslash int\_while\_do:nn \{(integer relation\)} \{(code\)} \texttt{\textbackslash int\_while\_do:nn} \texttt{\textbackslash int\_while\_do:nn}\text{ Evaluates the \texttt{(integer relation)} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is true. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is false.
7 Integer step functions

\int_step_function:nN \star \int_step_function:nnN \star \int_step_function:nnnN

Rev: 2012-06-04
Updated: 2018-04-22

This function first evaluates the \langle initial value\rangle, \langle step\rangle and \langle final value\rangle, all of which should be integer expressions. The \langle function\rangle is then placed in front of each \langle value\rangle from the \langle initial value\rangle to the \langle final value\rangle in turn (using \langle step\rangle between each \langle value\rangle). The \langle step\rangle must be non-zero. If the \langle step\rangle is positive, the loop stops when the \langle value\rangle becomes larger than the \langle final value\rangle. If the \langle step\rangle is negative, the loop stops when the \langle value\rangle becomes smaller than the \langle final value\rangle. The \langle function\rangle 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]

The functions \int_step_function:nN and \int_step_function:nnN both use a fixed \langle step\rangle of 1, and in the case of \int_step_function:nN the \langle initial value\rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_inline:nn
\int_step_inline:nn
\int_step_inline:nn

Rev: 2012-06-04
Updated: 2018-04-22

This function first evaluates the \langle initial value\rangle, \langle step\rangle and \langle final value\rangle, all of which should be integer expressions. Then for each \langle value\rangle from the \langle initial value\rangle to the \langle final value\rangle in turn (using \langle step\rangle between each \langle value\rangle), the \langle code\rangle is inserted into the input stream with \#1 replaced by the current \langle value\rangle. Thus the \langle code\rangle should define a function of one argument (\#1).

The functions \int_step_inline:nn and \int_step_inline:nn both use a fixed \langle step\rangle of 1, and in the case of \int_step_inline:nn the \langle initial value\rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_variable:nN
\int_step_variable:nN
\int_step_variable:nnnN

Rev: 2012-06-04
Updated: 2018-04-22

This function first evaluates the \langle initial value\rangle, \langle step\rangle and \langle final value\rangle, all of which should be integer expressions. Then for each \langle value\rangle from the \langle initial value\rangle to the \langle final value\rangle in turn (using \langle step\rangle between each \langle value\rangle), the \langle code\rangle is inserted into the input stream, with the \langle tl var\rangle defined as the current \langle value\rangle. Thus the \langle code\rangle should make use of the \langle tl var\rangle.

The functions \int_step_variable:nN and \int_step_variable:nnnN both use a fixed \langle step\rangle of 1, and in the case of \int_step_variable:nN the \langle initial value\rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.
8 Formatting integers

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

\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 \{ 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 \{ integer expression \} \{ total symbols \} \{ value to symbol mapping \}
This is the low-level function for conversion of an \{ integer expression \} into a symbolic form (often 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 \}
 \ldots
 \{ 26 \} \{ z \}}
\texttt{\int_to_bin:n} \texttt{\{\langle integer expression\rangle\}}
\begin{flushleft}
Calculates the value of the \langle integer expression\rangle and places the binary representation of the result in the input stream.
\end{flushleft}
\texttt{\int_to_hex:n} \texttt{\{\langle integer expression\rangle\}}
\begin{flushleft}
Calculates the value of the \langle integer expression\rangle 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 \texttt{\int_to_hex:n} and upper case ones for \texttt{\int_to_Hex:n}. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
\end{flushleft}
\texttt{\int_to_oct:n} \texttt{\{\langle integer expression\rangle\}}
\begin{flushleft}
Calculates the value of the \langle integer expression\rangle 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).
\end{flushleft}
\texttt{\int_to_base:nn} \texttt{\{\langle integer expression\rangle\}} \texttt{\{\langle base\rangle\}}
\begin{flushleft}
Calculates the value of the \langle integer expression\rangle and converts it into the appropriate representation in the \langle base\rangle; the latter 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 \texttt{\int_to_base:n} and upper case ones for \texttt{\int_to_Base:n}. The maximum \langle base\rangle value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
\end{flushleft}
\texttt{\int_to_roman:n} \texttt{\{\langle integer expression\rangle\}}
\begin{itemize}
\item \texttt{\int_to_roman:n}
\item \texttt{\int_to_Roman:n}
\end{itemize}
\begin{flushleft}
Places the value of the \langle integer expression\rangle in the input stream as Roman numerals, either lower case \texttt{\int_to_roman:n} or upper case \texttt{\int_to_Roman:n}. If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are \texttt{mdclxvi}, repeated as needed: the notation with bars (such as \texttt{\overline{v}} for 5000) is not used. For instance \texttt{\int_to_roman:n \{ 8249 \} } expands to \texttt{mmmmmmmcxlix}.
\end{flushleft}
\texttt{\int_from_alph:n} \texttt{\{\langle letters\rangle\}}
\begin{itemize}
\item \texttt{\int_from_alph:n}
\item \texttt{\int_from_Alph:n}
\end{itemize}
\begin{flushleft}
Converts the \langle letters\rangle into the integer (base 10) representation and leaves this in the input stream. The \langle letters\rangle 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 \texttt{\int_to_alph:n} and \texttt{\int_to_Alph:n}.
\end{flushleft}

9 Converting from other formats to integers
\textbf{10 Random integers}

\textbf{\textbackslash int\_from\_bin:n \star}
\textit{\textbackslash int\_from\_bin:n \langle binary number \rangle}
\textbackslash int\_from\_bin:n

Converts the \langle binary number \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle binary number \rangle 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 \textbackslash int\_to\_bin:n.

\textbf{\textbackslash int\_from\_hex:n \star}
\textit{\textbackslash int\_from\_hex:n \langle hexadecimal number \rangle}
\textbackslash int\_from\_hex:n

Converts the \langle hexadecimal number \rangle into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the \langle hexadecimal number \rangle by upper or lower case letters. The \langle hexadecimal number \rangle 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 \textbackslash int\_to\_hex:n and \textbackslash int\_to\_Hex:n.

\textbf{\textbackslash int\_from\_oct:n \star}
\textit{\textbackslash int\_from\_oct:n \langle octal number \rangle}
\textbackslash int\_from\_oct:n

Converts the \langle octal number \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle octal number \rangle 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 \textbackslash int\_to\_oct:n.

\textbf{\textbackslash int\_from\_roman:n \star}
\textit{\textbackslash int\_from\_roman:n \langle roman numeral \rangle}
\textbackslash int\_from\_roman:n

Converts the \langle roman numeral \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle roman numeral \rangle is first converted to a string, with no expansion. The \langle roman numeral \rangle may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value is −1. This is the inverse function of \textbackslash int\_to\_roman:n and \textbackslash int\_to\_Roman:n.

\textbf{\textbackslash int\_from\_base:nn \star}
\textit{\textbackslash int\_from\_base:nn \langle number \rangle \{\langle base \rangle\}}
\textbackslash int\_from\_base:nn

Converts the \langle number \rangle expressed in \langle base \rangle into the appropriate value in base 10. The \langle number \rangle is first converted to a string, with no expansion. The \langle number \rangle should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum \langle base \rangle value is 36. This is the inverse function of \textbackslash int\_to\_base:nn and \textbackslash int\_to\_Base:nn.

\textbf{\textbackslash int\_rand:nn \star}
\textit{\textbackslash int\_rand:nn \langle intexpr1 \rangle \{\langle intexpr2 \rangle\}}
\textbackslash int\_rand:nn

Evaluates the two \langle integer expressions \rangle and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X\texttt{e}\TeX.

\textbf{\textbackslash int\_rand:n \star}
\textit{\textbackslash int\_rand:n \langle intexpr \rangle}
\textbackslash int\_rand:n

Evaluates the \langle integer expression \rangle then produces a pseudo-random number between 1 and the \langle intexpr \rangle (included). This is not available in older versions of X\texttt{e}\TeX.
11 Viewing integers

\int_show:N \langle integer \rangle
Displays the value of the \langle integer \rangle on the terminal.

\int_show:n \{\langle integer expression \rangle\}
Displays the result of evaluating the \langle integer expression \rangle on the terminal.

\int_log:N \langle integer \rangle
\int_log:c
\int_log:n \{\langle integer expression \rangle\}
Writes the value of the \langle integer \rangle in the log file.
Writes the result of evaluating the \langle integer expression \rangle in the log file.

12 Constant integers

\c_zero_int \c_one_int
New: 2018-05-07

The maximum value that can be stored as an integer.

\c_max_int

Maximum number of registers.

\c_max_char_int
Maximum character code completely supported by the engine.

13 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.
13.1 Direct number expansion

\int_value:w * \int_value:w \langle integer \rangle
\int_value:w \langle integer denotation \rangle \langle optional space \rangle

Expands the following tokens until an \langle integer \rangle is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The \langle integer \rangle can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any \TeX register except \texttt{\toks}) or
- explicit digits (or by \texttt{\langle octal digits \rangle} or \texttt{\langle hexadecimal digits \rangle} or \texttt{\langle character \rangle}).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in f-expansion, and so \texttt{\exp_stop_f:} may be employed as an end marker. Note that protected functions are expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, \texttt{\int_eval:n} is the preferred approach to generating numbers.

\TeXhackers note: This is the \TeX primitive \texttt{\number}.

14 Primitive conditionals

\if_int_compare:w * \if_int_compare:w \langle integer_1 \rangle \langle relation \rangle \langle integer_2 \rangle
\langle true code \rangle
\else:\langle false code \rangle
\fi:

Compare two integers using \langle relation \rangle, which must be one of \texttt{=} \texttt{<} or \texttt{>} with category code 12. The \texttt{\else:} branch is optional.

\TeXhackers note: These are both names for the \TeX primitive \texttt{\ifnum}.

\if_case:w * \if_case:w \langle integer \rangle \langle case_0 \rangle
\or: \langle case_1 \rangle
\or: ...\langle case_i \rangle
\or: \langle default \rangle
\fi:

Selects a case to execute based on the value of the \langle integer \rangle. The first case (\langle case_0 \rangle) is executed if \langle integer \rangle is 0, the second (\langle case_1 \rangle) if the \langle integer \rangle is 1, etc. The \langle integer \rangle may be a literal, a constant or an integer expression (e.g. using \texttt{\int_eval:n}).

\TeXhackers note: These are the \TeX primitives \texttt{\ifcase} and \texttt{\or}.
\if_int_odd:w * \if_int_odd:w \langle tokens \rangle \ (optional space) \langle true code \rangle \else: \langle true code \rangle \fi:

Expands \langle tokens \rangle until a non-numeric token or a space is found, and tests whether the resulting \langle integer \rangle is odd. If so, \langle true code \rangle is executed. The \else: branch is optional.

\TeXhackers note: This is the \TeX primitive \ifodd.
Part XII
The l3flag package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its \(\text{height}\). In expansion-only contexts, a flag can only be “raised”: this increases the \(\text{height}\) by 1. The \(\text{height}\) can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a \(\text{name}\) such as \texttt{str\_missing}. The \(\text{name}\) is used as part of \texttt{use:\_c constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.}

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by \texttt{l3str\_convert}, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

1 Setting up flags

\begin{verbatim}
\flag_new:n \flag_new:n \{\text{name}\}\n\end{verbatim}

Creates a new flag with a name given by \(\text{name}\), or raises an error if the name is already taken. The \(\text{name}\) may not contain spaces. The declaration is global, but flags are always local variables. The \(\text{flag}\) initially has zero height.

\begin{verbatim}
\flag_clear:n \flag_clear:n \{\text{name}\}\n\end{verbatim}

The \(\text{flag}\)'s height is set to zero. The assignment is local.

\begin{verbatim}
\flag_clear_new:n \flag_clear_new:n \{\text{name}\}\n\end{verbatim}

Ensures that the \(\text{flag}\) exists globally by applying \texttt{\flag_new:n} if necessary, then applies \texttt{\flag_clear:n}, setting the height to zero locally.

\begin{verbatim}
\flag_show:n \flag_show:n \{\text{name}\}\n\end{verbatim}

Displays the \(\text{flag}\)'s height in the terminal.

\begin{verbatim}
\flag_log:n \flag_log:n \{\text{name}\}\n\end{verbatim}

Writes the \(\text{flag}\)'s height to the log file.
2 Expandable flag commands

\flag_if_exist:p:n ∗ \flag_if_exist:n \{⟨flag name⟩\}
This function returns true if the ⟨flag name⟩ references a flag that has been defined previously, and false otherwise.

\flag_if_exist:p:n \# \flag_if_exist:n \{⟨flag name⟩\} ∗

\flag_if_raised:p:n ∗ \flag_if_raised:n \{⟨flag name⟩\}
This function returns true if the ⟨flag⟩ has non-zero height, and false if the ⟨flag⟩ has zero height.

\flag_if_raised:p:n \# \flag_if_raised:n \{⟨flag name⟩\} ∗

\flag_height:n \{⟨flag name⟩\}
Expands to the height of the ⟨flag⟩ as an integer denotation.

\flag_height:n ∗

\flag_raise:n \{⟨flag name⟩\}
The ⟨flag⟩’s height is increased by 1 locally.

\flag_raise:n ∗
Part XIII
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 \texttt{(true)} and \texttt{(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 \texttt{(true)} or \texttt{(false)}. For example, the function \texttt{\cs_if_free_p:N} checks whether the control sequence given as its argument is free and then returns the boolean \texttt{(true)} or \texttt{(false)} values to be used in testing with \texttt{\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 \texttt{\cs_if_free:NTF} which also takes one argument (the $N$) and then executes either \texttt{true} or \texttt{false} depending on the result.

\TeX\text{hackers note:} The arguments are executed after exiting the underlying \texttt{\if...\fi:} structure.

### 1 Defining a set of conditional functions

\begin{verbatim}
\prg_new_conditional:Npnn \prg_new_protected_conditional:Npnn
\prg_set_conditional:Npnn \prg_set_protected_conditional:Npnn
\end{verbatim}

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

\begin{verbatim}
\prg_new_protected_conditional:Npnn \prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn \prg_set_protected_conditional:Npnn
\end{verbatim}

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

The conditionals are defined by \texttt{\prg_new_conditional:Npnn} and friends as:
- \(\langle\text{name}\rangle\_p:\langle\text{arg spec}\rangle\) — a predicate function which will supply either a logical \text{true} or logical \text{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 \text{protected} conditionals.

- \(\langle\text{name}\rangle:\langle\text{arg spec}\rangle T\) — a function with one more argument than the original \langle\text{arg spec}\rangle demands. The \text{true branch} code in this additional argument will be left on the input stream only if the test is \text{true}.

- \(\langle\text{name}\rangle:\langle\text{arg spec}\rangle F\) — a function with one more argument than the original \langle\text{arg spec}\rangle demands. The \text{false branch} code in this additional argument will be left on the input stream only if the test is \text{false}.

- \(\langle\text{name}\rangle:\langle\text{arg spec}\rangle TF\) — a function with two more argument than the original \langle\text{arg spec}\rangle demands. The \text{true branch} code in the first additional argument will be left on the input stream if the test is \text{true}, while the \text{false branch} code in the second argument will be left on the input stream if the test is \text{false}.

The \text{(code)} of the test may use \text{(parameters)} as specified by the second argument to \textbackslash{prg\_set\_conditional:Npnn}: this should match the \text{(argument specification)} but this is not enforced. The \text{Nnn} versions infer the number of arguments from the argument specification given (\textit{cf.} \textbackslash{cs\_new:Nn}, etc.). Within the \text{(code)}, the functions \textbackslash{prg\_return\_true}: and \textbackslash{prg\_return\_false}: are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

\textbackslash{prg\_set\_conditional:Npnn} \textbackslash{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 \textbackslash{foo\_if\_bar\_p:NN}, \textbackslash{foo\_if\_bar\_NNTF} and \textbackslash{foo\_if\_bar\_NNT} but not \textbackslash{foo\_if\_bar\_NF} (because \text{F} is missing from the \text{(conditions)} list). The return statements take care of resolving the remaining \text{\else:} and \text{\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.

\textbackslash{prg\_new\_eq\_conditional:Nnn} \textbackslash{\langle\text{name1}\rangle:(\text{arg spec1}) \langle\text{name2}\rangle:(\text{arg spec2}) \langle\text{conditions}\rangle}

These functions copy a family of conditionals. The \text{new} version checks for existing definitions (\textit{cf.} \textbackslash{cs\_new\_eq:Nn}) whereas the \text{set} version does not (\textit{cf.} \textbackslash{cs\_set\_eq:Nn}). The conditionals copied are depended on the comma-separated list of \text{(conditions)}, which should be one or more of \text{p}, \text{T}, \text{F} and \text{TF}.  

105
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.

\prg_return_true: ⋆ \prg_return_false: ⋆

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 generally means being constructed from predicate functions and booleans, possibly nested).

\textbf{TeX hackers note:} The \texttt{bool} data type is not implemented using the \texttt{iffalse/ifftrue} primitives, in contrast to \newif, etc., in plain \TeX, \LaTeX{} and so on. Programmers should not base use of \texttt{bool} switches on any particular expectation of the implementation.

\bool_new:N \bool_new:c

Creates a new \langle boolean\rangle or raises an error if the name is already taken. The declaration is global. The \langle boolean\rangle is initially \texttt{false}. 

106
\bool_const:Nn
\bool_const:cn

\bool_const:Nn \{\boolean\} \{\boolexpr\}

Creates a new constant \{\boolean\} or raises an error if the name is already taken. The value of the \{\boolean\} is set globally to the result of evaluating the \{\boolexpr\}.

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

\bool_set_false:N \{\boolean\}

Sets \{\boolean\} logically false.

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

\bool_set_true:N \{\boolean\}

Sets \{\boolean\} logically true.

\bool_set_eq:NN
\bool_set_eq:{cN|Nc|cc}
\bool_gset_eq:NN
\bool_gset_eq:{cN|Nc|cc}

\bool_set_eq:NN \{\boolean_1\} \{\boolean_2\}

Sets \{\boolean_1\} to the current value of \{\boolean_2\}.

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

\bool_set:Nn \{\boolean\} \{\boolexpr\}

Evaluates the \{\boolean expression\} as described for \bool_if: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
\bool_show:n
\bool_show:cn

\bool_show:N \{\boolean\}

Displays the logical truth of the \{\boolean\} on the terminal.

\bool_show:n \{\boolean expression\}

Displays the logical truth of the \{\boolean expression\} on the terminal.

\bool_log:N
\bool_log:c
\bool_log:n
\bool_log:cn

\bool_log:N \{\boolean\}

Writes the logical truth of the \{\boolean\} in the log file.
\bool_log:n \{ ⟨boolean expression⟩ \}

Writes the logical truth of the ⟨boolean expression⟩ in the log file.

\bool_if_exist_p:N \{ ⟨boolean⟩ \}
\bool_if_exist:NTF \{ ⟨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.

\section{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 || and prefix !
with their usual precedences (namely, && binds more tightly than ||). In addition to
this, parentheses can be used to isolate sub-expressions. For example,

\begin{verbatim}
\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 }
\end{verbatim}

is a valid boolean expression.

Contrarily to some other programming languages, the operators && and || evaluate
both operands in all cases, even when the first operand is enough to determine the result.
This “eager” evaluation should be contrasted with the “lazy” evaluation of \bool_lazy_- 
... functions.

\TeXhacks {m note: The eager evaluation of boolean expressions is unfortunately necessary
in \TeX. Indeed, a lazy parser can get confused if && or || or parentheses appear as (unbraced)
arguments of some predicates. For instance, the innocuous-looking expression below would break
(in a lazy parser) if \#1 were a closing parenthesis and \l_tmpa_bool were true.

\begin{verbatim}
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X \#1 )
\end{verbatim}
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

```latex
\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 } % skipped
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `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`.

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.

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` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two `<boolean expressions>`.

Implements the “And” operation between two boolean expressions, hence is true if both are true. Contrarily to the infix operator `&&`, the `<boolean expr>` is only 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>`.
4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_until_do:Nn \*
\bool_until_do:cn \*

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_while:Nn \*
\bool_do_while:cn \*

\bool_do_while:Nn \* \bool_do_while:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \*
\bool_do_until:cn \*

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
Updated: 2017-07-15

\bool_until_do:Nn \* \bool_until_do:cn \*
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn \*
\bool_do_until:nn \{boolean \ expression\} \{\langle \ association \ expression\rangle \}

Places the \langle \ association \ expression\rangle in the input stream for \TeX{} to process, and then checks the logical value of the \langle boolean \ expression\rangle as described for \bool_if:nTF. If it is false then the \langle \ association \ expression\rangle is inserted into the input stream again and the process loops until the \langle boolean \ expression\rangle evaluates to true.

\bool_do_while:nn \{boolean \ expression\} \{\langle \ association \ expression\rangle \}

Places the \langle \ association \ expression\rangle in the input stream for \TeX{} to process, and then checks the logical value of the \langle boolean \ expression\rangle as described for \bool_if:nTF. If it is true then the \langle \ association \ expression\rangle is inserted into the input stream again and the process loops until the \langle boolean \ expression\rangle evaluates to false.

\bool_until_do:nn \{boolean \ expression\} \{\langle \ association \ expression\rangle \}

This function firsts checks the logical value of the \langle boolean \ expression\rangle (as described for \bool_if:nTF). If it is false the \langle \ association \ expression\rangle is placed in the input stream and expanded. After the completion of the \langle \ association \ expression\rangle the truth of the \langle boolean \ expression\rangle is re-evaluated. The process then loops until the \langle boolean \ expression\rangle is true.

\bool_while_do:nn \{boolean \ expression\} \{\langle \ association \ expression\rangle \}

This function firsts checks the logical value of the \langle boolean \ expression\rangle (as described for \bool_if:nTF). If it is true the \langle \ association \ expression\rangle is placed in the input stream and expanded. After the completion of the \langle \ association \ expression\rangle the truth of the \langle boolean \ expression\rangle is re-evaluated. The process then loops until the \langle boolean \ expression\rangle is false.

5 Producing multiple copies

\prg_replicate:nn \{integer \ expression\} \{\langle \ token \ list\rangle \}

Evaluates the \langle integer \ expression\rangle (which should be zero or positive) and creates the resulting number of copies of the \langle token \ list\rangle. 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 \{\langle \ true \ association \ expression\rangle\} \{\langle \ false \ association \ expression\rangle\}

Detects if \TeX{} is currently in horizontal mode.

\mode_if_inner_p: \mode_if_inner:TF \{\langle \ true \ association \ expression\rangle\} \{\langle \ false \ association \ expression\rangle\}

Detects if \TeX{} is currently in inner mode.

\mode_if_math_p: \mode_if_math:TF \{\langle \ true \ association \ expression\rangle\} \{\langle \ false \ association \ expression\rangle\}

Detects if \TeX{} is currently in maths mode.
Detects if \TeX{} is currently in vertical mode.

7 Primitive conditionals

\if_predicate:w \langle predicate \rangle \{ \langle true \rangle \} \{ \langle false \rangle \} \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 \langle predicate \rangle but to make the coding clearer this should be done through \if_bool:N.)

\if_bool:N \langle boolean \rangle \{ \langle true \rangle \} \{ \langle false \rangle \} \fi:

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

8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in expl3. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

\prg_break_point:Nn \langle type \rangle _map_break: \{ \langle code \rangle \}

Used to mark the end of a recursion or mapping: the functions \langle type \rangle _map_break: and \langle type \rangle _map_break:n use this to break out of the loop (see \prg_map_break:Nn for how to set these up). After the loop ends, the \langle code \rangle is 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 \langle type \rangle _map_break: \{ \langle user code \rangle \}

\prg_break_point:Nn \langle type \rangle _map_break: \{ \langle ending code \rangle \}

Breaks a recursion in mapping contexts, inserting in the input stream the \langle user code \rangle after the \langle ending code \rangle for the loop. The function breaks loops, inserting their \langle ending code \rangle, until reaching a loop with the same \langle type \rangle as its first argument. This \langle type \rangle _-map_break: argument must be defined; it is simply used as a recognizable marker for the \langle type \rangle.

For types with mappings defined in the kernel, \langle type \rangle _map_break: and \langle type \rangle _-map_break:n are defined as \prg_map_break:Nn \langle type \rangle _map_break: {} and the same with {} omitted.

8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.
\prg_break_point: * New: 2018-03-27

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

\prg_break: * New: 2018-03-27
\prg_break:n *

\prg_break:n \langle code \rangle \ldots \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 the \langle code \rangle in the input stream.

\group_align_safe_begin: * Updated: 2011-08-11
\group_align_safe_end: *

These functions are used to enclose material in a \TeX{} alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the \& token inside \halign. This is necessary to allow grabbing of tokens for testing purposes, as \TeX{} uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as \peek_after:Nw would 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.

9 Internal programming functions
Part XIV

The \texttt{l3sys} package: System/runtime functions

1. The name of the job

\begin{verbatim}
\c_sys_jobname_str
\end{verbatim}

Constant that gets the “job name” assigned when \TeX{} starts.

\textbf{\TeX{}hackers note:} This copies the contents of the primitive \jobname. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

2. Date and time

\begin{verbatim}
\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int
\end{verbatim}

The date and time at which the current job was started: these are all reported as integers.

\textbf{\TeX{}hackers 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.

3. Engine

\begin{verbatim}
\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 *
\end{verbatim}

Conditional 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 \texttt{\textepsilon}-p\TeX{} and \texttt{\textepsilon}-up\TeX{} as expl3 requires the \texttt{\textepsilon}-\TeX{} extensions. Each conditional is true for \textit{exactly one} supported engine. In particular, \texttt{\sys_if_engine_ptex_p:} is true for \texttt{\textepsilon}-p\TeX{} but false for \texttt{\textepsilon}-up\TeX{}.

\begin{verbatim}
\c_sys_engine_str
\end{verbatim}

The current engine given as a lower case string: one of \texttt{luatex, pdftex, ptex, uptex or xetex}.
4 Output format

\sys_if_output_dvi:TF \{true code\} \{false code\}

Conditionals which give the current output mode the \TeX run is operating in. This is always 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.

The current output mode given as a lower case string: one of dvi or pdf.

5 Platform

\sys_if_platform_unix:TF \{true code\} \{false code\}

Conditionals which allow platform-specific code to be used. The names follow the Lua os.type() function, i.e. all Unix-like systems are unix (including Linux and MacOS).

The current platform given as a lower case string: one of unix, windows or unknown.

6 Random numbers

\sys_rand_seed:

Expands to the current value of the engine’s random seed, a non-negative integer. In engines without random number support this expands to 0.

\sys_gset_rand_seed:n \{intexpr\}

Globally sets the seed for the engine’s pseudo-random number generator to the \textit{integer expression}. This random seed affects all \ldots_rand functions (such as \int_rand:nn or \clist_rand_item:n) as well as other packages relying on the engine’s random number generator. In engines without random number support this produces an error.

\textbf{\TeX hackers note:} While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond \(2^{28}\) is divided by an appropriate power of 2. We recommend using an integer in \([0, 2^{28} − 1] \).
7 Access to the shell

\sys_get_shell::nnN
\sys_get_shell::nnNTF

\sys_get_shell::nnN {(shell command)} ⟨(setup)⟩ ⟨tl var⟩
\sys_get_shell::nnNTF {(shell command)} ⟨(setup)⟩ ⟨tl var⟩ ⟨{true code}⟩ ⟨{false code}⟩

Defines ⟨tl⟩ to the text returned by the ⟨shell command⟩. The ⟨shell command⟩ is converted to a string using \tl_to_str:n. Category codes may need to be set appropriately via the ⟨setup⟩ argument, which is run just before running the ⟨shell command⟩ (in a group). If shell escape is disabled, the ⟨tl var⟩ will be set to \q_no_value in the non-branching version. Note that quote characters (“”) cannot be used inside the ⟨shell command⟩. The \sys_get_shell::nnNTF conditional returns true if the shell is available and no quote is detected, and false otherwise.

This variable exposes the internal triple of the shell escape status. The possible values are:

0 Shell escape is disabled
1 Unrestricted shell escape is enabled
2 Restricted shell escape is enabled

\sys_if_shell::p: \sys_if_shell::TF
\sys_if_shell_unrestricted::p: \sys_if_shell_unrestricted::TF
\sys_if_shell_restricted::p: \sys_if_shell_restricted::TF

\sys_if_shell::p: ⟨true code⟩ ⟨false code⟩
\sys_if_shell::TF ⟨true code⟩ ⟨false code⟩
\sys_if_shell_unrestricted::p: ⟨true code⟩ ⟨false code⟩
\sys_if_shell_unrestricted::TF ⟨true code⟩ ⟨false code⟩
\sys_if_shell_restricted::p: ⟨true code⟩ ⟨false code⟩
\sys_if_shell_restricted::TF ⟨true code⟩ ⟨false code⟩

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

Performs a check for whether unrestricted shell escape is enabled.

Performs a check for whether restricted shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use \sys_if_shell::.

\sys_shell_now::n
\sys_shell_now::x

\sys_shell_now::n ⟨tokens⟩
\sys_shell_now::x

Execute ⟨tokens⟩ through shell escape immediately.

\sys_shell_shipout::n
\sys_shell_shipout::x

\sys_shell_shipout::n ⟨tokens⟩
\sys_shell_shipout::x

Execute ⟨tokens⟩ through shell escape at shipout.
7.1 Loading configuration data

\sys_load_backend:n \{\texttt{-backend}\}
\texttt{sys_load_backend:n \{\texttt{backend}\}}

Loads the additional configuration file needed for backend support. If the \texttt{(backend)} is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

\c_sys_backend_str
Set to the name of the backend in use by \sys_load_backend:n when issued.

\sys_load_debug:
\sys_load_deprecation:
Load the additional configuration files for debugging support and rolling back deprecations, respectively.

7.2 Final settings

\sys_finalise:
\sys_finalise:
Finalises all system-dependent functionality: required before loading a backend.
Part XV

The \texttt{l3clist} package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of
the list. This data type allows basic list manipulations such as adding/removing items,
applying a function to every item, removing duplicate items, extracting a given item,
using the comma list with specified separators, and so on. Sequences (defined in \texttt{l3seq})
are safer, faster, and provide more features, so they should often be preferred to comma
lists. Comma lists are mostly useful when interfacing with \LaTeX\ or other code that
expects or provides comma list data.

Several items can be added at once. To ease input of comma lists from data provided
by a user outside an \\texttt{ExplSyntaxOn} ... \texttt{ExplSyntaxOff} block, spaces are removed
from both sides of each comma-delimited argument upon input. Blank arguments are
ignored, to allow for trailing commas or repeated commas (which may otherwise arise
when concatenating comma lists “by hand”). In addition, a set of braces is removed if
the result of space-trimming is braced: this allows the storage of any item in a comma
list. For instance,

\begin{verbatim}
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { -a- , -{b}- , c-\d } \\
\clist_put_right:Nn \l_my_clist { -{e-} , , {{f}} , }
\end{verbatim}

results in \texttt{\l_my_clist} containing \texttt{a,b,c-\d,e-} and \texttt{{f}}. Comma lists normally do not contain empty items so the following
gives an empty comma list:

\begin{verbatim}
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , - , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
\end{verbatim}

and it leaves \texttt{true} in the input stream. To include an “unsafe” item (empty, or one that
contains a comma, or starts or ends with a space, or is a single brace group), surround
it with braces.

Almost all operations on comma lists are noticeably slower than those on sequences
so converting the data to sequences using \texttt{seq_set_from_clist:Nn} (see \texttt{l3seq}) may be
advisable if speed is important. The exception is that \texttt{clist_if_in:NnTF} and \texttt{clist_-}
\texttt{remove_duplicates:N} may be faster than their sequence analogues for large lists. However,
these functions work slowly for “unsafe” items that must be braced, and may pro-
duce errors when their argument contains \{, \} or \# (assuming the usual \TeX\ category
codes apply). In addition, comma lists cannot store quarks \texttt{\q\_mark} or \texttt{\q\_stop}. The
sequence data type should thus certainly be preferred to comma lists to store such items.

1 Creating and initialising comma lists

\begin{verbatim}
\clist_new:N \clist_new:N \clist_new:c
\end{verbatim}

\begin{verbatim}
\clist_new:N \clist_new:N (comma list) \clist_new:c
\end{verbatim}

Creates a new \texttt{(comma list)} or raises an error if the name is already taken. The declaration
is global. The \texttt{(comma list)} initially contains no items.
\clist_const:Nn \clist_const:<(N|c|n|c|x) New: 2014-07-05
\clist_const:Nn \clist_const:<(N|c|n|c|x) \clist const:Nn \clist const:<(N|c|n|c|x)
Creates a new constant \clist const:<(N|c|n|c|x) or raises an error if the name is already taken. The
value of the \clist const:<(N|c|n|c|x) is set globally to the \comma list.

\clist_clear:N \clist_clear:c \clist gclear:N \clist gclear:c
\clist_clear:N \clist_clear:c \clist gclear:N \clist gclear:c
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 \clist_clear_new:c \clist gclear_new:N \clist gclear_new:c
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 \clist_set_eq:<(cN|Nc|cc) \clist gset_eq:NN \clist gset_eq:<(cN|Nc|cc)
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)
\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 \clist_set_from_seq:<(cN|Nc|cc) \clist gset_from_seq:NN \clist gset_from_seq:<(cN|Nc|cc)
Converts the data in the \sequence into a \comma list: the original \sequence is un-
changed. Items which contain either spaces or commas are surrounded by braces.

\clist_concat:NNN \clist_concat:<(cN|c|c|c) \clist gconcat:NNN \clist gconcat:<(cN|c|c|c)
\clist_concat:NNN \clist_concat:<(cN|c|c|c) \clist gconcat:NNN \clist gconcat:<(cN|c|c|c)
Concatenates the content of \comma list_1 and \comma list_2 together and saves the
result in \comma list_1. The items in \comma list_2 are 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:<(c)
\clist_if_exist_p:N \clist_if_exist_p:<(c) \clist_if_exist:NTF \clist_if_exist:<(c)
\clist_if_exist_p:N \clist_if_exist_p:<(c) \clist_if_exist:NTF \clist_if_exist:<(c)
\clist_if_exist_p:N \clist_if_exist_p:<(c) \clist_if_exist:NTF \clist_if_exist:<(c)
\clist_if_exist:NTF \clist_if_exist:<(c) \clist_if_exist:<(c)
Tests whether the \comma list is currently defined. This does not check that the \comma list
really is a comma list.

New: 2012-03-03
2 Adding data to comma lists

\clist_set:Nn \clist_set:Nn \clist_set: \clist_gset:Nn
\clist_set:Nn \clist_set: \clist_set: \clist_gset: \clist_gset: \clist_gset:
\clist_gset:Nn \clist_gset:Nn \clist_gset:Nn
\clist_gset:Nn \clist_gset:Nn \clist_gset:Nn

Sets \langle \text{comma list} \rangle to contain the \langle \text{items} \rangle, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some \langle \text{tokens} \rangle as a single \langle \text{item} \rangle even if the \langle \text{tokens} \rangle contain commas or spaces, add a set of braces: \clist_set:Nn \langle \text{comma list} \rangle \{ \{ \langle \text{tokens} \rangle \} \}.

\clist_put_left:Nn \clist_put_left:Nn \clist_gput_left:Nn \clist_put_left:Nn
\clist_put_left:Nn \clist_gput_left:Nn \clist_gput_left:Nn
\clist_gput_left:Nn \clist_gput_left:Nn \clist_gput_left:Nn

Updated: 2011-09-06

Appends the \langle \text{items} \rangle to the left of the \langle \text{comma list} \rangle. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some \langle \text{tokens} \rangle as a single \langle \text{item} \rangle even if the \langle \text{tokens} \rangle contain commas or spaces, add a set of braces: \clist_put_left:Nn \langle \text{comma list} \rangle \{ \{ \langle \text{tokens} \rangle \} \}.

\clist_put_right:Nn \clist_put_right:Nn \clist_gput_right:Nn \clist_put_right:Nn
\clist_put_right:Nn \clist_gput_right:Nn \clist_gput_right:Nn
\clist_gput_right:Nn \clist_gput_right:Nn \clist_gput_right:Nn

Updated: 2011-09-06

Appends the \langle \text{items} \rangle to the right of the \langle \text{comma list} \rangle. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some \langle \text{tokens} \rangle as a single \langle \text{item} \rangle even if the \langle \text{tokens} \rangle contain commas or spaces, add a set of braces: \clist_put_right:Nn \langle \text{comma list} \rangle \{ \{ \langle \text{tokens} \rangle \} \}.

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:N \langle comma list \rangle
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

Removes duplicate items from the \langle comma list \rangle, leaving the left most copy of each item in the \langle comma list \rangle. The \langle item \rangle comparison takes place on a token basis, as for \texttt{tl_if_eq:nn(TF)}.

\textbf{\LaTeX}hackers note: This function iterates through every item in the \langle comma list \rangle and does a comparison with the \langle items \rangle already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the \langle comma list \rangle contains \{, \}, or \# (assuming the usual \LaTeX category codes apply).

\clist_remove_all:Nn \clist_remove_all:Nn \langle comma list \rangle \{ \langle item \rangle \}
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn

Removes every occurrence of \langle item \rangle from the \langle comma list \rangle. The \langle item \rangle comparison takes place on a token basis, as for \texttt{tl_if_eq:nn(TF)}.

\textbf{\LaTeX}hackers note: The function may fail if the \langle item \rangle contains \{, \}, or \# (assuming the usual \LaTeX category codes apply).

\clist_reverse:N \clist_reverse:N \langle comma list \rangle
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

Reverses the order of items stored in the \langle comma list \rangle.

\textbf{New: 2014-07-18}

\clist_reverse:n \langle comma list \rangle
\langle comma list \rangle in the input stream in reverse order. Contrarily to other what is done for other \texttt{n-type} \langle comma list \rangle arguments, braces and spaces are preserved by this process.

\textbf{\LaTeX}hackers note: The result is returned within \texttt{\unexpanded}, which means that the comma list does not expand further when appearing in an \texttt{x-type} or \texttt{e-type} argument expansion.

\clist_sort:Nn \clist_sort:cn \langle clist var \rangle \{ \langle comparison code \rangle \}
\clist_gsort:Nn
\clist_gsort:cn

Sorts the items in the \langle clist var \rangle according to the \langle comparison code \rangle, and assigns the result to \langle clist var \rangle. The details of sorting comparison are described in Section 1.
4 Comma list conditionals

\clist_if_empty_p:N ∗ \clist_if_empty_p:N (comma list) \clist_if_empty_p:NTF (comma list) {⟨true code⟩} {⟨false code⟩}
Tests if the ⟨comma list⟩ is empty (containing no items).

\clist_if_empty:n ∗ \clist_if_empty: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 {⟨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⟩, the usual rules of space trimming and brace stripping apply. Hence,

\clist_if_in:nnTF { a , {b}~, {b} , c } { b } {true} {false}
yields true.

\TeXhackers note: The function may fail if the ⟨item⟩ contains {,}, or # (assuming the usual \TeX category codes apply).

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e. any local assignments made by the ⟨function⟩ or ⟨code⟩ discussed below remain in effect after the loop.

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 the comma list that is being mapped is \{a_{u_{ij}},\{b_{i_{ij}}\},\i_{ij}\{c\}_{ij}\}, then the arguments passed to the mapped function are ‘a’, ‘\{b_{i_{ij}}\}’, 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:NN (comma list) (function) \clist_map_function:cN ∗ \clist_map_function:nN ∗ \clist_map_inline:Nn
\clist_map_function:NN (comma list) (function)
Applies ⟨function⟩ to every ⟨item⟩ stored in the ⟨comma list⟩. The ⟨function⟩ receives 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.
\clist_map_inline:Nn \clist_map_inline:cn \clist_map_inline:nn
\list_map_variable:NNn \clist_map_variable:NNn \clist_map_variable:NNn
\clist_map_break: \clist_map_break: \clist_map_break:

\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 receives the \item\ as #1. The \items\ are returned from left to right.

\clist_map_variable:NNn \comma list \variable\ \{\code\}
Stores each \item\ of the \comma list\ in turn in the (token list) \variable\ and applies the \code\). The \code\ will usually make use of the \variable\, but this is not enforced. The assignments to the \variable\ are local. Its value after the loop is the last \item\ in the \comma list\, or its original value if there were no \item\). The \items\ are returned from left to right.

\clist_map_break:
Used to terminate a \clist_map\ function before all entries in the \comma list\ have been processed. This normally takes place within a conditional statement, for example

\clist_map_inline:Nn \l_my_clist
\texttt{\str_if_eq:nnTF \#1 \{ bingo \}
\{ \clist_map_break: \}
\{ % Do something useful
\}

Use outside of a \clist_map\ scenario leads to low level \TeX\ errors.

\TeX\hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.
\clist_map_break:n \{\code\}\n
Used to terminate a \clist_map... function before all entries in the ⟨comma list⟩ have been processed, inserting the ⟨code⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

\clist_map_inline:Nn \l_my_clist
{\str_if_eq:nnTF { #1 } { bingo } { \clist_map_break:n \{\code\} }\{% Do something useful\}\}

Use outside of a \clist_map... scenario leads to low level \TeX{} errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨code⟩ is inserted into the input stream. This depends on the design of the mapping function.

\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⟩ includes those which are duplicates, i.e. every item in a ⟨comma list⟩ is counted.

6 Using the content of comma lists directly

\clist_use:Nnnn \clist_use:cnnn \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 is 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- }

inserts “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 \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the ⟨items⟩ do not expand further when appearing in an \texttt{x}-type argument expansion.
\clist_use:Nn \clist_use:cn
\clist_get:NN \clist_pop:NN
\clist_gpop:NN
\clist_pop:NNTF

\clist_use:Nn \clist_var \langle \text{separator} \rangle
Places the contents of the \clist_var in the input stream, with the \langle separator \rangle 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 is raised if the variable does not exist or if it is invalid.

For example,
\begin{verbatim}
\clist_set:Nn \l_tmpa_clist { a , b , , c , \{de\} , f }
\clist_use:Nn \l_tmpa_clist \{ -and- \}
\end{verbatim}
inserts “a and b and c and de and f” in the input stream.

\TeXhackers note: The result is returned within the \texttt{\unexpanded} primitive (\exp_not:n), which means that the \langle items \rangle do not expand further when appearing in an \texttt{x}-type argument expansion.

\section{Comma lists as stacks}
Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

\clist_get:NN \clist_get:cN
\clist_get:NN \clist_pop:NN \clist_pop:cN
\clist_gpop:NN \clist_gpop:cN
\clist_pop:NNTF \clist_pop:NN \clist_pop:cN

\clist_get:NN \langle comma list \rangle \langle token list variable \rangle
Stores the left-most item from a \langle comma list \rangle in the \langle token list variable \rangle without removing it from the \langle comma list \rangle. The \langle token list variable \rangle is assigned locally. In the non-branching version, if the \langle comma list \rangle is empty the \langle token list variable \rangle is set to the marker value \texttt{\q_no_value}.

\clist_pop:NN \langle comma list \rangle \langle token list variable \rangle
Pops the left-most item from a \langle comma list \rangle into the \langle token list variable \rangle, i.e. removes the item from the comma list and stores it in the \langle token list variable \rangle. Both of the variables are assigned locally.

\clist_gpop:NN \langle comma list \rangle \langle token list variable \rangle
Pops the left-most item from a \langle comma list \rangle into the \langle token list variable \rangle, i.e. removes the item from the comma list and stores it in the \langle token list variable \rangle. The \langle comma list \rangle is modified globally, while the assignment of the \langle token list variable \rangle is local.

\clist_pop:NNTF \langle comma list \rangle \langle token list variable \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}
If the \langle comma list \rangle is empty, leaves the \langle false code \rangle in the input stream. The value of the \langle token list variable \rangle is not defined in this case and should not be relied upon. If the \langle comma list \rangle is non-empty, pops the top item from the \langle comma list \rangle in the \langle token list variable \rangle, i.e. removes the item from the \langle comma list \rangle. Both the \langle comma list \rangle and the \langle token list variable \rangle are assigned locally.
\clist_gpop:NNTF \clist_gpop:cN
\clist_gpop:NTF

\clist_gpop:NN \clist_gpop:cN
\clist_gpop:NTF

\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_item:Nn \clist_item:cn \clist_item:nn
\clist_rand_item:N \clist_rand_item:c \clist_rand_item:n

\clist_show:N \clist_show:c

\newdate{2012-05-14}
\newdate{2014-07-17}
\newdate{2016-12-06}
\clist_show:n \clist_show:n \{(tokens)\}
Displays the entries in the comma list in the terminal.

\clist_log:N \clist_log:N \{comma list\}
\clist_log:c
 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 \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.

10 Constant and scratch comma lists

\c_empty_clist
Constant that is always empty.

\l_tmpa_clist \l_tmpb_clist
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
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 XVI

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 has two primary function categories: \token_ for anything that deals with tokens and \peek_ for looking ahead in the token stream.

Most functions we 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 takes 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 7.

1 Creating character tokens

\cs_set_active_eq:NN \char_set_active_eq:Nc \char_gset_active_eq:NN \char_gset_active_eq:Nc

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

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

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

128
\char_generate:nn {\charcode} {\catcode}

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)
- 13 (active)

and other values raise an error. The \charcode may be any one valid for the engine in use. Active characters cannot be generated in older versions of Xe\TeX.

\textbf{\TeXhackers note:} Exactly two expansions are needed to produce the character.

\char_lowercase:N \langle char \rangle

Converts the \langle char \rangle to the equivalent case-changed character as detailed by the function name (see \str_foldcase:n and \text_titlecase:n for details of these terms). The case mapping is carried out with no context-dependence (cf. \text_uppercase:n, etc.) The str versions always generate “other” (category code 12) characters, whilst the standard versions generate characters with the category code of the \langle char \rangle (i.e. only the character code changes).

\c_catcode_other_space_tl

Token list containing one character with category code 12, (“other”), and character code 32 (space).
2 Manipulating and interrogating character tokens

\char_set_catcode_escape:N \char_set_catcode_letter:N \langle character \rangle
\char_set_catcode_group_begin:N \char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N \char_set_catcodeAlignment: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_comment:N \char_set_catcode_invalid:N

Updated: 2015-11-11

Sets the category code of the \langle character \rangle to that indicated in the function name. Depending on the current category code of the \langle token \rangle the escape token may also be needed:

\char_set_catcode_other:N \%

The assignment is local.

\char_set_catcode_escape:n \char_set_catcode_letter:n \{\langle integer expression \rangle\}
\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

Updated: 2015-11-11

Sets the category code of the \langle character \rangle which has character code as given by the \langle integer expression \rangle. This version can be used to set up characters which cannot otherwise be given (cf. the N-type variants). The assignment is local.
These functions set the category code of the \textit{character} which has character code as given by the \textit{integer expression}. The first \textit{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 \texttt{\char_set_catcode\_\textit{type}} should be preferred, but there are cases where these lower-level functions may be useful.

\texttt{\char_set_catcode:nn \{\texttt{intexpr}_1\} \{\texttt{intexpr}_2\}}

Updated: 2015-11-11

Expands to the current category code of the \textit{character} with character code given by the \textit{integer expression}.

\texttt{\char_value_catcode:n \{\texttt{integer expression}\}}

Displays the current category code of the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\texttt{\char_set_lccode:nn \{\texttt{intexpr}_1\} \{\texttt{intexpr}_2\}}

Sets up the behaviour of the \textit{character} when found inside \texttt{\text_lowercase:n}, such that \texttt{\textit{character}_1} will be converted into \texttt{\textit{character}_2}. The two \textit{characters} may be specified using an \textit{integer expression} for the character code concerned. This may include the \TeX{} \texttt{\textit{character}} method for converting a single character into its character code:

\begin{verbatim}
\char_set_lccode:nn \{ \texttt{\textbackslash A} \} \{ \texttt{\textbackslash a} \} \% Standard behaviour
\char_set_lccode:nn \{ \texttt{\textbackslash A} \} \{ \texttt{\textbackslash A + 32} \}
\char_set_lccode:nn \{ 50 \} \{ 60 \}
\end{verbatim}

The setting applies within the current \TeX{} group.

\texttt{\char_value_lccode:n \{\texttt{integer expression}\}}

\texttt{\char_show_value_lccode:n \{\texttt{integer expression}\}}

Displays the current lower case code of the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\texttt{\char_set_uccode:nn \{\texttt{intexpr}_1\} \{\texttt{intexpr}_2\}}

Sets up the behaviour of the \textit{character} when found inside \texttt{\text_uppercase:n}, such that \texttt{\textit{character}_1} will be converted into \texttt{\textit{character}_2}. The two \textit{characters} may be specified using an \textit{integer expression} for the character code concerned. This may include the \TeX{} \texttt{\textit{character}} method for converting a single character into its character code:

\begin{verbatim}
\char_set_uccode:nn \{ \texttt{\textbackslash a} \} \{ \texttt{\textbackslash A} \} \% Standard behaviour
\char_set_uccode:nn \{ \texttt{\textbackslash A} \} \{ \texttt{\textbackslash A - 32} \}
\char_set_uccode:nn \{ 60 \} \{ 50 \}
\end{verbatim}

The setting applies within the current \TeX{} group.
\texttt{\char_value_uccode:n}\star \char_value_uccode:n \{(integer expression)\}
Expands to the current upper case code of the \langle character \rangle with character code given by the \langle integer expression \rangle.

\texttt{\char_show_uccode:n}\star \char_show_uccode:n \{(integer expression)\}
Displays the current upper case code of the \langle character \rangle with character code given by the \langle integer expression \rangle on the terminal.

\texttt{\char_set_mathcode:nn}\star \char_set_mathcode:nn \{(intexpr)\} \{(intexpr)\}
This function sets up the math code of \langle character \rangle. The \langle character \rangle is specified as an \langle integer expression \rangle which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\texttt{\char_value_mathcode:n}\star \char_value_mathcode:n \{(integer expression)\}
Expands to the current math code of the \langle character \rangle with character code given by the \langle integer expression \rangle.

\texttt{\char_show_value_mathcode:n}\star \char_show_value_mathcode:n \{(integer expression)\}
Displays the current math code of the \langle character \rangle with character code given by the \langle integer expression \rangle on the terminal.

\texttt{\char_set_sfcode:nn}\star \char_set_sfcode:nn \{(intexpr)\} \{(intexpr)\}
This function sets up the space factor for the \langle character \rangle. The \langle character \rangle is specified as an \langle integer expression \rangle which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\texttt{\char_value_sfcode:n}\star \char_value_sfcode:n \{(integer expression)\}
Expands to the current space factor for the \langle character \rangle with character code given by the \langle integer expression \rangle.

\texttt{\char_show_value_sfcode:n}\star \char_show_value_sfcode:n \{(integer expression)\}
Displays the current space factor for the \langle character \rangle with character code given by the \langle integer expression \rangle on the terminal.

\texttt{\l_char_active_seq}
Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category \langle active \rangle (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.

\texttt{\l_char_special_seq}
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 \langle letter \rangle (catcode 11) or \langle other \rangle (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.

132
3 Generic tokens

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.

\begin{verbatim}
\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
\end{verbatim}

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.

\begin{verbatim}
\c_catcode_letter_token
\c_catcode_other_token
\end{verbatim}

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

\begin{verbatim}
\token_to_meaning:N \token_to_meaning:c
\end{verbatim}

Inserts the current meaning of the \langle token\rangle into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the \langle token\rangle, thus for example both functions defined by \texttt{\cs_set_nopar:Npn} and token list variables defined using \texttt{\tl_new:N} are described as macros.

\TeXhackers note: This is the \TeX primitive \texttt{\meaning}. The \langle token\rangle can thus be an explicit space tokens or an explicit begin-group or end-group character token (\{ or \} when normal \TeX category codes apply) even though these are not valid \texttt{N}-type arguments.

\begin{verbatim}
\token_to_str:N \token_to_str:c
\end{verbatim}

Converts the given \langle token\rangle into a series of characters with category code 12 (other). If the \langle token\rangle is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the \langle token\rangle). This function requires only a single expansion.

\TeXhackers note: \texttt{\token_to_str:N} is the \TeX primitive \texttt{\string} renamed. The \langle token\rangle can thus be an explicit space tokens or an explicit begin-group or end-group character token (\{ or \} when normal \TeX category codes apply) even though these are not valid \texttt{N}-type arguments.
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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \textit{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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \textit{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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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_parameter:NTF \token \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \{\text{true code}\} \{\text{false code}\}

Tests if \textit{(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 \textit{N}-type argument.
Tests if \langle token \rangle has the category code of a letter token.

Tests if \langle token \rangle has the category code of an “other” token.

Tests if \langle token \rangle has the category code of an active character.

Tests if the two \langle tokens \rangle have the same category code.

Tests if the two \langle tokens \rangle have the same character code.

Tests if the two \langle tokens \rangle have the same meaning when expanded.

Tests if the \langle token \rangle is a \TeX{} macro.

Tests if the \langle token \rangle is a control sequence.

Tests if the \langle token \rangle is expandable. This test returns \langle false \rangle for an undefined token.

Tests if the \langle token \rangle is a long macro.

Tests if the \langle token \rangle is a protected macro: for a macro which is both protected and long this returns \langle false \rangle.
Tests if the \{token\} is a protected long macro.

Tests if the \{token\} is defined to be a chardef.

\TeXhacks note: Boolean, boxes and small integer constants are implemented as \chardef s.

Tests if the \{token\} is defined to be a mathchardef.

Tests if the \{token\} is defined to be a dimension register.

Tests if the \{token\} is defined to be an integer register.

\TeXhacks note: Constant integers may be implemented as integer registers, \chardef s, or \mathchardef s depending on their value.

Tests if the \{token\} is defined to be a muskip register.

Tests if the \{token\} is defined to be a skip register.
Tests if the \langle \text{token} \rangle is defined to be a toks register (not used by \LaTeX3).

Tests if the \langle \text{token} \rangle 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 \texttt{\textbackslash peek\_after:}\texttt{Nw} is provided along with a family of predefined tests for common cases. As peeking ahead does \textit{not} skip spaces the predefined tests include both a space-respecting and space-skipping version.

\texttt{\textbackslash peek\_after:}\texttt{Nw} \langle \text{function} \rangle \langle \text{token} \rangle

Locally sets the test variable \texttt{\_\_peek\_token} equal to \langle \text{token} \rangle (as an implicit token, \textit{not} as a token list), and then expands the \langle \text{function} \rangle. The \langle \text{token} \rangle remains in the input stream as the next item after the \langle \text{function} \rangle. The \langle \text{token} \rangle here may be \texttt{\_\_}, \{ or \} (assuming normal \TeX category codes), \textit{i.e.} it is not necessarily the next argument which would be grabbed by a normal function.

\texttt{\textbackslash peek\_after:}\texttt{Nw} \langle \text{function} \rangle \langle \text{token} \rangle

Globally sets the test variable \texttt{\_\_\_peek\_token} equal to \langle \text{token} \rangle (as an implicit token, \textit{not} as a token list), and then expands the \langle \text{function} \rangle. The \langle \text{token} \rangle remains in the input stream as the next item after the \langle \text{function} \rangle. The \langle \text{token} \rangle here may be \texttt{\_\_\_}, \{ or \} (assuming normal \TeX category codes), \textit{i.e.} it is not necessarily the next argument which would be grabbed by a normal function.

\texttt{\_\_\_peek\_token} Token set by \texttt{\textbackslash peek\_after:}\texttt{Nw} and available for testing as described above.

\texttt{\_\_\_\_\_peek\_token} Token set by \texttt{\textbackslash peek\_after:}\texttt{Nw} and available for testing as described above.

\texttt{\textbackslash peek\_catcode:}\texttt{NTF} \langle \text{test token} \rangle \langle \text{true code} \rangle \langle \text{false code} \rangle

Tests if the next \langle \text{token} \rangle in the input stream has the same category code as the \langle \text{test token} \rangle (as defined by the test \texttt{\textbackslash token\_if\_eq\_catcode:\texttt{NTF}}). Spaces are respected by the test and the \langle \text{token} \rangle is left in the input stream after the \langle \text{true code} \rangle or \langle \text{false code} \rangle (as appropriate to the result of the test).
Tests if the next non-space \langle test token \rangle in the input stream has the same category code as the \langle test token \rangle (as defined by the test \texttt{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 \langle token \rangle is left in the input stream after the \langle true code \rangle or \langle false code \rangle (as appropriate to the result of the test).

Tests if the next \langle token \rangle in the input stream has the same category code as the \langle test token \rangle (as defined by the test \texttt{token_if_eq_catcode:NNTF}). Spaces are respected by the test and the \langle token \rangle is removed from the input stream if the test is true. The function then places either the \langle true code \rangle or \langle false code \rangle in the input stream (as appropriate to the result of the test).

Tests if the next non-space \langle token \rangle in the input stream has the same character code as the \langle test token \rangle (as defined by the test \texttt{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 \langle token \rangle is left in the input stream after the \langle true code \rangle or \langle false code \rangle (as appropriate to the result of the test).

Tests if the next \langle token \rangle in the input stream has the same character code as the \langle test token \rangle (as defined by the test \texttt{token_if_eq_charcode:NNTF}). Spaces are respected by the test and the \langle token \rangle is removed from the input stream if the test is true. The function then places either the \langle true code \rangle or \langle false code \rangle in the input stream (as appropriate to the result of the test).
\peek_charcode_remove_ignore_spaces:NTF
Updated: 2012-12-20
\peek_charcode_remove_ignore_spaces:NTF ⟨test token⟩ \{⟨true code⟩\} \{⟨false code⟩\}
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⟩ is removed from the input stream if the test is true.
The function then places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as
appropriate to the result of the test).

\peek_meaning:NTF
Updated: 2011-07-02
\peek_meaning:NTF ⟨test token⟩ \{⟨true code⟩\} \{⟨false code⟩\}
Tests if the next ⟨token⟩ in the input stream has the same meaning as the ⟨test token⟩ (as
defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and
the ⟨token⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate
to the result of the test).

\peek_meaning_ignore_spaces:NTF
Updated: 2012-12-05
\peek_meaning_ignore_spaces:NTF ⟨test token⟩ \{⟨true code⟩\} \{⟨false code⟩\}
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⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate
to the result of the test).

\peek_meaning_remove:NTF
Updated: 2011-07-02
\peek_meaning_remove:NTF ⟨test token⟩ \{⟨true code⟩\} \{⟨false code⟩\}
Tests if the next ⟨token⟩ in the input stream has the same meaning as the ⟨test token⟩
(as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test
and the ⟨token⟩ is removed from the input stream if the test is true. The function then
places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as appropriate to
the result of the test).

\peek_meaning_remove_ignore_spaces:NTF
Updated: 2012-12-05
\peek_meaning_remove_ignore_spaces:NTF ⟨test token⟩ \{⟨true code⟩\} \{⟨false code⟩\}
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⟩ is removed from the input stream if the test is true. The function
then places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as appropriate to
the result of the test).
Tests if the next \langle \textit{token} \rangle in the input stream can be safely grabbed as an N-type argument. The test is \langle false \rangle if the next \langle \textit{token} \rangle 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 \langle true \rangle in all other cases. Note that a \langle true \rangle result ensures that the next \langle \textit{token} \rangle is a valid N-type argument. However, if the next \langle \textit{token} \rangle is for instance \texttt{c_space_token}, the test takes the \langle false \rangle branch, even though the next \langle \textit{token} \rangle is in fact a valid N-type argument. The \langle \textit{token} \rangle is left in the input stream after the \langle true code \rangle or \langle false code \rangle (as appropriate to the result of the test).

7 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 \LaTeX’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, \texttt{\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).

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

- Expanding \texttt{\the\font} results in a token that looks identical to the command that was used to select the current font (such as \texttt{\tenrm}) but it differs from it in shape.

- A “frozen” \texttt{relax}, which differs from the primitive in shape (but has the same meaning), is inserted when the closing \texttt{\fi} of a conditional is encountered before the conditional is evaluated.

- Expanding \texttt{\noexpand \token} (when the \langle \textit{token} \rangle is expandable) results in an internal token, displayed (temporarily) as \texttt{\notexpanded: \token}, whose shape coincides with the \langle \textit{token} \rangle and whose meaning differs from \texttt{relax}.

- An \texttt{\outer endtemplate} can be encountered when peeking ahead at the next token; this expands to another internal token, \texttt{end of alignment template}.

- Tricky programming might access a frozen \texttt{\endwrite}.

\footnote{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.}
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 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 (\texttt{group\_begin}, often \{),
2. end-group character (\texttt{group\_end}, often \}),
3. math shift character (\texttt{math\_toggle}, often \$),
4. alignment tab character (\texttt{alignment}, often &),
5. macro parameter character (\texttt{parameter}, often \#),
6. superscript character (\texttt{math\_superscript}, often ^),
7. subscript character (\texttt{math\_subscript}, often _),
8. blank space (space, often character code 32),
9. the letter (letter, such as A),
10. the character (other, such as 0).

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

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

\langle properties \rangle \texttt{macro:} \langle parameters \rangle \rightarrow \langle replacement \rangle
where \(\langle\text{properties}\rangle\) is among \texttt{\protect\textbackslash long}\texttt{\protect\textbackslash outer}, \(\langle\text{parameters}\rangle\) describes parameters that the macro expects, such as \#1\#2\#3, and \(\langle\text{replacement}\rangle\) 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 :\texttt{N}.
Part XVII
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 \langle key \rangle and an associated \langle value \rangle. The \langle key \rangle and \langle value \rangle may both be any \langle balanced text \rangle. 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 \langle key \rangle: if an entry is added to a property list which already contains the \langle key \rangle then the new entry overwrites the existing one. The \langle keys \rangle are compared on a string basis, using the same method as \texttt{\textbackslash 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 \texttt{input} parsed by the \texttt{keys} module.

1 Creating and initialising property lists

\prop_new:N \langle property list \rangle

Creates a new \langle property list \rangle or raises an error if the name is already taken. The declaration is global. The \langle property list \rangle initially contains no entries.

\prop_clear:N \langle property list \rangle
\prop_clear:c \langle property list \rangle
\prop_gclear:N \langle property list \rangle
\prop_gclear:c \langle property list \rangle

Clears all entries from the \langle property list \rangle.

\prop_clear_new:N \prop_clear_new:c \prop_gclear_new:N \prop_gclear_new:c \langle property list \rangle

Ensures that the \langle property list \rangle exists globally by applying \prop_new:N if necessary, then applies \prop_(g)clear:N to leave the list empty.

\prop_set_eq:NN \langle property list \rangle \langle property list 2 \rangle
\prop_set_eq:ncN \langle property list \rangle \langle property list 2 \rangle
\prop_gset_eq:NN \langle property list \rangle \langle property list 2 \rangle
\prop_gset_eq:ncN \langle property list \rangle \langle property list 2 \rangle

Sets the content of \langle property list \rangle equal to that of \langle property list 2 \rangle.

\prop_set_from_keyval:Nn \langle prop var \rangle
\prop_set_from_keyval:cn \langle prop var \rangle
\prop_gset_from_keyval:Nn \langle prop var \rangle
\prop_gset_from_keyval:cn \langle prop var \rangle

Sets \langle prop var \rangle to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.
\prop_const_from_keyval:Nn (prop var)
{
⟨key1⟩ = ⟨value1⟩ ,
⟨key2⟩ = ⟨value2⟩ , ...
}

Creates a new constant ⟨prop var⟩ or raises an error if the name is already taken. The ⟨prop var⟩ is set globally to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.

2 Adding entries to property lists

\prop_put:Nnn
\prop_gput:Nnn
\prop_put_if_new:Nnn
\prop_gput_if_new:Nnn

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_gput_if_new:Nnn

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_pop:NnN

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⟩ is set to 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

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⟩ is set to 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.
Recovers the $\langle$value$\rangle$ stored with $\langle$key$\rangle$ from the $\langle$property list$\rangle$, and places this in the $\langle$token list variable$\rangle$. If the $\langle$key$\rangle$ is not found in the $\langle$property list$\rangle$ then the $\langle$token list variable$\rangle$ is set to the special marker $\langle$q_no_value$\rangle$. The $\langle$key$\rangle$ and $\langle$value$\rangle$ are then deleted from the property list. The $\langle$property list$\rangle$ is modified globally, while the assignment of the $\langle$token list variable$\rangle$ is local. See also $\langle$prop_gpop:NnNTF$\rangle$.

Expands to the $\langle$value$\rangle$ corresponding to the $\langle$key$\rangle$ in the $\langle$property list$\rangle$. If the $\langle$key$\rangle$ is missing, this has an empty expansion.

**TexHackers note:** This function is slower than the non-expandable analogue $\langle$prop_get:Nn$\rangle$. The result is returned within the $\langle$unexpanded$\rangle$ primitive $\langle$exp_not:n$\rangle$, which means that the $\langle$value$\rangle$ does not expand further when appearing in an $x$-type argument expansion.

Leaves the number of key–value pairs in the $\langle$property list$\rangle$ in the input stream as an $\langle$integer denotation$\rangle$.

### 4 Modifying property lists

Removes the entry listed under $\langle$key$\rangle$ from the $\langle$property list$\rangle$. If the $\langle$key$\rangle$ is not found in the $\langle$property list$\rangle$ no change occurs, i.e. there is no need to test for the existence of a key before deleting it.

### 5 Property list conditionals

Tests whether the $\langle$property list$\rangle$ is currently defined. This does not check that the $\langle$property list$\rangle$ really is a property list variable.

Tests if the $\langle$property list$\rangle$ is empty (containing no entries).
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 value. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}
\prop_get:NnN ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

Updated: 2012-05-19

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:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}
\prop_pop:NnN ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}
\prop_gpop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}
\prop_gpop:NnN ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

Updated: 2012-05-19

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.
7 Mapping to property lists

All mappings are done at the current group level, i.e. any local assignments made by the ⟨function⟩ or ⟨code⟩ discussed below remain in effect after the loop.

\prop_map_function:NN (property list) ⟨function⟩
Applies ⟨function⟩ to every ⟨entry⟩ stored in the ⟨property list⟩. The ⟨function⟩ receives two arguments for each iteration: the (key) and associated (value). The order in which ⟨entries⟩ are returned is not defined and should not be relied upon. To pass further arguments to the ⟨function⟩, see \prop_map_tokens:Nn.

\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 receives 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_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 } }

expands 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_map_break:
Used to terminate a \prop_map_... function before all entries in the ⟨property list⟩ have been processed. This normally takes 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 leads to low level \TeX errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.
\prop_map_break:n \prop_map_break:n \prop_map_break:n \prop_map_break:n

Used to terminate a \prop_map\... function before all entries in the \langle property list \rangle have been processed, inserting the \langle code \rangle after the mapping has ended. This normally takes place within a conditional statement, for example

\prop_map_inline:Nn \l_my_prop
{\str_if_eq:nnTF { #1 } { bingo } \prop_map_break:n \langle code \rangle
{ % Do something useful
}%
}

Use outside of a \prop_map\... scenario leads to low level \TeX{} errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the \langle code \rangle is inserted into the input stream. This depends on the design of the mapping function.

8 Viewing property lists

\prop_show:N \prop_show:N \prop_show:N
\prop_show:N \langle property list \rangle
Displays the entries in the \langle property list \rangle in the terminal.

\prop_log:N \prop_log:N \prop_log:N
\prop_log:N \langle property list \rangle
Writes the entries in the \langle property list \rangle in the log file.

9 Scratch property lists

\l_tmpa_prop \l_tmpb_prop
\l_tmpa_prop \l_tmpb_prop
New: 2012-06-23
Updated: 2015-08-01
Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX\-3-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
\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 \LaTeX\-3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
10 Constants

\texttt{\textbackslash c\_empty\_prop} A permanently-empty property list used for internal comparisons.
Part XVIII

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The l3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by l3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message class has to be made, for example error, warning or info.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, \ may be used to force a new line and \ causes 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 to filter out specifically messages from the submodule.
Tests whether the \langle message \rangle for the \langle module \rangle is currently defined.

### 2 Contextual information for messages

- \texttt{\textbackslash msg\_line\_context:} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash 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 \texttt{on line}.

- \texttt{\textbackslash msg\_line\_number:} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash msg\_line\_number:}
    - Prints the current line number when a message is given.

- \texttt{\textbackslash msg\_fatal\_text:n} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash msg\_fatal\_text:n \{\langle module\rangle\}}
    - Produces the standard text
      - \texttt{Fatal Package \langle module\rangle Error}

    - This function can be redefined to alter the language in which the message is given, using \texttt{#1} as the name of the \langle module \rangle to be included.

- \texttt{\textbackslash msg\_critical\_text:n} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash msg\_critical\_text:n \{\langle module\rangle\}}
    - Produces the standard text
      - \texttt{Critical Package \langle module\rangle Error}

    - This function can be redefined to alter the language in which the message is given, using \texttt{#1} as the name of the \langle module \rangle to be included.

- \texttt{\textbackslash msg\_error\_text:n} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash msg\_error\_text:n \{\langle module\rangle\}}
    - Produces the standard text
      - \texttt{Package \langle module\rangle Error}

    - This function can be redefined to alter the language in which the message is given, using \texttt{#1} as the name of the \langle module \rangle to be included.

- \texttt{\textbackslash msg\_warning\_text:n} \texttt{\ngthx{\textdagger}}
  - \texttt{\textbackslash msg\_warning\_text:n \{\langle module\rangle\}}
    - Produces the standard text
      - \texttt{Package \langle module\rangle Warning}

    - This function can be redefined to alter the language in which the message is given, using \texttt{#1} as the name of the \langle module \rangle to be included. The \langle type \rangle of \langle module \rangle may be adjusted. \texttt{Package} is the standard outcome: see \texttt{\textbackslash msg\_module\_type:n}.

\textcopyright{} 2012-03-03
\msg_info_text:n \msg_info_text:n \{\langle\text{module}\rangle\}\n
Produces the standard text:

Package \langle\text{module}\rangle\ Info

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle\ to be included. The \langle\text{type}\rangle\ of \langle\text{module}\rangle\ may be adjusted:

Package is the standard outcome: see \msg_module_type:n.\n
\msg_module_name:n \msg_module_name:n \{\langle\text{module}\rangle\}\n
Expands to the public name of the \langle\text{module}\rangle\ as defined by \g_msg_module_name_prop (or otherwise leaves the \langle\text{module}\rangle\ unchanged).\n
\msg_module_type:n \msg_module_type:n \{\langle\text{module}\rangle\}\n
Expands to the description which applies to the \langle\text{module}\rangle, for example a Package or Class. The information here is defined in \g_msg_module_type_prop, and will default to Package if an entry is not present.\n
\msg_see_documentation_text:n \msg_see_documentation_text:n \{\langle\text{module}\rangle\}\n
Produces the standard text

See the \langle\text{module}\rangle\ documentation for further information.\n
This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle\ to be included. The name of the \langle\text{module}\rangle\ may be altered by use of \g_msg_module_documentation_prop\n
\g_msg_module_name_prop \n
Provides a mapping between the module name used for messages, and that for documentation. For example, E\TeX\3 core messages are stored in the reserved LaTeX tree, but are printed as LaTeX3.\n
\g_msg_module_type_prop \n
Provides a mapping between the module name used for messages, and that type of module. For example, for E\TeX\3 core messages, an empty entry is set here meaning that they are not described using the standard Package text.\n
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 are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the x-type variants should be used to expand material.
Issues \texttt{\module\ error\ \message,\ passing\ \arg\ one\ to\ \arg\ four\ to\ the\ text-creating\ functions.\ After\ issuing\ a\ fatal\ error\ the\ \TeX\ run\ halts.\ No\ PDF\ file\ will\ be\ produced\ in\ this\ case\ (DVI\ mode\ runs\ may\ produce\ a\ truncated\ DVI\ file).}

\TeXhackers\ note: The \TeX\ \texttt{\endinput} primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.
Issues \textit{(module)} information \textit{(message)}, passing \textit{(arg one)} to \textit{(arg four)} to the text-creating functions. The information text is added to the log file.

Issues \textit{(module)} information \textit{(message)}, passing \textit{(arg one)} to \textit{(arg four)} to the text-creating functions. The information text is added to the log file: the output is briefer than \texttt{msg\textunderscore info:nnnnnn}.

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

\texttt{msg\_new:nnnn \{module\} \{my\textunderscore message\} \{Some\textunderscore text\} \{Some\textunderscore more\textunderscore text\}}

to define a message, with

\texttt{msg\_error:nn \{module\} \{my\textunderscore message\}}

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

\texttt{msg\_redirect\_class:nn \{error\} \{warning\}}

to turn all errors into warnings, or with

\texttt{msg\_redirect\_module:nnn \{module\} \{error\} \{warning\}}
to alter only messages from that module, or even

\msg_redirect_name:nn { 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 raises an error 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 $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

\msg_redirect_class:nn
\msg_redirect_module:nn
\msg_redirect_name: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:nn {⟨module⟩} {⟨class one⟩} {⟨class two⟩}
Redirects message of ⟨class one⟩ for ⟨module⟩ to act as though they were from ⟨class two⟩. Messages of ⟨class one⟩ from sources other than ⟨module⟩ are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the warning messages of ⟨module⟩ could be turned off with:

\msg_redirect_module:nn { module } { warning } { none }

\msg_redirect_name:nn {⟨module⟩} {⟨message⟩} {⟨class⟩}
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:nn { module } { annoying-message } { none }
Part XIX
The \texttt{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 \texttt{\file...}, while others are used to work with
files on a line by line basis and have prefix \texttt{\ior...} (reading) or \texttt{\iow...} (writing).

It is important to remember that when reading external files \TeX{} attempts to locate
them using 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 \texttt{⟨file name⟩} argument, this argument may contain both
literal items and expandable content, which should on full expansion be the desired
file name. Active characters (as declared in \texttt{\l_char_active_seq}) are not expanded,
allowing the direct use of these in file names. Quote tokens (") are not permitted in file
names as they are reserved for internal use by some \TeX{} primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the
fact that some file systems do not allow or interact unpredictably with spaces in these
positions. When no extension is given, this will trim spaces from the start of the name
only.

1 Input–output stream management

As \TeX{} engines have a limited number of input and 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 mod-
ules. 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.

\begin{verbatim}
\ior_new:N \ior_new:c \iow_new:N \iow_new:c
New: 2011-09-26
Updated: 2011-12-27
\end{verbatim}

\begin{verbatim}
\ior_open:Nn \ior_open:cn
\iow_open:Nn \iow_open:cn
Updated: 2012-02-10
\end{verbatim}

\begin{verbatim}
\ior_new:N ⟨stream⟩
\ior_new:c ⟨stream⟩
\iow_new:N ⟨stream⟩
\iow_new:c ⟨stream⟩

Globally reserves the name of the ⟨stream⟩, either for reading or for writing as appropri-
ate. The ⟨stream⟩ is not opened until the appropriate \texttt{\ Streaming}{\open:} function is used.

Attempting to use a ⟨stream⟩ which has not been opened is an error, and the ⟨stream⟩
will behave as the corresponding \texttt{\c_term...}.

\ior_open:Nn ⟨stream⟩ {⟨file name⟩}
\ior_open:cn ⟨stream⟩

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 \texttt{\ior_-
close:N} instruction is given or the \TeX{} run ends. If the file is not found, an error is
raised.

156
\ior_open:NnTF \ior_open:cnTF
\ior_close:N \ior_close:c \ior_close:N \ior_close:c
\ior_open:cn \ior_open:cn
\ior_close:N \ior_close:N
\ior_show_list: \ior_log_list: \ior_show_list: \ior_log_list:
\ior_open:Nn \ior_open:cn
\ior_open:Nn \ior_open:cn
\ior_open:Nn \ior_open:cn
\ior_open:Nn \ior_open:cn
\ior_open:Nn \ior_open:cn
\ior_open:Nn \ior_open:cn

1.1 Reading from files

Reading from files and reading from the terminal are separate processes in \texttt{expl3}. The functions \texttt{\ior_get:NN} and \texttt{\ior_str_get:NN}, and their branching equivalents, are designed to work with \texttt{files}. 
Function that reads one or more lines (until an equal number of left and right braces are found) from the file input \texttt{(stream)} and stores the result locally in the \texttt{(token list)} variable. The material read from the \texttt{(stream)} is tokenized by \TeX{} according to the category codes and \texttt{\textbackslash{endlinechar}} in force when the function is used. Assuming normal settings, any lines which do not end in a comment character \texttt{\%} have the line ending converted to a space, so for example input
\begin{verbatim}
a b c
\end{verbatim}
results in a token list \texttt{a/b/c}. Any blank line is converted to the token \texttt{\textbackslash{par}}. Therefore, blank lines can be skipped by using a test such as
\begin{verbatim}
\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
\end{verbatim}
Also notice that if multiple lines are read to match braces then the resulting token list can contain \texttt{\textbackslash{par}} tokens. In the non-branching version, where the \texttt{(stream)} is not open the \texttt{(tl var)} is set to \texttt{\textbackslash{q_no_value}}.

\textbf{\TeX{}hackers note:} This protected macro is a wrapper around the \TeX{} primitive \texttt{\textbackslash{read}}. Regardless of settings, \TeX{} replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code \texttt{\textbackslash{endlinechar}}, omitted if \texttt{\textbackslash{endlinechar}} is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

Function that reads one line from the file input \texttt{(stream)} and stores the result locally in the \texttt{(token list)} variable. The material is read from the \texttt{(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 always only reads one line and any blank lines in the input result in the \texttt{(token list variable)} being empty. Unlike \texttt{\ior_get:NN}, line ends do not receive any special treatment. Thus input
\begin{verbatim}
a b c
\end{verbatim}
results in a token list \texttt{a b c} with the letters \texttt{a}, \texttt{b}, and \texttt{c} having category code 12. In the non-branching version, where the \texttt{(stream)} is not open the \texttt{(tl var)} is set to \texttt{\textbackslash{q_no_value}}.

\textbf{\TeX{}hackers note:} This protected macro is a wrapper around the \TeX{} primitive \texttt{\textbackslash{readline}}. Regardless of settings, \TeX{} removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of \texttt{\ior_str_get:NN}.

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \texttt{(function)} or \texttt{(code)} discussed below remain in effect after the loop.
\ior_map_inline:Nn \ior_map_inline:Nn \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_inline:Nn

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:

\ior_map_break:

\ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn \ior_map_break:
\ior_map_break:n \ior_map_break:n \langle code \rangle
Used to terminate a \ior_map\ldots function before all lines in the \langle stream \rangle have been processed, inserting the \langle code \rangle after the mapping has ended. This normally takes place within a conditional statement, for example

\ior_map_inline:Nn \l_my_ior
{\str_if_eq:nnTF {#1} {bingo}\
{\ior_map_break:n \langle code \rangle}\
{\% Do something useful}
}

Use outside of a \ior_map\ldots scenario leads to low level \TeX errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the \langle code \rangle is inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N  \ior_if_eof_p:N \langle stream \rangle
\ior_if_eof:NTF  \ior_if_eof:NTF  \langle stream \rangle  \langle \{true code\} \langle false code\} \rangle
Tests if the end of a file \langle stream \rangle has been reached during a reading operation. The test also returns a true value if the \langle stream \rangle is not open.

1.2 Writing to files

\iow_now:Nn  \iow_now:Nn  \langle stream \rangle  \langle \{tokens\} \rangle
This function writes \langle tokens \rangle to the specified \langle stream \rangle immediately (i.e. the write operation is called on expansion of \iow_now:Nn).

\iow_log:n  \iow_log:n  \langle \{tokens\} \rangle
This function writes the given \langle tokens \rangle to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_term:n  \iow_term:n  \langle \{tokens\} \rangle
This function writes the given \langle tokens \rangle 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 \langle\text{stream}\rangle\{\langle\text{tokens}\rangle\}

This function writes \langle\text{tokens}\rangle to the specified \langle\text{stream}\rangle when the current page is finalised (\textit{i.e.} at shipout). The x-type variants expand the \langle\text{tokens}\rangle at the point where the function is used but \textit{not} when the resulting tokens are written to the \langle\text{stream}\rangle (\textit{cf.} \iow_shipout_x:NN).

\textbf{\LaTeX} hackers 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 are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

\iow_shipout_x:NN \iow_shipout_x:(Nx|cn|cx)
\iow_shipout_x:NN \langle\text{stream}\rangle\{\langle\text{tokens}\rangle\}

This function writes \langle\text{tokens}\rangle to the specified \langle\text{stream}\rangle when the current page is finalised (\textit{i.e.} at shipout). The \langle\text{tokens}\rangle 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).

\textbf{\LaTeX} hackers note: This is a wrapper around the \LaTeX 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 are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

\iow_char:N \star
\iow_char:N \langle\text{char}\rangle

Inserts \langle\text{char}\rangle into the output stream. Useful when trying to write difficult characters such as \%, {, }, \textit{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 (\textit{e.g.} in the second argument of \iow_now:NN).

\iow_newline: \star
\iow_newline:

Function to add a new line within the \langle\text{tokens}\rangle written to a file. The function has no effect if writing is taking place without expansion (\textit{e.g.} in the second argument of \iow_now:NN).

\textbf{\LaTeX} hackers note: When using expl3 with a format other than \LaTeX, the character inserted by \iow_newline: is not recognized by \LaTeX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects \iow_shipout:Nn, \iow_shipout_x:NN and direct uses of primitive operations.
1.3 Wrapping lines in output

This function wraps the ⟨text⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨run-on text⟩ is inserted. The line character count targeted is 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:

- \ or \iow_newline: 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_allow_break: may be used to allow a line-break without inserting a space (this is experimental),
- \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 is typically a wrapper around a write operation. The output of \iow_wrap:nnnN (i.e. the argument passed to the ⟨function⟩) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does not expand further when written to a file.

\TeX{}hackers 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 {⟨text⟩}

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 \TeX{}live and \MiKTeX{} systems.
1.4 Constant input–output streams, and variables

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any \TeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_iow
\g_tmpb_iow
New: 2017-12-11

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

\c_log_iow
\c_term_iow

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any \TeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_iow
\g_tmpb_iow
New: 2017-12-11

1.5 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 File operation functions

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (i.e. if it is in the \TeX search path), and does not end in / other than the case that it is exactly equal to the root directory. The ⟨name⟩ and ⟨ext⟩ parts together make up the file name, thus the ⟨name⟩ part may be thought of as the “job name” for the current file. Note that \TeX does not provide information on the ⟨ext⟩ part for the main (top level) file and that this file always has an empty ⟨dir⟩ component. Also, the ⟨name⟩ here will be equal to \c_sys_jobname_str, which may be different from the real file name (if set using --jobname, for example).
Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

\texttt{\LaTeXhackers note:} When working as a package in \LaTeX{} 2\epsilon, expl3 will automatically append the current \texttt{\input@path} to the set of values from \texttt{\l_file_search_path_seq}.

\texttt{\file_if_exist:nTF} \texttt{\{\{file name\}\{\{true code\}\{false code\}\}}

Searches for (\texttt{file name}) using the current \LaTeX{} search path and the additional paths controlled by \texttt{\l_file_search_path_seq}.

\texttt{\file_get:nnN} \texttt{\\{\{filename\}\{\{setup\}\{tl\}\}}
\texttt{\file_get:nnNTF} \texttt{\{\{filename\}\{\{setup\}\{tl\}\{\{true code\}\{false code\}\}}

Defines (\texttt{tl}) to the contents of (\texttt{filename}). Category codes may need to be set appropriately via the (\texttt{setup}) argument. The non-branching version sets the (\texttt{tl}) to \texttt{\q_no_value} if the file is not found. The branching version runs the (true code) after the assignment to (\texttt{tl}) if the file is found, and (false code) otherwise.

\texttt{\file_get_full_name:nN} \texttt{\\{\{file name\}\}}
\texttt{\file_get_full_name:VN} \texttt{\\{\{file name\}\}}
\texttt{\file_get_full_name:nNTF} \texttt{\{\{file name\}\{\{true code\}\{false code\}\}}
\texttt{\file_get_full_name:VNTF} \texttt{\{\{file name\}\{\{true code\}\{false code\}\}}

Searches for (\texttt{file name}) in the path as detailed for \texttt{\file_if_exist:nTF}, and if found sets the (\texttt{tl var}) the fully-qualified name of the file, i.e. the path and file name. This includes an extension .\texttt{tex} when the given (\texttt{file name}) has no extension but the file found has that extension. In the non-branching version, the (\texttt{tl var}) will be set to \texttt{\q_no_value} in the case that the file does not exist.

\texttt{\file_full_name:n} \texttt{\{\{file name\}\}}
\texttt{\file_full_name:V} \texttt{\{\{file name\}\}}

Searches for (\texttt{file name}) in the path as detailed for \texttt{\file_if_exist:nTF}, and if found leaves the fully-qualified name of the file, i.e. the path and file name, in the input stream. This includes an extension .\texttt{tex} when the given (\texttt{file name}) has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.
Parses the \emph{full name} and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The \emph{dir}: everything up to the last `/ (path separator) in the \emph{file path}. As with system \TeX{} variables and related functions, the \emph{dir} does \textit{not} include the trailing `/ unless it points to the root directory. If there is no path (only a file name), \emph{dir} is empty.
- The \emph{name}: everything after the last `/ up to the last "." where both of those characters are optional. The \emph{name} may contain multiple "," characters. It is empty if \emph{full name} consists only of a directory name.
- The \emph{ext}: everything after the last "." (including the dot). The \emph{ext} is empty if there is no "." after the last `/.

This function does not expand the \emph{full name} before turning it to a string. It assume that the \emph{full name} either contains no quote (" ) characters or is surrounded by a pair of quotes.

Searches for \emph{file name} using the current \TeX{} search path and the additional paths controlled by \texttt{\_file_search_path_seq}. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most \TeX{} behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The \{\emph{start index}\} and \{\emph{end index}\} values work as described for \texttt{\str_range:nnn}.

Sets the \emph{tl var} to the result of applying \texttt{\file_hex_dump:n/\file_hex_dump:nnn} to the \emph{file}. If the file is not found, the \emph{tl var} will be set to \texttt{\q_no_value}.

Searches for \emph{file name} using the current \TeX{} search path and the additional paths controlled by \texttt{\_file_search_path_seq}. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most \TeX{} behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

Sets the \emph{tl var} to the result of applying \texttt{\file_mdfive_hash:n} to the \emph{file}. If the file is not found, the \emph{tl var} will be set to \texttt{\q_no_value}.
\file_size:n \{(file name)\}

Searches for \texttt{(file name)} using the current \TeX{} search path and the additional paths controlled by \texttt{\_file_search_path_seq}. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

\file_get_size:nN \{(file name)\} \langle tl var \rangle

Sets the \langle tl var \rangle to the result of applying \file_size:n to the \langle file \rangle. If the file is not found, the \langle tl var \rangle will be set to \texttt{\_no\_value}. This is not available in older versions of \Xe\TeX{}.

\file_timestamp:n \{(file name)\}

Searches for \texttt{(file name)} using the current \TeX{} search path and the additional paths controlled by \texttt{\_file_search_path_seq}. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form `D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minute⟩⟨second⟩⟨offset⟩`, where the latter may be `Z` (UTC) or `⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'`. When the file is not found, the result of expansion is empty. This is not available in older versions of \Xe\TeX{}.

\file_get_timestamp:nN \langle tl var \rangle

Sets the \langle tl var \rangle to the result of applying \file_timestamp:n to the \langle file \rangle. If the file is not found, the \langle tl var \rangle will be set to \texttt{\_no\_value}. This is not available in older versions of \Xe\TeX{}.

\file_compare_timestamp:nNn \langle file-1 \rangle \langle comparator \rangle \langle file-2 \rangle \langle true code \rangle \langle false code \rangle

Compares the file stamps on the two \langle files \rangle as indicated by the \langle comparator \rangle, and inserts either the \langle true code \rangle or \langle false case \rangle as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

\file_compare_timestamp:nNnT \{ source-file \} > \{ derived-file \}

\{
% Code to regenerate derived file
\}

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of \Xe\TeX{}.

\file_input:n \{(file name)\}

Searches for \texttt{(file name)} in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional \La\TeX{} source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.
\texttt{\textbackslash file\_if\_exist\_input:n} \{\texttt{file name}\} \texttt{\textbackslash file\_if\_exist\_input:nF} \{\texttt{false code}\}

Searches for \texttt{file name} using the current \TeX{} search path and the additional paths controlled by \texttt{\textbackslash file\_path\_include:n}. If found then reads in the file as additional \LaTeX{} source as described for \texttt{\textbackslash file\_input:n}, otherwise inserts the \texttt{(false code)}. Note that these functions do not raise an error if the file is not found, in contrast to \texttt{\textbackslash file\_input:n}.

\texttt{\textbackslash file\_input\_stop:}

Ends the reading of a file started by \texttt{\textbackslash file\_input:n} or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \texttt{\msg\_critical:nn(nn)} should be preferred.

\textbf{\TeX{}hackers note:} This function must be used on a line on its own: \TeX{} reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

\texttt{\textbackslash file\_show\_list:} \texttt{\textbackslash file\_log\_list:}

These functions list all files loaded by \LaTeX{}\texttt{2e} commands that populate \texttt{@filelist} or by \texttt{\textbackslash file\_input:n}. While \texttt{\textbackslash file\_show\_list:} displays the list in the terminal, \texttt{\textbackslash file\_log\_list:} outputs it to the log file only.
Part XX
The l3skip package
Dimensions and skips

\LaTeXe provides two general length variables: \texttt{dim} and \texttt{skip}. Lengths stored as \texttt{dim} variables have a fixed length, whereas \texttt{skip} lengths have a rubber (stretch/shrink) component. In addition, the \texttt{muskip} type is available for use in math mode: this is a special form of \texttt{skip} where the lengths involved are determined by the current math font (in \texttt{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 \texttt{dim} variables

\begin{verbatim}
\dim_new:N \dim_new:c
\end{verbatim}

Creates a new \texttt{dimension} or raises an error if the name is already taken. The declaration is global. The \texttt{dimension} is initially equal to 0 pt.

\begin{verbatim}
\dim_const:Nn \dim_const:cn
\end{verbatim}

Creates a new constant \texttt{dimension} or raises an error if the name is already taken. The value of the \texttt{dimension} is set globally to the \texttt{dimension expression}.

\begin{verbatim}
\dim_zero:N \dim_zero:c \dim_gzero:N \dim_gzero:c
\end{verbatim}

Sets \texttt{dimension} to 0 pt.

\begin{verbatim}
\dim_zero_new:N \dim_zero_new:c \dim_gzero_new:N \dim_gzero_new:c
\end{verbatim}

Ensures that the \texttt{dimension} exists globally by applying \texttt{dim_new:N} if necessary, then applies \texttt{dim_(g)zero:N} to leave the \texttt{dimension} set to zero.

\begin{verbatim}
\dim_if_exist_p:N \dim_if_exist:NTF \dim_if_exist:N \dim_if_exist:c
\end{verbatim}

Tests whether the \texttt{dimension} is currently defined. This does not check that the \texttt{dimension} really is a dimension variable.
2 Setting dim variables

\input{sections/dim.tex}

\section{Utilities for dimension calculations}

\input{sections/dimexpressions.tex}
\dim_ratio:nn \{\dimexpr1\}\{\dimexpr2\}

Parses the two \langle \text{dimension expressions} \rangle and converts the ratio of the two to a form suitable for use inside a \langle \text{dimension expression} \rangle. This ratio is then left in the input stream, allowing syntax such as

\dim_set:Nn \l_my_dim
\{10\text{ pt} \ast \dim_ratio:nn \{5\text{ pt}\}\{10\text{ pt}\}\}

The output of \dim_ratio:nn on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

\tl_set:Nx \l_my_tl \{ \dim_ratio:nn \{5\text{ pt}\}\{10\text{ pt}\}\}
\tl_show:N \l_my_tl

displays 327680/655360 on the terminal.

4 Dimension expression conditionals

\dim_compare_p:nNn \{\dimexpr1\}\{\relax\}\{\dimexpr2\}\}
\dim_compare:nNnTF
\{\dimexpr1\}\{\relax\}\{\dimexpr2\}\}
\{\text{true code}\}\{\text{false code}\}\}

This function first evaluates each of the \langle \text{dimension expressions} \rangle as described for \dim_eval:n. The two results are then compared using the \langle \text{relation} \rangle:

Equal =
Greater than >
Less than <

This function is less flexible than \dim_compare:nTF but around 5 times faster.
\texttt{\dim_compare_p:n} \texttt{\dim_compare:nTF} *

\begin{align*}
&\{ \\
&\langle \text{dimexpr}_1 \rangle \text{ relation}_1 \\
&\quad \ldots \\
&\langle \text{dimexpr}_N \rangle \text{ relation}_N \\
&\langle \text{dimexpr}_{N+1} \rangle \\
&\}\ \\
&\texttt{\dim_compare:nTF} \\
&\{ \\
&\langle \text{dimexpr}_1 \rangle \text{ relation}_1 \\
&\quad \ldots \\
&\langle \text{dimexpr}_N \rangle \text{ relation}_N \\
&\langle \text{dimexpr}_{N+1} \rangle \\
&\}\ \\
&\{\langle \text{true code} \rangle\} \{\langle \text{false code} \rangle\}
\end{align*}

This function evaluates the \textit{(dimension expressions)} as described for \texttt{\dim_eval:n} and compares consecutive result using the corresponding \textit{(relation)}, namely it compares \textit{\langle \text{dimexpr}_1 \rangle} and \textit{\langle \text{dimexpr}_2 \rangle} using the \textit{(relation}_1\rangle), then \textit{\langle \text{dimexpr}_2 \rangle} and \textit{\langle \text{dimexpr}_3 \rangle} using the \textit{(relation}_2\rangle), until finally comparing \textit{\langle \text{dimexpr}_N \rangle} and \textit{\langle \text{dimexpr}_{N+1} \rangle} using the \textit{(relation}_N\rangle). The test yields \texttt{true} if all comparisons are \texttt{true}. Each \textit{(dimension expression)} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \texttt{false}, then no other \textit{(dimension expression)} is evaluated and no other comparison is performed. The \textit{(relations)} can be any of the following:

\begin{itemize}
\item Equal \quad = \text{ or } ==
\item Greater than or equal to \quad >=
\item Greater than \quad >
\item Less than or equal to \quad <=
\item Less than \quad <
\item Not equal \quad !=
\end{itemize}

This function is more flexible than \texttt{\dim_compare:nNnTF} but around 5 times slower.
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 and other cases are discarded. 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

\begin{verbatim}
\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! }
\end{verbatim}

leaves “Medium” in the input stream.

### 5 Dimension expression loops

\begin{verbatim}
\dim_do_until:nNnn \{ \dimexpr \text{expression}_1 \} \{ \dimexpr \text{expression}_2 \} \{ \text{code} \}
\end{verbatim}

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 \texttt{false} then the \(code\) is inserted into the input stream again and a loop occurs until the \(relation\) is \texttt{true}.

\begin{verbatim}
\dim_do_while:nNnn \{ \dimexpr \text{expression}_1 \} \{ \dimexpr \text{expression}_2 \} \{ \text{code} \}
\end{verbatim}

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 \texttt{true} then the \(code\) is inserted into the input stream again and a loop occurs until the \(relation\) is \texttt{false}.

\begin{verbatim}
\dim_until_do:nNnn \{ \dimexpr \text{expression}_1 \} \{ \dimexpr \text{expression}_2 \} \{ \text{code} \}
\end{verbatim}

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 \texttt{false}. After the \(code\) has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{true}. 
\texttt{\textbackslash \text{dim}{\_\text{while}}:nNn} \quad \texttt{\textbackslash \text{dim}{\_\text{while}}:nNn \{\text{dimexpr}_1\} \{\text{relation}\} \{\text{dimexpr}_2\} \{\text{code}\}}

Evaluates the relationship between the two \textit{(dimension expressions)} as described for \texttt{\textbackslash \text{dim}{\_\text{compare}}:nNnTF}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \texttt{true}. After the \textit{(code)} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{false}.

\texttt{\textbackslash \text{dim}{\_\text{do}}:nn} \quad \texttt{\textbackslash \text{dim}{\_\text{do}}:nn \{\text{dimension relation}\} \{\text{code}\}}

Places the \textit{(code)} in the input stream for \TeX to process, and then evaluates the \textit{(dimension relation)} as described for \texttt{\textbackslash \text{dim}{\_\text{compare}}:nNnTF}. If the test is \texttt{false} then the \textit{(code)} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \texttt{true}.

\texttt{\textbackslash \text{dim}{\_\text{while}}:nn} \quad \texttt{\textbackslash \text{dim}{\_\text{while}}:nn \{\text{dimension relation}\} \{\text{code}\}}

Places the \textit{(code)} in the input stream for \TeX to process, and then evaluates the \textit{(dimension relation)} as described for \texttt{\textbackslash \text{dim}{\_\text{compare}}:nNnTF}. If the test is \texttt{true} then the \textit{(code)} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \texttt{false}.

\texttt{\textbackslash \text{until}_\text{do}:nn} \quad \texttt{\textbackslash \text{until}_\text{do}:nn \{\text{dimension relation}\} \{\text{code}\}}

Evaluates the \textit{(dimension relation)} as described for \texttt{\textbackslash \text{dim}{\_\text{compare}}:nNnTF}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \texttt{false}. After the \textit{(code)} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash \text{while}_\text{do}:nn} \quad \texttt{\textbackslash \text{while}_\text{do}:nn \{\text{dimension relation}\} \{\text{code}\}}

Evaluates the \textit{(dimension relation)} as described for \texttt{\textbackslash \text{dim}{\_\text{compare}}:nNnTF}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \texttt{true}. After the \textit{(code)} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{false}.

6 Dimension step functions

\texttt{\textbackslash \text{dim}{\_\text{step}}:nNnN} \quad \texttt{\textbackslash \text{dim}{\_\text{step}}:nNnN \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{function}\}}

This function first evaluates the \textit{(initial value)}, \textit{(step)} and \textit{(final value)}, all of which should be dimension expressions. The \textit{(function)} is then placed in front of each \textit{(value)} from the \textit{(initial value)} to the \textit{(final value)} in turn (using \textit{(step)} between each \textit{(value)}). The \textit{(step)} must be non-zero. If the \textit{(step)} is positive, the loop stops when the \textit{(value)} becomes larger than the \textit{(final value)}. If the \textit{(step)} is negative, the loop stops when the \textit{(value)} becomes smaller than the \textit{(final value)}. The \textit{(function)} should absorb one argument.

\texttt{\textbackslash \text{dim}{\_\text{step}}:nNnN} \quad \texttt{\textbackslash \text{dim}{\_\text{step}}:nNnN \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{code}\}}

This function first evaluates the \textit{(initial value)}, \textit{(step)} and \textit{(final value)}, all of which should be dimension expressions. Then for each \textit{(value)} from the \textit{(initial value)} to the \textit{(final value)} in turn (using \textit{(step)} between each \textit{(value)}), the \textit{(code)} is inserted into the input stream with \texttt{\#1} replaced by the current \textit{(value)}. Thus the \textit{(code)} should define a function of one argument \texttt{\#1}.

173
This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be dimension expressions. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream, with the \langle tl var \rangle defined as the current \langle value \rangle. Thus the \langle code \rangle should make use of the \langle tl var \rangle.

### 7 Using dim expressions and variables

\verbatim\dim_step_variable:nnnNn
\verbatim{\langle initial value \rangle \langle step \rangle \langle final value \rangle \langle tl var \rangle \langle code \rangle}

\dim_eval:n \{\langle dimension expression \rangle\}

Evaluates the \langle dimension expression \rangle, expanding any dimensions and token list variables within the \langle expression \rangle to their content (without requiring \texttt{\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 \langle dimension denotation \rangle after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \TeX{}-style assignment as it is not an \langle internal dimension \rangle.

\dim_sign:n \{\langle dimexpr \rangle\}

Evaluates the \langle dimexpr \rangle then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\dim_use:N \{} \dim_use:c \{

Recovers the content of a \langle dimension \rangle and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a \langle dimension \rangle is required (such as in the argument of \texttt{\dim_eval:n}).

\TeX{}hackers note: \texttt{\dim_use:N} is the \TeX{} primitive \texttt{\the}: this is one of several \texttt{\LaTeX3} names for this primitive.

\dim_to_decimal:n \{\langle dimexpr \rangle\}

Evaluates the \langle dimension expression \rangle, 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

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

leaves 1.00374 in the input stream, \textit{i.e.} the magnitude of one “big point” when converted to (\TeX{}) points.

\textbf{New: 2018-11-03}
\dim_to_decimal_in_bp:n \{\dimexpr\}\n
Evaluates the \textit{(dimension expression)}, and leaves the result, expressed in big points (bp) in the input stream, \textit{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 \{1\text{pt}\}

leaves 0.99628 in the input stream, \textit{i.e.} the magnitude of one (\TeX{)} point when converted to big points.

\dim_to_decimal_in_sp:n \{\dimexpr\}\n
Evaluates the \textit{(dimension expression)}, and leaves the result, expressed in scaled points (sp) in the input stream, \textit{with no units}. The result is necessarily an integer.

\dim_to_decimal_in_unit:nn \{\dimexpr\}\{\dimexpr\}\n
Evaluates the \textit{(dimension expressions)}, and leaves the value of \textit{(dimexpr)}\textsubscript{1}, expressed in a unit given by \textit{(dimexpr)}\textsubscript{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 \{1\text{bp}\}\{1\text{mm}\}

leaves 0.35277 in the input stream, \textit{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 \epsilon-\TeX{} primitives, which is required internally by \dim_to_decimal_in_unit:nn.

\dim_to_fp:n \{\dimexpr\}\n
Expands to an internal floating point number equal to the value of the \textit{(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 and a smaller range are acceptable.

8 Viewing \textit{dim} variables

\dim_show:N \dim_show:c

Displays the value of the \textit{(dimension)} on the terminal.
\dim_show:n \dim_show:n \{(dimension expression)\}
Displays the result of evaluating the \{(dimension expression)\} on the terminal.
\dim_log:N \dim_log:N \dim_log:c
\dim_log:n \dim_log:n \{(dimension expression)\}
Writes the value of the \{(dimension)\} in the log file.
\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.

10 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.

11 Creating and initialising skip variables
\skip_new:N \skip_new:N \{(skip)\}
Creates a new \{(skip)\} or raises an error if the name is already taken. The declaration is global. The \{(skip)\} is initially equal to 0pt.
\skip_const:Nn \skip_const:cn
New: 2012-03-05
\skip_zero:N \skip_zero:c \skip_gzero:N \skip_gzero:c
\skip_zero_new:N \skip_zero_new:c \skip_gzero_new:N \skip_gzero_new:c
New: 2012-01-07
\skip_if_exist_p:N * \skip_if_exist_p:c * \skip_if_exist:NTF \skip_if_exist:cIF *
New: 2012-03-03
\skip_add:Nn \skip_add:cn \skip_gadd:Nn \skip_gadd:cn
Updated: 2011-10-22
\skip_set:Nn \skip_set:cn \skip_gset:Nn \skip_gset:cn
Updated: 2011-10-22
\skip_set_eq:NN \skip_set_eq:(cN|Nc|cc) \skip_gset_eq:NN \skip_gset_eq:(cN|Nc|cc)
\skip_sub:Nn \skip_sub:cn \skip_gsub:Nn \skip_gsub:cn
Updated: 2011-10-22

\skip_const:Nn \skip (skip) \{\langle \text{skip expression} \rangle\}
Creates a new constant \langle \text{skip} \rangle or raises an error if the name is already taken. The value of the \langle \text{skip} \rangle is set globally to the \langle \text{skip expression} \rangle.

\skip_zero:N \skip (skip)
Sets \langle \text{skip} \rangle to 0 pt.

\skip_zero_new:N \skip (skip)
Ensures that the \langle \text{skip} \rangle exists globally by applying \skip_new:N if necessary, then applies \skip_(g)zero:N to leave the \langle \text{skip} \rangle set to zero.

\skip_if_exist_p:N \skip_if_exist:NTF \langle \text{true code} \rangle \{\langle \text{false code} \rangle\}
Tests whether the \langle \text{skip} \rangle is currently defined. This does not check that the \langle \text{skip} \rangle really is a skip variable.

12 Setting skip variables

\skip_add:Nn \skip (skip) \{\langle \text{skip expression} \rangle\}
Adds the result of the \langle \text{skip expression} \rangle to the current content of the \langle \text{skip} \rangle.

\skip_set:Nn \skip (skip) \{\langle \text{skip expression} \rangle\}
Sets \langle \text{skip} \rangle to the value of \langle \text{skip expression} \rangle, 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 \langle \text{skip}1 \rangle \langle \text{skip}2 \rangle
Sets the content of \langle \text{skip}1 \rangle equal to that of \langle \text{skip}2 \rangle.

\skip_sub:Nn \skip (skip) \{\langle \text{skip expression} \rangle\}
Subtracts the result of the \langle \text{skip expression} \rangle from the current content of the \langle \text{skip} \rangle.
13 Skip expression conditionals

\skip_if_eq_p:nn \skip_if_eq:nnTF
\skip_if_finite_p:n \skip_if_finite:nTF

This function first evaluates each of the \langle skip expressions \rangle as described for \skip_eval:n. The two results are then compared for exact equality, \ie 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

Evaluates the \langle skip expression \rangle as described for \skip_eval:n, and then tests if all of its components are finite.

14 Using skip expressions and variables

\skip_eval:n \skip_use:N \skip_use:c

Evaluates the \langle skip expression \rangle, expanding any skips and token list variables within the \langle expression \rangle 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 \langle glue denotation \rangle after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \TeX-style assignment as it is not an \langle internal glue \rangle.

\skip_use:N \skip_use:c

Recovers the content of a \langle skip \rangle and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a \langle dimension \rangle or \langle skip \rangle 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 \TeX primitives.

15 Viewing skip variables

\skip_show:N \skip_show:c

Displays the value of the \langle skip \rangle on the terminal.

\skip_show:n

Displays the result of evaluating the \langle skip expression \rangle on the terminal.
16 Constant skips

\c_max_skip
\c_zero_skip

Updated: 2012-11-02
Updated: 2012-11-01

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

17 Scratch skips

\l_tmpa_skip
\l_tmpb_skip
\g_tmpa_skip
\g_tmpb_skip

Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any \LTTEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any \LTTEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

18 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n

Updated: 2011-10-22

Inserts a horizontal \langle skip \rangle into the current list. The argument can also be a \langle dim \rangle.

\texttt{T\TeX}hackers note: \texttt{\skip_horizontal:N} is the \texttt{T\TeX} primitive \texttt{\hskip} renamed.
\skip_vertical:N \skip_vertical:C \skip_vertical:n

Inserted a vertical \langle skip \rangle into the current list. The argument can also be a \langle dim \rangle.

\TeX{}hackers note: \skip_vertical:N is the \TeX{} primitive \vskip renamed.

19 Creating and initialising \texttt{muskip} variables

\miskkip_new:N \miskkip_new:C

Creates a new \langle muskip \rangle or raises an error if the name is already taken. The declaration is global. The \langle muskip \rangle is initially equal to 0 mu.

\miskkip_const:Nn \miskkip_const:cn

Creates a new constant \langle muskip \rangle or raises an error if the name is already taken. The value of the \langle muskip \rangle is set globally to the \langle muskip expression \rangle.

\miskkip_zero:N \miskkip_zero:C \miskkip_gzero:N \miskkip_gzero:C

Sets \langle muskip \rangle to 0 mu.

\miskkip_zero_new:N \miskkip_zero_new:C \miskkip_gzero_new:N \miskkip_gzero_new:C

Ensures that the \langle muskip \rangle exists globally by applying \miskkip_new:N if necessary, then applies \miskkip_(g)zero:N to leave the \langle muskip \rangle set to zero.

\miskkip_if_exist_p:N \miskkip_if_exist_p:C \miskkip_if_exist:NTF \miskkip_if_exist:CTF

Tests whether the \langle muskip \rangle is currently defined. This does not check that the \langle muskip \rangle really is a muskip variable.

20 Setting \texttt{muskip} variables

\miskkip_add:Nn \miskkip_add:cn \miskkip_gadd:Nn \miskkip_gadd:cn

Adds the result of the \langle muskip expression \rangle to the current content of the \langle muskip \rangle.
Sets \langle \textit{muskip} \rangle to the value of \langle \textit{muskip expression} \rangle, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

\textbf{21 Using muskip expressions and variables}

Evaluates the \langle \textit{muskip expression} \rangle, expanding any skips and token list variables within the \langle \textit{expression} \rangle to their content (without requiring \texttt{\mbox{\textbackslash muskip\textbackslash use:N}}/\texttt{\mbox{\textbackslash tl\textbackslash use:N}}) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a \langle \textit{muglue denotation} \rangle after two expansions. This is expressed in mu, and requires suitable termination if used in a \LaTeX{}-style assignment as it is not an \langle \textit{internal muglue} \rangle.

\textbf{22 Viewing muskip variables}

Displays the value of the \langle \textit{muskip} \rangle on the terminal.
\muskip\_show:n \muskip\_show:n \{\muskip\ expression\}\n
Displays the result of evaluating the \langle\muskip\ expression\rangle on the terminal.

\muskip\_log:N  \muskip\_log:N \muskip

\muskip\_log:c

New: 2014-08-22
Updated: 2015-08-07

\muskip\_log:n \muskip\_log:n \{\muskip\ expression\}\n
New: 2014-08-22
Updated: 2015-08-07

\muskip\_log:n \muskip\_log:n \{\muskip\ expression\}\n
New: 2014-08-22
Updated: 2015-08-07

23 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.

24 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.

25 Primitive conditional

\if\dim:w \if\dim:w \{\dimen1\} \{\text{relation}\} \{\dimen2\}
\{\text{true code}\}
\else:
\{false\}
\fi:

Compare two dimensions. The \langle\text{relation}\rangle is one of \text{<, = or >} with category code 12.

\LaTeXhackers\ note: This is the \LaTeX primitive \texttt{\textbackslashifdim}.
Part XXI

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 is 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. As 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
}

creates a key called \l_mymodule_tmp_tl, and not one called key.

1 Creating keys

\keys_define:nn
\keys_define:nn {⟨module⟩} {⟨keyval list⟩}

Parses the ⟨keyval list⟩ and defines the keys listed there for ⟨module⟩. The ⟨module⟩ name is treated as a string. 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., override one another. Some other properties are mutually exclusive, notably .value_required:n and .value_forbidden:n, and so they 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 }
{

Note that with the exception of the special .undefine property, all key properties define the key within the current TeX scope.

\begin{verbatim}
keyname .value_required:n = true,
keyname .code:n = Some-code-using-#1
\end{verbatim}

\begin{verbatim}
.\bool_set:N
.\bool_set:c
.\bool_gset:N
.\bool_gset:c
\end{verbatim}

Updated: 2013-07-08

\begin{verbatim}
(key) .\bool_set_inverse:N = (boolean)
\end{verbatim}

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.

\begin{verbatim}
(key) .\bool_set_inverse:N = (boolean)
\end{verbatim}

\begin{verbatim}
\choices:nn = { (choices) } { (code) }
\end{verbatim}

Sets \(key\) to act as a choice key. Each valid choice for \(key\) must then be created, as discussed in section 3.

\begin{verbatim}
(key) .\clist_set:N = (comma list variable)
\end{verbatim}

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 is created globally at the point that the key is set up.

\begin{verbatim}
(key) .\code:n = { (code) }
\end{verbatim}

Stores the \(code\) for execution when \(key\) is used. The \(code\) can include one parameter (#1), which will be the \(value\) given for the \(key\).

\begin{verbatim}
(key) .\cs_set:Np = { (control sequence) } { arg. spec. }
\end{verbatim}

Defines \(key\) to set \(control sequence\) to have \(arg. spec.\) and replacement text \(value\).
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 does not trigger an error.

\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 does not trigger an error.

\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 does not trigger an error.

\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 does not trigger an error.


Initialises the \textit{key} with the \textit{value}, equivalent to
\begin{verbatim}
\keys_set:nn \{\textit{module}\} \{ \textit{key} = \textit{value} \}
\end{verbatim}

\begin{verbatim}
\int_set:N \int_set:c \int_gset:N \int_gset:c
\end{verbatim}
Defines \textit{key} to set \textit{integer} to \textit{value} (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

\begin{verbatim}
\meta:n
\end{verbatim}
Makes \textit{key} a meta-key, which will set \textit{keyval list} in one go. The \textit{keyval list} can refer as \#1 to the value given at the time the \textit{key} is used (or, if no value is given, the \textit{key}'s default value).

\begin{verbatim}
\meta:nn
\end{verbatim}
Makes \textit{key} a meta-key, which will set \textit{keyval list} in one go using the \textit{path} in place of the current one. The \textit{keyval list} can refer as \#1 to the value given at the time the \textit{key} is used (or, if no value is given, the \textit{key}'s default value).

\begin{verbatim}
\multichoice:
\end{verbatim}
Sets \textit{key} to act as a multiple choice key. Each valid choice for \textit{key} must then be created, as discussed in section 3.

\begin{verbatim}
\multichoice:nn \multichoice:c \multichoice:
\end{verbatim}
Sets \textit{key} to act as a multiple choice key, and defines a series \textit{choices} which are implemented using the \textit{code}. Inside \textit{code}, \texttt{\l_keys_choice_tl} will be the name of the choice made, and \texttt{\l_keys_choice_int} will be the position of the choice in the list of \textit{choices} (indexed from 1). Choices are discussed in detail in section 3.

\begin{verbatim}
\muskip_set:N \muskip_set:c \muskip_gset:N \muskip_gset:c
\end{verbatim}
Defines \textit{key} to set \textit{muskip} to \textit{value} (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

\begin{verbatim}
\prop_put:N \prop_put:c \prop_gput:N \prop_gput:c
\end{verbatim}
Defines \textit{key} to put the \textit{value} onto the \textit{property list} stored under the \textit{key}. If the variable does not exist, it is created globally at the point that the key is set up.
\[ \text{key} \].skip_set:N = \langle \text{skip} \rangle \\
Defines \langle \text{key} \rangle to set \langle \text{skip} \rangle to \langle \text{value} \rangle (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

\[ \text{key} \].tl_set:N = \langle \text{token list variable} \rangle \\
Defines \langle \text{key} \rangle to set \langle \text{token list variable} \rangle to \langle \text{value} \rangle. If the variable does not exist, it is created globally at the point that the key is set up.

\[ \text{key} \].tl_set_x:N = \langle \text{token list variable} \rangle \\
Defines \langle \text{key} \rangle to set \langle \text{token list variable} \rangle to \langle \text{value} \rangle, which will be subjected to an \text{x-type expansion (i.e. using} \text{tl_set:Nx)}. If the variable does not exist, it is created globally at the point that the key is set up.

\[ \text{key} \].undefine: \\
Removes the definition of the \langle \text{key} \rangle within the current scope.

\[ \text{key} \].value_forbidden:n = \text{true}|\text{false} \\
Specifies that \langle \text{key} \rangle cannot receive a \langle \text{value} \rangle when used. If a \langle \text{value} \rangle is given then an error will be issued. Setting the property \text{false} cancels the restriction.

\[ \text{key} \].value_required:n = \text{true}|\text{false} \\
Specifies that \langle \text{key} \rangle must receive a \langle \text{value} \rangle when used. If a \langle \text{value} \rangle is not given then an error will be issued. Setting the property \text{false} cancels 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:

\[
\text{\textbackslash keys_define:nn} \{ \text{mymodule} / \text{subgroup} \} \\
\{ \text{key .code:n = code} \}
\]

or to the key name:

\[
\text{\textbackslash keys_define:nn} \{ \text{mymodule} \} \\
\{ \text{subgroup} / \text{key .code:n = code} \}
\]

As illustrated, the best choice of token for sub-dividing keys in this way is \text{/}. 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 \text{mymodule/subgroup/key}. As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.
3 Choice and multiple choice keys

The \texttt{keys} 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 \texttt{.choice:} property:

\begin{verbatim}
\keys_define:nn { mymodule }
  { key .choice: }
\end{verbatim}

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 \texttt{.choices:nn} property.

\begin{verbatim}
\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. }
  }
\end{verbatim}

The index \texttt{\l_keys_choice_int} in the list of choices starts at 1.

\begin{verbatim}
\\l_keys_choice_int
\\l_keys_choice_tl
\end{verbatim}

Inside the code block for a choice generated using \texttt{.choices:nn}, the variables \texttt{\l_keys_choice_tl} and \texttt{\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 \texttt{.code:n}, the value passed to the key (i.e. the choice name) is also available as \texttt{#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 \texttt{.choice:} property of a key, then manually defining sub-keys.

\begin{verbatim}
\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,
  }
\end{verbatim}

It is possible to mix the two methods, but manually-created choices should \texttt{not} use \texttt{\l_keys_choice_tl} or \texttt{\l_keys_choice_int}. These variables do not have defined
behaviour when used outside of code created using \texttt{choices:nn} (\textit{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 \texttt{unknown} choice. The general behavior of the \texttt{unknown} key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

\begin{verbatim}
\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:nxxx { mymodule } { unknown-choice }
    { key } \% Name of choice key
    { choice-a , choice-b , choice-c } \% Valid choices
    { \exp_not:n {#1} } \% Invalid choice given
}
\end{verbatim}

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties \texttt{.multichoice:} and \texttt{.multichoices:nn}. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

\begin{verbatim}
\keys_define:nn { mymodule }
{
  key .multichoices:nn =
  { choice-a, choice-b, choice-c }
  { You-gave-choice-`\\l_use:N \l_keys_choice_tl',-
    which-is-in-position-
    \int_use:N \l_keys_choice_int \c_space_tl
    in-the-list. }
}
\end{verbatim}

and

\begin{verbatim}
\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,
}
\end{verbatim}

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.

\section{Setting keys}

\keys_set:nn \keys_set:{(nV|nv|no)}
\keys_set:nn \keys_set:nn \keys_set:nn \keys_set:nn
\keys_set:nn \keys_set:nn \keys_set:nn \keys_set:nn
\keys_set:nn \keys_set:nn

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_str. 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_str.

\section{Handling of unknown keys}

If a key has not previously been defined (is unknown), \keys_set:nn looks 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.

```tex
\keys_define:nn { mymodule }
{  
    unknown .code:n =  
        You-tried-to-set-key-’\l_keys_key_str’-to-’\#1’.  
}
```

These functions set keys which are known for the ⟨module⟩, and simply ignore other keys. The \keys_set_known:nn 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. In addition, \keys_set_known:nnN stores the key–value pairs in the ⟨tl⟩ in comma-separated form (i.e. an edited version of the ⟨keyval list⟩). When a ⟨root⟩ is given (\keys_set_known:nnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

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

```tex
\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 attempts 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

```tex
\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 ,  
```
assigns 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:nnn \keys_set_filter:nnnnN \keys_set_filter:nnnN

Activates key filtering in an “opt-out” sense: keys assigned to any of the \langle groups \rangle specified are ignored. The \langle groups \rangle are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the \langle tl \rangle in a comma-separated form (i.e. an edited version of the \langle keyval list \rangle). The \keys_set_filter:nnn version skips this stage.

Use of \keys_set_filter:nnnN can be nested, with the correct residual \langle keyval list \rangle returned at each stage. In the version which takes a \langle root \rangle argument, the key list is returned relative to that point in the key tree. In the cases without a \langle root \rangle argument, only the key names and values are returned.

\keys_set_groups:nnn \keys_set_groups:nnnnN

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the \langle groups \rangle specified are set. The \langle groups \rangle are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

7 Utility functions for keys

\keys_if_exist_p:nn \keys_if_exist:nnTF

Tests if the \langle key \rangle exists for \langle module \rangle, i.e. if any code has been defined for \langle key \rangle.
Tests if the \langle choice \rangle is defined for the \langle key \rangle within the \langle module \rangle, i.e. if any code has been defined for \langle key \rangle/\langle choice \rangle. The test is \textbf{false} if the \langle key \rangle itself is not defined.

\section*{8 Low-level interface for parsing key–val lists}

To re-cap from earlier, a key–value list is input of the form

\begin{verbatim}
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
\end{verbatim}

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 \langle key–value list \rangle into \langle keys \rangle and associated \langle values \rangle. 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 have exactly one set removed (after space trimming), thus

\begin{verbatim}
key = {value here},
\end{verbatim}

and

\begin{verbatim}
key = value here,
\end{verbatim}
are treated identically.

\keyval_parse:NNn \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) is used to process keys given with no value and (function_2) is used to process keys given with a value. The order of the (keys) in the (key–value list) is preserved. Thus

\keyval_parse:NNn \function:n \function:nn
{ key1 = value1, key2 = value2, key3 = , key4 }

is 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 XXII
The \texttt{l3intarray} package: fast global integer arrays

1 \texttt{l3intarray} documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast \texttt{l3seq}, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (\textit{i.e.} one power lower than the usual $\verb|\c_max_int|$ ceiling of $2^{31} - 1$)

The use of \texttt{intarray} data is therefore recommended for cases where the need for fast access is of paramount importance.

\begin{verbatim}
\intarray_new:Nn \intarray_new:cn
New: 2018-03-29
\end{verbatim}

Evaluates the integer expression \texttt{(size)} and allocates an \texttt{integer array variable} with that number of (zero) entries. The variable name should start with \texttt{\g_} because assignments are always global.

\begin{verbatim}
\intarray_count:N \intarray_count:c
New: 2018-03-29
\end{verbatim}

Expands to the number of entries in the \texttt{integer array variable}. Contrarily to \texttt{\seq_count:N} this is performed in constant time.

\begin{verbatim}
\intarray_gset:Nnn \intarray_gset:cn
New: 2018-03-29
\end{verbatim}

Stores the result of evaluating the integer expression \texttt{(value)} into the \texttt{integer array variable} at the (integer expression) \texttt{(position)}. If the \texttt{(position)} is not between 1 and the \texttt{\intarray_count:N}, or the \texttt{(value)}’s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

\begin{verbatim}
\intarray_const_from_clist:Nn \intarray_const_from_clist:cn
New: 2018-05-04
\end{verbatim}

Creates a new constant \texttt{integer array variable} or raises an error if the name is already taken. The \texttt{integer array variable} is set (globally) to contain as its items the results of evaluating each \texttt{integer expression} in the \texttt{comma list}.

\begin{verbatim}
\intarray_gzero:N \intarray_gzero:c
New: 2018-05-04
\end{verbatim}

Sets all entries of the \texttt{integer array variable} to zero. Assignments are always global.
\texttt{\textbackslash intarray\_item:} \texttt{Nn} \texttt{\{intarray var\} \{(position)\}}

Expands to the integer entry stored at the (integer expression) \texttt{(position)} in the \texttt{(integer array variable)}. If the \texttt{(position)} is not between 1 and the \texttt{\textbackslash intarray\_count:N}, an error occurs.

\texttt{\textbackslash intarray\_rand\_item:N} \texttt{\{intarray var\}}

Selects a pseudo-random item of the \texttt{(integer array)}. If the \texttt{(integer array)} is empty, produce an error.

\texttt{\textbackslash intarray\_show:N} \texttt{\{intarray var\}}
\texttt{\textbackslash intarray\_log:N} \texttt{\{intarray var\}}

Displays the items in the \texttt{(integer array variable)} in the terminal or writes them in the log file.

1.1 Implementation notes

It is a wrapper around the \texttt{\fontdimen} primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to \texttt{l3seq} sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive \texttt{\fontdimen} stores dimensions but the \texttt{l3intarray} package transparently converts these from/to integers. Assignments are always global.

While Lua\TeX{}’s memory is extensible, other engines can “only” deal with a bit less than $4 \times 10^6$ entries in all \texttt{\fontdimen} arrays combined (with default \TeX{}Live settings).
Part XXIII

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 \( x \times y \), division \( x/y \), square root \( \sqrt{x} \), and parentheses.
- Comparison operators: \( x < y \), \( x \leq y \), \( x > y \), \( x \geq y \), etc.
- Boolean logic: sign \( \text{sign} x \), negation \( !x \), conjunction \( x \& \& y \), disjunction \( x \mid \mid y \), ternary operator \( x ? y : z \).
- Exponentials: \( \exp x \), \( \ln x \), \( x^y \), \( \log_b x \).
- Integer factorial: \( \text{fact} x \).
- Trigonometry: \( \sin x \), \( \cos x \), \( \tan x \), \( \cot x \), \( \sec x \), \( \csc x \) expecting their arguments in radians, and \( \text{sind} x \), \( \text{cosd} x \), \( \text{tand} x \), \( \text{cotd} x \), \( \text{secd} x \), \( \text{cscd} x \) expecting their arguments in degrees.
- Inverse trigonometric functions: \( \text{asin} x \), \( \text{acos} x \), \( \text{atan} x \), \( \text{acot} x \), \( \text{asec} x \), \( \text{acsc} x \) giving a result in radians, and \( \text{asind} x \), \( \text{acosd} x \), \( \text{atand} x \), \( \text{acotd} x \), \( \text{asecd} x \), \( \text{acscd} x \) giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: \( \sinh x \), \( \cosh x \), \( \tanh x \), \( \coth x \), \( \text{sech} x \), \( \text{csch} x \), and \( \text{asinh} x \), \( \text{acosh} x \), \( \text{atanh} x \), \( \text{acoth} x \), \( \text{asech} x \), \( \text{acsch} x \).
- Extrema: \( \max(x_1, x_2, \ldots) \), \( \min(x_1, x_2, \ldots) \), \( \text{abs}(x) \).
- Rounding functions, controlled by two optional values, \( n \) (number of places, 0 by default) and \( t \) (behavior on a tie, NaN by default):
  - \( \text{trunc}(x, n) \) rounds towards zero,
  - \( \text{floor}(x, n) \) rounds towards \( -\infty \),
  - \( \text{ceil}(x, n) \) rounds towards \( +\infty \),
  - \( \text{round}(x, n, t) \) 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”.
- Random numbers: \( \text{rand}() \), \( \text{randint}(m, n) \).
- Constants: \( \pi \), \( \text{deg} \) (one degree in radians).
- Dimensions, automatically expressed in points, e.g., \( \text{pc} \) is 12.
• Automatic conversion (no need for \texttt{\textbackslash type\_use:N}) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.

• Tuples: \((x_1,\ldots,x_n)\) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as \texttt{1.234e-34}, or \texttt{-0.0001}), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. 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} \$. 

The operation \texttt{round} can be used to limit the result’s precision. Adding \texttt{+0} avoids the possibly undesirable output \texttt{-0}, replacing it by \texttt{+0}. However, the \texttt{l3fp} 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.

\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m } { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum { 2 pi * \sin ( 2.3 ^ 5 ) }
\end{document}

See the documentation of \texttt{siunitx} for various options of \texttt{\num}.

1 Creating and initialising floating point variables

\texttt{\fp_new:N} \langle \texttt{fp var} \rangle
\texttt{\fp_new:c}

\texttt{\fp const:Nn} \langle \texttt{fp var} \rangle \{ \langle \texttt{floating point expression} \rangle \}
\texttt{\fp const:cn}

\texttt{\fp_zero:N} \langle \texttt{fp var} \rangle
\texttt{\fp_zero:c}
\texttt{\fp_gzero:N}
\texttt{\fp_gzero:c}

Updated: 2012-05-08
2 Setting floating point variables

\fp_set:Nn \fp_set:cn \fp_gset:Nn \fp_gset:cn
Updated: 2012-05-08

\fp_set_eq:NN \fp_set_eq:cn \fp_gset_eq:NN \fp_gset_eq:cn
Updated: 2012-05-08

\fp_add:Nn \fp_add:cn \fp_gadd:Nn \fp_gadd:cn
Updated: 2012-05-08

\fp_sub:Nn \fp_sub:cn \fp_gsub:Nn \fp_gsub:cn
Updated: 2012-05-08

3 Using floating points

\fp_eval:n \fp_eval:n (fp var)

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 ±∞ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_eval:n and they are combined as ((fp1), (fp2), … (fpn)) if n > 1 and ((fp1),) or () for fewer items. This function is identical to \fp_to_decimal:n.
\fp_sign:n \{\langle pexpr \rangle \}

Evaluates the \langle pexpr \rangle and leaves its sign in the input stream using \fp_eval:n {\text{sign(\langle result \rangle)}}: +1 for positive numbers and for +\infty, -1 for negative numbers and for -\infty, \pm 0 for \pm 0. If the operand is a tuple or is \text{NaN}, then “invalid operation” occurs and the result is 0.

\fp_decimal:N \langle fp \ var \rangle
\fp_decimal:n \{\langle floating point expression \rangle \}

Evaluates the \langle floating point expression \rangle 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 \text{NaN} trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_decimal:n and they are combined as \langle fp_1 \rangle, \langle fp_2 \rangle, \ldots \langle fp_n \rangle if n > 1 and \langle fp_1 \rangle, or () for fewer items.

\fp_to_decimal:N \fp_to_decimal:c \fp_to_decimal:n \⋆

Updated: 2012-07-08

\fp_to_dim:N \langle fp \ var \rangle
\fp_to_dim:c \fp_to_dim:n \⋆

Updated: 2016-03-22

\fp_to_int:N \langle fp \ var \rangle
\fp_to_int:c \fp_to_int:n \⋆

Updated: 2012-07-08

\fp_to_scientific:N \langle fp \ var \rangle
\fp_to_scientific:c \fp_to_scientific:n \⋆

Updated: 2016-03-22

\fp_to_scientific:n \{\langle floating point expression \rangle \}

Evaluates the \langle floating point expression \rangle and expresses the result in scientific notation: \langle optional \ -\rangle \langle digit \rangle . \langle 15 \ digits \rangle \langle optional \ sign \rangle \langle exponent \rangle

The leading \langle digit \rangle is non-zero except in the case of \pm 0. The values \pm \infty and \text{NaN} trigger an “invalid operation” exception. Normal category codes apply: thus the e is category code 11 (a letter). For a tuple, each item is converted using \fp_to_scientific:n and they are combined as \langle fp_1 \rangle, \langle fp_2 \rangle, \ldots \langle fp_n \rangle if n > 1 and \langle fp_1 \rangle, or () for fewer items.
4 Floating point conditionals

\fp_compare:nNnTF \langle \textrm{expression} \rangle \{ \langle \textrm{true code} \rangle \} \{ \langle \textrm{false code} \rangle \}

Tests whether the \langle \textit{fp var} \rangle is currently defined. This does not check that the \langle \textit{fp var} \rangle really is a floating point variable.

\textbf{Tuples are equal if they have the same number of items and items compare equal (in particular there must be no } \texttt{NaN}). At present any other comparison with tuples yields \texttt{?} (not ordered). This is experimental.

This function is less flexible than \texttt{\fp_compare:nTF} but slightly faster. It is provided for consistency with \texttt{\int_compare:nNnTF} and \texttt{\dim_compare:nNnTF}.
\[\text{\texttt{fp\_compare\_p\_n}} \star \text{\texttt{fp\_compare\_nTF}} \star\]

Updated: 2013-12-14

\[\fp\text{\_compare\_p\_n}\]
\begin{verbatim}
{ 
  \texttt{fpexpr}_1 \texttt{\{relation\}_1} \\
  \ldots \\
  \texttt{fpexpr}_N \texttt{\{relation\}_N} \\
  \texttt{fpexpr}_{N+1}
}
\end{verbatim}

\[\fp\text{\_compare\_nTF}\]
\begin{verbatim}
{ 
  \texttt{fpexpr}_1 \texttt{\{relation\}_1} \\
  \ldots \\
  \texttt{fpexpr}_N \texttt{\{relation\}_N} \\
  \texttt{fpexpr}_{N+1} \\
}
\end{verbatim}

\begin{verbatim}
\{\{true code\}\} \{\{false code\}\}
\end{verbatim}

Evaluates the \textit{floating point expressions} as described for \texttt{fp\_eval\_n} and compares consecutive result using the corresponding \texttt{\{relation\}}, namely it compares \texttt{\{intexpr\}_1} and \texttt{\{intexpr\}_2} using the \{\texttt{relation\}_1\}, then \texttt{\{intexpr\}_2} and \texttt{\{intexpr\}_3} using the \{\texttt{relation\}_2\}, until finally comparing \texttt{\{intexpr\}_N} and \texttt{\{intexpr\}_{N+1}} using the \{\texttt{relation\}_N\}. The test yields \texttt{true} if all comparisons are \texttt{true}. Each \textit{floating point expression} is evaluated only once.

Contrarily to \texttt{\texttt{int\_compare\_nTF}}, all \textit{floating point expressions} are computed, even if one comparison is \texttt{false}. Two floating points \(x\) and \(y\) may obey four mutually exclusive relations: \(x \lt y\), \(x = y\), \(x \gt y\), or \(x \neq y\) (“not ordered”). The last case occurs exactly if one or both operands is \texttt{NaN} or is a tuple, unless they are equal tuples. Each \texttt{\{relation\}} can be any (non-empty) combination of \(<\), \(=\), \(>\), and \(?\), plus an optional leading \(!\) (which negates the \texttt{\{relation\}}), with the restriction that the \texttt{\{relation\}} may not start with \(?\), as this symbol has a different meaning (in combination with :\) within floating point expressions. The comparison \(x \texttt{\{relation\}} y\) is then \texttt{true} if the \texttt{\{relation\}} does not start with \(!\) and the actual relation (\(<\), \(=\), \(>\), or \(?\)) between \(x\) and \(y\) appears within the \texttt{\{relation\}}, or on the contrary if the \texttt{\{relation\}} starts with \(!\) and the relation between \(x\) and \(y\) does not appear within the \texttt{\{relation\}}. Common choices of \texttt{\{relation\}} include >= (greater or equal), != (not equal), !? or <= (comparable).

This function is more flexible than \texttt{fp\_compare\_nNnTF} and only slightly slower.

### 5 Floating point expression loops

\[\text{\texttt{fp\_do\_until\_nNn}} \star\]

Places the \texttt{\{code\}} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{floating point expressions} as described for \texttt{fp\_compare\_nNnTF}. If the test is \texttt{false} then the \texttt{\{code\}} is inserted into the input stream again and a loop occurs until the \texttt{\{relation\}} is \texttt{true}.

\[\text{\texttt{fp\_do\_while\_nNn}} \star\]

Places the \texttt{\{code\}} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{floating point expressions} as described for \texttt{fp\_compare\_nNnTF}. If the test is \texttt{true} then the \texttt{\{code\}} is inserted into the input stream again and a loop occurs until the \texttt{\{relation\}} is \texttt{false}.
\texttt{\textbackslash fp\_until\_do:nNnn} \quad \texttt{\textbackslash fp\_until\_do:nNnn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash fp\_while\_do:nNnn} \quad \texttt{\textbackslash fp\_while\_do:nNnn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\texttt{\textbackslash fp\_do\_until:nn} \quad \texttt{\textbackslash fp\_do\_until:nn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \texttt{false} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{true}.

\texttt{\textbackslash fp\_do\_while:nn} \quad \texttt{\textbackslash fp\_do\_while:nn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \texttt{true} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{false}.

\texttt{\textbackslash fp\_until\_do:nn} \quad \texttt{\textbackslash fp\_until\_do:nn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash fp\_while\_do:nn} \quad \texttt{\textbackslash fp\_while\_do:nn \{fpexpr\}_1 \{relation\} \{fpexpr\}_2 \{\texttt{(code)}\}}

Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.
This function first evaluates the \textit{initial value}, \textit{step} and \textit{final value}, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The \textit{function} is then placed in front of each \textit{value} from the \textit{initial value} to the \textit{final value} in turn (using \textit{step} between each \textit{value}). \textit{Step} must be non-zero. If the \textit{step} is positive, the loop stops when the \textit{value} becomes larger than the \textit{final value}. If the \textit{step} is negative, the loop stops when the \textit{value} becomes smaller than the \textit{final value}. The \textit{function} should absorb one numerical argument. For example

\begin{verbatim}
\cs_set:Npn \my_func:n #1 { \[I saw #1\] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
\end{verbatim}

would print

\begin{verbatim}
[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]
\end{verbatim}

\textbf{\TeX}hacker's note: Due to rounding, it may happen that adding the \textit{step} to the \textit{value} does not change the \textit{value}; such cases give an error, as they would otherwise lead to an infinite loop.

This function first evaluates the \textit{initial value}, \textit{step} and \textit{final value}, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each \textit{value} from the \textit{initial value} to the \textit{final value} in turn (using \textit{step} between each \textit{value}), the \textit{code} is inserted into the input stream with \texttt{#1} replaced by the current \textit{value}. Thus the \textit{code} should define a function of one argument (\texttt{#1}).

This function first evaluates the \textit{initial value}, \textit{step} and \textit{final value}, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each \textit{value} from the \textit{initial value} to the \textit{final value} in turn (using \textit{step} between each \textit{value}), the \textit{code} is inserted into the input stream, with the \texttt{tl var} defined as the current \textit{value}. Thus the \textit{code} should make use of the \texttt{tl var}.

6 Some useful constants, and scratch variables

Zero, with either sign.

One as an \texttt{fp}: useful for comparisons in some places.
Infinity, with either sign. These can be input directly in a floating point expression as \texttt{inf} and \texttt{-inf}.

The value of the base of the natural logarithm, \texttt{e} = \exp(1).

The value of \pi. This can be input directly in a floating point expression as \texttt{pi}.

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 \texttt{deg}.

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.

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.

\section{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 relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- \textbf{Overflow} occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in ±\infty.
- \textbf{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.
- \textbf{Invalid operation} occurs for operations with no defined outcome, for instance 0/0 or \sin(\infty), and results in a NaN. It also occurs for conversion functions whose target type does not have the appropriate infinite or NaN value (\texttt{e.g.}, \texttt{fp_to_dim:n}).
- \textbf{Division by zero} occurs when dividing a non-zero number by 0, or when evaluating functions at poles, \texttt{e.g.}, \ln(0) or \cot(0). This results in ±\infty.
(not yet) 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 \LaTeX{3}.

To each exception we associate a “flag”: \texttt{fp\_overflow}, \texttt{fp\_underflow}, \texttt{fp\_invalid\_operation} and \texttt{fp\_division\_by\_zero}. The state of these flags can be tested and modified with commands from \texttt{l3flag}.

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using \texttt{\fp\_trap:nn}) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

\begin{verbatim}
\fp\_trap:nn {⟨exception⟩} {⟨trap type⟩}
\end{verbatim}

All occurrences of the \texttt{⟨exception⟩} (\texttt{overflow}, \texttt{underflow}, \texttt{invalid\_operation} or \texttt{division\_by\_zero}) within the current group are treated as \texttt{⟨trap type⟩}, which can be

\begin{itemize}
  \item none: the \texttt{⟨exception⟩} will be entirely ignored, and leave no trace;
  \item flag: the \texttt{⟨exception⟩} will turn the corresponding flag on when it occurs;
  \item error: additionally, the \texttt{⟨exception⟩} will halt the \LaTeX{} run and display some information about the current operation in the terminal.
\end{itemize}

\textit{This function is experimental, and may be altered or removed.}

Flags denoting the occurrence of various floating-point exceptions.

8 Viewing floating points

\begin{verbatim}
\fp\_show:N ⟨fp var⟩
\fp\_show:c ⟨floating point expression⟩
\fp\_show:n
\end{verbatim}

Evaluates the \texttt{⟨floating point expression⟩} and displays the result in the terminal.

\begin{verbatim}
\fp\_log:N ⟨fp var⟩
\fp\_log:c ⟨floating point expression⟩
\fp\_log:n
\end{verbatim}

Evaluates the \texttt{⟨floating point expression⟩} and writes the result in the log file.
9  Floating point expressions

9.1  Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- $\pm 0$, 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);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle \text{sign} \rangle$: a possibly empty string of + and – characters;
- $\langle \text{significand} \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle \text{exponent} \rangle$: 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 $\langle \text{sign} \rangle$ contains an even number of –, and – otherwise, hence, an empty $\langle \text{sign} \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle \text{significand} \rangle$ 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 $\langle \text{significand} \rangle$ has at most 16 digits. The stored $\langle \text{exponent} \rangle$ is obtained by combining the input $\langle \text{exponent} \rangle$ (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 $\langle \text{exponent} \rangle$ 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 $\pm 0$).

The result is thus $\pm 0$ if and only if $\langle \text{significand} \rangle$ contains no non-zero digit (i.e., consists only in characters 0, and an optional period), 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.

The $\langle \text{significand} \rangle$ must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as e and should be input as $\exp(1)$ or $\texttt{\textbackslash c\_e\_fp}$.

Special numbers are input as follows:

- inf represents $+\infty$, and can be preceded by any $\langle \text{sign} \rangle$, yielding $\pm \infty$ as appropriate.
- nan represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a NaN.
- Note that commands such as $\texttt{\textbackslash infty}$, $\texttt{\textbackslash pi}$, or $\texttt{\textbackslash sin}$ do not work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.
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 +, −, !.
- Implicit multiplication by juxtaposition (2pi) when neither factor is in parentheses.
- Binary * and /, implicit multiplication by juxtaposition with parentheses (for instance 3(4+5)).
- Binary + and −.
- Comparisons >=, !=, <?, etc.
- Logical and, denoted by &&.
- Logical or, denoted by ||.
- Ternary operator ?,: (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$1/2\pi = 1/(2\pi),$$
$$1/2\pi(\pi + \pi) = (2\pi)^{-1}(\pi + \pi) \simeq 1,$$
$$\sin2\pi = \sin(2\pi) \neq 0,$$
$$2^{-2\max(3,5)} = 2^2 \max(3,5) = 20,$$
$$1\text{in}/1\text{cm} = (1\text{in})/(1\text{cm}) = 2.54.$$  

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 or a tuple such as (0, 0). Tuples are only supported to some extent by operations that work with truth values (?, |, &&, !), by comparisons (!<=?), and by +, −, *, /. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a NaN result.
The ternary operator `?:` results in \( \langle \text{operand} 2 \rangle \) if \( \langle \text{operand} 1 \rangle \) is true (not \( \pm 0 \)), and \( \langle \text{operand} 3 \rangle \) if \( \langle \text{operand} 1 \rangle \) is false (\( \pm 0 \)). All three \( \langle \text{operands} \rangle \) are evaluated in all cases; they may be tuples. The operator is right associative, hence

\[
\begin{align*}
\text{\texttt{fp\_eval:n}} & \{ \langle \text{operand}_1 \rangle \ ? \langle \text{operand}_2 \rangle : \langle \text{operand}_3 \rangle \} \\
\end{align*}
\]

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.

If \( \langle \text{operand}_1 \rangle \) is true (not \( \pm 0 \)), use that value, otherwise the value of \( \langle \text{operand}_2 \rangle \). Both \( \langle \text{operands} \rangle \) are evaluated in all cases; they may be tuples. In \( \langle \text{operand}_1 \rangle \ || \langle \text{operand}_2 \rangle \ || \ldots \ || \langle \text{operands}_n \rangle \), the first true (nonzero) \( \langle \text{operand} \rangle \) is used and if all are zero the last one (\( \pm 0 \)) is used.

If \( \langle \text{operand}_1 \rangle \) is false (equal to \( \pm 0 \)), use that value, otherwise the value of \( \langle \text{operand}_2 \rangle \). Both \( \langle \text{operands} \rangle \) are evaluated in all cases; they may be tuples. In \( \langle \text{operand}_1 \rangle \ && \langle \text{operand}_2 \rangle \ && \ldots \ && \langle \text{operands}_n \rangle \), the first false (\( \pm 0 \)) \( \langle \text{operand} \rangle \) is used and if none is zero the last one is used.

Each \( \langle \text{relation} \rangle \) consists of a non-empty string of \(<\), \(=\), \(>\), and \(?\), optionally preceded by \(!\), and may not start with \(?\). This evaluates to \(+1\) if all comparisons \( \langle \text{operand}_i \rangle \langle \text{relation}_i \rangle \langle \text{operand}_{i+1} \rangle \) are true, and \(+0\) otherwise. All \( \langle \text{operands} \rangle \) are evaluated (once) in all cases. See \texttt{fp\_compare:nTF} for details.

Computes the sum or the difference of its two \( \langle \text{operands} \rangle \). The “invalid operation” exception occurs for \( \infty - \infty \). “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is \( \text{NaN} \).
* \fp_eval:n \{ ⟨operand1⟩ * ⟨operand2⟩ \}
/ \fp_eval:n \{ ⟨operand1⟩ / ⟨operand2⟩ \}

Computes the product or the ratio of its two (operands). The “invalid operation” exception occurs for \(∞/∞\), 0/0, or 0 * ∞. “Division by zero” occurs when dividing a finite non-zero number by ±0. “Underflow” and “overflow” occur when appropriate. When ⟨operand1⟩ is a tuple and ⟨operand2⟩ is a floating point number, each item of ⟨operand1⟩ is multiplied or divided by ⟨operand2⟩. Multiplication also supports the case where ⟨operand1⟩ is a floating point number and ⟨operand2⟩ a tuple. Other combinations yield an “invalid operation” exception and a NaN result.

\(+\) \fp_eval:n \{ + ⟨operand⟩ \}
\(-\) \fp_eval:n \{ − ⟨operand⟩ \}
\(!\) \fp_eval:n \{ ! ⟨operand⟩ \}

The unary + does nothing, the unary − changes the sign of the ⟨operand⟩ (for a tuple, of all its components), and ! ⟨operand⟩ evaluates to 1 if ⟨operand⟩ is false (is ±0) and 0 otherwise (this is the not boolean function). Those operations never raise exceptions.

\(**\) \fp_eval:n \{ ⟨operand1⟩ ** ⟨operand2⟩ \}
\(^\sim\) \fp_eval:n \{ ⟨operand1⟩ ^⟨operand2⟩ \}

Raises ⟨operand1⟩ to the power ⟨operand2⟩. This operation is right associative, hence 2 ** 2 ** 3 equals 2\(^2^3\) = 256. If ⟨operand1⟩ is negative or −0 then: the result’s sign is + if the ⟨operand2⟩ is infinite and (−1)\(^p/\(^q\) with p, q integers; the result is +0 if abs((⟨operand1⟩)**⟨operand2⟩) evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ±0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

\(\abs\) \fp_eval:n \{ abs( ⟨fpexpr⟩ ) \}

Computes the absolute value of the ⟨fpexpr⟩. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also \fp_abs:n.

\(\exp\) \fp_eval:n \{ exp( ⟨fpexpr⟩ ) \}

Computes the exponential of the ⟨fpexpr⟩. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\(\text{fact}\) \fp_eval:n \{ fact( ⟨fpexpr⟩ ) \}

Computes the factorial of the ⟨fpexpr⟩. If the ⟨fpexpr⟩ is an integer between −0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give +∞ with “overflow”, while fact(+∞) = +∞ and fact(nan) = nan with no exception. All other inputs give NaN with the “invalid operation” exception.

\(\ln\) \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) = −∞. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
\texttt{\texttt{logb}} \hspace{1em} * \hspace{1em} \texttt{\texttt{\texttt{fp_eval:n \{ logb( \textbackslash{fpexpr} ) \}}}}

Determines the exponent of the \textbackslash{fpexpr}, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating \texttt{logb(\pm0) = -\infty}. Other special values are \texttt{logb(\pm\infty) = +\infty} and \texttt{logb(NaN) = NaN}. If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is NaN.

\texttt{\texttt{max}} \hspace{1em} \texttt{\texttt{\texttt{fp_eval:n \{ max( \textbackslash{fpexpr}1, \textbackslash{fpexpr}2, \ldots ) \}}}}

\texttt{\texttt{min}} \hspace{1em} \texttt{\texttt{\texttt{fp_eval:n \{ min( \textbackslash{fpexpr}1, \textbackslash{fpexpr}2, \ldots ) \}}}}

Evaluates each \textbackslash{fpexpr} and computes the largest (smallest) of those. If any of the \textbackslash{fpexpr} is a NaN or tuple, the result is NaN. If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

\texttt{\texttt{\texttt{\texttt{round}}}} \hspace{1em} \texttt{\texttt{\texttt{fp_eval:n \{ round( \textbackslash{fpexpr}1, \textbackslash{fpexpr}2, \textbackslash{fpexpr}3 ) \}}}}

Only \texttt{round} accepts a third argument. Evaluates \texttt{(fpexpr1) = x and (fpexpr2) = n and (fpexpr3) = t} then rounds \texttt{x} to \texttt{n} places. If \texttt{n} is an integer, this rounds \texttt{x} to a multiple of \texttt{10^{-n}}; if \texttt{n} = +\infty, this always yields \texttt{x}; if \texttt{n} = -\infty, this yields one of \texttt{\pm0}, \texttt{\pm\infty}, or NaN; if \texttt{n} = NaN, this yields NaN; if \texttt{n} is neither \texttt{\pm\infty} nor an integer, then an “invalid operation” exception is raised. When \texttt{(fpexpr3)} is omitted, \texttt{n = 0}, i.e., \texttt{(fpexpr1)} is rounded to an integer. The rounding direction depends on the function.

- \texttt{round} yields the multiple of \texttt{10^{-n}} closest to \texttt{x}, with ties \texttt{x half-way between two such multiples} rounded as follows. If \texttt{t} is \texttt{nan} (or not given) the even multiple is chosen (“ties to even”), if \texttt{t} = \texttt{\pm0} the multiple closest to \texttt{0} is chosen (“ties to zero”), if \texttt{t} is positive/negative the multiple closest to \texttt{\infty/\hfill -\infty} is chosen (“ties towards positive/negative infinity”).
- \texttt{floor} yields the largest multiple of \texttt{10^{-n}} smaller or equal to \texttt{x} (“round towards negative infinity”);
- \texttt{ceil} yields the smallest multiple of \texttt{10^{-n}} greater or equal to \texttt{x} (“round towards positive infinity”);
- \texttt{trunc} yields a multiple of \texttt{10^{-n}} with the same sign as \texttt{x} and with the largest absolute value less that that of \texttt{x} (“round towards zero”).

“Overflow” occurs if \texttt{x} is finite and the result is infinite (this can only happen if \texttt{(fpexpr3) < -9984}). If any operand is a tuple, “invalid operation” occurs.

\texttt{\texttt{\texttt{sign}} \hspace{1em} \texttt{\texttt{\texttt{fp_eval:n \{ sign( \textbackslash{fpexpr} ) \}}}}}

Evaluates the \textbackslash{fpexpr} and determines its sign: +1 for positive numbers and for \texttt{+\infty}, \texttt{-1} for negative numbers and for \texttt{-\infty}, \texttt{\pm0} for \texttt{\pm0}, and NaN for NaN. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

212
\fp_eval:n { \sin \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \cos \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \tan \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \cot \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \csc \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \sec \langle \mathit{fpexpr} \rangle } 

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \langle \mathit{fpexpr} \rangle given in radians. For arguments given in degrees, see \texttt{sind}, \texttt{cosd}, etc. Note that since \pi is irrational, \sin(8\pi) is not quite zero, while its analogue \texttt{sind}(8 \times 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. If the operand is a tuple, “invalid operation” occurs.

\fp_eval:n { \sind \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \cosd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \tand \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \cotd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \cscd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \secd \langle \mathit{fpexpr} \rangle } 

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \langle \mathit{fpexpr} \rangle given in degrees. For arguments given in radians, see \texttt{sin}, \texttt{cos}, etc. Note that since \pi is irrational, \sin(8\pi) is not quite zero, while its analogue \texttt{sind}(8 \times 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. If the operand is a tuple, “invalid operation” occurs.

\fp_eval:n { \asind \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \acosd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \acscd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \asecd \langle \mathit{fpexpr} \rangle } 

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arccosecant of the \langle \mathit{fpexpr} \rangle and returns the result in radians, in the range \([-\pi/2,\pi/2]\) for \texttt{asin} and \texttt{acsc} and \([0,\pi]\) for \texttt{acos} and \texttt{asec}. For a result in degrees, use \texttt{asind}, etc. If the argument of \texttt{asin} or \texttt{acos} lies outside the range \([-1,1]\), or the argument of \texttt{acsc} or \texttt{asec} inside the range \((-1,1)\), an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\fp_eval:n { \asind \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \acosd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \acscd \langle \mathit{fpexpr} \rangle } \\
\fp_eval:n { \asecd \langle \mathit{fpexpr} \rangle } 

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arccosecant of the \langle \mathit{fpexpr} \rangle and returns the result in degrees, in the range \([-90,90]\) for \texttt{asin} and \texttt{acsc} and \([0,180]\) for \texttt{acos} and \texttt{asec}. For a result in radians, use \texttt{asin}, etc. If the argument of \texttt{asin} or \texttt{acos} lies outside the range \([-1,1]\), or the argument of \texttt{acsc} or \texttt{asec} inside the range \((-1,1)\), an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

213
\texttt{atan} \begin{verbatim}
\fp_eval:n { atan( \fpeval ) }
\end{verbatim}
\texttt{acot} \begin{verbatim}
\fp_eval:n { acot( \fpeval ) }
\end{verbatim}
\begin{verbatim}
\fp_eval:n { acot( \fpeval , \fpeval ) }
\end{verbatim}
When both expressions yield \(+\infty\), the resulting angle is one of \(\{\pm\pi/4, \pm3\pi/4\}\) depending on signs.
\texttt{atand} \begin{verbatim}
\fp_eval:n { atand( \fpeval ) }
\end{verbatim}
\texttt{acotd} \begin{verbatim}
\fp_eval:n { acotd( \fpeval ) }
\end{verbatim}
\begin{verbatim}
\fp_eval:n { acotd( \fpeval , \fpeval ) }
\end{verbatim}
Those functions yield an angle in degrees: \texttt{atand} and \texttt{acotd} are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the \(\langle \fpeval \rangle\): 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 \((\fpeval_2, \fpeval_1)\): this is the arctangent of \(\fpeval_1/\fpeval_2\), possibly shifted by \(\pi\) depending on the signs of \(\fpeval_1\) and \(\fpeval_2\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\fpeval_1, \fpeval_2)\), equal to the arccotangent of \(\fpeval_1/\fpeval_2\), possibly shifted by \(\pi\). Both two-argument functions take values in the wider range \([-\pi, \pi]\). The ratio \(\langle \fpeval_1 \rangle/\langle \fpeval_2 \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm0\), or when both yield \(\pm\infty\), the resulting angle is one of \(\{\pm\pi/4, \pm3\pi/4\}\) depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.
\texttt{sqrt} \begin{verbatim}
\fp_eval:n { sqrt( \fpeval ) }
\end{verbatim}
\begin{verbatim}
\end{verbatim}
Computes the square root of the \(\langle \fpeval \rangle\). The “invalid operation” is raised when the \(\langle \fpeval \rangle\) is negative or is a tuple; no other exception can occur. Special values yield \(\sqrt{-0} = -0\), \(\sqrt{+0} = +0\), \(\sqrt{\pm\infty} = +\infty\) and \(\sqrt{NaN} = NaN\).
\texttt{rand} \hspace{1cm} \texttt{\textbackslash fp\_eval:n \{ \texttt{rand()} \}}

\texttt{New: 2016-12-05}

Produces a pseudo-random floating-point number (multiple of $10^{-16}$) between 0 included and 1 excluded. This is not available in older versions of \TeX{}. The random seed can be queried using \texttt{\textbackslash sys\_rand\_seed:} and set using \texttt{\textbackslash sys\_gset\_rand\_seed:n}.

\textbf{\TeX{}hackers note:} This is based on pseudo-random numbers provided by the engine's primitive \texttt{\textbackslash pdfuniformdeviate} in pdf\TeX{}, \texttt{\textbackslash uniformdeviate} in up\TeX{} and \texttt{\textbackslash uniformdeviate} in Lua\TeX{} and X\TeX{}. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than \texttt{\textbackslash uniformdeviate} to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

\texttt{randint} \hspace{1cm} \texttt{\textbackslash fp\_eval:n \{ \texttt{randint(\langle fpexpr\rangle )} \}}

\texttt{\textbackslash fp\_eval:n \{ \texttt{randint(\langle fpexpr\rangle 1, \langle fpexpr\rangle 2)} \}}

\texttt{New: 2016-12-05}

Produces a pseudo-random integer between 1 and \langle fpexpr \rangle or between \langle fpexpr \rangle 1 and \langle fpexpr \rangle 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. See \texttt{rand} for important comments on how these pseudo-random numbers are generated.

\underline{inf} \hspace{1cm} The special values $+\infty$, $-\infty$, and NaN are represented as inf, -inf and nan (see \texttt{\textbackslash c\_inf\_fp}, \texttt{\textbackslash c\_minus\_inf\_fp} and \texttt{\textbackslash c\_nan\_fp}).

\underline{nan} \hspace{1cm}

\underline{pi} \hspace{1cm} The value of $\pi$ (see \texttt{\textbackslash c\_pi\_fp}).

\underline{deg} \hspace{1cm} The value of $1^\circ$ in radians (see \texttt{\textbackslash c\_one\_degree\_fp}).
Those units of measurement are equal to their values in pt, namely

\[
\begin{align*}
1 \text{ in} &= 72.27 \text{ pt} \\
1 \text{ pt} &= 1 \text{ pt} \\
1 \text{ pc} &= 12 \text{ pt} \\
1 \text{ cm} &= \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt} \\
1 \text{ mm} &= \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt} \\
1 \text{ dd} &= 0.376065 \text{ mm} = 1.07000856496063 \text{ pt} \\
1 \text{ cc} &= 12 \text{ dd} = 12.84010277952756 \text{ pt} \\
1 \text{ nd} &= 0.375 \text{ mm} = 1.066978346456693 \text{ pt} \\
1 \text{ nc} &= 12 \text{ nd} = 12.80374015748031 \text{ pt} \\
1 \text{ bp} &= \frac{1}{72} \text{ in} = 1.00375 \text{ pt} \\
1 \text{ sp} &= 2^{-16} \text{ pt} = 1.52587890625e-5 \text{ pt}.
\end{align*}
\]

The values of the (font-dependent) units \texttt{em} and \texttt{ex} are gathered from \TeX{} when the surrounding floating point expression is evaluated.

Other names for 1 and +0.

\begin{align*}
\texttt{\textbackslash fp_abs:n} & \quad \{(\text{floating point expression})\} \\
\texttt{\textbackslash fp_abs:n} & \quad \text{New: 2012-05-14} \\
\texttt{\textbackslash fp_abs:n} & \quad \text{Updated: 2012-07-08} \\
\texttt{\textbackslash fp_max:nn} & \quad \{(\text{fp expression 1})\} \{(\text{fp expression 2})\} \\
\texttt{\textbackslash fp_min:nn} & \quad \text{New: 2012-09-26}
\end{align*}

\texttt{\textbackslash fp_abs:n} evaluates the \{(\text{floating point expression})\} as described for \texttt{\textbackslash fp_eval:n} and leaves the absolute value of the result in the input stream. If the argument is ±\infty, NaN or a tuple, “invalid operation” occurs. Within floating point expressions, \texttt{abs()} can be used; it accepts ±\infty and NaN as arguments.

\texttt{\textbackslash fp_max:nn} \texttt{\textbackslash fp_min:nn} evaluates the \{(\text{floating point expressions})\} as described for \texttt{\textbackslash fp_eval:n} and leaves the resulting larger (\texttt{max}) or smaller (\texttt{min}) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, \texttt{max()} and \texttt{min()} can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789+, or if it receives a \TeX{} primitive conditional affected by \texttt{\exp_not:N}.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.
• Support signalling \texttt{nan}.

• Modulo and remainder, and rounding function \texttt{quantize} (and its friends analogous to \texttt{trunc}, \texttt{ceil}, \texttt{floor}).

• \texttt{\fp_format:nn}{\langle\texttt{fpexpr}\rangle}{\langle\texttt{format}\rangle}, but what should \langle\texttt{format}\rangle be? More general pretty printing?

• Add \texttt{and}, \texttt{or}, \texttt{xor}? Perhaps under the names \texttt{all}, \texttt{any}, and \texttt{xor}?

• Add \texttt{log}(x,b) for logarithm of $x$ in base $b$.

• \texttt{hypot} (Euclidean length). Cartesian-to-polar transform.

• Hyperbolic functions \texttt{cosh}, \texttt{sinh}, \texttt{tanh}.

• Inverse hyperbolics.

• Base conversion, input such as \texttt{0xAB.CDEF}.

• Factorial (not with \texttt{!}), gamma function.

• Improve coefficients of the \texttt{sin} and \texttt{tan} series.

• Treat upper and lower case letters identically in identifiers, and ignore underscores.

• Add an \texttt{array}(1,2,3) and \texttt{i=complex}(0,1).

• Provide an experimental \texttt{map} function? Perhaps easier to implement if it is a single character, \texttt{@sin}(1,2)?

• Provide an \texttt{isnan} function analogue of \texttt{\fp_if_nan:nTF}?

• Support keyword arguments?

\texttt{Pgfmath} also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

• Check that functions are monotonic when they should.

• Add exceptions to ?:, !<=>?, &&, ||, and !.

• Logarithms of numbers very close to 1 are inaccurate.

• When rounding towards $-\infty$, \texttt{\dim_to_fp:n}{\langle0pt\rangle} should return $-0$, not $+0$.

• The result of $(\pm0) + (\pm0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.

• \texttt{0e9999999999} gives a \texttt{TpX} “number too large” error.

• Subnormals are not implemented.

Possible optimizations/improvements.

• Document that \texttt{l3trial/l3fp-types} introduces tools for adding new types.

• In subsection 9.1, write a grammar.
• 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 \( c = \frac{2000}{([200x]+1)} \in [10, 95] \) 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 \texttt{__fp_basics_pack_weird_low:NNNW} and \texttt{__fp_basics_pack_weird_high:NNNNNNNW} better. Move the other \texttt{basics_pack} auxiliaries to \texttt{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., \texttt{@/} or whatever)?
Part XXIV

The l3fparray package: fast global floating point arrays

1 l3fparray documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). The interface is very close to that of l3intarray. The size of the array is fixed and must be given at point of initialisation.

\fparray_new:Nn \fparray_new:Nn \langle \text{fparray var} \rangle \{\langle \text{size} \rangle\}

Evaluates the integer expression \langle \text{size} \rangle and allocates an \langle \text{floating point array variable} \rangle with that number of (zero) entries. The variable name should start with \texttt{\_g} because assignments are always global.

\fparray_count:N \fparray_count:N \langle \text{fparray var} \rangle

Expands to the number of entries in the \langle \text{floating point array variable} \rangle. This is performed in constant time.

\fparray_gset:Nnn \fparray_gset:Nnn \langle \text{fparray var} \rangle \{\langle \text{position} \rangle\} \{\langle \text{value} \rangle\}

Stores the result of evaluating the floating point expression \langle \text{value} \rangle into the \langle \text{floating point array variable} \rangle at the (integer expression) \langle \text{position} \rangle. If the \langle \text{position} \rangle is not between 1 and the \fparray_count:N, an error occurs. Assignments are always global.

\fparray_gzero:N \fparray_gzero:N \langle \text{fparray var} \rangle

Sets all entries of the \langle \text{floating point array variable} \rangle to +0. Assignments are always global.

\fparray_item:Nn \fparray_item:Nn \langle \text{fparray var} \rangle \{\langle \text{position} \rangle\}

\fparray_item_to_tl:Nn \fparray_item_to_tl:Nn \langle \text{fparray var} \rangle \{\langle \text{position} \rangle\}

Applies \texttt{\_fp_use:N} or \texttt{\_fp_to_tl:N} (respectively) to the floating point entry stored at the (integer expression) \langle \text{position} \rangle in the \langle \text{floating point array variable} \rangle. If the \langle \text{position} \rangle is not between 1 and the \fparray_count:N, an error occurs.
The \texttt{l3sort} package

Sorting functions

1 Controlling sorting

\LaTeX{} comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

\begin{verbatim}
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{\int_compare:nNnTF { #1 } > { #2 }
{" \sort_return_swapped: }
{ \sort_return_same: }
}
\end{verbatim}

results in $\l_foo_clist$ holding the values $\{-2, 01, +1, 3, 5\}$ sorted in non-decreasing order.

The code defining the comparison should call $\texttt{\sort_return_swapped:}$ if the two items given as $\texttt{#1}$ and $\texttt{#2}$ are not in the correct order, and otherwise it should call $\texttt{\sort_return_same:}$ to indicate that the order of this pair of items should not be changed.

For instance, a $\langle$\texttt{comparison code}$\rangle$ consisting only of $\texttt{\sort_return_same:}$ with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a $\langle$\texttt{comparison code}$\rangle$ consisting only of $\texttt{\sort_return_swapped:}$ reverses the list (in a fairly inefficient way).

\TeX{}hackers note: The current implementation is limited to sorting approximately 20000 items (40000 in \texttt{luatex}), depending on what other packages are loaded.

Internally, the code from \texttt{l3sort} stores items in $\texttt{\toks}$ registers allocated locally. Thus, the $\langle$\texttt{comparison code}$\rangle$ should not call $\texttt{\newtoks}$ or other commands that allocate new $\texttt{\toks}$ registers. On the other hand, altering the value of a previously allocated $\texttt{\toks}$ register is not a problem.

\begin{verbatim}
\sort_return_same: \sort_return_swapped:
\end{verbatim}

\texttt{$\langle$ comparison code $\rangle$} $\langle$ \texttt{seq var}$\rangle$

\begin{verbatim}
... \sort_return_same: or \sort_return_swapped: ...
\end{verbatim}

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the $\texttt{\sort_return_...}$ functions should be used by the code, according to the results of some tests on the items $\texttt{#1}$ and $\texttt{#2}$ to be compared.
Part XXVI

The \texttt{l3tl-analysis} package: Analysing token lists

\section{\texttt{l3tl-analysis} documentation}

This module mostly provides internal functions for use in the \texttt{l3regex} module. However, it provides as a side-effect a user debugging function, very similar to the \texttt{\ShowTokens} macro from the \texttt{ted} package.

\texttt{\tl_analysis_show:n \langle token list \rangle}

Displays to the terminal the detailed decomposition of the \texttt{\langle token list \rangle} into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

\texttt{\tl_analysis_map_inline:nn \langle token list \rangle \{ \langle inline function \rangle \}}

Applies the \texttt{\langle inline function \rangle} to each individual \texttt{\langle token \rangle} in the \texttt{\langle token list \rangle}. The \texttt{\langle inline function \rangle} receives three arguments:

- \texttt{\langle tokens \rangle}, which both o-expand and x-expand to the \texttt{\langle token \rangle}. The detailed form of \texttt{\langle token \rangle} may change in later releases.
- \texttt{\langle char code \rangle}, a decimal representation of the character code of the token, $-1$ if it is a control sequence (with \texttt{\langle catcode \rangle} 0).
- \texttt{\langle catcode \rangle}, a capital hexadecimal digit which denotes the category code of the \texttt{\langle token \rangle} (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C:other, D:active).

As all other mappings the mapping is done at the current group level, \textit{i.e.} any local assignments made by the \texttt{\langle inline function \rangle} remain in effect after the loop.
The l3regex package: Regular expressions in \TeX

The l3regex package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that \TeX manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After
\begin{verbatim}
\tl_set:Nn \l_my_tl { That\textendash cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
\end{verbatim}
the token list variable \c{l}_my_tl holds the text “This cat.”, where the first occurrence of “at” was replaced by “is”. A more complicated example is a pattern to emphasize each word and add a comma after it:
\begin{verbatim}
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
\end{verbatim}
The \c{w} sequence represents any “word” character, and + indicates that the \c{w} sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, \c{0} denotes the full match (here, a word). The command \c{emph} is inserted using \c{emph}, and its argument \c{0} is put between braces \cB\{ and \cE\}.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using \regex_const:Nn. For example,
\begin{verbatim}
\regex_const:Nn \c_foo_regex { \c{begin} \cB. \c[^BE].* \cE. }
\end{verbatim}
stores in \c_foo_regex a regular expression which matches the starting marker for an environment: \c{begin}, followed by a begin-group token (\cB.), then any number of tokens which are neither begin-group nor end-group character tokens (\c[^BE].*), ending with an end-group token (\cE.). As explained in the next section, the parentheses “capture” the result of \c[^BE].*, giving us access to the name of the environment when doing replacements.

1 Syntax of regular expressions

We start with a few examples, and encourage the reader to apply \regex_show:n to these regular expressions.

- \texttt{Cat} matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use \texttt{\bCat\b} to match a complete word only.

- \texttt{[abc]} matches one letter among “a”, “b”, “c”; the pattern \texttt{(a|b|c)} matches the same three possible letters (but see the discussion of submatches below).

- \texttt{[A-Za-z]*} matches any number (due to the quantifier \star) of Latin letters (not accented).
• \c{[A-Za-z]*} matches a control sequence made of Latin letters.

• \_\[-\_]\*\_ matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to \_\.*\?\_ where . matches arbitrary characters and the lazy quantifier *? means to match as few characters as possible, thus avoiding matching underscores.

• \[\+\-]\?\d+ matches an explicit integer with at most one sign.

• \[\+\-\_]\*\d+\_\* matches an explicit integer with any number of + and – signs, with spaces allowed except within the mantissa, and surrounded by spaces.

• \[\+\-\_]\*(\d+|\d*.\d+)\_\* matches an explicit integer or decimal number; using \[\_\] instead of \_ would allow the comma as a decimal marker.

• \[\+\-\_]\*(\d+|\d*.\d+)\_\*(\(?i)pt|in|cm|m|ex|bs|dn|pcn)c\_\* matches an explicit dimension with any unit that \TeX{} knows, where (?i) means to treat lowercase and uppercase letters identically.

• \[\+\-\_]\*((?i)nan|inf|\d+|\d*.\d+)\_\* matches an explicit floating point number or the special values nan and inf (with signs and spaces allowed).

• \[\+\-\_]\*(\d+|\cC.)\_\* matches an explicit integer or control sequence (without checking whether it is an integer variable).

• \G.*?\K at the beginning of a regular expression matches and discards (due to \G{}) everything between the end of the previous match (\G) and what is matched by the rest of the regular expression; this is useful in \regex_replace_all:nnN when the goal is to extract matches or submatches in a finer way than with \regex_extract_all:nnN.

While it is impossible for a regular expression to match only integer expressions, \[\+\-\]\*(\d+|\d*.\d+)\* matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (e.g., \* matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance \d matches any digit). As a rule,

• every alphanumeric character (A–Z, a–z, 0–9) matches exactly itself, and should not be escaped, because \A, \B, ... have special meanings;

• non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (e.g., use \l, \L, \?, \.);

• spaces should always be escaped (even in character classes);

• any other character may be escaped or not, without any effect: both versions match exactly that character.
Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into \TeX\ under normal category codes. For instance, $\backslash abc\%$ matches the characters $\abc\%$ (with arbitrary category codes), but does not match the control sequence $\abc$ followed by a percent character. Matching control sequences can be done using the $\c\{\{\text{regex}\}\}$ syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

$\\backslash\{\text{hex...}\}$ Character with hex code hh\ldots

$\\backslash \text{hh}$ Character with hex code hh.

$\backslash a$ Alarm (hex 07).

$\backslash e$ Escape (hex 1B).

$\backslash f$ Form-feed (hex 0C).

$\backslash n$ New line (hex 0A).

$\backslash r$ Carriage return (hex 0D).

$\backslash t$ Horizontal tab (hex 09).

Character types.

. A single period matches any token.

$\backslash d$ Any decimal digit.

$\backslash h$ Any horizontal space character, equivalent to [$\backslash \\text{^I}$]: space and tab.

$\backslash s$ Any space character, equivalent to [$\backslash \text{^I}\text{^J}\text{^L}\text{^M}$].

$\backslash v$ Any vertical space character, equivalent to [$\text{^K}\text{^L}\text{^M}$]. Note that $\text{^K}$ is a vertical space, but not a space, for compatibility with Perl.

$\backslash w$ Any word character, \textit{i.e.}, alphanumerics and underscore, equivalent to the explicit class $[A\text{-}Za\text{-}z0\text{-}9\_]$.

$\backslash \text{D}$ Any token not matched by $\backslash d$.

$\backslash H$ Any token not matched by $\backslash h$.

$\backslash N$ Any token other than the $\backslash n$ character (hex 0A).

$\backslash S$ Any token not matched by $\backslash s$.

$\backslash V$ Any token not matched by $\backslash v$.

$\backslash W$ Any token not matched by $\backslash w$.

Of those, $., \backslash D, \backslash H, \backslash M, \backslash S, \backslash W$, and $\backslash W$ match arbitrary control sequences.

Character classes match exactly one token in the subject.

$[\ldots]$ Positive character class. Matches any of the specified tokens.

224
[^... ] Negative character class. Matches any token other than the specified characters.

x-y Within a character class, this denotes a range (can be used with escaped characters).

[:<name>:] Within a character class (one more set of brackets), this denotes the POSIX character class \langle name\rangle, which can be \texttt{alnum}, \texttt{alpha}, \texttt{ascii}, \texttt{blank}, \texttt{cntrl}, \texttt{digit}, \texttt{graph}, \texttt{lower}, \texttt{print}, \texttt{punct}, \texttt{space}, \texttt{upper}, \texttt{word}, or \texttt{xdigit}.

[::<name>:] Negative POSIX character class.

For instance, \[a-oq-z\cC\.] matches any lowercase latin letter except p, as well as control sequences (see below for a description of \c).

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

* 0 or more, greedy.

*? 0 or more, lazy.

+ 1 or more, greedy.

+? 1 or more, lazy.

\{n\} Exactly n.

\{n,\} n or more, greedy.

\{n,\}? n or more, lazy.

\{n,m\} At least n, no more than m, greedy.

\{n,m\}? At least n, no more than m, lazy.

Anchors and simple assertions.

\b Word boundary: either the previous token is matched by \w and the next by \W, or the opposite. For this purpose, the ends of the token list are considered as \W.

\B Not a word boundary: between two \w tokens or two \W tokens (including the boundary).

\^ or \A Start of the subject token list.

\$ or \z End of the subject token list.

\G Start of the current match. This is only different from \^ in the case of multiple matches: for instance \regex_count:nnN { \G a } { aaba } \l_tmpa_int yields 2, but replacing \G by \^ would result in \l_tmpa_int holding the value 1.

Alternation and capturing groups.

A|B|C Either one of A, B, or C.

(....) Capturing group.

(?:....) Non-capturing group.
Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

The \c escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;
- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- 0 for others; and
- A for active characters.

The \c escape sequence is used as follows.

\c\{\texttt{regex}\} A control sequence whose csname matches the \texttt{regex}, anchored at the beginning and end, so that \c\{\texttt{begin}\} matches exactly \texttt{begin}, and nothing else.

\cX Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category X (any of \texttt{CBEMTPUDSLOA}). For instance, \c\L[A-Z]\d matches uppercase letters and digits of category code letter, \cC. matches any control sequence, and \cO(abc) matches abc where each character has category other.

\c\{XYZ\} Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of \texttt{CBEMTPUDSLOA}). For instance, \c\{LSO\}(\ldots) matches two tokens of category letter, space, or other.

\c\[^\texttt{XYZ}\] Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of \texttt{CBEMTPUDSLOA}). For instance, \c\[^O\]\d matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, \c\O\d \c\L\cL[0-9][A-F] matches what \TeX{} considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, \c\L(ab\cO\*cd) matches ab\*cd where all characters are of category letter, except \* which has category other.

The \u escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters.
Namely, \{tl var name\} matches the exact contents of the token list \{tl var\}. Within a \{...\} control sequence matching, the \u escape sequence only expands its argument once, in effect performing \tl_to_str:v. Quantifiers are not supported directly: use a group.

The option (?i) makes the match case insensitive (identifying A–Z with a–z; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using (?-i). For instance, in (?i)(a(?-i)b|c)d, the letters a and d are affected by the i option. Characters within ranges and classes are affected individually: (?i)[Y–\] is equivalent to [YZ][\yz], and (?i)[^aeiou] matches any character which is not a vowel. Neither character properties, nor \c{...} nor \u{...} are affected by the i option.

In character classes, only [, ^, ], \ and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (\d, \D, etc.) is supported in character classes. If the first character is ^, then the meaning of the character class is inverted; ^ appearing anywhere else in the range is not special. If the first character (possibly following a leading ^) is ] then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using ^, for instance, \[\D 0–5] and \[^6–9] are equivalent.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance \regex_extract_once:nnNTF.

The \K escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

\regex_extract_all:nnN \{ a \K . \} \{ a123aaxyz \} \l_foo_seq

results in \l_foo_seq containing the items \{1\} and \{a\}: the true matches are \{a1\} and \{aa\}, but they are trimmed by the use of \K. The \K command does not affect capturing groups: for instance,

\regex_extract_once:mnN \{ (. \K c)+ \d \} \{ acbc3 \} \l_foo_seq

results in \l_foo_seq containing the items \{c3\} and \{bc\}: the true match is \{acbc3\}, with first submatch \{bc\}, but \K resets the beginning of the match to the last position where it appears.

2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

• \0 is the whole match;
• \1 is the submatch that was matched by the first (capturing) group (...); similarly for \2, ..., \9 and \g{number};
• \  inserts a space (spaces are ignored when not escaped);
• \a, \e, \f, \n, \r, \t, \xhh, \x{hhh} correspond to single characters as in regular expressions;
• \c{⟨cs name⟩} inserts a control sequence;
• \c⟨category⟩⟨character⟩ (see below);
• \u{⟨tl var name⟩} inserts the contents of the ⟨tl var⟩ (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for TeX, for instance use \#). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

\tl_set:Nn \l_my_tl { Hello,-world! }
\regex_replace_all:nnN { ([er]l|o) . } { (\0--\1) } \l_my_tl

results in \l_my_tl holding H(ell--el)(o,--o) w(or--o)(1d--1)!

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n-th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through \ use category code 10, while spaces inserted through \x{20} or \x{20} have category code 12. The escape sequence \c allows to insert characters with arbitrary category codes, as well as control sequences.

\cX(...) Produces the characters “…” with category X, which must be one of CBEMTPUDSLOA as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance \cL(Hello\cS\ world)! gives this text with standard category codes.

\c{⟨text⟩} Produces the control sequence with csname ⟨text⟩. The ⟨text⟩ may contain references to the submatches \0, \1, and so on, as in the example for \u below.

The escape sequence \u{⟨tl var name⟩} allows to insert the contents of the token list with name ⟨tl var name⟩ directly into the replacement, giving an easier control of category codes. When nested in \c{⟨...⟩} and \u{⟨...⟩} constructions, the \u and \c escape sequences perform \tl_to_str:v, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of \c and \u. For instance,

\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} } }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { ^[\-]+ } { \u{l_my_\0_tl} } \l_my_tl

results in \l_my_tl holding first,\emph{second},first,first.
3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the \l3regex module’s functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

\regex_new:N  ⟨regex var⟩

Creates a new ⟨regex var⟩ or raises an error if the name is already taken. The declaration is global. The ⟨regex var⟩ is initially such that it never matches.

\regex_set:Nn  ⟨regex var⟩  ⟨⟨regex⟩⟩

Stores a compiled version of the ⟨regular expression⟩ in the ⟨regex var⟩. For instance, this function can be used as

\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ular\ expression) }

The assignment is local for \regex_set:Nn and global for \regex_gset:Nn. Use \regex_const:Nn for compiled expressions which never change.

\regex_show:n  ⟨⟨regex⟩⟩

Shows how \l3regex interprets the ⟨regex⟩. For instance, \regex_show:n \{\A X|Y\} shows

+-branch
  anchor at start (\A)
  char code 88
+-branch
  char code 89

indicating that the anchor \A only applies to the first branch: the second branch is not anchored to the beginning of the match.

4 Matching

All regular expression functions are available in both :n and :N variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by \regex_(g)set:Nn.

\regex_match:nnTF  ⟨⟨regex⟩⟩  ⟨⟨token list⟩⟩  ⟨⟨true code⟩⟩  ⟨⟨false code⟩⟩

Tests whether the ⟨regular expression⟩ matches any part of the ⟨token list⟩. For instance,

\regex_match:nnTF \{ b \{cde\}* \} \{ abedcex \} \{ TRUE \} \{ FALSE \}
\regex_match:nnTF \{ \{b-dq-w\} \} \{ example \} \{ TRUE \} \{ FALSE \}

leaves TRUE then FALSE in the input stream.
Sets \( \text{int var} \) within the current \TeX{} group level equal to the number of times \( \text{regular expression} \) appears in \( \text{token list} \). The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

\begin{verbatim}
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbabcbbb } \l_foo_int
\end{verbatim}

results in \( \l_foo_int \) taking the value 5.

\section{Submatch extraction}

Finds the first match of the \( \text{regular expression} \) in the \( \text{token list} \). If it exists, the match is stored as the first item of the \( \text{seq var} \), and further items are the contents of capturing groups, in the order of their opening parenthesis. The \( \text{seq var} \) is assigned locally. If there is no match, the \( \text{seq var} \) is cleared. The testing versions insert the \( \text{true code} \) into the input stream if a match was found, and the \( \text{false code} \) otherwise. For instance, assume that you type

\begin{verbatim}
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
\end{verbatim}

Then the regular expression (anchored at the start with \( \A \) and at the end with \( \Z \)) must match the whole token list. The first capturing group, \( (La)? \), matches \( La \), and the second capturing group, \( (!*) \), matches \( !!! \). Thus, \( \l_foo_seq \) contains as a result the items \{LaTeX!!!\}, \{La\}, and \{!!!\}, and the \text{true} branch is left in the input stream. Note that the \( n \)-th item of \( \l_foo_seq \), as obtained using \( \seq_item:Nn \), correspond to the submatch numbered \( (n - 1) \) in functions such as \( \regex_replace_once:nnN \).

Finds all matches of the \( \text{regular expression} \) in the \( \text{token list} \), and stores all the sub-match information in a single sequence (concatenating the results of multiple \( \regex_extract_once:nnN \) calls). The \( \text{seq var} \) is assigned locally. If there is no match, the \( \text{seq var} \) is cleared. The testing versions insert the \( \text{true code} \) into the input stream if a match was found, and the \( \text{false code} \) otherwise. For instance, assume that you type

\begin{verbatim}
\regex_extract_all:nnNTF { \w+ } { Hello,-world! } \l_foo_seq
{ true } { false }
\end{verbatim}

Then the regular expression matches twice, the resulting sequence contains the two items \{Hello\} and \{world\}, and the \text{true} branch is left in the input stream.
Splits the \( \langle \text{token list} \rangle \) into a sequence of parts, delimited by matches of the \( \langle \text{regular expression} \rangle \). If the \( \langle \text{regular expression} \rangle \) has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to \( \langle \text{seq var} \rangle \) is local. If no match is found the resulting \( \langle \text{seq var} \rangle \) has the \( \langle \text{token list} \rangle \) as its sole item. If the \( \langle \text{regular expression} \rangle \) matches the empty token list, then the \( \langle \text{token list} \rangle \) is split into single tokens. The testing versions insert the \( \langle \text{true code} \rangle \) into the input stream if a match was found, and the \( \langle \text{false code} \rangle \) otherwise. For example, after

\[
\text{\seq_new:N} \ \l_path_seq
\text{\regex_split:nnNTF} \ { / } \ { the/path/for/this/file.tex } \ \l_path_seq
\{ \text{true} \} \ { \text{false} }
\]

the sequence \( \l_path_seq \) contains the items \{the\}, \{path\}, \{for\}, \{this\}, and \{file.tex\}, and the \text{true} branch is left in the input stream.

6 Replacement

Searches for the \( \langle \text{regular expression} \rangle \) in the \( \langle \text{token list} \rangle \) and replaces the first match with the \( \langle \text{replacement} \rangle \). The result is assigned locally to \( \langle \text{tl var} \rangle \). In the \( \langle \text{replacement} \rangle \), \( \backslash 0 \) represents the full match, \( \backslash 1 \) represent the contents of the first capturing group, \( \backslash 2 \) of the second, etc.

Replaces all occurrences of the \( \langle \text{regular expression} \rangle \) in the \( \langle \text{token list} \rangle \) by the \( \langle \text{replacement} \rangle \), where \( \backslash 0 \) represents the full match, \( \backslash 1 \) represent the contents of the first capturing group, \( \backslash 2 \) of the second, etc. Every match is treated independently, and matches cannot overlap. The result is assigned locally to \( \langle \text{tl var} \rangle \).

7 Constants and variables

Scratch regex 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.

Scratch regex 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 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
  Additional error-checking to come.
- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register \c{max_register_int}.
- Find out whether the fact that \w and friends match the end-marker leads to bugs. Possibly update \__regex_item_reverse:n.
- The empty cs should be matched by \c{}, not by \c{csname.?endcsname}s?.
  Code improvements to come.
- Shift arrays so that the useful information starts at position 1.
- Only build \c{...} once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should \__regex_action_free_group:n only be used for greedy {n,} quantifier? (I think not.)
- Quantifiers for \u and assertions.
- When matching, keep track of an explicit stack of current_state and current_submatches.
- If possible, when a state is reused by the same thread, kill other subthreads.
- Use an array rather than \__regex_balance_tl to build the function \__regex_replacement_balance_one_match:n.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (abcade should give two states, for abc and ade). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single \__regex_action_free:n.
- Optimize the use of \__regex_action_success: by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of \int_step... functions.
• Groups don’t capture within regexes for csnames; optimize and document.
• Better “show” for anchors, properties, and catcode tests.
• Does \K really need a new state for itself?
• When compiling, use a boolean \in_cs and less magic numbers.
• Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with \cs_if_exist tests.

The following features are likely to be implemented at some point in the future.
• General look-ahead/behind assertions.
• Regex matching on external files.
• Conditional subpatterns with look ahead/behind: “if what follows is […], then […].”
• (*. .) and (? . .) sequences to set some options.
• UTF-8 mode for pdf\TeX.
• Newline conventions are not done. In particular, we should have an option for . not to match newlines. Also, \A should differ from ^, and \Z, \z and $ should differ.
• Unicode properties: \p{. .} and \P{. .}: \X which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.
• Provide a syntax such as \ur{l_my_regex} to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
• Allowing \u{l_my_tl} in more places, for instance as the number of repetitions in a quantifier.

The following features of \texttt{pcre} or Perl may or may not be implemented.
• Callout with (?C . . .) or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential \regex_break: and then of playing well with \tl_map_break: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since \fontdimen are global.
• Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn’t it?
• Named subpatterns: \TeX programmers have lived so far without any need for named macro parameters.

The following features of \texttt{pcre} or Perl will definitely not be implemented.
• Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
• Recursion: this is a non-regular feature.

• Atomic grouping, possessive quantifiers: those tools, mostly meant to fix cata-
  tropheic backtracking, are unnecessary in a non-backtracking algorithm, and difficult
to implement.

• Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking al-
  gorithm, in particular because the corresponding group should be treated as atomic.

• Backtracking control verbs: intrinsically tied to backtracking.

• \ddd, matching the character with octal code ddd: we already have \{...\} and
  the syntax is confusingly close to what we could have used for backreferences (\1,
\2, ...), making it harder to produce useful error message.

• \cx, similar to TeX’s own \^{x}.

• Comments: TeX already has its own system for comments.

• \Q...\E escaping: this would require to read the argument verbatim, which is not
  in the scope of this module.

• \C single byte in UTF-8 mode: XeTeX and LuaTeX serve us characters directly,
  and splitting those into bytes is tricky, encoding dependent, and most likely not
  useful anyways.
Part XXVIII

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:N <box>
\box_new:c

Creates a new <box> or raises an error if the name is already taken. The declaration is
global. The <box> is initially void.

\box_clear:N \box_clear:c
\box_gclear:N \box_gclear:c
\box_clear_new:N \box_clear_new:c
\box_gclear_new:N \box_gclear_new:c

Clears the content of the <box> by setting the box equal to \c_empty_box.

\box_set_eq:NN \box_set_eq:nc \box_gset_eq:NN \box_gset_eq:nc
\box_gset_eq:cc

Sets the content of <box1> equal to that of <box2>.

\box_if_exist_p:N \box_if_exist:NTF \box_if_exist:p:N \box_if_exist:c:TF
\box_if_exist:c:TF

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. An error is
raised if the variable does not exist or if it is invalid.

\TeXhackers note: This is the \TeX primitive \copy.
This function operates in vertical mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced horizontally by the given \texttt{dimexpr} from the reference point for typesetting, to the right or left as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \vbox<n> or a “raw” box specification such as \vbox:n \{ xyz \}}.

This function operates in horizontal mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced vertically by the given \texttt{dimexpr} from the reference point for typesetting, up or down as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \vbox<n> or a “raw” box specification such as \vbox:n \{ xyz \}}.

3 Measuring and setting box dimensions

Calculates the depth (below the baseline) of the \texttt{box} in a form suitable for use in a \texttt{dimension expression}.

\textbf{\LaTeXeX hackers note:} This is the \LaTeX primitive \texttt{dp}.

Calculates the height (above the baseline) of the \texttt{box} in a form suitable for use in a \texttt{dimension expression}.

\textbf{\LaTeXeX hackers note:} This is the \LaTeX primitive \texttt{ht}.

Calculates the width of the \texttt{box} in a form suitable for use in a \texttt{dimension expression}.

\textbf{\LaTeXeX hackers note:} This is the \LaTeX primitive \texttt{wd}.

Set the depth (below the baseline) of the \texttt{box} to the value of the \texttt{dimexpr}.

Set the height (above the baseline) of the \texttt{box} to the value of the \texttt{dimexpr}.
\texttt{\textbackslash box\_set\_wd:}\texttt{Nn} \hspace{1cm} \texttt{\textbackslash box\_set\_wd:cn} \hspace{1cm} \texttt{\textbackslash box\_gset\_wd:}\texttt{Nn} \hspace{1cm} \texttt{\textbackslash box\_gset\_wd:cn}

Set the width of the \texttt{\langle box\rangle} to the value of the \texttt{\textbackslash{dimension expression}}.

\texttt{\textbackslash box\_set\_wd:}\texttt{Nn} \hspace{1cm} \texttt{\textbackslash box\_set\_wd:cn} \hspace{1cm} \texttt{\textbackslash box\_gset\_wd:}\texttt{Nn} \hspace{1cm} \texttt{\textbackslash box\_gset\_wd:cn}

Updated: 2019-01-22

4 Box conditionals

\texttt{\textbackslash box\_if\_empty\_p:}\texttt{N} \hspace{1cm} \texttt{\textbackslash box\_if\_empty\_p:c} \hspace{1cm} \texttt{\textbackslash box\_if\_empty:}\texttt{NTF} \hspace{1cm} \texttt{\textbackslash box\_if\_empty:}\texttt{cTF}

Tests if \texttt{\langle box\rangle} is a empty (equal to \texttt{\textbackslash c\_empty\_box}).

\texttt{\textbackslash box\_if\_horizontal\_p:}\texttt{N} \hspace{1cm} \texttt{\textbackslash box\_if\_horizontal\_p:c} \hspace{1cm} \texttt{\textbackslash box\_if\_horizontal:}\texttt{NTF} \hspace{1cm} \texttt{\textbackslash box\_if\_horizontal:}\texttt{cTF}

Tests if \texttt{\langle box\rangle} is a horizontal box.

\texttt{\textbackslash box\_if\_vertical\_p:}\texttt{N} \hspace{1cm} \texttt{\textbackslash box\_if\_vertical\_p:c} \hspace{1cm} \texttt{\textbackslash box\_if\_vertical:}\texttt{NTF} \hspace{1cm} \texttt{\textbackslash box\_if\_vertical:}\texttt{cTF}

Tests if \texttt{\langle box\rangle} is a vertical box.

5 The last box inserted

\texttt{\textbackslash box\_set\_to\_last:}\texttt{N} \hspace{1cm} \texttt{\textbackslash box\_set\_to\_last:c} \hspace{1cm} \texttt{\textbackslash box\_gset\_to\_last:}\texttt{N} \hspace{1cm} \texttt{\textbackslash box\_gset\_to\_last:c}

Sets the \texttt{\langle box\rangle} 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 \texttt{\langle box\rangle} is always void as it is not possible to recover the last added item.

6 Constant boxes

\texttt{\textbackslash c\_empty\_box}

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

\TeXhacker{} note: At the \TeX{} level this is a void box.

237
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:cn

\New: 2012-05-11

\box_show:Nnn (box) \{intexpr\} \{intexpr\}

Display the contents of (box) in the terminal, showing the first \langle intexpr\rangle items of the box, and descending into \langle intexpr\rangle 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:cn

\New: 2012-05-11

\box_log:Nnn (box) \{intexpr\} \{intexpr\}

Writes the contents of (box) to the log, showing the first \langle intexpr\rangle items of the box, and descending into \langle intexpr\rangle group levels.

9 Boxes and color

All \LaTeX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

10 Horizontal mode boxes

\hbox:n

Updated: 2017-04-05

\hbox:n \{contents\}

Typesets the \{contents\} into a horizontal box of natural width and then includes this box in the current list for typesetting.
\texttt{\textbackslash hbox\_to\_wd:n} \texttt{\textbackslash hbox\_to\_wd:n \{\langle\text{dimexpr}\rangle\} \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle into a horizontal box of width \langle\text{dimexpr}\rangle and then includes this box in the current list for typesetting.

\texttt{\textbackslash hbox\_to\_zero:n} \texttt{\textbackslash hbox\_to\_zero:n \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle into a horizontal box of zero width and then includes this box in the current list for typesetting.

\texttt{\textbackslash hbox\_set:Nn} \texttt{\textbackslash hbox\_set:cn} \texttt{\textbackslash hbox\_gset:Nn} \texttt{\textbackslash hbox\_gset:cn}

\texttt{\textbackslash hbox\_set:Nn \langle\text{box}\rangle \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle at natural width and then stores the result inside the \langle\text{box}\rangle.

\texttt{\textbackslash hbox\_set_to\_wd:Nnn} \texttt{\textbackslash hbox\_set_to\_wd:cnn} \texttt{\textbackslash hbox\_gset_to\_wd:Nnn} \texttt{\textbackslash hbox\_gset_to\_wd:cnn}

\texttt{\textbackslash hbox\_set_to\_wd:Nnn \langle\text{box}\rangle \{\langle\text{dimexpr}\rangle\} \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle to the width given by the \langle\text{dimexpr}\rangle and then stores the result inside the \langle\text{box}\rangle.

\texttt{\textbackslash hbox\_overlap\_right:n} \texttt{\textbackslash hbox\_overlap\_right:n \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle into a horizontal box of zero width such that material protrudes to the right of the insertion point.

\texttt{\textbackslash hbox\_overlap\_left:n} \texttt{\textbackslash hbox\_overlap\_left:n \{\langle\text{contents}\rangle\}}

Typesets the \langle\text{contents}\rangle into a horizontal box of zero width such that material protrudes to the left of the insertion point.

\texttt{\textbackslash hbox\_set:Nw} \texttt{\textbackslash hbox\_set:cw} \texttt{\textbackslash hbox\_set_end:} \texttt{\textbackslash hbox\_gset:Nw} \texttt{\textbackslash hbox\_gset:cw} \texttt{\textbackslash hbox\_gset_end:}

\texttt{\textbackslash hbox\_set:Nw \langle\text{box}\rangle \{\langle\text{contents}\rangle\} \textbackslash hbox\_set\_end:}

Typesets the \langle\text{contents}\rangle at natural width and then stores the result inside the \langle\text{box}\rangle. In contrast to \texttt{\textbackslash hbox\_set:Nn} this function does not absorb the argument when finding the \langle\text{content}\rangle, and so can be used in circumstances where the \langle\text{content}\rangle may not be a simple argument.

\texttt{\textbackslash hbox\_set_to\_wd:Nnw} \texttt{\textbackslash hbox\_set_to\_wd:cnw} \texttt{\textbackslash hbox\_gset_to\_wd:Nnw} \texttt{\textbackslash hbox\_gset_to\_wd:cnw}

\texttt{\textbackslash hbox\_set_to\_wd:Nnw \langle\text{box}\rangle \{\langle\text{dimexpr}\rangle\} \{\langle\text{contents}\rangle\} \textbackslash hbox\_set\_end:}

Typesets the \langle\text{contents}\rangle to the width given by the \langle\text{dimexpr}\rangle and then stores the result inside the \langle\text{box}\rangle. In contrast to \texttt{\textbackslash hbox\_set_to\_wd:Nnn} this function does not absorb the argument when finding the \langle\text{content}\rangle, and so can be used in circumstances where the \langle\text{content}\rangle may not be a simple argument.

\texttt{\textbackslash hbox\_unpack:N} \texttt{\textbackslash hbox\_unpack:c}

\texttt{\textbackslash hbox\_unpack:N \langle\text{box}\rangle}

Unpacks the content of the horizontal \langle\text{box}\rangle, retaining any stretching or shrinking applied when the \langle\text{box}\rangle was set.

\textbf{\LaTeX}hacker note: This is the \LaTeX primitive \texttt{\textbackslash unhcopy}. 

239
11 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 has 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 is typically non-zero.

\vbox:n \{\textit{contents}\}

Typesets the \textit{contents} into a vertical box of natural height and includes this box in the current list for typesetting.

\vbox_top:n \{\textit{contents}\}

Typesets the \textit{contents} into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the first item added to the box.

\vbox_to_ht:nn \{\textit{dimexpr}\} \{\textit{contents}\}

Typesets the \textit{contents} into a vertical box of height \textit{dimexpr} and then includes this box in the current list for typesetting.

\vbox_to_zero:n \{\textit{contents}\}

Typesets the \textit{contents} into a vertical box of zero height and then includes this box in the current list for typesetting.

\vbox_set:Nn \{\textit{contents}\}

Typesets the \textit{contents} at natural height and then stores the result inside the \textit{box}.

\vbox_set_top:Nn \{\textit{contents}\}

Typesets the \textit{contents} at natural height and then stores the result inside the \textit{box}. The baseline of the box is equal to that of the first item added to the box.

\vbox_set_to_ht:Nnn \{\textit{box}\} \{\textit{dimexpr}\} \{\textit{contents}\}

Typesets the \textit{contents} to the height given by the \textit{dimexpr} and then stores the result inside the \textit{box}.
\vbox_set:Nw \vbox_set:cw \vbox_set_end:
Typesets the \langle contents \rangle at natural height and then stores the result inside the \langle box \rangle. In contrast to \vbox_set:Nn this function does not absorb the argument when finding the \langle content \rangle, and so can be used in circumstances where the \langle content \rangle may not be a simple argument.

\vbox_set_to_ht:Nnw \vbox_set_to_ht:cnw \vbox_set_to_ht:cnw \vbox_gset_to_ht:Nnw \vbox_gset_to_ht:cnw
Updated: 2017-04-05

\vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \vbox_gset_split_to_ht:NNn \vbox_gset_split_to_ht:NNn
Updated: 2018-12-29

\vbox_unpack:N \vbox_unpack:c
Unpacks the content of the vertical \langle box \rangle, retaining any stretching or shrinking applied when the \langle box \rangle was set.

\TeXhackers\ note: This is the \TeX\ primitive \unvcopy.

12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other expl3 variable. However, for efficiency reasons, it is also useful to have functions which drop box contents on use. When a box is dropped, the box becomes empty at the group level where the box was originally set rather than necessarily at the current group level. For example, with

\hbox_set:Nn \l_tmpa_box { A } \group_begin: \hbox_set:Nn \l_tmpa_box { B } \group_begin: \box_use_drop:N \l_tmpa_box \group_end: \box_show:N \l_tmpa_box \group_end: \box_show:N \l_tmpa_box

241
the first use of \texttt{\box_show:N} will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of \texttt{drop} functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

\begin{verbatim}
\box_use_drop:N
\box_use_drop:c
\end{verbatim}

Inserts the current content of the \texttt{\langle box\rangle} onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

\textbf{\TeXhackers note:} This is the \texttt{\box} primitive.

\begin{verbatim}
\box_set_eq_drop:NN
\box_set_eq_drop:(cN|Nc|cc)
\end{verbatim}

Sets the content of \texttt{\langle box_1\rangle} equal to that of \texttt{\langle box_2\rangle}, then drops \texttt{\langle box_2\rangle}.

\textbf{\TeXhackers note:} This is the \texttt{\box} primitive.

\begin{verbatim}
\box_gset_eq_drop:NN
\box_gset_eq_drop:(cN|Nc|cc)
\end{verbatim}

Sets the content of \texttt{\langle box_1\rangle} globally equal to that of \texttt{\langle box_2\rangle}, then drops \texttt{\langle box_2\rangle}.

\textbf{\TeXhackers note:} This is the \texttt{\box} primitive.

\begin{verbatim}
\hbox_unpack_drop:N
\hbox_unpack_drop:c
\end{verbatim}

Unpacks the content of the horizontal \texttt{\langle box\rangle}, retaining any stretching or shrinking applied when the \texttt{\langle box\rangle} was set. The original \texttt{\langle box\rangle} is then dropped.

\textbf{\TeXhackers note:} This is the \texttt{\TeX} primitive \texttt{\unhbox}.

\begin{verbatim}
\vbox_unpack_drop:N
\vbox_unpack_drop:c
\end{verbatim}

Unpacks the content of the vertical \texttt{\langle box\rangle}, retaining any stretching or shrinking applied when the \texttt{\langle box\rangle} was set. The original \texttt{\langle box\rangle} is then dropped.

\textbf{\TeXhackers note:} This is the \texttt{\TeX} primitive \texttt{\unvbox}.

### 13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in \texttt{\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.
Resizes the \( \langle \text{box} \rangle \) to fit within the given \( \langle x\text{-size} \rangle \) (horizontally) and \( \langle y\text{-size} \rangle \) (vertically); both of the sizes are dimension expressions. The \( \langle y\text{-size} \rangle \) is the height only: it does not include any depth. The updated \( \langle \text{box} \rangle \) is an hbox, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. The final size of the \( \langle \text{box} \rangle \) is the smaller of \( \{\langle x\text{-size} \rangle \} \) and \( \{\langle y\text{-size} \rangle \} \), i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle y\text{-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and vice versa.

Resizes the \( \langle \text{box} \rangle \) to fit within the given \( \langle x\text{-size} \rangle \) (horizontally) and \( \langle y\text{-size} \rangle \) (vertically); both of the sizes are dimension expressions. The \( \langle y\text{-size} \rangle \) is the total vertical size (height plus depth). The updated \( \langle \text{box} \rangle \) is an hbox, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. The final size of the \( \langle \text{box} \rangle \) is the smaller of \( \{\langle x\text{-size} \rangle \} \) and \( \{\langle y\text{-size} \rangle \} \), i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle y\text{-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and vice versa.

Resizes the \( \langle \text{box} \rangle \) to \( \langle y\text{-size} \rangle \) (vertically), scaling the horizontal size by the same amount; \( \langle y\text{-size} \rangle \) is a dimension expression. The \( \langle y\text{-size} \rangle \) is the height only: it does not include any depth. The updated \( \langle \text{box} \rangle \) is an hbox, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. A negative \( \langle y\text{-size} \rangle \) causes the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle y\text{-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and vice versa.
Resizes the \( \langle \text{box} \rangle \) to \( \langle \text{y-size} \rangle \) (vertically), scaling the horizontal size by the same amount: \( \langle \text{y-size} \rangle \) is a dimension expression. The \( \langle \text{y-size} \rangle \) is the total vertical size (height plus depth). The updated \( \langle \text{box} \rangle \) is an \texttt{hbox}, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. A negative \( \langle \text{y-size} \rangle \) causes the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle \text{y-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \( \langle \text{box} \rangle \) to \( \langle \text{x-size} \rangle \) (horizontally), scaling the vertical size by the same amount: \( \langle \text{x-size} \rangle \) is a dimension expression. The updated \( \langle \text{box} \rangle \) is an \texttt{hbox}, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. A negative \( \langle \text{x-size} \rangle \) causes the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle \text{x-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \( \langle \text{box} \rangle \) to \( \langle \text{x-size} \rangle \) (horizontally) and \( \langle \text{y-size} \rangle \) (vertically): both of the sizes are dimension expressions. The \( \langle \text{y-size} \rangle \) is the height only and does not include any depth. The updated \( \langle \text{box} \rangle \) is an \texttt{hbox}, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. Negative sizes cause the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle \text{y-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \( \langle \text{box} \rangle \) to \( \langle \text{x-size} \rangle \) (horizontally) and \( \langle \text{y-size} \rangle \) (vertically): both of the sizes are dimension expressions. The \( \langle \text{y-size} \rangle \) is the total vertical size (height plus depth). The updated \( \langle \text{box} \rangle \) is an \texttt{hbox}, irrespective of the nature of the \( \langle \text{box} \rangle \) before the resizing is applied. Negative sizes cause the material in the \( \langle \text{box} \rangle \) to be reversed in direction, but the reference point of the \( \langle \text{box} \rangle \) is unchanged. Thus a negative \( \langle \text{y-size} \rangle \) results in the \( \langle \text{box} \rangle \) having a depth dependent on the height of the original and \textit{vice versa}.
\texttt{\texttt{\textbackslash box\_rotate:Nn}} \texttt{\texttt{\textbackslash box\_rotate:cn}} \texttt{\texttt{\textbackslash box\_grotate:Nn}} \texttt{\texttt{\textbackslash box\_grotate:cn}}

Rotates the \texttt{\textbackslash box} by \texttt{\textit{angle}} (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated \texttt{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the rotation is applied.

\texttt{\texttt{\textbackslash box\_scale:Nnn}} \texttt{\texttt{\textbackslash box\_scale:cn}} \texttt{\texttt{\textbackslash box\_gscale:Nnn}} \texttt{\texttt{\textbackslash box\_gscale:cn}}

Scales the \texttt{\textbackslash box} by factors \texttt{x\_scale} and \texttt{y\_scale} in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated \texttt{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the scaling is applied. Negative scalings cause the material in the \texttt{\textbackslash box} to be reversed in direction, but the reference point of the \texttt{\textbackslash box} is unchanged. Thus a negative \texttt{y\_scale} results in the \texttt{\textbackslash box} having a depth dependent on the height of the original and \textit{vice versa}.

14 Primitive box conditionals

\texttt{\texttt{\textbackslash if\_hbox:N}} \texttt{\texttt{\textbackslash if\_hbox:N}} \texttt{\texttt{\textbackslash if\_hbox:N}} \texttt{\texttt{\textbackslash if\_hbox:N}}

\texttt{\texttt{\texttt{\textbackslash if\_hbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_hbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_hbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_hbox:N}}}

Tests is \texttt{\textbackslash box} is a horizontal box.

\texttt{T\textsuperscript{P}Xhackers note:} This is the \texttt{\textsc{\texttt{\textbackslash T\textsc{e}X}}} primitive \texttt{\texttt{\textbackslash if\_hbox}.}

\texttt{\texttt{\textbackslash if\_vbox:N}} \texttt{\texttt{\textbackslash if\_vbox:N}} \texttt{\texttt{\textbackslash if\_vbox:N}} \texttt{\texttt{\textbackslash if\_vbox:N}}

\texttt{\texttt{\texttt{\textbackslash if\_vbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_vbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_vbox:N}}} \texttt{\texttt{\texttt{\textbackslash if\_vbox:N}}}

Tests is \texttt{\textbackslash box} is a vertical box.

\texttt{T\textsuperscript{P}Xhackers note:} This is the \texttt{\textsc{\texttt{\textbackslash T\textsc{e}X}}} primitive \texttt{\texttt{\textbackslash if\_vbox}.}

\texttt{\texttt{\texttt{\textbackslash if\_box\_empty:N}} \texttt{\texttt{\textbackslash if\_box\_empty:N}} \texttt{\texttt{\textbackslash if\_box\_empty:N}} \texttt{\texttt{\textbackslash if\_box\_empty:N}}}

\texttt{\texttt{\texttt{\texttt{\textbackslash if\_box\_empty:N}}} \texttt{\texttt{\texttt{\textbackslash if\_box\_empty:N}}} \texttt{\texttt{\texttt{\textbackslash if\_box\_empty:N}}} \texttt{\texttt{\texttt{\textbackslash if\_box\_empty:N}}}

Tests is \texttt{\textbackslash box} is an empty (void) box.

\texttt{T\textsuperscript{P}Xhackers note:} This is the \texttt{\textsc{\texttt{\textbackslash T\textsc{e}X}}} primitive \texttt{\texttt{\textbackslash if\_void}.}
Part XXIX

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

Creates a new \texttt{coffin} or raises an error if the name is already taken. The declaration is global. The \texttt{coffin} is initially empty.

\coffin_clear:N \coffin_clear:c
\coffin_gclear:N \coffin_gclear:c

New: 2011-08-17
Updated: 2019-01-21

\coffin_set_eq:NN \coffin_set_eq:cn \coffin_gset_eq:NN \coffin_gset_eq:cn

New: 2011-08-17
Updated: 2019-01-21

\coffin_if_exist_p:N \coffin_if_exist:NTF \coffin_if_exist:TF \coffin_if_exist:c

New: 2012-06-20

2 Setting coffin content and poles

\hcoffin_set:Nn \hcoffin_set:cn
\hcoffin_gset:Nn \hcoffin_gset:cn

New: 2011-08-17
Updated: 2019-01-21

\hcoffin_set:Nn \hcoffin_set:cn \hcoffin_gset:Nn \hcoffin_gset:cn

\coffin_set_eq:NN \coffin_set_eq:cn \coffin_gset_eq:NN \coffin_gset_eq:cn

\coffin_if_exist_p:N \coffin_if_exist:NTF \coffin_if_exist:TF \coffin_if_exist:c

New: 2011-08-17
Updated: 2019-01-21

\coffin_set_eq:NN \coffin_set_eq:cn \coffin_gset_eq:NN \coffin_gset_eq:cn

Sets both the content and poles of \texttt{coffin}_1 equal to those of \texttt{coffin}_2.

\coffin_if_exist_p:N \coffin_if_exist:NTF \coffin_if_exist:TF \coffin_if_exist:c

New: 2012-06-20

Typesets the \texttt{material} in horizontal mode, storing the result in the \texttt{coffin}. The standard poles for the \texttt{coffin} are then set up based on the size of the typeset material.
\hcoffin_set:Nw \hcoffin_set:cw \hcoffin_set_end:
\hcoffin_gset:Nw \hcoffin_gset:cw \hcoffin_gset_end:

Typesets the \langle material \rangle in horizontal mode, storing the result in the \langle coffin \rangle. The standard poles for the \langle coffin \rangle 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 \vcoffin_set_end:
\vcoffin_gset:Nnn \vcoffin_gset:cnn

Typesets the \langle material \rangle in vertical mode constrained to the given \langle width \rangle and stores the result in the \langle coffin \rangle. The standard poles for the \langle coffin \rangle are then set up based on the size of the typeset material.

\vcoffin_set:Nnw \vcoffin_set:cnw \vcoffin_set_end:
\vcoffin_gset:Nnw \vcoffin_gset:cnw

Typesets the \langle material \rangle in vertical mode constrained to the given \langle width \rangle and stores the result in the \langle coffin \rangle. The standard poles for the \langle coffin \rangle 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 \coffin_gset_horizontal_pole:Nnn \coffin_gset_horizontal_pole:cnn

Sets the \langle pole \rangle to run horizontally through the \langle coffin \rangle. The \langle pole \rangle is placed at the \langle offset \rangle from the bottom edge of the bounding box of the \langle coffin \rangle. The \langle offset \rangle should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn \coffin_set_vertical_pole:cnn \coffin_gset_vertical_pole:Nnn \coffin_gset_vertical_pole:cnn

Sets the \langle pole \rangle to run vertically through the \langle coffin \rangle. The \langle pole \rangle is placed at the \langle offset \rangle from the left-hand edge of the bounding box of the \langle coffin \rangle. The \langle offset \rangle should be given as a dimension expression.
3 Coffin affine transformations

\coffin_resize:Nnn \coffin_resize:cnn
\coffin_gresize:Nnn \coffin_gresize:cnn

Updated: 2019-01-23

\coffin_rotate:Nn \coffin_rotate:cn
\coffin_grotate:Nn \coffin_grotate:cn

\coffin_scale:Nnn \coffin_scale:cnn
\coffin_gscale:Nnn \coffin_gscale:cnn

Updated: 2019-01-23

Resized the \(\text{coffin}\) to \(\text{width}\) and \(\text{total-height}\), both of which should be given as dimension expressions.

\coffin_rotate:Nn \coffin_rotate:cn
\coffin_grotate:Nn \coffin_grotate:cn

Rotates the \(\text{coffin}\) by the given \(\text{angle}\) (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

\coffin_scale:Nnn \coffin_scale:cnn
\coffin_gscale:Nnn \coffin_gscale:cnn

Scales the \(\text{coffin}\) by a factors \(\text{x-scale}\) and \(\text{y-scale}\) in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

4 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:{cnnNnnnn}[Nnnncccc][ccnnnccc] \coffin_attach:NnnNnnnn
\coffin_gattach:NnnNnnnn
\coffin_gattach:{cnnNnnnn}[Nnnncccc][ccnnnccc]

Updated: 2019-01-22

This function attaches \(\text{coffin}_2\) to \(\text{coffin}_1\) such that the bounding box of \(\text{coffin}_1\) is not altered, i.e. \(\text{coffin}_2\) can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating \(\text{handle}_1\), the point of intersection of \(\text{coffin}_1\) and \(\text{coffin}_2\), and \(\text{handle}_2\), the point of intersection of \(\text{coffin}_2\) and \(\text{coffin}_2\). \(\text{coffin}_2\) is then attached to \(\text{coffin}_1\) such that the relationship between \(\text{handle}_1\) and \(\text{handle}_2\) is described by the \(\text{x-offset}\) and \(\text{y-offset}\). The two offsets should be given as dimension expressions.

\coffin_join:NnnNnnnn
\coffin_join:{cnnNnnnn}[Nnnncccc][ccnnnccc] \coffin_join:NnnNnnnn
\coffin_gjoin:NnnNnnnn
\coffin_gjoin:{cnnNnnnn}[Nnnncccc][ccnnnccc]

Updated: 2019-01-22

This function joins \(\text{coffin}_2\) to \(\text{coffin}_1\) such that the bounding box of \(\text{coffin}_1\) may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating \(\text{handle}_1\), the point of intersection of \(\text{coffin}_1\) and \(\text{coffin}_2\), and \(\text{handle}_2\), the point of intersection of \(\text{coffin}_2\) and \(\text{coffin}_2\). \(\text{coffin}_2\) is then attached to \(\text{coffin}_1\) such that the relationship between \(\text{handle}_1\) and \(\text{handle}_2\) is described by the \(\text{x-offset}\) and \(\text{y-offset}\). The two offsets should be given as dimension expressions.
Typesetting is carried out by first calculating \( \langle \text{handle} \rangle \), the point of intersection of \( \langle \text{pole}_1 \rangle \) and \( \langle \text{pole}_2 \rangle \). The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the \( \langle \text{handle} \rangle \) is described by the \( \langle x\text{-offset}\rangle \) and \( \langle y\text{-offset}\rangle \). 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.

5 Measuring coffins

\texttt{\coffin_dp:N} \langle \text{coffin} \rangle
\texttt{\coffin_dp:C}
Calculates the depth (below the baseline) of the \( \langle \text{coffin} \rangle \) in a form suitable for use in a \langle dimension expression \rangle.

\texttt{\coffin_ht:N} \langle \text{coffin} \rangle
\texttt{\coffin_ht:C}
Calculates the height (above the baseline) of the \( \langle \text{coffin} \rangle \) in a form suitable for use in a \langle dimension expression \rangle.

\texttt{\coffin_wd:N} \langle \text{coffin} \rangle
\texttt{\coffin_wd:C}
Calculates the width of the \( \langle \text{coffin} \rangle \) in a form suitable for use in a \langle dimension expression \rangle.

6 Coffin diagnostics

\texttt{\coffin_display_handles:Nn} \langle \text{coffin} \rangle \{\langle\text{color}\rangle\}
This function first calculates the intersections between all of the \( \langle \text{poles} \rangle \) of the \( \langle \text{coffin} \rangle \) to give a set of \( \langle \text{handles} \rangle \). It then prints the \( \langle \text{coffin} \rangle \) at the current location in the source, with the position of the \( \langle \text{handles} \rangle \) marked on the coffin. The \( \langle \text{handles} \rangle \) are labelled as part of this process; the locations of the \( \langle \text{handles} \rangle \) and the labels are both printed in the \( \langle \text{color} \rangle \) specified.

\texttt{\coffin_mark_handle:Nnnn} \langle \text{coffin} \rangle \{\langle \text{pole}_1 \rangle\} \{\langle \text{pole}_2 \rangle\} \{\langle \text{color} \rangle\}
This function first calculates the \( \langle \text{handle} \rangle \) for the \( \langle \text{coffin} \rangle \) as defined by the intersection of \( \langle \text{pole}_1 \rangle \) and \( \langle \text{pole}_2 \rangle \). It then marks the position of the \( \langle \text{handle} \rangle \) on the \( \langle \text{coffin} \rangle \). The \( \langle \text{handle} \rangle \) are labelled as part of this process: the location of the \( \langle \text{handle} \rangle \) and the label are both printed in the \( \langle \text{color} \rangle \) specified.

\texttt{\coffin_show_structure:N} \langle \text{coffin} \rangle
This function shows the structural information about the \( \langle \text{coffin} \rangle \) 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.

249
This function writes the structural information about the \texttt{coffin} in the log file. See also \texttt{coffin\_show\_structure} which displays the result in the terminal.

7 Constants and variables

\texttt{c\_empty\_coffin}

A permanently empty coffin.

\texttt{l\_tmpa\_coffin} \texttt{l\_tmpb\_coffin}

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

\texttt{g\_tmpa\_coffin} \texttt{g\_tmpb\_coffin}

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX{}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Part XXX
The \texttt{l3color-base} package
Color support

This module provides support for color in \LaTeXe. At present, the material here is mainly intended to support a small number of low-level requirements in other \texttt{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.

\begin{verbatim}
\color_group_begin:
\color_group_end:
\end{verbatim}

Creates a color group: one used to “trap” color settings.

\begin{verbatim}
\color_ensure_current:
\end{verbatim}

Ensures that material inside a box uses 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 \texttt{\color_group_begin: ... \color_group_end:} group.
Part XXXI

The \texttt{l3luatex} package:
\texttt{LuaTEX}-specific functions

The \texttt{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 \texttt{pdfTEX}, \texttt{pTEX}, \texttt{upTEX} or \texttt{XeTEX} these raise an error: use \texttt{\sys_if_engine_luatex:T} to avoid this. Details on using Lua with the \texttt{LuaTEX} engine are given in the \texttt{LuaTEX} manual.

1 Breaking out to Lua

\begin{quote}
\texttt{\lua_now:n \{token list\}}
\end{quote}

The \texttt{\{token list\}} is first tokenized by \TeX, which includes converting line ends to spaces in the usual \TeX manner and which respects currently-applicable \TeX category codes. The resulting \texttt{\{Lua input\}} is passed to the Lua interpreter for processing. Each \texttt{\lua_now:n} block is treated by Lua as a separate chunk. The Lua interpreter executes the \texttt{\{Lua input\}} immediately, and in an expandable manner.

\texttt{\textsc{TeX}hackers note:} \texttt{\lua_now:e} is a macro wrapper around \texttt{\directlua}: when \texttt{LuaTEX} is in use two expansions are required to yield the result of the Lua code.

\begin{quote}
\texttt{\lua_shipout:n \{token list\}}
\end{quote}

The \texttt{\{token list\}} is first tokenized by \TeX, which includes converting line ends to spaces in the usual \TeX manner and which respects currently-applicable \TeX category codes. The resulting \texttt{\{Lua input\}} is passed to the Lua interpreter when the current page is finalised \textit{(i.e.} at shipout\textit{)}. Each \texttt{\lua_shipout:n} block is treated by Lua as a separate chunk. The Lua interpreter will execute the \texttt{\{Lua input\}} during the page-building routine: no \TeX expansion of the \texttt{\{Lua input\}} will occur at this stage.

In the case of the \texttt{\lua_shipout_e:n} version the input is fully expanded by \TeX in an \texttt{e}-type manner during the shipout operation.

\texttt{\textsc{TeX}hackers note:} At a \TeX level, the \texttt{\{Lua input\}} is stored as a “whatsit”.

\begin{quote}
\texttt{\lua_escape:n \{token list\}}
\end{quote}

Converts the \texttt{\{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 \texttt{\textbackslash n} and \texttt{\textbackslash r}, respectively.

\texttt{\textsc{TeX}hackers note:} \texttt{\lua_escape:e} is a macro wrapper around \texttt{\luaescapestring}: when \texttt{LuaTEX} is in use two expansions are required to yield the result of the Lua code.
2 Lua interfaces

As well as interfaces for \TeX, there are a small number of Lua functions provided here.

\begin{itemize}
  \item \texttt{\texttt{\texttt{l3kernel}}} All public interfaces provided by the module are stored within the \texttt{l3kernel} table.
  \item \texttt{\texttt{l3kernel.charcat}}\(\langle\text{charcode}\rangle, \langle\text{catcode}\rangle\) Constructs a character of \texttt{\langle\text{charcode}\rangle} and \texttt{\langle\text{catcode}\rangle} and returns the result to \TeX.
  \item \texttt{\texttt{l3kernel.elapsedtime}}\()\) Returns the CPU time in \texttt{(scaled seconds)} since the start of the \TeX\ run or since \texttt{l3kernel.resettimer} was issued. This only measures the time used by the CPU, not the real time, e.g., waiting for user input.
  \item \texttt{\texttt{l3kernel.filedump}}\(\langle\text{file}\rangle, \langle\text{offset}\rangle, \langle\text{length}\rangle\) Returns the uppercase hexadecimal representation of the content of the \texttt{\langle\text{file}\rangle} read as bytes. If the \texttt{\langle\text{length}\rangle} is given, only this part of the file is returned; similarly, one may specify the \texttt{\langle\text{offset}\rangle} from the start of the file. If the \texttt{\langle\text{length}\rangle} is not given, the entire file is read starting at the \texttt{\langle\text{offset}\rangle}.
  \item \texttt{\texttt{l3kernel.filemdfivesum}}\(\langle\text{file}\rangle\) Returns the \texttt{MD5} sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal \TeX\ behaviour. If the \texttt{\langle\text{file}\rangle} is not found, nothing is returned with \texttt{no error raised}.
  \item \texttt{\texttt{l3kernel.filemoddate}}\(\langle\text{file}\rangle\) Returns the of the date/time of last modification of the \texttt{\langle\text{file}\rangle} in the format
    \[
    D:\langle\text{year}\rangle\langle\text{month}\rangle\langle\text{day}\rangle\langle\text{hour}\rangle\langle\text{minute}\rangle\langle\text{second}\rangle\langle\text{offset}\rangle
    \]
    where the latter may be \texttt{Z} (UTC) or \texttt{(plus-minus)\langle\text{hours}\rangle'\langle\text{minutes}\rangle'}. If the \texttt{\langle\text{file}\rangle} is not found, nothing is returned with \texttt{no error raised}.
  \item \texttt{\texttt{l3kernelfilesize}}\(\langle\text{file}\rangle\) Returns the size of the \texttt{\langle\text{file}\rangle} in bytes. If the \texttt{\langle\text{file}\rangle} is not found, nothing is returned with \texttt{no error raised}.
  \item \texttt{\texttt{l3kernel.resettimer}}\()\) Resets the timer used by \texttt{l3kernel.elapsedtime}.
  \item \texttt{\texttt{l3kernel.shellescape}}\(\langle\text{cmd}\rangle\) Executes the \texttt{\langle\text{cmd}\rangle} and prints to the log as for pd\TeX.
  \item \texttt{\texttt{l3kernel.strcmp}}\(\langle\text{str one}\rangle, \langle\text{str two}\rangle\) Compares the two strings and returns \texttt{0} to \TeX\ if the two are identical.
\end{itemize}
Part XXXII

The \texttt{l3unicode} package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.
Part XXXIII

The l3text package: text processing

1 l3text documentation

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the \textlangle text\rangle are normalized and become \textbraceleft and \textbraceright, respectively.

1.1 Expanding text

\textexpand:n \langle text\rangle

Takes user input \textlangle text\rangle and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in \texttt{l_text_math_delims_tl} or as the argument to commands listed in \texttt{l_text_math_arg_tl}). Commands which are neither engine- nor \LaTeX{} protected are expanded exhaustively. Any commands listed in \texttt{l_text_expand_exclude_tl}, \texttt{l_text_accents_tl} and \texttt{l_text_letterlike_tl} are excluded from expansion.

\textdeclareexpand equivalent:Nn \langle cmd\rangle \{(replacement)\}

\textdeclareexpand equivalent:cn

Declares that the \textlangle replacement\rangle tokens should be used whenever the \textlangle cmd\rangle (a single token) is encountered. The \textlangle replacement\rangle tokens should be expandable.
1.2 Case changing

\text_lowercase:n \text_uppercase:n \text_titlecase:n \text_titlecase_first:n \text_lowercase:nn \text_uppercase:nn \text_titlecase:nn \text_titlecase_first:nn

Takes user input (\text) first applies \text_expand, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process (at least where they can be represented by the engine as a single token: 8-bit engines may require active characters).

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the (tokens) to uppercase and the rest to lowercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example ij in Dutch which becomes IJ. The titlecase_first variant does not attempt any case changing at all after the first letter has been processed.

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_lowercase:n, \str_uppercase:n and \str_foldcase:n.

Case changing does not take place within math mode material so for example

\text_uppercase:n \{ Some-text-$y = mx + c$ WITH {Braces} \}

becomes

\text_uppercase:n \{ SOME TEXT $y = mx + c$ WITH \{BRACES\} \}

The arguments of commands listed in \l_text_case_exclude_arg_tl are excluded from case changing; the latter are entirely non-textual content (such as labels).

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 Xe\LaTeX{} or Lua\LaTeX{} 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 á can be case-changed using pdf\LaTeX{}. For \pdf\LaTeX{} only the ASCII range is covered as the engine treats input outside of this range as east Asian.

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 lowercasing I-dot and introduced when upper casing i-dotless.

- German (de-alt). An alternative mapping for German in which the lowercase Eszett maps to a großes Eszett. This transformation is available using pdf\LaTeX{} as there is a T1 slot for the result.

- Greek (el). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place.

- Lithuanian (lt). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing 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 titlecased input produces IJ rather than Ij. The output retains two separate letters, thus this transformation is available using pdf\LaTeX{}.
# 1.3 Control variables

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\l_text_accents_tl</td>
<td>Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the \LaTeX2ε package.)</td>
</tr>
<tr>
<td>\l_text_letterlike_tl</td>
<td>Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the \LaTeX2ε package.)</td>
</tr>
<tr>
<td>\l_text_math_arg_tl</td>
<td>Lists commands present in the ⟨text⟩ where the argument of the command should be treated as math mode material. The treatment here is similar to \l_text_math_delims_tl but for a command rather than paired delimiters.</td>
</tr>
<tr>
<td>\l_text_math_delims_tl</td>
<td>Lists pairs of tokens which delimit (in-line) math mode content; such content may be excluded from processing.</td>
</tr>
<tr>
<td>\l_text_expand_exclude_tl</td>
<td>Lists commands which are excluded from expansion.</td>
</tr>
</tbody>
</table>
Part XXXIV

The \texttt{l3legacy} package

Interfaces to legacy concepts

There are a small number of \TeX{} or \LaTeX{} concepts which are not used in \texttt{expl3} code but which need to be manipulated when working as a \LaTeX{} package. To allow these to be integrated cleanly into \texttt{expl3} code, a set of legacy interfaces are provided here.

\begin{verbatim}
\legacy_if_p:n \legacy_if:nTF
\end{verbatim}

Tests if the \LaTeX{} conditional (generated by \texttt{\newif}) if \texttt{true} or \texttt{false} and branches accordingly. The \texttt{name} of the conditional should \textit{omit} the leading \texttt{if}.
Part XXXV

The \texttt{l3candidates} package

Experimental additions to \texttt{l3kernel}

1 Important notice

This module provides a space in which functions can be added to \texttt{l3kernel} (\texttt{expl3}) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in \texttt{l3kernel} in the future.

In contrast to the material in \texttt{l3experimental}, the functions here are all \textit{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 \textbf{not necessarily stable} and we may have to adjust them!

2 Additions to \texttt{l3box}

2.1 Viewing part of a box

\begin{verbatim}
\box_clip:N \box_clip:c
\box_gclip:N \box_gclip:c
\end{verbatim}

Updated: 2019-01-23

\begin{verbatim}
\box_clip:N \langle box \rangle
\end{verbatim}

Clips the \langle box \rangle in the output so that only material inside the bounding box is displayed in the output. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the clipping is applied.

These functions require the \LaTeXX native drivers: they do not work with the \LaTeX2ε graphics drivers!

\textbf{T\textsc{p}X}hackers 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_set_trim:Nnnnn \box_set_trim:cnmnn \box_gset_trim:Nnnnn \box_gset_trim:cnmnn

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 outside of the bounding box is still displayed in the output unless \box_clip:N is subsequently applied. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the trim operation is applied. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

\box_set_viewport:Nnnnn \box_set_viewport:cnmnn \box_gset_viewport:Nnnnn \box_gset_viewport:cnmnn

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 outside of the bounding box is still displayed in the output unless \box_clip:N is subsequently applied. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the viewport operation is applied.

3 Additions to l3expan

\exp_args_generate:n

\exp_args_generate:n {⟨variant argument specifiers⟩}

Defines \exp_args:N⟨variant⟩ functions for each ⟨variant⟩ given in the comma list {⟨variant argument specifiers⟩}. Each ⟨variant⟩ should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the ⟨variant⟩. This is only useful for cases where \cs_generate_variant:Nn is not applicable.

4 Additions to l3fp

\fp_if_nan:n \fp_if_nan:n \fp_if_nan:n \fp_if_nan:n

\fp_if_nan:n ⟨⟨fpexpr⟩⟩

Evaluates the ⟨fpexpr⟩ and tests whether the result is exactly NaN. The test returns false for any other result, even a tuple containing NaN.

5 Additions to l3file

\iow_allow_break:

\iow_allow_break:

In the first argument of \iow_wrap:nnnN (for instance in messages), inserts a break-point that allows a line break. In other words this is a zero-width breaking space.
Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the \langle token list variable \rangle. Tokenization occurs as described for \ior_get:NN or \ior_str_get:NN, respectively. When the \langle prompt \rangle is empty, \TeX{} will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. \iorw_term:n. Where the \langle prompt \rangle is given, it will appear in the terminal followed by an =, e.g.

\begin{verbatim}
prompt=
\end{verbatim}

Opens the pseudo-file created by the output of the \langle shell command \rangle for reading using \langle stream \rangle as the control sequence for access. If the \langle stream \rangle was already open it is closed before the new operation begins. The \langle stream \rangle is available for access immediately and will remain allocated to \langle shell command \rangle until a \ior_close:N instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \langle shell command \rangle, see \sys_get_shell:nnN(TF).

6 Additions to l3flag

\begin{verbatim}
define \flag_raise_if_clear:n {\langle flag name \rangle}
\end{verbatim}

Ensures the \langle flag \rangle is raised by making its height at least 1, locally.

7 Additions to l3intarray

\begin{verbatim}
define \intarray_gset_rand:Nnn  \langle intarray var \rangle \langle minimum \rangle \langle maximum \rangle
\intarray_gset_rand:cn \langle intarray var \rangle \langle maximum \rangle
\end{verbatim}

Evaluates the integer expressions \langle minimum \rangle and \langle maximum \rangle then sets each entry (independently) of the \langle integer array variable \rangle to a pseudo-random number between the two (with bounds included). If the absolute value of either bound is bigger than \(2^{30} - 1\), an error occurs. Entries are generated in the same way as repeated calls to \int_rand:nn or \int_rand:n respectively, in particular for the second function the \langle minimum \rangle is 1. Assignments are always global. This is not available in older versions of Xe\TeX{}.

7.1 Working with contents of integer arrays

\begin{verbatim}
define \intarray_to_clist:N \langle intarray var \rangle
\end{verbatim}

Converts the \langle intarray \rangle to integer denotations separated by commas. All tokens have category code other. If the \langle intarray \rangle has no entry the result is empty; otherwise the result has one fewer comma than the number of items.
8 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, short-hands such as \{ or \ do not work, 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, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn * \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.

\msg_show_eval:Nn \msg_log_eval:Nn

\msg_show: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 is shown on the terminal and the \TeX{} run is interrupted in a manner similar to \tl_show:n. This is used in conjunction with \msg_show_item:n and similar functions to print complex variable contents completely. If the formatted text does not contain >~ at the start of a line, an additional line >~ will be put at the end. In addition, a final period is added if not present.
Used in the text of messages for \texttt{\msg_show:nnxxxx} to show or log a list of items or key–value pairs. The one-argument functions are used for sequences, clist or token lists and the others for property lists. These functions turn their arguments to strings.

\section{Additions to l3prg}

\verb|\bool_set_inverse:N| \langle \text{boolean} \rangle
Toggles the \langle \text{boolean} \rangle from \texttt{true} to \texttt{false} and conversely: sets it to the inverse of its current value.

\verb|\bool_case_true:n| \langle \text{boolexpr case} \rangle \{ \langle \text{code case} \rangle \} \langle \text{true code} \rangle
\verb|\bool_case_false:n| \langle \text{boolexpr case} \rangle \{ \langle \text{code case} \rangle \} \langle \text{false code} \rangle
Evaluates in turn each of the \langle \text{boolean expression cases} \rangle until the first one that evaluates to \texttt{true} or to \texttt{false}, for \verb|\bool_case_true:n| and \verb|\bool_case_false:n|, respectively. The \langle \text{code} \rangle associated to this first case is left in the input stream, followed by the \langle \text{true code} \rangle, and other cases are discarded. If none of the cases match then only the \langle \text{false code} \rangle is inserted. The functions \verb|\bool_case_true:n| and \verb|\bool_case_false:n|, which do nothing if there is no match, are also available. For example

\begin{verbatim}
\bool_case_true:nF
{ \dim_compare_p:n { \l_mypkg_wd_dim <= 10pt } } \{ Fits \}
{ \int_compare_p:n { \l_mypkg_total_int >= 10 } } \{ Many \}
{ \_mypkg_special_bool } \{ Special \}
{ No idea! }
\end{verbatim}

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if \ldots elseif \ldots elseif \ldots else \ldots”.

264
10 Additions to \texttt{l3prop}

\prop_rand_key_value:N \prop_rand_key_value:c

Selects a pseudo-random key–value pair from the \texttt{property list} and returns \{⟨key⟩\} and \{⟨value⟩\}. If the \texttt{property list} is empty the result is empty. This is not available in older versions of Xe\TeX.

\textbf{\texttt{\TeX}hackers note:} The result is returned within the \texttt{\unexpanded} primitive (\exp_not:n), which means that the \langle value \rangle does not expand further when appearing in an \texttt{x}-type argument expansion.

11 Additions to \texttt{l3seq}

\seq_mapthread_function:NNN \star \seq_mapthread_function:NNN (seq_1) (seq_2) (function)

\seq_set_filter:NNn \seq_gset_filter:NNn

\seq_set_map:NNn \seq_gset_map:NNn

Applies \langle function \rangle to every pair of items \langle seq_1-item \rangle–\langle seq_2-item \rangle from the two sequences, returning items from both sequences from left to right. The \langle function \rangle receives two \texttt{n}-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (\textit{i.e.} whichever sequence has fewer items determines how many iterations occur).

\textbf{\texttt{\TeX}hackers note:} Contrarily to other mapping functions, \texttt{\seq_map_break:} cannot be used in this function, and would lead to low-level \TeX errors.

As such, the code in \langle inline function \rangle should be expandable.

\textbf{\texttt{\TeX}hackers note:} Contrarily to other mapping functions, \texttt{\seq_map_break:} cannot be used in this function, and would lead to low-level \TeX errors.
Sets the \( \text{seq} \) equal to a sequence whose items are obtained by \( \times \)-expanding \( \langle \text{loop code} \rangle \) \( \langle \text{function} \rangle \). This expansion must result in successive calls to the \( \langle \text{function} \rangle \) with no nonexpandable tokens in between. More precisely the \( \langle \text{function} \rangle \) is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The \( \langle \text{loop code} \rangle \) must be expandable; it can be for example \( \\texttt{tl_map_function:NN} \) \( \langle \text{tl var} \rangle \) or \( \\texttt{clist_map_function:nN} \) \{\langle clist \rangle \} or \( \\texttt{int_step_function:nnnN} \) \{\langle initial value \rangle \} \{\langle step \rangle \} \{\langle final value \rangle \}.

Sets the \( \text{seq} \) equal to a sequence whose items are obtained by \( \times \)-expanding \( \langle \text{loop code} \rangle \) applied to a \( \langle \text{function} \rangle \) derived from the \( \langle \text{inline code} \rangle \). A \( \langle \text{function} \rangle \) is defined, that takes one argument, \( \times \)-expands the \( \langle \text{inline code} \rangle \) with that argument as \#1, then adds appropriate separators to turn the result into an item of the sequence. The \( \times \)-expansion of \( \langle \text{loop code} \rangle \) \( \langle \text{function} \rangle \) must result in successive calls to the \( \langle \text{function} \rangle \) with no nonexpandable tokens in between. The \( \langle \text{loop code} \rangle \) must be expandable; it can be for example \( \\texttt{tl_map_function:NN} \) \( \langle \text{tl var} \rangle \) or \( \\texttt{clist_map_function:nN} \) \{\langle clist \rangle \} or \( \\texttt{int_step_function:nnnN} \) \{\langle initial value \rangle \} \{\langle step \rangle \} \{\langle final value \rangle \}, but not the analogous “inline” mappings.

Applies \( \langle \text{function} \rangle \) to every entry in the \( \langle \text{sequence variable} \rangle \). The \( \langle \text{function} \rangle \) should have signature :nn. It receives two arguments for each iteration: the \( \langle \text{index} \rangle \) (namely 1 for the first entry, then 2 and so on) and the \( \langle \text{item} \rangle \).

Applies \( \langle \text{inline function} \rangle \) to every entry in the \( \langle \text{sequence variable} \rangle \). The \( \langle \text{inline function} \rangle \) should consist of code which receives the \( \langle \text{index} \rangle \) (namely 1 for the first entry, then 2 and so on) as \#1 and the \( \langle \text{item} \rangle \) as \#2.
12 Additions to \texttt{l3sys}

\begin{verbatim}
\c_sys_engine_version_str
New: 2018-05-02
\end{verbatim}

The version string of the current engine, in the same form as given in the banner issued when running a job. For \texttt{pdf\TeX} and \texttt{Lua\TeX} this is of the form

\(\langle \text{major} \rangle.\langle \text{minor} \rangle.\langle \text{revision} \rangle\)

For \texttt{Xe\TeX}, the form is

\(\langle \text{major} \rangle.\langle \text{minor} \rangle\)

For \texttt{p\TeX} and \texttt{up\TeX}, only releases since \TeX{} Live 2018 make the data available, and the form is more complex, as it comprises the \texttt{p\TeX} version, the \texttt{up\TeX} version and the \texttt{e-p\TeX} version.

\(p\langle \text{major} \rangle.\langle \text{revision} \rangle-u\langle \text{major} \rangle.\langle \text{minor} \rangle-\langle \text{e-p\TeX} \rangle\)

where the \texttt{u} part is only present for \texttt{up\TeX}.

\begin{verbatim}
\sys_if_rand_exist_p: *
\sys_if_rand_exist:T/F *
New: 2017-05-27
\end{verbatim}

Tests if the engine has a pseudo-random number generator. Currently this is the case in \texttt{pdf\TeX}, \texttt{Lua\TeX}, \texttt{p\TeX}, \texttt{up\TeX} and recent releases of \texttt{Xe\TeX}.

\begin{verbatim}
\sys_if_rand_exist_p:
\sys_if_rand_exist:T/F {\langle \text{true code} \rangle} {\langle \text{false code} \rangle}
\end{verbatim}
13 Additions to \l3tl

\tl_range_braced:Nnn \tl_range_braced:nnn \tl_range_unbraced:Nnn \tl_range_unbraced:nnn

Leaves in the input stream the items from the \textit{start index} to the \textit{end index} inclusive, using the same indexing as \texttt{\tl_range:nnn}. Spaces are ignored. Regardless of whether items appear with or without braces in the \textit{token list}, the \texttt{\tl_range_braced:nnn} function wraps each item in braces, while \texttt{\tl_range_unbraced:nnn} does not (overall it removes an outer set of braces). For instance,

\begin{verbatim}
\io_term:x { \tl_range_braced:nnn { abcd-{e{}}f } { 2 } { 5 } }
\io_term:x { \tl_range_braced:nnn { abcd-{e{}}f } { -4 } { -1 } }
\io_term:x { \tl_range_braced:nnn { abcd-{e{}}f } { -2 } { -1 } }
\io_term:x { \tl_range_braced:nnn { abcd-{e{}}f } { 0 } { -1 } }
\end{verbatim}

prints \{b}{c}{d}{e{}}, \{c}{d}{e{}}{f}, \{e{}}{f}, and an empty line to the terminal, while

\begin{verbatim}
\io_term:x { \tl_range_unbraced:nnn { abcd-{e{}}f } { 2 } { 5 } }
\io_term:x { \tl_range_unbraced:nnn { abcd-{e{}}f } { -4 } { -1 } }
\io_term:x { \tl_range_unbraced:nnn { abcd-{e{}}f } { -2 } { -1 } }
\io_term:x { \tl_range_unbraced:nnn { abcd-{e{}}f } { 0 } { -1 } }
\end{verbatim}

prints bcde{}, cde{}f, e{}f, and an empty line to the terminal. Because braces are removed, the result of \texttt{\tl_range_unbraced:nnn} may have a different number of items as for \texttt{\tl_range:nnn} or \texttt{\tl_range_braced:nnn}. In cases where preserving spaces is important, consider the slower function \texttt{\tl_range:nnn}.

\textit{TexHackers note:} The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \textit{item} does not expand further when appearing in an \texttt{x}-type argument expansion.

\begin{verbatim}
\tl_build_begin:N \tl_build_gbegin:N
\tl_build_clear:N \tl_build_gclear:N
\end{verbatim}

Clears the \texttt{\tl var} and sets it up to support other \texttt{\tl_build…} functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard \l3tl functions. Until \texttt{\tl_build_end:N \tl var} is called, applying any function from \l3tl other than \texttt{\tl_build…} will lead to incorrect results. The \texttt{begin} and \texttt{gbegin} functions must be used for local and global \texttt{(tl var)} respectively.

Clears the \texttt{\tl var} and sets it up to support other \texttt{\tl_build…} functions. The \texttt{clear} and \texttt{gclear} functions must be used for local and global \texttt{(tl var)} respectively.

268
\tl_build_put_left:Nn \tl_build_put_left:Nx \tl_build_gput_left:Nn \tl_build_gput_left:Nx \tl_build_put_right:Nn \tl_build_put_right:Nx \tl_build_gput_right:Nn \tl_build_gput_right:Nx

New: 2018-04-01

\tl_build_get:N \tl_build_get:NN

New: 2018-04-01

\tl_build_put_left:Nn \tl_build_put_right:Nn \tl_build_gput_left:Nn \tl_build_gput_right:Nn

Add \langle tokens\rangle to the left or right side of the current contents of \langle tl \ var \rangle. The \langle tl \ var \rangle must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The \textit{put} and \textit{gput} functions must be used for local and global \langle tl \ var \rangle respectively. The \textit{right} functions are about twice faster than the \textit{left} functions.

\newline

\tl_build_get:N \tl_build_get:NN

Stores the contents of the \langle tl \ var_1 \rangle in the \langle tl \ var_2 \rangle. The \langle tl \ var_1 \rangle must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The \langle tl \ var_2 \rangle is a “normal” token list variable, assigned locally using \tl_set:Nn.

\newline

\tl_build_end:N \tl_build_gend:N

New: 2018-04-01

14 Additions to \texttt{l3token}

Token list containing one character with category code 13, (“active”), and character code 32 (space).

\newline

\char_to_utfviii_bytes:n \star

New: 2020-01-09

Converts the (Unicode) \langle codepoint \rangle to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups \#1 and \#2 filled and \#3 and \#4 empty.

\newline

\char_to_nfd:N \star

New: 2020-01-02

Converts the \langle char \rangle to the Unicode Normalization Form Canonical Decomposition. The category code of the generated character is the same as the \langle char \rangle. With 8-bit engines, no change is made to the character.
Collects and removes tokens from the input stream until finding a token that does not match the \(\textit{test token}\) (as defined by the test \texttt{\small token_if_eq_catcode:NNTF} or \texttt{\small token_if_eq_charcode:NNTF} or \texttt{\small token_if_eq_meaning:NNTF}). The collected tokens are passed to the \(\textit{inline code}\) as \#1. When begin-group or end-group tokens (usually \{ or \}) are collected they are replaced by implicit \texttt{\small c_group_begin_token} and \texttt{\small c_group_end_token}, and when spaces (including \texttt{\small c_space_token}) are collected they are replaced by explicit spaces.

For example the following code prints “Hello” to the terminal and leave “, world!” in the input stream.

\begin{verbatim}
\peek_catcode_collect_inline:Nn \{ \iow_term:n {#1} \} Hello,~world!
\end{verbatim}

Another example is that the following code tests if the next token is \*, ignoring intervening spaces, but putting them back using \#1 if there is no \*.

\begin{verbatim}
\peek_meaning_collect_inline:Nn \c_space_token \{ \peek_charcode:NTF \* \{ star \} \{ no-star \#1 \} \}
\end{verbatim}
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>211</th>
<th>210</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kk</td>
<td>211</td>
<td>B</td>
</tr>
<tr>
<td>*</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>210, 211</td>
<td>264</td>
</tr>
<tr>
<td>-</td>
<td>210, 211</td>
<td>264</td>
</tr>
<tr>
<td>/</td>
<td>211</td>
<td>264</td>
</tr>
<tr>
<td>?</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:N</td>
<td>37</td>
<td>110</td>
</tr>
<tr>
<td>:V</td>
<td>37</td>
<td>110</td>
</tr>
<tr>
<td>:V_unbraced</td>
<td>37</td>
<td>111</td>
</tr>
<tr>
<td>:c</td>
<td>37</td>
<td>111</td>
</tr>
<tr>
<td>:t</td>
<td>37</td>
<td>111</td>
</tr>
<tr>
<td>:e</td>
<td>37</td>
<td>111</td>
</tr>
<tr>
<td>:e_unbraced</td>
<td>37</td>
<td>185</td>
</tr>
<tr>
<td>:error</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td>:f</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:f_unbraced</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:n</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:o_unbraced</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:p</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:v</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:v_unbraced</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:x</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>:x_unbraced</td>
<td>37</td>
<td>107</td>
</tr>
<tr>
<td>&lt;</td>
<td>210</td>
<td>109</td>
</tr>
<tr>
<td>=</td>
<td>210</td>
<td>109</td>
</tr>
<tr>
<td>&gt;</td>
<td>210</td>
<td>109</td>
</tr>
<tr>
<td>?</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>? commands:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>211</td>
<td>109</td>
</tr>
<tr>
<td>\</td>
<td>210</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>211</td>
</tr>
<tr>
<td>acos</td>
<td>213</td>
</tr>
<tr>
<td>acosd</td>
<td>213</td>
</tr>
<tr>
<td>acot</td>
<td>214</td>
</tr>
<tr>
<td>acotd</td>
<td>214</td>
</tr>
<tr>
<td>acsc</td>
<td>213</td>
</tr>
<tr>
<td>acscd</td>
<td>213</td>
</tr>
<tr>
<td>asec</td>
<td>213</td>
</tr>
<tr>
<td>asecd</td>
<td>213</td>
</tr>
<tr>
<td>asind</td>
<td>213</td>
</tr>
<tr>
<td>asind</td>
<td>213</td>
</tr>
<tr>
<td>atand</td>
<td>214</td>
</tr>
</tbody>
</table>

271
\ExplFileDescription .............................. 7
\ExplFileName ................................. 7
\ExplFileVersion .............................. 7
\ExplSyntaxOn .................................. 4, 7, 7, 118
\ExplSyntaxOff .................................. 4, 7, 7, 118

F

fact ................................. 211
false .................................. 216

fi commands:
\fi: ........................ 23, 100, 100, 101,
                      105, 112, 113, 163, 182, 245, 245, 245

file commands:
\file_compare_timestamp:nnNn ... 166
\file_compare_timestamp:nnNnTF ... 166
\file_compare_timestamp_p:nnNn ... 166
\g_file_curr_dir_str ............ 163
\g_file_curr_ext_str ............ 163
\g_file_curr_name_str .......... 163
\file_full_name:n ............... 164
\file_get:nnN ....................... 164
\file_get:nnNTF .................... 164
\file_get_full_name:nnN .......... 164
\file_get_full_name:nnNTF ........ 164
\file_get_hex_dump:nnN .......... 165
\file_get_hex_dump:nnnNTF ........ 165
\file_get_hex_dump:p:nnN ........ 165
\file_get_mdfive_hash:nnN ........ 165
\file_get_mdfive_hash:nnNTF ..... 165
\file_get_size:nn .................. 166
\file_get_size:nnNTF .............. 166
\file_get_timestamp:nn ........... 166
\file_get_timestamp:nnNTF ........ 166
\file_hex_dump:n ................. 165, 165
\file_hex_dump:nn ................. 165, 165
\file_if_exist:nnTF ............... 164, 164, 164, 166
\file_if_exist:nnTF ............... 164, 164, 164, 166
\file_input:n ....................... 167
\file_input:stop ................... 167
\file_log_list ...................... 167
\file_mdfive_hash:nn .............. 165, 165
\file_parse_full_name:nnNNN ...... 165
\file_path_include:n .............. 167
\l_file_search_path_seq ........... 164, 164, 165, 165, 166, 166
\file_show_list .................... 167
\file_size:n ....................... 166, 166
\file_timestamp:n ................. 166, 166

flag commands:
\flag_clear:n ...................... 102, 102
\flag_clear_new:n .................. 102
\flag_height:n ..................... 103
\flag_new:n ....................... 102, 102
\flag_raise:n ...................... 103
\flag_raise_if_clear:n ............ 262
\flag_show:n ...................... 102

flag fp commands:
\flag_fp_division_by_zero .......... 207
\flag_fp_invalidate_operation ...... 207
\flag_fp_overflow ................... 207
\flag_fp_underflow ................ 207

floor ............................... 212

fp commands:
\c_e_fp ................................ 206, 208
\fp_abs:n ................................ 211, 216
\fp_add:NN ................................ 200
\fp_compare:nnNTF .................... 200
\fp_compare:nnp ....................... 203
\fp_compare:p ......................... 203
\fp_const:N ................................ 199
\fp_do_until:nn ....................... 204
\fp_do_until:nNN ...................... 203
\fp_do_while:nn ....................... 204
\fp_eval:n ................................ 200
\fp_format:n ......................... 217
\fp_gadd:NN ................................ 200
\fp_gset:N ............................. 186
\fp_gset:NN ................................ 200
\fp_gset_eq:NN ......................... 200
\fp_gsub:NN ................................ 200
\fp_gzero:N ................................ 199
\fp_gzero_new:N ....................... 200
\fp_if_exist:nnTF ..................... 202
\fp_if_exist:p ......................... 202
\fp_if_nan:n ......................... 261
\fp_log:N ................................ 207
\fp_max:nn ............................. 216
\fp_min:nn ............................. 216

277
group commands:
\group_align_safe_begin: \ldots 113
\group_align_safe_end: \ldots 113
\group_begin: \ldots 9
\group_end: \ldots 9, 9
\c_group_begin_token \ldots 133, 270
\c_group_end_token \ldots 133, 270
\group_insert_after:N \ldots 9
groups commands:
.groups:n \ldots 186

hbox commands:
\hbox:n \ldots 238
\hbox_gset:Nn \ldots 239
\hbox_gset:Nw \ldots 239
\hbox_gset_end: \ldots 239
\hbox_set_to_fd:Nnn \ldots 239
\hbox_set_to_fd:Nw \ldots 239
\hbox_overlap_left:n \ldots 239
\hbox_overlap_right:n \ldots 239
\hbox_set:Nn \ldots 239, 239
\hbox_set:Nw \ldots 239, 239
\hbox_set_end: \ldots 239, 239
\hbox_set_to_fd:Nnn \ldots 239, 239
\hbox_set_to_fd:Nw \ldots 239, 239
\hbox_set_to_fd:Nw \ldots 239
\hbox_to_fd:n \ldots 239
\hbox_to_zero:n \ldots 239
\hbox_unpack:N \ldots 242
\hbox_unpack_drop:N \ldots 242

hcoffin commands:
\hcoffin_gset:Nn \ldots 246
\hcoffin_gset:Nw \ldots 247
\hcoffin_gset_end: \ldots 247
\hcoffin_set:N \ldots 246
\hcoffin_set:Nw \ldots 247
\hcoffin_set_end: \ldots 247

if commands:
\if:w \ldots 23, 128
\if:bool:N \ldots 112, 112
\if:box_empty:N \ldots 245
\if:case:w \ldots 100
\if:catcode:w \ldots 23
\if:charcode:w \ldots 23, 128
\if:cs_exist:N \ldots 23
\if:cs_exist:w \ldots 23
\if:dim:w \ldots 182
\if:eof:w \ldots 163
\if:false: \ldots 23, 106
\if:hbox:N \ldots 245
\if:int_compare:w \ldots 22, 100
\if:int:odd:w \ldots 101
\prg_set_protected_conditional:Npn

prop commands:
\c_empty_prop ................. 149
\prop_clear:N .......... 143, 143
\prop_clear_new:N .......... 143
\prop_const_from_keyval:Nn .. 144
\prop_count:N .......... 145
\prop_gclear:N .......... 143
\prop_gclear_new:N .......... 143
\prop_get:Nn .......... 143, 144, 145
\prop_get:NnTF .......... 144, 146, 146
\prop_gpop:Nn .......... 144, 145
\prop_gpop:NnTF .......... 145, 146
\prop_gput:N .......... 144, 147
\prop_gput_if_new:Nn .......... 144
\prop_gremove:N .......... 145
\prop_gset_eq:NN .......... 143
\prop_gset_from_keyval:Nn .. 143
\prop_if_empty:NTF .......... 145
\prop_if_empty:p:N .......... 145
\prop_if_exist:NTF .......... 145
\prop_if_exist:p:N .......... 145
\prop_if_in:NnTF .......... 146
\prop_if_in:p:N .......... 146
\prop_item:Nn .......... 145, 147
\prop_log:N .......... 148
\prop_map_break: ............ 147
\prop_map_break:n .......... 148
\prop_map_function:NN .......... 147, 147, 264
\prop_map_inline:Nn .......... 147
\prop_map_tokens:Nn .......... 147, 147
\prop_new:N .......... 143, 143
\prop_pop:Nn .......... 144
\prop_pop:NnTF .......... 144, 146
\prop_put:N .......... 187
\prop_put:Nn .......... 144
\prop_put_if_new:Nn .......... 144
\prop_rand_key_value:N .......... 265
\prop_remove:N .......... 145
\prop_set_eq:NN .......... 143
\prop_set_from_keyval:Nn .. 143
\prop_show:N .......... 148
\g_tempa_prop .......... 148
\l_tempa_prop .......... 148
\g_tempb_prop .......... 148
\l_tempb_prop .......... 148
\ProvidesExplFile ............ 7
\ProvidesExplPackage ........ 7
\regenics

Q
\q_mark ................. 71, 118
\q_nil ............. 21, 21, 53, 71, 71
\q_no_value ........ 70,
71, 71, 71, 71, 77, 77, 77, 77, 83, 83, 116, 125, 144, 144, 144, 145,
158, 158, 164, 164, 165, 165, 166, 166
\quark_if_nil:NTF ............ 71
\quark_if_nil:nTF ........ 71
\quark_if_nil:p:N .......... 71
\quark_if_nil:p:n .......... 71
\quark_if_no_value:NTF ........ 71
\quark_if_no_value:nTF ........ 71
\quark_if_no_value:p:N ........ 71
\quark_if_no_value:p:n .......... 71
\quark_if recursion_tail_-
break:NN ............... 72
\quark_if recursion_tail_-
break:NN ............... 72
\quark_if recursion_tail_stop:N ... 72
\quark_if recursion_tail_stop:n ... 72
\quark_if recursion_tail_stop_- 
do:Nn ............... 72
\quark_if recursion_tail_stop_- 
do:nn ............... 72
\quark_new:N ............ 70
\q_recursion_stop ........ 21, 21, 72, 72, 72, 72, 72, 73
\q_recursion_tail ........ 74, 74
\q_no_value ........ 230
\q_recursion_stop ........ 230
\s_stop ............ 74, 74

R
rand ................. 215
randint .............. 215
regex commands:
\c_foo_regex ................. 222
\regex_gset:Nn ........ 222, 222
\regex_const:Nn ........ 230
\regex_count:Nn ........ 230
\regex_count:nN ........ 230
\regex_extract_all:Nn .......... 230
\regex_extract_all:nN .......... 223, 230
\regex_extract_all:nNTF .......... 230
\regex_extract_all:nnN .......... 230
\regex_extract_all:nnNTF .......... 230
\regex_extract_once:nn .......... 230
\regex_extract:nn .......... 230
\regex_extract:nnTF .......... 229
\regex_extract:nnTF .......... 229
\sys_gset_rand_seed:n
\sys_get_shell:nnNTF
\sys_get_shell:nnN(TF)
\sys_get_shell:nnN
\c_sys_engine_version_str
\c_sys_engine_str
\c_sys_day_int
\l_tmpb_str
\g_tmpb_str
\l_tmpa_str
\g_tmpa_str
\c_tilde_str
\str_use:N
\str_range:nnn
\str_range:Nnn
\str_put_right:Nn
\str_put_left:Nn
\str_new:N
\str_map_variable:nNn
\str_map_variable:NNn
\str_map_inline:Nn
\str_map_function:NN
\str_map_break:n
\str_map_break:
\c_sys_backend_str
\str_uppercase:n
\str_tail_ignore_spaces:n
\str_show:n
\str_set_eq:NN
\str_set_convert:NnnnTF
\str_set_convert:Nnnn
\str_replace_once:Nnn
\str_remove_once:Nn
\str_remove_all:Nn
\str_log:N
\str_if_exist_p:N
\str_item_ignore_spaces:nn
\str_item:nn
\str_item:Nn
\str_if_in:nnTF
\str_if_in:NnTF
\c_sys_hour_int
\char
\box
\@secondoftwo
\@gobbbletwo
\@firstoftwo
\@firstofone
\c_sys_minute_int
\c_sys_month_int
\c_sys_jobname_str
\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted_p:
\sys_if_shell:TF
\sys_if_rand_exist_p:
\sys_if_rand_exist:TF
\sys_if_platform_windows:TF
\sys_if_platform_unix_p:
\sys_if_output_pdf_p:
\sys_if_output_dvi_p:
\sys_if_engine_xetex_p:
\sys_if_engine_uptex_p:
\sys_if_engine_ptex_p:
\sys_if_engine_pdftex_p:
\sys_if_engine_luatex_p:
\sys_if_engine_luatex:TF
\sys_load_backend:n
\sys_if_debug:p
\sys_if_deprecation:
\c_sys_minute_int
\c_sys_month_int
\c_sys_output_str
\c_sys_platform_str
\sys_rand_seed:...
\sys_shell_shipout:n
\sys_shell_now:n
\c_sys_shell_escape_int
\c_sys_platform_str
\c_sys_jobname_str
tl commands:
\c_empty_tl 53 \tl_if_empty_p:N 42
\c_my_tl 222, 228 \tl_if_empty_p:n 42
\c_novalue_tl 42, 53 \tl_if_eq:nT(nTF) 121, 121
\c_space_tl 53 \tl_if_eq:NnTF 42, 43, 70
\tl_analysis_map_inline:Nn 221 \tl_if_eq:NnTF 42, 79, 79
\tl_analysis_map_inline:nn 221 \tl_if_eq_p:NN 42
\tl_analysis_show:N 221 \tl_if_exist:NTF 39
\tl_analysis_show:n 221 \tl_if_exist_p:N 39
\tl_build_begin:N 268, 269, 269, 269 \tl_if_head_eq_catcode:nNTF 50
\tl_build_clear:N 268 \tl_if_head_eq_catcode:p:nN 50
\tl_build_end:N 268, 269, 269, 269 \tl_if_head_eq_charcode:nNTF 50
\tl_build_gbegin:N 268 \tl_if_head_eq_charcode:p:nN 50
\tl_build_gclear:N 268 \tl_if_head_is_group:NTF 50
\tl_build_gend:N 269 \tl_if_head_is_group:p:n 50
\tl_build_get:N 269 \tl_if_head_is_N_type:nTF 50
\tl_build_get:NN 269 \tl_if_head_is_space:nTF 50
\tl_build_gput_left:Nn 269 \tl_if_head_is_space:p:n 50
\tl_build_gput_right:Nn 269 \tl_if_in:NnTF 42
\tl_build_put_left:Nn 269 \tl_if_in:nTF 42
\tl_build_put_right:Nn 269 \tl_if_noempty:nTF 42
\tl_case:Nn 49 \tl_if_noempty_p:n 42
\tl_case:NnTF 49 \tl_if_noempty_p:nN 42
\tl_clear:N 38, 39 \tl_if_single:nTF 43
\tl_clear_new:N 39 \tl_if_single:nT(p) 43
\tl_concat:NWN 39 \tl_if_single:nTF 43
\tl_count:N 38 \tl_if_single:p:n 43
\tl_count_tokens:n 47 \tl_if_single_token:nTF 43
\tl_count:N 27, 43, 46, 47 \tl_if_single_token:nTF 43
\tl_count:n 27, 43, 46, 47 \tl_if_single_token:nTF 43
\tl_gclear:N 38 \tl_map:function:NN 43
\tl_gclear_new:N 39 \tl_map:break: 45, 233
\tl_gconcat:NNN 39 \tl_map:break:n 45, 45
\tl_gclear_new:Nnn 39 \tl_gcremove_all:Nn 40
\tl_gconcat:NNN 39 \tl_ggetmove_all:Nn 40
\tl_gclear:N 38 \tl_ggetmove_one:Nn 40
\tl_gclear_new:N 39 \tl_ggetplace_all:Nnm 40
\tl_gconcat:NWN 39 \tl_ggetplace_one:Nnn 40
\tl_gclear:N 38 \tl_ggetplace_one:Nnn 40
\tl_gconcat:NN 39 \tl_ggetvariable:Nn 44
\tl_gclear:N 38 \tl_ggetvariable:Nn 44
\tl_gconcat:NN 39 \tl_grevers:e:N 47
\tl_gclear:N 38 \tl_gmap:inl ine:Nn 44, 44, 44
\tl_gconcat:NWN 39 \tl_gmap:inl ine:n 44, 44, 72
\tl_gclear:N 38 \tl_gmap:tokens:Nn 44
\tl_gclear_new:N 39 \tl_gmap:tokens:n 44
\tl_gconcat:NN 41 \tl_gset:rescan:Nnn 44
\tl_gclear:N 38 \tl_gset:variable:Nn 44
\tl_gconcat:NN 41 \tl_gset:variable:Nn 44
\tl_gclear:N 38 \tl_gset:N 188
\tl_gclear:N 39, 76 \tl_gset_eq:NN 39
\tl_gconcat:NN 39 \tl_gset_rescan:Nnn 41
\tl_gclear:N 188 \tl_gset:N 48
\tl_gconcat:NN 48 \tl_gset:rescan:Nnn 41
\tl_gclear:N 48 \tl_gset:rescan:Nnn 41
\tl_gconcat:NN 48 \tl_gset:rescan:Nnn 41
\tl_gclear:N 49 \tl_gset:x:N 48
\tl_gconcat:NN 49 \tl_gset:x:N 48
\tl_gclear:N 49 \tl_new:N 38, 39, 133
\tl_gconcat:NN 49 \tl_get:spaces:N 48
\tl_gclear:N 49 \tl_put_left:N 39
\tl_gconcat:NN 49 \tl_put_right:N 39
\tl_gclear:N 49 \tl_rand_item:N 51
\tl_gconcat:NN 49 \tl_rand_item:n 51
\tl_gclear:N 49 \tl_remove:Nn 52, 63, 268
\tl_gconcat:NN 49 \tl_remove:NNn 52, 268
\tl_gclear:N 49 \tl_remove:NNnn 52, 268