The \LaTeX3 Sources
The \LaTeX3 Project
Released 2020-05-15

Abstract

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

The \TeX\ and \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\ \\tex. In time, a \LaTeX3\ format will be produced based on this code. This allows the code to be used in \LaTeX2\ packages now while a stand-alone \LaTeX3\ is developed.

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

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

---

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

I Introduction to expl3 and this document 1

1 Naming functions and variables 1
   1.1 Terminological inexactitude 3

2 Documentation conventions 4

3 Formal language conventions which apply generally 5

4 \TeX concepts not supported by \LaTeX3 6

II The l3bootstrap package: Bootstrap code 7

1 Using the \LaTeX3 modules 7

III The l3names package: Namespace for primitives 8

1 Setting up the \LaTeX3 programming language 8

IV The l3basics package: Basic definitions 9

1 No operation functions 9

2 Grouping material 9

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

4 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 l3expan package: Argument expansion
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 l3quark package: Quarks
1. Quarks 38
2. Defining quarks 38
3. Quark tests 39
4. Recursion 39
5. An example of recursion with quarks 40
6. Scan marks 41

### VII The l3tl package: Token lists
1. Creating and initialising token list variables 43
2. Adding data to token list variables 44
3. Modifying token list variables 45
4. Reassigning token list category codes 45
5. Token list conditionals 46
6. Mapping to token lists 48
7. Using token lists 51
VIII The l3str package: Strings
1 Building strings
2 Adding data to string variables
3 Modifying string variables
4 String conditionals
5 Mapping to strings
6 Working with the content of strings
7 String manipulation
8 Viewing strings
9 Constant token lists
10 Scratch strings

IX The l3str-convert package: string encoding conversions
1 Encoding and escaping schemes
2 Conversion functions
3 Creating 8-bit mappings
4 Possibilities, and things to do

X The l3seq package: Sequences and stacks
1 Creating and initialising sequences
2 Appending data to sequences
3 Recovering items from sequences
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Recovering values from sequences with branching</td>
<td>78</td>
</tr>
<tr>
<td>5</td>
<td>Modifying sequences</td>
<td>79</td>
</tr>
<tr>
<td>6</td>
<td>Sequence conditionals</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>Mapping to sequences</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>Using the content of sequences directly</td>
<td>82</td>
</tr>
<tr>
<td>9</td>
<td>Sequences as stacks</td>
<td>83</td>
</tr>
<tr>
<td>10</td>
<td>Sequences as sets</td>
<td>84</td>
</tr>
<tr>
<td>11</td>
<td>Constant and scratch sequences</td>
<td>85</td>
</tr>
<tr>
<td>12</td>
<td>Viewing sequences</td>
<td>86</td>
</tr>
<tr>
<td>XI</td>
<td>The \texttt{l3int} package: Integers</td>
<td>87</td>
</tr>
<tr>
<td>1</td>
<td>Integer expressions</td>
<td>88</td>
</tr>
<tr>
<td>2</td>
<td>Creating and initialising integers</td>
<td>89</td>
</tr>
<tr>
<td>3</td>
<td>Setting and incrementing integers</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>Using integers</td>
<td>91</td>
</tr>
<tr>
<td>5</td>
<td>Integer expression conditionals</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>Integer expression loops</td>
<td>93</td>
</tr>
<tr>
<td>7</td>
<td>Integer step functions</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>Formatting integers</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>Converting from other formats to integers</td>
<td>97</td>
</tr>
<tr>
<td>10</td>
<td>Random integers</td>
<td>98</td>
</tr>
<tr>
<td>11</td>
<td>Viewing integers</td>
<td>99</td>
</tr>
<tr>
<td>12</td>
<td>Constant integers</td>
<td>99</td>
</tr>
<tr>
<td>13</td>
<td>Scratch integers</td>
<td>99</td>
</tr>
<tr>
<td>13.1</td>
<td>Direct number expansion</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>Primitive conditionals</td>
<td>100</td>
</tr>
<tr>
<td>XII</td>
<td>The \texttt{l3flag} package: Expandable flags</td>
<td>102</td>
</tr>
<tr>
<td>1</td>
<td>Setting up flags</td>
<td>102</td>
</tr>
</tbody>
</table>
XIII The \texttt{l3prg} package: Control structures
\begin{itemize}
  \item 1 Defining a set of conditional functions
  \item 2 The boolean data type
  \item 3 Boolean expressions
  \item 4 Logical loops
  \item 5 Producing multiple copies
  \item 6 Detecting \TeX’s mode
  \item 7 Primitive conditionals
  \item 8 Nestable recursions and mappings
    \begin{itemize}
    \item 8.1 Simple mappings
    \end{itemize}
  \item 9 Internal programming functions
\end{itemize}

XIV The \texttt{l3sys} package: System/runtime functions
\begin{itemize}
  \item 1 The name of the job
  \item 2 Date and time
  \item 3 Engine
  \item 4 Output format
  \item 5 Platform
  \item 6 Random numbers
  \item 7 Access to the shell
    \begin{itemize}
    \item 7.1 Loading configuration data
    \item 7.2 Final settings
    \end{itemize}
\end{itemize}

XV The \texttt{l3clist} package: Comma separated lists
\begin{itemize}
  \item 1 Creating and initialising comma lists
  \item 2 Adding data to comma lists
  \item 3 Modifying comma lists
  \item 4 Comma list conditionals
\end{itemize}
4 Floating point conditionals 202
5 Floating point expression loops 203
6 Some useful constants, and scratch variables 205
7 Floating point exceptions 206
8 Viewing floating points 207
9 Floating point expressions 208
  9.1 Input of floating point numbers 208
  9.2 Precedence of operators 209
  9.3 Operations 209
10 Disclaimer and roadmap 216

XXIV The \texttt{l3fparray} package: fast global floating point arrays 219
  1 \texttt{l3fparray} documentation 219

XXV The \texttt{l3sort} package: Sorting functions 220
  1 Controlling sorting 220

XXVI The \texttt{l3tl-analysis} package: Analysing token lists 221
  1 \texttt{l3tl-analysis} documentation 221

XXVII The \texttt{l3regex} package: Regular expressions in \TeX{} 222
  1 Syntax of regular expressions 222
  2 Syntax of the replacement text 227
  3 Pre-compiling regular expressions 229
  4 Matching 229
  5 Submatch extraction 230
  6 Replacement 231
  7 Constants and variables 231
  8 Bugs, misfeatures, future work, and other possibilities 232

XXVIII The \texttt{l3box} package: Boxes 235
1 \texttt{l3bootstrap} implementation  \hspace{1cm} 270
1.1 Format-specific code ........................................ 270
1.2 The \texttt{pdfstrcmp} primitive in Xe\TeX{} .................. 271
1.3 Loading support Lua code .................................... 271
1.4 Engine requirements ........................................ 272
1.5 Extending allocators ........................................ 274
1.6 Character data ............................................. 274
1.7 The \texttt{I\TeX{}X} code environment ....................... 276

2 \texttt{l3names} implementation  \hspace{1cm} 277
2.1 Deprecated functions ........................................ 301

3 Internal kernel functions  \hspace{1cm} 313

4 Kernel backend functions  \hspace{1cm} 319

5 \texttt{l3basics} implementation  \hspace{1cm} 320
5.1 Renaming some \TeX{} primitives (again) .................... 320
5.2 Defining some constants .................................... 322
5.3 Defining functions ......................................... 323
5.4 Selecting tokens ............................................ 323
5.5 Gobbling tokens from input ................................ 325
5.6 Debugging and patching later definitions .................. 325
5.7 Conditional processing and definitions ..................... 326
5.8 Dissecting a control sequence ............................. 332
5.9 Exist or free ................................................ 334
5.10 Preliminaries for new functions ............................ 336
5.11 Defining new functions ..................................... 337
5.12 Copying definitions ........................................ 339
5.13 Undefining functions ...................................... 339
5.14 Generating parameter text from argument count .......... 340
5.15 Defining functions from a given number of arguments ... 341
5.16 Using the signature to define functions ................... 342
5.17 Checking control sequence equality ........................ 344
5.18 Diagnostic functions ....................................... 344
5.19 Decomposing a macro definition ............................ 346
5.20 Doing nothing functions ................................... 346
5.21 Breaking out of mapping functions ........................ 347
5.22 Starting a paragraph ....................................... 347

6 \texttt{l3expan} implementation  \hspace{1cm} 348
6.1 General expansion ........................................... 348
6.2 Hand-tuned definitions ....................................... 351
6.3 Last-unbraced versions ...................................... 355
6.4 Preventing expansion ........................................ 357
6.5 Controlled expansion ........................................ 358
6.6 Emulating e-type expansion ................................ 358
6.7 Defining function variants ................................. 365
6.8 Definitions with the automated technique .................. 376
10.5.2 Escape methods ........................................ 456
10.6 Encoding definitions ..................................... 458
  10.6.1 UTF-8 support ........................................ 458
  10.6.2 UTF-16 support ....................................... 463
  10.6.3 UTF-32 support ....................................... 468
  10.6.4 ISO 8859 support ..................................... 471

11 l3seq implementation ...................................... 487
  11.1 Allocation and initialisation ............................ 488
  11.2 Appending data to either end ........................... 491
  11.3 Modifying sequences ................................... 492
  11.4 Sequence conditionals ................................ 494
  11.5 Recovering data from sequences ....................... 496
  11.6 Mapping to sequences .................................. 500
  11.7 Using sequences ........................................ 502
  11.8 Sequence stacks ....................................... 503
  11.9 Viewing sequences ..................................... 504
  11.10 Scratch sequences .................................... 505

12 l3int implementation ...................................... 505
  12.1 Integer expressions .................................... 506
  12.2 Creating and initialising integers ..................... 508
  12.3 Setting and incrementing integers ..................... 510
  12.4 Using integers ......................................... 511
  12.5 Integer expression conditionals ....................... 511
  12.6 Integer expression loops ................................ 515
  12.7 Integer step functions .................................. 516
  12.8 Formatting integers .................................... 518
  12.9 Converting from other formats to integers .......... 523
  12.10 Viewing integer .......................................
  12.11 Random integers ..................................... 526
  12.12 Constant integers .................................... 527
  12.13 Scratch integers ..................................... 527
  12.14 Integers for earlier modules ......................... 527

13 l3flag implementation ..................................... 528
  13.1 Non-expandable flag commands ......................... 528
  13.2 Expandable flag commands .............................. 529

14 l3prg implementation ..................................... 530
  14.1 Primitive conditionals .................................. 530
  14.2 Defining a set of conditional functions ............... 530
  14.3 The boolean data type .................................. 530
  14.4 Internal auxiliaries .................................... 531
  14.5 Boolean expressions .................................... 533
  14.6 Logical loops .......................................... 537
  14.7 Producing multiple copies .............................. 538
  14.8 Detecting \TeX\’s mode .................................. 540
  14.9 Internal programming functions ......................... 541
19 **l3msg implementation** 606  
19.1 Internal auxiliaries 607  
19.2 Creating messages 607  
19.3 Messages: support functions and text 608  
19.4 Showing messages: low level mechanism 609  
19.5 Displaying messages 611  
19.6 Kernel-specific functions 620  
19.7 Expandable errors 628  

20 **l3file implementation** 629  
20.1 Input operations 630  
20.1.1 Variables and constants 630  
20.1.2 Stream management 631  
20.1.3 Reading input 633  
20.2 Output operations 636  
20.2.1 Variables and constants 636  
20.2.2 Internal auxiliaries 637  
20.3 Stream management 638  
20.3.1 Deferred writing 639  
20.3.2 Immediate writing 640  
20.3.3 Special characters for writing 640  
20.3.4 Hard-wrapping lines to a character count 641  
20.4 File operations 650  
20.4.1 Internal auxiliaries 652  
20.5 GetIfInfo 668  
20.6 Messages 669  
20.7 Functions delayed from earlier modules 670  

21 **l3skip implementation** 671  
21.1 Length primitives renamed 671  
21.2 Internal auxiliaries 671  
21.3 Creating and initialising dim variables 671  
21.4 Setting dim variables 672  
21.5 Utilities for dimension calculations 673  
21.6 Dimension expression conditionals 674  
21.7 Dimension expression loops 676  
21.8 Dimension step functions 677  
21.9 Using dim expressions and variables 679  
21.10Viewing dim variables 680  
21.11Constant dimensions 681  
21.12Scratch dimensions 681  
21.13Creating and initialising skip variables 681  
21.14Setting skip variables 682  
21.15Skip expression conditionals 683  
21.16Using skip expressions and variables 684  
21.17Inserting skips into the output 684  
21.18Viewing skip variables 684  
21.19Constant skips 685  
21.20Scratch skips 685  
21.21Creating and initialising muskip variables 685
21.22 Setting muskip variables ........................................... 686
21.23 Using muskip expressions and variables ....................... 687
21.24 Viewing muskip variables ........................................... 687
21.25 Constant muskips ..................................................... 687
21.26 Scratch muskips ....................................................... 687

22 l3keys Implementation .................................................. 688
22.1 Low-level interface .................................................... 688
22.2 Constants and variables .............................................. 694
  22.2.1 Internal auxiliaries .............................................. 696
22.3 The key defining mechanism ........................................ 697
22.4 Turning properties into actions ..................................... 699
22.5 Creating key properties .............................................. 705
22.6 Setting keys ........................................................... 709
22.7 Utilities ............................................................... 718
22.8 Messages .............................................................. 720

23 l3intarray implementation ............................................. 721
23.1 Allocating arrays ..................................................... 721
23.2 Array items ........................................................... 722
23.3 Working with contents of integer arrays ......................... 724
23.4 Random arrays ....................................................... 726

24 l3fp implementation ..................................................... 727

25 l3fp-aux implementation ............................................... 727
25.1 Access to primitives .................................................. 727
25.2 Internal representation .............................................. 728
25.3 Using arguments and semicolons ................................... 729
25.4 Constants, and structure of floating points ...................... 730
25.5 Overflow, underflow, and exact zero .............................. 732
25.6 Expanding after a floating point number ......................... 732
25.7 Other floating point types .......................................... 734
25.8 Packing digits ......................................................... 737
25.9 Decimate (dividing by a power of 10) ............................. 739
25.10 Functions for use within primitive conditional branches ...... 741
25.11 Integer floating points .............................................. 743
25.12 Small integer floating points ...................................... 743
25.13 Fast string comparison .............................................. 744
25.14 Name of a function from its l3fp-parse name ................... 744
25.15 Messages ............................................................. 745

26 l3fp-traps Implementation .............................................. 745
26.1 Flags ................................................................. 745
26.2 Traps ................................................................. 745
26.3 Errors ............................................................... 749
26.4 Messages ............................................................ 749

27 l3fp-round implementation ............................................ 750
27.1 Rounding tools ........................................................ 750
27.2 The round function .................................................... 754
<table>
<thead>
<tr>
<th>28</th>
<th><strong>l3fp-parse implementation</strong></th>
<th>758</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.1</td>
<td>Work plan</td>
<td>758</td>
</tr>
<tr>
<td>28.1.1</td>
<td>Storing results</td>
<td>759</td>
</tr>
<tr>
<td>28.1.2</td>
<td>Precedence and infix operators</td>
<td>760</td>
</tr>
<tr>
<td>28.1.3</td>
<td>Prefix operators, parentheses, and functions</td>
<td>763</td>
</tr>
<tr>
<td>28.1.4</td>
<td>Numbers and reading tokens one by one</td>
<td>764</td>
</tr>
<tr>
<td>28.2</td>
<td>Main auxiliary functions</td>
<td>766</td>
</tr>
<tr>
<td>28.3</td>
<td>Helpers</td>
<td>766</td>
</tr>
<tr>
<td>28.4</td>
<td>Parsing one number</td>
<td>767</td>
</tr>
<tr>
<td>28.4.1</td>
<td>Numbers: trimming leading zeros</td>
<td>774</td>
</tr>
<tr>
<td>28.4.2</td>
<td>Number: small significand</td>
<td>775</td>
</tr>
<tr>
<td>28.4.3</td>
<td>Number: large significand</td>
<td>777</td>
</tr>
<tr>
<td>28.4.4</td>
<td>Number: beyond 16 digits, rounding</td>
<td>779</td>
</tr>
<tr>
<td>28.4.5</td>
<td>Number: finding the exponent</td>
<td>781</td>
</tr>
<tr>
<td>28.5</td>
<td>Constants, functions and prefix operators</td>
<td>785</td>
</tr>
<tr>
<td>28.5.1</td>
<td>Prefix operators</td>
<td>785</td>
</tr>
<tr>
<td>28.5.2</td>
<td>Constants</td>
<td>788</td>
</tr>
<tr>
<td>28.5.3</td>
<td>Functions</td>
<td>789</td>
</tr>
<tr>
<td>28.6</td>
<td>Main functions</td>
<td>789</td>
</tr>
<tr>
<td>28.7</td>
<td>Infix operators</td>
<td>792</td>
</tr>
<tr>
<td>28.7.1</td>
<td>Closing parentheses and commas</td>
<td>793</td>
</tr>
<tr>
<td>28.7.2</td>
<td>Usual infix operators</td>
<td>795</td>
</tr>
<tr>
<td>28.7.3</td>
<td>Juxtaposition</td>
<td>796</td>
</tr>
<tr>
<td>28.7.4</td>
<td>Multi-character cases</td>
<td>796</td>
</tr>
<tr>
<td>28.7.5</td>
<td>Ternary operator</td>
<td>797</td>
</tr>
<tr>
<td>28.7.6</td>
<td>Comparisons</td>
<td>797</td>
</tr>
<tr>
<td>28.8</td>
<td>Tools for functions</td>
<td>799</td>
</tr>
<tr>
<td>28.9</td>
<td>Messages</td>
<td>801</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>29</th>
<th><strong>l3fp-assign implementation</strong></th>
<th>802</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.1</td>
<td>Assigning values</td>
<td>802</td>
</tr>
<tr>
<td>29.2</td>
<td>Updating values</td>
<td>803</td>
</tr>
<tr>
<td>29.3</td>
<td>Showing values</td>
<td>804</td>
</tr>
<tr>
<td>29.4</td>
<td>Some useful constants and scratch variables</td>
<td>804</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
<th><strong>l3fp-logic Implementation</strong></th>
<th>805</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.1</td>
<td>Syntax of internal functions</td>
<td>805</td>
</tr>
<tr>
<td>30.2</td>
<td>Tests</td>
<td>805</td>
</tr>
<tr>
<td>30.3</td>
<td>Comparison</td>
<td>806</td>
</tr>
<tr>
<td>30.4</td>
<td>Floating point expression loops</td>
<td>809</td>
</tr>
<tr>
<td>30.5</td>
<td>Extrema</td>
<td>812</td>
</tr>
<tr>
<td>30.6</td>
<td>Boolean operations</td>
<td>814</td>
</tr>
<tr>
<td>30.7</td>
<td>Ternary operator</td>
<td>815</td>
</tr>
</tbody>
</table>
### 34 l3fp-trig Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.1 Direct trigonometric functions</td>
<td>886</td>
</tr>
<tr>
<td>34.1.1 Filtering special cases</td>
<td>886</td>
</tr>
<tr>
<td>34.1.2 Distinguishing small and large arguments</td>
<td>889</td>
</tr>
<tr>
<td>34.1.3 Small arguments</td>
<td>890</td>
</tr>
<tr>
<td>34.1.4 Argument reduction in degrees</td>
<td>890</td>
</tr>
<tr>
<td>34.1.5 Argument reduction in radians</td>
<td>892</td>
</tr>
<tr>
<td>34.1.6 Computing the power series</td>
<td>899</td>
</tr>
<tr>
<td>34.2 Inverse trigonometric functions</td>
<td>902</td>
</tr>
<tr>
<td>34.2.1 Arctangent and arccotangent</td>
<td>903</td>
</tr>
<tr>
<td>34.2.2 Arcsine and arccosine</td>
<td>908</td>
</tr>
<tr>
<td>34.2.3 Arccosecant and arcsecant</td>
<td>910</td>
</tr>
</tbody>
</table>

### 35 l3fp-convert implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.1 Dealing with tuples</td>
<td>911</td>
</tr>
<tr>
<td>35.2 Trimming trailing zeros</td>
<td>912</td>
</tr>
<tr>
<td>35.3 Scientific notation</td>
<td>912</td>
</tr>
<tr>
<td>35.4 Decimal representation</td>
<td>913</td>
</tr>
<tr>
<td>35.5 Token list representation</td>
<td>915</td>
</tr>
<tr>
<td>35.6 Formatting</td>
<td>916</td>
</tr>
<tr>
<td>35.7 Convert to dimension or integer</td>
<td>917</td>
</tr>
<tr>
<td>35.8 Convert from a dimension</td>
<td>917</td>
</tr>
<tr>
<td>35.9 Use and eval</td>
<td>918</td>
</tr>
<tr>
<td>35.10 Convert an array of floating points to a comma list</td>
<td>919</td>
</tr>
</tbody>
</table>

### 36 l3fp-random Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.1 Engine support</td>
<td>920</td>
</tr>
<tr>
<td>36.2 Random floating point</td>
<td>924</td>
</tr>
<tr>
<td>36.3 Random integer</td>
<td>924</td>
</tr>
</tbody>
</table>

### 37 l3fparray implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.1 Allocating arrays</td>
<td>929</td>
</tr>
<tr>
<td>37.2 Array items</td>
<td>930</td>
</tr>
</tbody>
</table>

### 38 l3sort implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>38.1 Variables</td>
<td>933</td>
</tr>
<tr>
<td>38.2 Finding available \texttt{toks} registers</td>
<td>935</td>
</tr>
<tr>
<td>38.3 Protected user commands</td>
<td>937</td>
</tr>
<tr>
<td>38.4 Merge sort</td>
<td>939</td>
</tr>
<tr>
<td>38.5 Expandable sorting</td>
<td>942</td>
</tr>
<tr>
<td>38.6 Messages</td>
<td>947</td>
</tr>
</tbody>
</table>
40.6.3 Framework .............................................. 1032
40.6.4 Submatches ........................................... 1034
40.6.5 Csnames in replacement .......................... 1036
40.6.6 Characters in replacement ....................... 1037
40.6.7 An error .............................................. 1041
40.7 User functions ........................................ 1041
  40.7.1 Variables and helpers for user functions ... 1043
  40.7.2 Matching ........................................... 1044
  40.7.3 Extracting submatches ......................... 1044
  40.7.4 Replacement ...................................... 1048
40.7.5 Storing and showing compiled patterns .... 1050
40.8 Messages .............................................. 1050
40.9 Code for tracing ...................................... 1056

41 l3box implementation .................................. 1057
  41.1 Support code ...................................... 1057
  41.2 Creating and initialising boxes ................ 1057
  41.3 Measuring and setting box dimensions ....... 1058
  41.4 Using boxes ....................................... 1059
  41.5 Box conditionals ................................... 1059
  41.6 The last box inserted ............................ 1059
  41.7 Constant boxes .................................... 1060
  41.8 Scratch boxes ..................................... 1060
  41.9 Viewing box contents ............................ 1060
  41.10 Horizontal mode boxes .......................... 1061
  41.11 Vertical mode boxes ............................. 1063
  41.12 Affine transformations .......................... 1066

42 l3coffins Implementation .............................. 1075
  42.1 Coffins: data structures and general variables 1075
  42.2 Basic coffin functions ........................... 1076
  42.3 Measuring coffins ................................ 1082
  42.4 Coffins: handle and pole management .......... 1082
  42.5 Coffins: calculation of pole intersections ... 1085
  42.6 Affine transformations .......................... 1088
  42.7 Aligning and typesetting of coffins .......... 1096
  42.8 Coffin diagnostics ................................ 1101
  42.9 Messages .......................................... 1107

43 l3color-base Implementation .......................... 1107

44 l3luatex implementation ................................ 1109
  44.1 Breaking out to Lua ............................... 1109
  44.2 Messages .......................................... 1110
  44.3 Lua functions for internal use ................ 1110
  44.4 Generic Lua and font support .................. 1114

45 l3unicode implementation .............................. 1114
46 \texttt{l3text implementation} \hspace{1em} 1118
  \begin{itemize}
    \item 46.1 Internal auxiliaries \hspace{1em} 1118
    \item 46.2 Utilities \hspace{1em} 1119
    \item 46.3 Configuration variables \hspace{1em} 1121
    \item 46.4 Expansion to formatted text \hspace{1em} 1123
  \end{itemize}

47 \texttt{l3text-case implementation} \hspace{1em} 1130
  \begin{itemize}
    \item 47.1 Case changing \hspace{1em} 1130
    \item 47.2 Case changing data for 8-bit engines \hspace{1em} 1147
  \end{itemize}

48 \texttt{l3text-purify implementation} \hspace{1em} 1154
  \begin{itemize}
    \item 48.1 Purifying text \hspace{1em} 1154
    \item 48.2 Accent and letter-like data for purifying text \hspace{1em} 1159
  \end{itemize}

49 \texttt{l3legacy Implementation} \hspace{1em} 1166

50 \texttt{l3candidates Implementation} \hspace{1em} 1166
  \begin{itemize}
    \item 50.1 Additions to \texttt{l3box} \hspace{1em} 1166
      \begin{itemize}
        \item 50.1.1 Viewing part of a box \hspace{1em} 1166
      \end{itemize}
    \item 50.2 Additions to \texttt{l3flag} \hspace{1em} 1169
    \item 50.3 Additions to \texttt{l3msg} \hspace{1em} 1169
    \item 50.4 Additions to \texttt{l3prg} \hspace{1em} 1171
    \item 50.5 Additions to \texttt{l3prop} \hspace{1em} 1172
    \item 50.6 Additions to \texttt{l3seq} \hspace{1em} 1172
    \item 50.7 Additions to \texttt{l3sys} \hspace{1em} 1175
    \item 50.8 Additions to \texttt{l3file} \hspace{1em} 1176
      \begin{itemize}
        \item 50.8.1 Building a token list \hspace{1em} 1177
        \item 50.8.2 Other additions to \texttt{l3tl} \hspace{1em} 1180
      \end{itemize}
    \item 50.9 Additions to \texttt{l3token} \hspace{1em} 1181
  \end{itemize}

51 \texttt{l3deprecation implementation} \hspace{1em} 1182
  \begin{itemize}
    \item 51.1 Helpers and variables \hspace{1em} 1182
    \item 51.2 Patching definitions to deprecate \hspace{1em} 1184
    \item 51.3 Removed functions \hspace{1em} 1186
    \item 51.4 Deprecated primitives \hspace{1em} 1189
    \item 51.5 Loading the patches \hspace{1em} 1190
    \item 51.6 Deprecated \texttt{l3box} functions \hspace{1em} 1190
    \item 51.7 Deprecated \texttt{l3int} functions \hspace{1em} 1191
    \item 51.8 Deprecated \texttt{l3luatex} functions \hspace{1em} 1192
    \item 51.9 Deprecated \texttt{l3msg} functions \hspace{1em} 1192
    \item 51.10 Deprecated \texttt{l3prg} functions \hspace{1em} 1194
    \item 51.11 Deprecated \texttt{l3str} functions \hspace{1em} 1194
    \item 51.11.1 Deprecated \texttt{l3tl} functions \hspace{1em} 1195
    \item 51.12 Deprecated \texttt{l3tl-analysis} functions \hspace{1em} 1196
    \item 51.13 Deprecated \texttt{l3token} functions \hspace{1em} 1197
    \item 51.14 Deprecated \texttt{l3file} functions \hspace{1em} 1197
  \end{itemize}

Index \hspace{1em} 1198

xxiv
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\LaTeX3 programming language is found in expl3.pdf.

1 Naming functions and variables

L\LaTeX3 does not use $\theta$ 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:

\begin{itemize}
\item \textbf{N} and \textbf{n} These mean \textit{no manipulation}, of a single token for \textbf{N} and of a set of tokens given in braces for \textbf{n}. Both pass the argument through exactly as given. Usually, if you use a single token for an \textbf{n} argument, all will be well.
\item \textbf{c} This means \texttt{csname}, and indicates that the argument will be turned into a csname before being used. So \texttt{\foo:c \{ArgumentOne\}} will act in the same way as \texttt{\foo:N \\ArgumentOne}.
\item \textbf{V} and \textbf{v} These mean \textit{value of variable}. The \textbf{V} and \textbf{v} specifiers are used to get the content of a variable without needing to worry about the underlying \TeX structure containing the data. A \textbf{V} argument will be a single token (similar to \textbf{N}), for example \texttt{\foo:V \\MyVariable}; on the other hand, using \textbf{v} a csname is constructed first, and then the value is recovered, for example \texttt{\foo:v \{\MyVariable\}}.
\item \textbf{o} This means \textit{expansion once}. In general, the \textbf{V} and \textbf{v} specifiers are favoured over \textbf{o} for recovering stored information. However, \textbf{o} is useful for correctly processing information with delimited arguments.
\item \textbf{x} The \textbf{x} specifier stands for \textit{exhaustive expansion}: every token in the argument is fully expanded until only unexpandable ones remain. The \TeX \texttt{\edef} primitive carries out this type of expansion. Functions which feature an \textbf{x}-type argument are \textit{not} expandable.
\item \textbf{e} The \textbf{e} specifier is in many respects identical to \textbf{x}, but with a very different implementation. Functions which feature an \textbf{e}-type argument may be expandable. The drawback is that \textbf{e} is extremely slow (often more than 200 times slower) in older engines, more precisely in non-Lua\TeX engines older than 2019.
\end{itemize}
The f specifier stands for full expansion, and in contrast to 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 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
\tl_set:Nn \l_mya_tl { A }
\tl_set:Nn \l_myb_tl { B }
\tl_set:Nf \l_mya_tl { \l_mya_tl \l_myb_tl }
will leave \l_mya_tl with the content A\l_myb_tl, as A cannot be expanded and so terminates expansion before \l_myb_tl is considered.

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

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

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

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

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

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

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

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module\footnote{The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the int module contains some scratch variables called \l_tmpa_int, \l_tmpb_int, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in \l_int_tmpa_int would be very unreadable.} name and then a descriptive part. Variables end with a short identifier to show the variable type:

clist Comma separated list.

dim “Rigid” lengths.

fp Floating-point values;

int Integer-valued count register.
**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 \texttt{V}-type or \texttt{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 \emph{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.\footnote{\TeX{}nically, functions with no arguments are \texttt{l\_long} while token list variables are not.} 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 \TeX{}book parlance).

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

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

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

```
\ExplSyntaxOn ...
\ExplSyntaxOff
```

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, \textit{(sequence)} indicates that \texttt{\seq_new:N} expects the name of a sequence. From the argument specifier, \texttt{\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 \texttt{x}-type or \texttt{e}-type argument (in plain \LaTeX{} terms, inside an \texttt{edef} or \texttt{\expanded}), as well as within an \texttt{f}-type argument. These fully expandable functions are indicated in the documentation by a star:

```
\cs_to_str:N *
```

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 \textit{(cs)}, shorthand for a \textit{(control sequence)}.

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

```
\seq_map_function:NN *
```

**Conditional functions** Conditional (if) functions are normally defined in three variants, with \texttt{T}, \texttt{F} and \texttt{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 TF indicates that three functions are available:

- \sys_if_engine_xetex:T
- \sys_if_engine_xetex:F
- \sys_if_engine_xetex:TF

Usually, the illustration will use the TF variant, and so both \textit{true code} and \textit{false code} will be shown. The two variant forms T and F take only \textit{true code} and \textit{false code}, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, 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 L\TeX or plain \TeX. In these cases, the text will include an extra “\TeXhackers note” section:

The normal description text.

\TeXhackers note: Detail for the experienced \TeX or L\TeX 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 expl3, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of expl3 after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way unless they impact on the expected behaviour.

3 Formal language conventions which apply generally

As this is a formal reference guide for L\TeX3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test if evaluated to give a logically TRUE or FALSE result. Depending on this result, either the \textit{true code} or the \textit{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  \LaTeX\ concepts not supported by \LaTeX\3

The \TeX\ concept of an “\texttt{\textbackslash outer}” macro is \textit{not supported} at all by \LaTeX\3. As such, the functions provided here may break when used on top of \LaTeX\2 if \texttt{\textbackslash outer} tokens are used in the arguments.
Part II
The l3bootstrap package
Bootstrap code

1 Using the \LaTeX3 modules

The modules documented in source3 are designed to be used on top of \LaTeX2ε and are loaded all as one with the usual \usepackage{expl3} or \RequirePackage{expl3} instructions. These modules will also form the basis of the \LaTeX3 format, but work in this area is incomplete and not included in this documentation at present.

As the modules use a coding syntax different from standard \LaTeX2ε it provides a few functions for setting it up.

\ExplSyntaxOn
⟨code⟩\ExplSyntaxOff

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

\requirepackage{expl3}
\ProvidesExplPackage{⟨package⟩}{{⟨date⟩}}{{⟨version⟩}}{{⟨description⟩}}

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

\requirepackage{l3bootstrap}
\getidinfo \$Id: ⟨SVN info field⟩ \$ {⟨description⟩}

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

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with \RequirePackage or 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 \GetIdInfo command you can use the information when loading a package with

\ProvidesExplPackage{\ExplFileName} % \ExplFileDate\ExplFileVersion{\ExplFileDescription}
Part III

The l3names package
Namespace for primitives

1 Setting up the \LaTeX3 programming language

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

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

This module is entirely dedicated to primitives, which should not be used directly within \LaTeX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The \TeXbook*, *\TeX by Topic* and the manuals for pdf\TeX, \Xe\TeX, Lua\TeX, \mup\TeX and up\TeX should be consulted for details of the primitives. These are named \texttt{\tex\langle name\rangle}:D, typically based on the primitive’s \langle name\rangle in pdf\TeX and omitting a leading pdf when the primitive is not related to pdf output.
Part IV
The \texttt{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

\begin{verbatim}
\prg_do_nothing: * \prg_do_nothing:
An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.
\end{verbatim}

\begin{verbatim}
\scan_stop: \scan_stop:
A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.
\end{verbatim}

2 Grouping material

\begin{verbatim}
\group_begin: \group_begin: \group_end: \group_end:
These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each \texttt{\group_begin}: must be matched by a \texttt{\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.
\end{verbatim}

\begin{verbatim}
\group_insert_after:N \group_insert_after:N \{ token \}
Adds \{token\} to the list \{tokens\} to be inserted when the current group level ends. The list of \{tokens\} to be inserted is empty at the beginning of a group: multiple applications of \texttt{\group_insert_after:N} may be used to build the inserted list one \{token\} at a time. The current group level may be closed by a \texttt{\group_end}: function or by a token with category code 2 (close-group), namely a \} if standard category codes apply.
\end{verbatim}
3 Control sequences and functions

As T\TeX{} is a macro language, creating new functions means creating macros. At point of
desire, 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 ex-
pand 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 T\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:

\textbf{N and n} No manipulation.
\textbf{T and F} Functionally equivalent to n (you are actually encouraged to use the family of
\prg_new_conditional: functions described in Section 1).

\textbf{p and w} These are special cases.

The \cs_new: functions below (and friends) do not stop you from using other argu-
ment 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.

10
3.2 Defining new functions using parameter text

\cs_new:Npn \cs_new:cpn \cs_new:Npx \cs_new:cpn

Creates \textit{(function)} to expand to \textit{(code)} as replacement text. Within the \textit{(code)}, the \textit{(parameters)} (\#1, \#2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the \textit{(function)} is already defined.

\cs_new_nopar:Npn \cs_new_nopar:cpn \cs_new_nopar:Npx \cs_new_nopar:cpn

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

\cs_new_protected:Npn \cs_new_protected:cpn \cs_new_protected:Npx \cs_new_protected:cpn

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

\cs_new_protected_nopar:Npn \cs_new_protected_nopar:nopar \cs_new_protected_nopar:Npx \cs_new_protected_nopar:nopar

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

\cs_set:Npn \cs_set:cpn \cs_set:Npx \cs_set:cpn

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

\cs_set_nopar:Npn \cs_set_nopar:cpn \cs_set_nopar:Npx \cs_set_nopar:cpn

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

\cs_set_protected:Npn \cs_set_protected:cpn \cs_set_protected:Npx \cs_set_protected:cpn

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

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

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

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

3.3 Defining new functions using the signature

Creates \langle function \rangle to expand to \langle code \rangle as replacement text. Within the \langle code \rangle, the number of \langle parameters \rangle is detected automatically from the function signature. These \langle parameters \rangle (#1, #2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the \langle function \rangle is already defined.
\cs_set_protected:Nn \cs_set_protected:(cnNx|cx)

Creating \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} (#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.

\cs_new_protected:Nn \cs_new_protected:(cnNx|cx)

Creating \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} (#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.

\cs_new_protected_nopar:Nn \cs_new_protected_nopar:(cnNx|cx)

Creating \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} (#1, #2, etc.) will be replaced by those absorbed by the function. The \texttt{function} will not expand within an \texttt{x}-type or \texttt{e}-type argument. The definition is global and an error results if the \texttt{function} is already defined.

\cs_set:Nn \cs_set:(cnNx|cx)

Sets \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} (#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.

\cs_set_nopar:Nn \cs_set_nopar:(cnNx|cx)

Sets \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} (#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 assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.

\cs_set_protected:Nn \cs_set_protected:(cnNx|cx)

Sets \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} (#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 assignment of a meaning to the \texttt{function} is restricted to the current \TeX{} group level.
\cs_set_protected_nopar:Nn \cs_set_protected_nopar:(cn|Nx|cx)
\cs_set_protected_nopar:Nn (function) \{\langle code\rangle\}

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

\cs_set_protected_nopar:Nn \cs_set_protected_nopar:(cn|Nx|cx)
\cs_set_protected_nopar:Nn (function) \{\langle code\rangle\}

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

\cs_set_protected_nopar:Nn \cs_set_protected_nopar:(cn|Nx|cx)
\cs_set_protected_nopar:Nn (function) \{\langle code\rangle\}

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

\cs_set_protected_nopar:Nn \cs_set_protected_nopar:(cn|Nx|cx)
\cs_set_protected_nopar:Nn (function) \{\langle code\rangle\}

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

\cs_set_protected_nopar:Nn \cs_set_protected_nopar:(cn|Nx|cx)
\cs_set_protected_nopar:Nn (function) \{\langle code\rangle\}

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

\cs_generate_from_arg_count:NNnn \cs_generate_from_arg_count:NNnn (function) (creator) \{\langle number\rangle\} \{\langle code\rangle\}

\cs_generate_from_arg_count:NNnn (function) \{\langle number\rangle\} \{\langle code\rangle\}

Updated: 2012-01-14

Uses the \textit{creator} function (which should have signature \texttt{Npn}, for example \texttt{cs_new:Npn}) to define a \textit{function} which takes \langle \texttt{number} \rangle arguments and has \langle code \rangle as replacement text. The \langle number \rangle of arguments is an integer expression, evaluated as detailed for \texttt{\textbackslash int_eval:n}.
3.4 Copying control sequences

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

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

\cs_new_eq:NN \cs_set_eq:NN \cs_gset_eq:NN

Globally creates \langle control sequence1 \rangle and sets it to have the same meaning as \langle control sequence2 \rangle or \langle token \rangle. The second control sequence may subsequently be altered without affecting the copy.

\cs_new_eq:(Nc|cN|cc) \cs_set_eq:(Nc|cN|cc) \cs_gset_eq:(Nc|cN|cc)

Sets \langle control sequence1 \rangle to have the same meaning as \langle control sequence2 \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 sequence1 \rangle is restricted to the current \TeX group level.

\cs_gset_eq:(Nc|cN|cc)

Globally sets \langle control sequence1 \rangle to have the same meaning as \langle control sequence2 \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 sequence1 \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 \cs_undefine:c

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

\cs_undefine:N \langle control sequence \rangle

3.6 Showing control sequences

\cs_meaning:N \cs_meaning:c

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

\cs_meaning:N \langle control sequence \rangle

\TeXhackers note: This is \TeX’s \meaning primitive. For tokens that are not control sequences, it is more logical to use \token_to_meaning:N. The c variant correctly reports undefined arguments.
\cs_show:N \cs_show:c
Displays the definition of the \textit{control sequence} on the terminal.

\textbf{\LaTeX{}hackers note:} This is similar to the \LaTeX{} primitive \texttt{\show}, wrapped to a fixed number of characters per line.

\cs_log:N \cs_log:c
Writes the definition of the \textit{control sequence} in the log file. See also \cs_show:N which displays the result in the terminal.

3.7 Converting to and from control sequences

\use:c \{\textit{control sequence name}\}\)
Expands the \textit{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 \textit{control sequence name} must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

\textbf{\LaTeX{}hackers note:} Protected macros that appear in a 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.

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

\begin{verbatim}
\use:c \{ a b c \}
\end{verbatim}

and

\begin{verbatim}
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c \{ \tl_use:N \l_my_tl \}
\end{verbatim}

would be equivalent to

\begin{verbatim}
abc
\end{verbatim}
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 \textit{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 \textit{control sequence} into the input stream followed by the \textit{true code}. Otherwise the \textit{false code} is used.
\cs:w \{control sequence name\} \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.

\textbf{\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 \{control sequence\}

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 \textit{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.

\section{Analysing control sequences}

\cs_split_function:N \{function\}

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{\textbackslash cs\_prefix\_spec:N} \texttt{\langle token\rangle}

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

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

\texttt{\textbackslash cs\_argument\_spec:N} \texttt{\langle token\rangle}

If the \langle 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 token\rangle is not a macro then \texttt{\textbackslash scan\_stop:} is left in the input stream.

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

\texttt{\textbackslash cs\_replacement\_spec:N} \texttt{\langle token\rangle}

If the \langle 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_y\#2} in the input stream. If the \langle token\rangle is not a macro then \texttt{\textbackslash scan\_stop:} is left in the input stream.

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

\section{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).

\begin{align*}
\texttt{\use:nn} & \equiv \{ (\text{group}_1) \} \\
\texttt{\use:nnn} & \equiv \{ (\text{group}_1) \} \{ (\text{group}_2) \} \\
\texttt{\use:nnnn} & \equiv \{ (\text{group}_1) \} \{ (\text{group}_2) \} \{ (\text{group}_3) \} \\
\texttt{\use:i:nn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \\
\texttt{\use:i:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:i:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:i:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:ii:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:i:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:i:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:ii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iii:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:ii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iv:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \}
\end{align*}

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.

\textbf{\TeX{}hackers note:} The \use:n function is equivalent to \LaTeX{}’s \@firstofone.

\begin{align*}
\texttt{\use:i:nn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \\
\texttt{\use:i:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:i:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \}
\end{align*}

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:i:nnn 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.

\textbf{\TeX{}hackers note:} These are equivalent to \LaTeX{}’s \@firstoftwo and \@secondoftwo.

\begin{align*}
\texttt{\use:i:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:ii:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:iii:nnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \\
\texttt{\use:ii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iv:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \}
\end{align*}

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.

\textbf{\TeX{}hackers note:} These are equivalent to \LaTeX{}’s \@firstoftwo, \@secondoftwo and \@secondoftwo.

\begin{align*}
\texttt{\use:i:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:ii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:ii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iii:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \} \\
\texttt{\use:iv:nnnn} & \equiv \{ (\text{arg}_1) \} \{ (\text{arg}_2) \} \{ (\text{arg}_3) \} \{ (\text{arg}_4) \}
\end{align*}

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.
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
abc { def }
i.e. the outer braces are removed and the third group is removed.

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.

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

**TEXhackers note:** These are equivalent to \( \LaTeX \)'s `\@gobble`, `\@gobbletwo`, etc.

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

**TEXhackers note:** `\use:e` is a wrapper around the primitive `\expanded` where it is available: it requires two expansions to complete its action. When `\expanded` is not available this function is very slow.

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

### 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 \textit{balanced text} from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

Absorb the \textit{balanced text} from the input stream delimited by the marker given in the function name, leaving \textit{inserted tokens} in the input stream for further processing.

\section{Predicates and conditionals}

\LaTeX{} 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 \textit{true code} or the \textit{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 \textit{true code} and/or \textit{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 {\textit{true code}} {\textit{false code}}
\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. 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.

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

6.1 Tests on control sequences

\begin{verbatim}
\bool_if:nTF {
    \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {(true code)} {(false code)}
\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_eq_p:NN \cs1 \cs2
\cs_if_eq:NNTF \cs1 \cs2 {\true code} {\false code}
\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_exist_p:N \cs
\cs_if_exist_p:c \cs
\cs_if_exist:NTF \cs \{\true code\} \{\false code\}
\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 \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}. 

\begin{verbatim}
\c_true_bool
\c_false_bool
\end{verbatim}
\if_true: \if_true: \true \else: \false \fi:
\if_false: \false \else: \false \fi:
\reverse_if:N
\if_true: always executes \true, while \if_false: always executes \false. \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in \l3int and used in case switches.

\TeXhackers note: These are equivalent to their corresponding \TeX primitive conditionals; \reverse_if:N is \TeX's \unless.

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

\if_cs_exist:N \if_cs_exist:w \true \else: \false \fi:
\if_mode_horizontal: \true \else: \false \fi:
Check if \true appears in the hash table or if the control sequence that can be formed from \tokens appears in the hash table. The latter function does not turn the control sequence in question into \scan_stop:! This can be useful when dealing with control sequences which cannot be entered as a single token.

\if_mode_horizontal: \true \else: \false \fi:
\if_mode_vertical: \true \else: \false \fi:
\if_mode_math: \true \else: \false \fi:
\if_mode_inner: \true \else: \false \fi:
Execute \true if currently in horizontal mode, otherwise execute \false. Similar for the other functions.

TEXhackers note: This is \TeX's \ifx.

TEXhackers note: These are equivalent to their corresponding \TeX primitive conditionals; \reverse_if:N is \TeX's \unless.
7 Starting a paragraph

\texttt{\textbackslash 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{}\epsilon leave\textbackslash mode approach, no box is used by the method implemented here.

7.1 Debugging support

\texttt{\debug\_on:n}

\texttt{\debug\_off:n}

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

- \textbf{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;
- \textbf{check-expressions} that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- \textbf{deprecation} that makes soon-to-be-deprecated commands produce errors;
- \textbf{log-functions} that logs function definitions;
- \textbf{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{}\epsilon package mode loaded with \texttt{\enable\dash debug} or another option implying it.

\texttt{\debug\_suspend: ... \debug\_resume:}

Suppress (locally) errors and logging from debug commands, except for the deprecated 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}.
Part V

The l3expan package

Argument expansion

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

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

1 Defining new variants

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

Internally preprocessing of arguments is done with functions of the form \texttt{\exp\_ - \ldots}. 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, 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 \textit{parent control sequence} for \LaTeX{} code-level macros. The \textit{parent control sequence} is first separated into the \textit{base name} and \textit{original argument specifier}. The comma-separated list of \textit{variant argument specifiers} is then used to define variants of the \textit{original argument specifier} if these are not already defined. For each \textit{variant} given, a function is created that expands its arguments as detailed and passes them to the \textit{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 \foo:{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 \textit{parent control sequence} is already defined. If the \textit{parent control sequence} is protected or if the \textit{variant} involves any \texttt{x} argument, then the \textit{variant control sequence} is also protected. The \textit{variant} is created globally, as is any \texttt{\exp_args:N(variant)} function needed to carry out the expansion.

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

\begin{itemize}
  \item \texttt{c} variant of an \texttt{N} parent;
  \item \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, \texttt{e}, or \texttt{x} variant of an \texttt{n} parent;
  \item \texttt{N}, \texttt{n}, \texttt{T}, \texttt{F}, or \texttt{p} argument unchanged.
\end{itemize}

This means the \textit{parent} of a \textit{variant} form is always unambiguous, even in cases where both an \texttt{n}-type parent and an \texttt{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 \texttt{n}, \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, \texttt{e}, or \texttt{x}-type variants of an \texttt{N}-type argument or \texttt{N} or \texttt{c}-type variants of an \texttt{n}-type argument. Both are deprecated. The first because passing more than one token to an \texttt{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 \texttt{V}-type or \texttt{v}-type variant instead of \texttt{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 \texttt{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{} registers. The \texttt{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 \texttt{V} specifier should be used. For those referred to by \texttt{(cs)name}, the \texttt{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 T\TeX{}’s \verb|\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 \verb|\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 T\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 \verb|\example:n|. For
this, one should define a variant using \verb|\cs_generate_variant:Nn \example:n { f }|, then do

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

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

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

results in the call

\example:n \{ 3 , \int_eval:n \{ 3 + 4 \} \}

while using \verb|\example:x| or \verb|\example:e| instead results in

\example:n \{ 3 , 7 \}

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 \verb|\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 \textit{token} in the argument it receives: it is conceptually similar to

\verb|\exp_after:wN <base function> \exp_after:wN \{ <argument> \}|

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

\verb|\tl_set:No \tl_tmpa_tl \{ \{ \g_tmpb_tl \} \}|

28
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 draw-
  back 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 \langle function\rangle name and the \langle tokens\rangle). The
\langle tokens\rangle 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 \langle function\rangle.
Thus the \langle function\rangle may take more than one argument: all others are left unchanged.

The :cc variant constructs the \langle function\rangle name in the same manner as described for
the \langle tokens\rangle.

\textbf{\texttt{\TeX} hackers note:} Protected macros that appear in a 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.
This function absorbs two arguments (the \textit{function} name and the \textit{tokens}). The \textit{tokens} are expanded once, and 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.

\textbf{TEX hackers 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.

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.

\textbf{TEX hackers note:} This relies on the \texttt{\expanded} primitive when available (in Lua\TeX{} 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 \textit{function} name and the \textit{tokens}) and exhaustively expands the \textit{tokens}. The result is inserted in braces into the input stream \textit{after} reinsertion of the \textit{function}. Thus the \textit{function} may take more than one argument: all others are left unchanged.

5 \hspace{1em} Manipulating two arguments

\texttt{\textbackslash exp\_args: Nx (function) \{\langle 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.

\texttt{\textbackslash exp\_args: Nnc \langle token_1 \rangle \langle token_2 \rangle \{\langle 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.

\texttt{\textbackslash exp\_args: Nnx \langle token_1 \rangle \langle token_2 \rangle \{\langle 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 \textit{x}-type argument.
6 Manipulating three arguments

\texttt{\exp_args:NNNo} \ (token_1) \ (token_2) \ (token_3) \ \{\tokens\}

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

\texttt{\exp_args:NNcf} \ (token_1) \ (token_2) \ \{\tokens_1\} \ \{\tokens\}

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

\texttt{\exp_args:NNx} \ (token_1) \ (token_2) \ \{\tokens_1\} \ \{\tokens_2\}

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, \textit{etc}. New: 2015-08-12
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{\TeX{}hackers 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 \texttt{f-}
expanded.

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.

\exp_not:N \token

Prevents expansion of the \token in a context where it would otherwise be expanded, for example an x-type argument or the first token in an o or e or f argument.

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

\exp_not:c \{\tokens\}

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

\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.

\exp_not:n \{\tokens\}

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

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

\exp_not:o \{\tokens\}

Expands the \tokens once, then prevents any further expansion in x-type or e-type arguments using \exp_not:n.

\exp_not:V \variable

Recover the content of the \variable, then prevents expansion of this material in x-type or e-type arguments using \exp_not:n.

34
\exp_not:v \{\langle tokens \rangle\}

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

\textbf{\LaTeX{}hackers note:} Protected macros that appear in a v-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.

\exp_not:e \{\langle tokens \rangle\}

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

\exp_not:f \{\langle tokens \rangle\}

Expands \langle tokens \rangle 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 x-type or e-type arguments using \exp_not:n.

\foo_bar:f \{\langle tokens \rangle \exp_stop_f: \langle more tokens \rangle\}

This function terminates an f-type expansion. Thus if a function \foo_bar:f starts an f-type expansion and all of \langle tokens \rangle are expandable \exp_stop_f: terminates the expansion of tokens even if \langle more tokens \rangle are also expandable. The function itself is an implicit space token. Inside an x-type expansion, it retains its form, but when typeset it produces the underlying space (\uni{2423}).

9 Controlled expansion

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

Of course, deep down \LaTeX{} 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 \exp_after:wN expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of \langle expandable-tokens \rangle as that will break badly if unexpandable tokens are encountered in that place!
\exp:w \langle expandable-tokens \rangle \exp_end:

Expands \langle expandable-tokens \rangle until reaching \exp_end: at which point expansion stops. The full expansion of \langle expandable-tokens \rangle has to be empty. If any token in \langle 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 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 expandable-tokens \rangle, but this should not be relied upon.

\exp:w \langle expandable-tokens \rangle \exp_end_continue_f:w \langle further-tokens \rangle

Expands \langle expandable-tokens \rangle until reaching \exp_end_continue_f:w \langle further-tokens \rangle at which point expansion continues as an f-type expansion expanding \langle further-tokens \rangle until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by \exp_\text{-stop_f:}). As with all f-type expansions a space ending the expansion gets removed.

The full expansion of \langle expandable-tokens \rangle has to be empty. If any token in \langle 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 \exp_stop_f: 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 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 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 expandable-tokens \rangle \exp_stop_f:

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.
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{\langle further-tokens\rangle} 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 \texttt{f}-type expansion.

10 Internal functions

\begin{verbatim}
\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }
\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
\end{verbatim}

Internal forms for the base expansion types. These names do \emph{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{verbatim}
\cs_new:Npn \exp_last_unbraced:No { \::n \::o_unbraced \::: }
\end{verbatim}

Internal forms for the expansion types which leave the terminal argument unbraced. These names do \emph{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 \texttt{l3quark} package

Quarks

Two special types of constants in \LaTeX{} are “quarks” and “scan marks”. By convention all constants of type quark start out with \texttt{\textbackslash q}, and scan marks start with \texttt{\textbackslash s}.

1 Quarks

Quarks are control sequences that expand to themselves and should therefore \textit{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’ (\textit{i.e.} \texttt{\textbackslash q_stop}). For example, when writing a macro to parse a user-defined date

\begin{verbatim}
\date_parse:n {19/June/1981}
\end{verbatim}

one might write a command such as

\begin{verbatim}
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
\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 \texttt{\q_no_value}. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using \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 quark \rangle

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

\texttt{\textbackslash q_stop}

Used as a marker for delimited arguments, such as

\begin{verbatim}
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
\end{verbatim}
\textbf{\texttt{\q_mark}} Used as a marker for delimited arguments when \texttt{\q_stop} is already in use.

\textbf{\texttt{\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 \texttt{\q_stop}, which is only ever used as a delimiter).

\textbf{\texttt{\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 \texttt{\prop_get:NnN} if there is no data to return.

\section{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.

\begin{verbatim}
\quark_if_nil_p:N \quark_if_nil:NTF \langle token \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}
\end{verbatim}

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

\begin{verbatim}
\quark_if_nil_p:n \quark_if_nil:p:(o|V) \quark_if_nil:n:TF \langle token \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}
\end{verbatim}

Tests if the \langle token list \rangle contains only \texttt{\q_nil} (distinct from \langle token list \rangle being empty or containing \texttt{\q_nil} plus one or more other tokens).

\begin{verbatim}
\quark_if_no_value_p:N \quark_if_no_value:p:c \quark_if_no_value:n:TF \langle token \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}
\end{verbatim}

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

\begin{verbatim}
\quark_if_no_value_p:n \quark_if_no_value:p:n \quark_if_no_value:n:NTF \langle token list \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}
\end{verbatim}

Tests if the \langle token list \rangle contains only \texttt{\q_no_value} (distinct from \langle token list \rangle being empty or containing \texttt{\q_no_value} plus one or more other tokens).

\section{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.

\textbf{\texttt{\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.

\[ \texttt{\textbackslash q\_recursion\_stop} \]

_tests if \langle token list \rangle contains only the marker \texttt{\textbackslash q\_recursion\_tail}, and if so uses \texttt{\use\textbackslash i\_delimit\textbackslash by\_q\_recursion\_stop:w} to terminate the recursion that this belongs to. The recursion input must include the marker tokens \texttt{\textbackslash q\_recursion\_tail} and \texttt{\textbackslash q\_\textbackslash -\_recursion\_stop} as the last two items.

\[ \texttt{\textbackslash q\_if\_recursion\_tail\_stop:o} \]

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

\[ \texttt{\textbackslash q\_if\_recursion\_tail\_break:nN} \]

Tests if \langle token list \rangle contains only \texttt{\textbackslash q\_recursion\_tail}, and if so terminates the recursion using \langle type \rangle\_map\_break:. The recursion end should be marked by \texttt{\prg\_\textbackslash break\_point:Nn \langle type \rangle\_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\} \{\texttt{[--#1--#2--]}\}} would produce “[\texttt{-a-b-}] [\texttt{-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 \texttt{#2}. Then initiate the recursion using an internal function. The token list \texttt{#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}
}
\end{verbatim}

Finally, recurse:

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

Note that contrarily to \LaTeX{} built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of \texttt{\textunderscore}_{\text{my_map_dbl}}_{\text{fn}\textcolon\text{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:Nn \scan_stop: \scan_stop:
\end{verbatim}

Creates a new \texttt{\langle\text{scan mark}\rangle} which is set equal to \texttt{\scan_stop}. The \texttt{\langle\text{scan mark}\rangle} is defined globally, and an error message is raised if the name was already taken by another scan mark.

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

41
\s_stop

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

\use_none_delimit_by_s_stop:w * \use_none_delimit_by_s_stop:w (tokens) \s_stop

Removes the (tokens) and \s_stop from the input stream. This leads to a low-level \LaTeX error if \s_stop is absent.
Part VII

The l3tl package

Token lists

\TeX{} works with tokens, and \LaTeX{} 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 \texttt{tokens} }
```

or may be stored in a so-called “token list variable”, which have the suffix \texttt{tl}: a token list variable can also be used as the argument to a function, for example

```
\foo:N \tl_{some\_tl}
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix \texttt{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 \texttt{\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 \texttt{N} argument, or \texttt{\{}, or \texttt{\}\} (assuming normal \TeX{} category codes). Thus for example

```
\{ Hello } \texttt{- world}
```

contains six items (\texttt{Hello, w, o, r, 1 and d}), but thirteen tokens (\texttt{\{, H, e, l, 1, o, \}, \texttt{\}}, \texttt{w, o, r, 1 and d}). Functions which act on items are often faster than their analogue acting directly on tokens.

1 Creating and initialising token list variables

```
\tl_new:N \tl_new:c
\tl_const:Nn \tl_const:c
\tl_clear:N \tl_clear:c
\tl_gclear:N \tl_gclear:c
```

```
\tl_new:N \tl const:Nn \tl_new:c \tl const:c \tl clear:N \tl clear:c
```

```
\tl_new:N \tl_new:c
\tl const:Nn \tl const:c
\tl clear:N \tl clear:c
```

\begin{itemize}
\item \texttt{\tl_new:N \tl_new:c} Creates a new \texttt{\{tl var\}} or raises an error if the name is already taken. The declaration is global. The \texttt{\{tl var\}} is initially empty.
\item \texttt{\tl const:Nn \tl const:c} Creates a new constant \texttt{\{tl var\}} or raises an error if the name is already taken. The value of the \texttt{\{tl var\}} is set globally to the \texttt{(token list)}.  
\item \texttt{\tl clear:N \tl clear:c} Clears all entries from the \texttt{\{tl var\}}.
\end{itemize}
Ensures that the \(tl\ var\) exists globally by applying \tl_new:N\ if necessary, then applies \tl_(g)clear:N\ to leave the \(tl\ var\) empty.

\(\text{tl_set_eq:NN}\ (\text{tl var}_1) (\text{tl var}_2)\)

Sets the content of \(\text{tl var}_1\) equal to that of \(\text{tl var}_2\).

\(\text{tl_concat:NNN} (\text{tl var}_1) (\text{tl var}_2) (\text{tl var}_3)\)

Concatenates the content of \(\text{tl var}_2\) and \(\text{tl var}_3\) together and saves the result in \(\text{tl var}_1\). The \(\text{tl var}_2\) is placed at the left side of the new token list.

\(\text{tl_set:Nn} (\text{tl var}) \{\text{tokens}\}\)

Sets \(\text{tl var}\) to contain \(\text{tokens}\), removing any previous content from the variable.

\(\text{tl_put_left:Nn} (\text{tl var}) \{\text{tokens}\}\)

Appends \(\text{tokens}\) to the left side of the current content of \(\text{tl var}\).

\(\text{tl_put_right:Nn} (\text{tl var}) \{\text{tokens}\}\)

Appends \(\text{tokens}\) to the right side of the current content of \(\text{tl var}\).

2 Adding data to token list variables

<table>
<thead>
<tr>
<th>(\text{tl_set:NN})</th>
<th>(\text{tl_set:NN} (\text{tl var}) {\text{tokens}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{tl_set:(NV</td>
<td>No</td>
</tr>
<tr>
<td>(\text{tl_gset:NN})</td>
<td>(\text{tl_gset:NN} (\text{tl var}) {\text{tokens}})</td>
</tr>
<tr>
<td>(\text{tl_gset:(NV</td>
<td>No</td>
</tr>
</tbody>
</table>

Sets \(\text{tl var}\) to contain \(\text{tokens}\), removing any previous content from the variable.

<table>
<thead>
<tr>
<th>(\text{tl_put_left:Nn})</th>
<th>(\text{tl_put_left:Nn} (\text{tl var}) {\text{tokens}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{tl_put_left:(NV</td>
<td>No</td>
</tr>
<tr>
<td>(\text{tl_gput_left:Nn})</td>
<td>(\text{tl_gput_left:Nn} (\text{tl var}) {\text{tokens}})</td>
</tr>
<tr>
<td>(\text{tl_gput_left:(NV</td>
<td>No</td>
</tr>
</tbody>
</table>

Appends \(\text{tokens}\) to the left side of the current content of \(\text{tl var}\).

<table>
<thead>
<tr>
<th>(\text{tl_put_right:Nn})</th>
<th>(\text{tl_put_right:Nn} (\text{tl var}) {\text{tokens}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{tl_put_right:(NV</td>
<td>No</td>
</tr>
<tr>
<td>(\text{tl_gput_right:Nn})</td>
<td>(\text{tl_gput_right:Nn} (\text{tl var}) {\text{tokens}})</td>
</tr>
<tr>
<td>(\text{tl_gput_right:(NV</td>
<td>No</td>
</tr>
</tbody>
</table>

Appends \(\text{tokens}\) to the right side of the current content of \(\text{tl var}\).
3 Modifying token list variables

\texttt{\textbackslash tl\_replace\_once:Nnn} \langle \texttt{tl var} \rangle \{\langle old tokens\rangle\} \{\langle new tokens\rangle\}

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).

\texttt{\textbackslash tl\_replace\_all:Nnn} \langle \texttt{tl var} \rangle \{\langle old tokens\rangle\} \{\langle new tokens\rangle\}

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 \texttt{\textbackslash tl\_remove\_all:Nn} for an example).

\texttt{\textbackslash tl\_remove\_once:Nn} \langle \texttt{tl var} \rangle \{\langle tokens\rangle\}

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).

\texttt{\textbackslash tl\_remove\_all:Nn} \langle \texttt{tl var} \rangle \{\langle tokens\rangle\}

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,

\texttt{\textbackslash tl\_set:Nn \l\_tmpa\_tl \{abbccd\} \textbackslash tl\_remove\_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 \texttt{\textbackslash char\_generate:nn}).
\tl_set_rescan:Nnn \tl_set_rescan:(Nno|Nnx|cnn|cno|cnx) \tl_gset_rescan:Nnn \tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)

Updated: 2015-08-11

Sets \texttt{\langle tl var \rangle} to contain \texttt{\langle tokens \rangle}, applying the category code régime specified in the \texttt{\langle setup \rangle} before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the \texttt{\langle setup \rangle} are those in force at the point of use of \texttt{\tl_set_rescan:Nnn}.) This allows the \texttt{\langle tl var \rangle} to contain material with category codes other than those that apply when \texttt{\langle tokens \rangle} are absorbed. The \texttt{\langle setup \rangle} 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}.

\TeXhackers note: The \texttt{\langle tokens \rangle} 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 32, before the user \texttt{\langle setup \rangle}), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

\tl_rescan:nn \tl_rescan:n \langle setup \rangle \langle tokens \rangle

Rescans \texttt{\langle tokens \rangle} applying the category code régime specified in the \texttt{\langle setup \rangle}, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the \texttt{\langle setup \rangle} are those in force at the point of use of \texttt{\tl_rescan:nn}.) The \texttt{\langle setup \rangle} 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{\langle tokens \rangle} argument of \texttt{\tl_rescan:nn}.

\TeXhackers note: The \texttt{\langle tokens \rangle} 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 32, before the user \texttt{\langle setup \rangle}), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

5 Token list conditionals

\tl_if_blank_p:n \tl_if_blank:nTF

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

46
\tl_if_empty_p:\N *
\tl_if_empty_p:\C *
\tl_if_empty:\NTF *
\tl_if_empty:cTF *

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

\tl_if_empty_p:\N *
\tl_if_empty_p:\{V|o\} *
\tl_if_empty:\ntF *
\tl_if_empty:cTF *

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

\tl_if_eq_p:\NN *
\tl_if_eq_p:\(Nc|cN|cc) *
\tl_if_eq:\ntF *
\tl_if_eq:cTF *

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

\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:\ntF \l_tmpa_tl \l_tmpb_tl { true } { false }
yields false.

\tl_if_eq:\ntF *
\tl_if_eq:cn *
\tl_if_eq:nn *

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

\tl_if_in:Nn *
\tl_if_in:cn *
\tl_if_in:nn *

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).

\tl_if_in:n *
\tl_if_in:nn *

Tests if \langle token list \rangle is found inside \langle token list \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).

\tl_if_novalue_p:n *
\tl_if_novalue:n *

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{tl var} 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 \texttt{function} or \texttt{code} discussed below remain in effect after the loop.
\mapfunction{nN} \{\textit{token list}\} \{\textit{function}\}

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

\mapinline{Nn} \{\textit{tl var}\} \{\textit{inline function}\}

Applies the \{\textit{inline function}\} to every \{\textit{item}\} stored within the \{\textit{tl var}\}. The \{\textit{inline function}\} should consist of code which receives the \{\textit{item}\} as #1. See also \mapfunction{NN}.

\mapinline{nn} \{\textit{token list}\} \{\textit{inline function}\}

Applies the \{\textit{inline function}\} to every \{\textit{item}\} stored within the \{\textit{token list}\}. The \{\textit{inline function}\} should consist of code which receives the \{\textit{item}\} as #1. See also \mapfunction{NN}.

\maptokens{Nn} \{\textit{tl var}\} \{\textit{code}\}
\maptokens{cn} \{\textit{tokens}\} \{\textit{code}\}
\maptokens{nn} \{\textit{token list}\} \{\textit{code}\}

Analogue of \mapfunction{NN} which maps several tokens instead of a single function. The \{\textit{code}\} receives each item in the \{\textit{tl var}\} or \{\textit{tokens}\} as two trailing brace groups. For instance,

\begin{verbatim}
\maptokens{Nn} \l_my_tl \{ \prg_replicate:nn \{ 2 \} \}
\end{verbatim}

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

\mapvariable{NNn} \{\textit{tl var}\} \{\textit{variable}\} \{\textit{code}\}
\mapvariable{cNn} \{\textit{tokens}\} \{\textit{code}\}
\mapvariable{nn} \{\textit{token list}\} \{\textit{code}\}

Stores each \{\textit{item}\} of the \{\textit{tl var}\} in turn in the \{\textit{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{item}\} in the \{\textit{tl var}\}, or its original value if the \{\textit{tl var}\} is blank. See also \mapinline{Nn}.

\mapvariable{nNn} \{\textit{token list}\} \{\textit{variable}\} \{\textit{code}\}

Stores each \{\textit{item}\} of the \{\textit{token list}\} in turn in the \{\textit{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{item}\} in the \{\textit{tl var}\}, or its original value if the \{\textit{tl var}\} is blank. See also \mapinline{nn}.
\tl_map_break:

Used to terminate a \tl_map... function before all entries in the \textit{token list variable} have been processed. This normally takes place within a conditional statement, for example

\begin{verbatim}
\tl_map_inline:Nn \l_my_tl
{\str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
% Do something useful
}
\end{verbatim}

See also \tl_map_break:n. Use outside of a \tl_map... scenario leads to low level \TeX{} errors.

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

\tl_map_break:n

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

\begin{verbatim}
\tl_map_inline:Nn \l_my_tl
{\str_if_eq:nnT { #1 } { bingo } { \tl_map_break:n { <code> } }
% Do something useful
}
\end{verbatim}

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

\textbf{\TeX{}hackers note:} When the mapping is broken, additional tokens may be inserted before the \textit{code} 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.

**TeXhackers note:** This is the \e-\TeX primitive \texttt{\detokenize}. 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

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

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

\tl_to_str:N \tl_to_str:c

\tl_to_str:n (tl var)

\tl_to_str:c

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{\tl_to_str:n}.

\tl_use:N \tl_use:c

\tl_use:N (tl var)

\tl_use:c

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 \tl_count:c

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

\tl_count:c

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{\tl_count:N}. This function requires three expansions, giving an \langle integer denotation\rangle.
\texttt{\tl_count:N \{tl \ var\}}

Counts the number of token groups in the \texttt{\{tl \ var\}} and leaves this information in the input stream. Unbraced tokens count as one element as do each token group \texttt{\{...\}}. This process ignores any unprotected spaces within the \texttt{\{tl \ var\}}. See also \texttt{\tl_count:n}. This function requires three expansions, giving an \textit{integer denotation}.

\texttt{\tl_count_tokens:n \{\tokens\}}

Counts the number of \TeX{} tokens in the \texttt{\{\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{\strut a\{-bc\}} is \texttt{6}.

\texttt{\tl_reverse:n \{\token \ list\}}

Reverses the order of the \texttt{\{items\}} in the \texttt{\{token \ list\}}, so that \texttt{\{item1\}\{item2\}\{item3\} ... \{itemn\}} becomes \texttt{\{itemn\} ... \{item3\}\{item2\}\{item1\}}. This process preserves unprotected space within the \texttt{\{token \ list\}}. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \texttt{\tl_reverse_items:n}. See also \texttt{\tl_reverse:N}.

\textbf{\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}\textsc{-type argument expansion}.

\texttt{\tl_reverse_items:n \{\token \ list\}}

Reverses the order of the \texttt{\{items\}} stored in \texttt{\{tl \ var\}}, so that \texttt{\{item1\}\{item2\}\{item3\} ... \{itemn\}} becomes \texttt{\{itemn\} ... \{item3\}\{item2\}\{item1\}}. This process removes any unprotected space within the \texttt{\{token \ list\}}. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function \texttt{\tl_reverse:n}.

\textbf{\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}\textsc{-type argument expansion}.

\texttt{\tl_trim_spaces:n \{\token \ list\}}

Removes any leading and trailing explicit space characters (explicit tokens with character code \texttt{32} and category code \texttt{10}) from the \texttt{\{token \ list\}} and leaves the result in the input stream.

\textbf{\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}\textsc{-type argument expansion}.

52
\tl_trim_spaces_apply:nN \star \tl_trim_spaces_apply:oN \star  
\textit{New: 2018-04-12}  
\tl_trim_spaces:N \tl_gtrim_spaces:N  
\textit{New: 2011-07-09}  
\tl_sort:Nn \tl_sort:cn \tl_gsort:Nn \tl_gsort:cn  
\textit{New: 2017-02-06}  
\tl_sort:nN \star  
\textit{New: 2017-02-06}  
\tl_sort:nN  
\tl_sort:cn  
\tl_gsort:nN  
\tl_gsort:cn  
\textit{New: 2011-07-09}  
\tl_sort:nN \star \tl_sort:cn \tl_gsort:nN \tl_gsort:cn  
\textit{New: 2017-02-06}  
\tl_sort:nN \star \tl_sort:cn \tl_gsort:nN \tl_gsort:cn

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

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

\tl_sort:Nn \langle tl var \rangle \{\langle comparison code\rangle\}  
Sorts the items in the \(\langle\text{tl var}\rangle\) according to the \(\langle\text{comparison code}\rangle\), and assigns the result to \(\langle\text{tl var}\rangle\). The details of sorting comparison are described in Section 1.

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

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

\section{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.
\resetcounters
\resetlabels

\ttlhead:N * \ttlhead:n \{ \langle \text{token list} \rangle \}
Leaves in the input stream the first \langle \text{item} \rangle in the \langle \text{token list} \rangle, discarding the rest of the \langle \text{token list} \rangle. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
\ttlhead:n \{ \ abc \ }
and
\ttlhead:n \{ - \ abc \ }
both leave \texttt{a} in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so
\ttlhead:n \{ - \{ - \ ab \} \ c \}
yields \texttt{ab}. A blank \langle \text{token list} \rangle (see \ttlifblank:nTF) results in \ttlhead:n leaving nothing in the input stream.

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

\ttltail:N * \ttltail:n \{ \langle \text{token list} \rangle \} \qstop
Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first \langle \text{item} \rangle in the \langle \text{token list} \rangle, and leaves the remaining tokens in the input stream. Thus for example
\ttltail:n \{ \ a \ - \{ \bc \} \ d \ }
and
\ttltail:n \{ \ - \ a \ - \{ \bc \} \ d \ }
both leave \texttt{\{\bc\}d} in the input stream. A blank \langle \text{token list} \rangle (see \ttltifblank:nTF) results in \ttltail:n leaving nothing in the input stream.

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

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

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

Tests if the first \langle token \rangle in the \langle token list \rangle is an explicit begin-group character (with category code 1 and any character code), in other words, if the \langle token list \rangle starts with a brace group. In particular, the test is \textit{false} if the \langle token list \rangle 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 \langle token \rangle in the \langle token list \rangle is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields \textit{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.
10 Using a single item

\tl_item:nn \{\langle token list\rangle\} \{\langle integer expression\rangle\}

Indexing items in the \langle token list\rangle from 1 on the left, this function evaluates the \langle integer expression\rangle and leaves the appropriate item from the \langle token list\rangle in the input stream. If the \langle integer expression\rangle 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 \unexpanded primitive (\exp_not:n), which means that the \langle item\rangle does not expand further when appearing in an \texttt{x}-type argument expansion.

\tl_rand_item:N \{\langle tl var\rangle\}
\tl_rand_item:n \{\langle token list\rangle\}

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

\textbf{\TeXhackers note:} 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 \texttt{x}-type argument expansion.
Leaves in the input stream the items from the \langle start index \rangle to the \langle end index \rangle inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here \langle start index \rangle and \langle end index \rangle should be \langle integer expressions \rangle. 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’, and a negative index means ‘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 \( s > l \).

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 \tl_trim_spaces:n to begin with.

Thus, with \( l = 7 \) as in the examples below, all of the following are equivalent and result in the whole token list
\[
\texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 1 \} \{ 7 \}
\]
\[
\texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 1 \} \{ 12 \}
\]
\[
\texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -7 \} \{ 7 \}
\]
\[
\texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -12 \} \{ 7 \}
\]

Here are some more interesting examples. The calls
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 2 \} \{ 5 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 2 \} \{ -3 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -6 \} \{ 5 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -6 \} \{ -3 \} \}
\]
are all equivalent and will print \( \texttt{bcd\{e{}f} \) on the terminal; similarly
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 2 \} \{ 5 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 2 \} \{ -3 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -6 \} \{ 5 \} \}
\]
\[
\texttt{iow_term:x } \{ \texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ -6 \} \{ -3 \} \}
\]
are all equivalent and will print \( \texttt{bcd \{e{}f} \) on the terminal (note the space in the middle).

To the contrary,
\[
\texttt{\tl_range:nnn \{ } abc\texttt{-}{e{}f} \texttt{g } \{ 2 \} \{ 4 \}
\]
will discard the space after ‘\texttt{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 \{ } <\texttt{tl}> \{ 3 \} \{ -1 \}. Similarly, for discarding the last item, we can do \texttt{\tl_range:nnn \{ } <\texttt{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 \langle item \rangle does not expand further when appearing in an \texttt{x}-type argument expansion.
11 Viewing token lists

\tl_show:N (tl var)
\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 \{\langle token list\rangle\}
Displays the \langle token list\rangle on the terminal.

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

\tl_log:N (tl var)
\tl_log:c
\tl_log:n \{\langle token list\rangle\}
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.

**TeXhackers note:** New: 2014-08-22
Updated: 2015-08-07

12 Constant token lists

\c_empty_tl
Constant that is always empty.

\c_novalue_tl
A marker for the absence of an argument. This constant \t1 can safely be typeset (\textit{cf.} \q-nil), with the result being \textit{-NoValue-}. It is important to note that \c_novalue_tl is constructed such that it will not match the simple text input \textit{-NoValue-}, \textit{i.e.} that

\tl_if_eq:VnTF \c_novalue_tl \{ \textit{-NoValue- } \}

is logically \textit{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 \t1.

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

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

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

The l3str package: Strings

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

A \TeX{} string (and thus an expl3 string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a \TeX{} string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix ...str. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using \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_...:N}, which expect a token list or string variable as their argument;
- \texttt{\str_...:n}, taking any token list (or string) as an argument;
- \texttt{\str_..._ignore_spaces:n}, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

\begin{verbatim}
\str_new:N \langle str var \rangle
\end{verbatim}

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

\begin{verbatim}
\str_const:Nn \langle str var \rangle { \langle token list \rangle }
\end{verbatim}

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

\str_clear_new:N \str_clear_new:c

\str_set_eq:NN \str_set_eq: \str_(g)clear:N \str_gset_eq:NN \str_gset_eq: (cN | Nc | cc)

\str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc

2 Adding data to string variables

\str_set:Nn \str_set:N \str_set:(NV|Nx|cn|cV|cx) \str_gset:Nn \str_gset:(NV|Nx|cn|cV|cx)

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

\\str_set:N \str_set:N \str_set:(NV|Nx|cn|cV|cx) \str_gset:(NV|Nx|cn|cV|cx)

\str_put_left:Nn \str_put_left:(NV|Nx|cn|cV|cx)

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

\str_clear_new:N \str_clear_new:c

\str_set_eq:NN \str_set_eq: \str_(g)clear:N \str_gset_eq:NN \str_gset_eq: (cN | Nc | cc)

\str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc

\\str_set:N \str_set:N \str_set:(NV|Nx|cn|cV|cx) \str_gset:(NV|Nx|cn|cV|cx)

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

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

\str_clear_new:N \str_clear_new:c

\str_set_eq:NN \str_set_eq: \str_(g)clear:N \str_gset_eq:NN \str_gset_eq: (cN | Nc | cc)

\str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc

\\str_set:N \str_set:N \str_set:(NV|Nx|cn|cV|cx) \str_gset:(NV|Nx|cn|cV|cx)

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

2 Adding data to string variables

\str_set:Nn \str_set:N \str_set:(NV|Nx|cn|cV|cx) \str_gset:Nn \str_gset:(NV|Nx|cn|cV|cx)

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

Clears the content of the \langle str var \rangle.

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

Sets the content of \langle str var1 \rangle equal to that of \langle str var2 \rangle.

Concatenates the content of \langle str var2 \rangle and \langle str var3 \rangle together and saves the result in \langle str var1 \rangle. The \langle str var2 \rangle is placed at the left side of the new string variable. The \langle str var2 \rangle and \langle str var3 \rangle must indeed be strings, as this function does not convert their contents to a string.

Converts the \langle token list \rangle to a \langle string \rangle, and stores the result in \langle str var \rangle.

Converts the \langle token list \rangle to a \langle string \rangle, and prepends the result to \langle str var \rangle. The current contents of the \langle str var \rangle are not automatically converted to a string.

2 Adding data to string variables

Converts the \langle token list \rangle to a \langle string \rangle, and stores the result in \langle str var \rangle.

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

### 3 Modifying string variables

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

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

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

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

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

results in \( \l_tmpa_str \) containing abcd.
4 String conditionals

\texttt{\textbackslash \text{str\_if\_exist\_p}:N \langle \text{str \ var} \rangle}
\texttt{\textbackslash \text{str\_if\_exist\_N}\text{TF} \langle \text{str \ var} \rangle \{\text{true code}\} \{\text{false code}\}}

Tests whether the \textlangle str \ var \rangle is currently defined. This does not check that the \textlangle str \ var \rangle really is a string.

\texttt{\textbackslash \text{str\_if\_empty\_p}:N \langle \text{str \ var} \rangle}
\texttt{\textbackslash \text{str\_if\_empty\_N}\text{TF} \langle \text{str \ var} \rangle \{\text{true code}\} \{\text{false code}\}}

Tests if the \textlangle string variable \rangle is entirely empty (i.e. contains no characters at all).

\texttt{\textbackslash \text{str\_if\_eq\_p}:NN \langle \text{str \ var} \rangle \langle \text{str \ var} \rangle}
\texttt{\textbackslash \text{str\_if\_eq\_NN}\text{TF} \langle \text{str \ var} \rangle \langle \text{str \ var} \rangle \{\text{true code}\} \{\text{false code}\}}

Compares the content of two \textlangle str variables \rangle and is logically true if the two contain the same characters in the same order. Thus for example

\textbackslash \text{str\_if\_eq\_p}:no \{ \text{abc} \} \{ \text{tl\_to\_str:n} \{ \text{abc} \} \}

is logically true.

\texttt{\textbackslash \text{str\_if\_in\_N}\text{TF} \langle \text{str \ var} \rangle \{\text{token list}\} \{\text{true code}\} \{\text{false code}\}}
\texttt{\textbackslash \text{str\_if\_in\_c}\text{TF} \langle \text{str \ var} \rangle \{\text{token list}\} \{\text{true code}\} \{\text{false code}\}}

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

\texttt{\textbackslash \text{str\_if\_in\_n}\text{TF} \langle \text{tl} \rangle \{\langle \text{tl} \rangle \} \{\text{true code}\} \{\text{false code}\}}
\texttt{\textbackslash \text{str\_if\_in\_n}\text{TF} \langle \text{tl} \rangle \{\langle \text{tl} \rangle \} \{\text{true code}\} \{\text{false code}\}}

Converts both \textlangle token lists \rangle to \textlangle strings \rangle and tests whether \textlangle string2 \rangle is found inside \textlangle string1 \rangle.
Compares the (test string) in turn with each of the (string cases) (all token lists are converted to strings). If the two are equal (as described for \str_if_eq:nnTF) 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 \str_case:nn, which does nothing if there is no match, is also available.

\str_case_e:nn ⋆
\str_case_e:nnTF ⋆
\str_map_function:NN ⋆
\str_map_function:CN ⋆
\str_map_function:nN ⋆

5 Mapping to strings

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.
\str_map_inline:Nn \str_map_inline:cn
\texttt{\str_map_inline:Nn (str var) \{inline function\}}
Applies the \{inline function\} to every \{character\} in the \{str var\} including spaces. The \{inline function\} should consist of code which receives the \{character\} as #1. See also \str_map_function:NN.

\str_map_variable:NNn \str_map_variable:cNn
\texttt{\str_map_variable:NNn (str var) \{variable\} \{code\}}
Stores each \{character\} of the \{string\} (including spaces) in turn in the (string or 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 \{character\} in the \{string\}, or its original value if the \{string\} is empty. See also \str_map_inline:Nn.

\str_map_variable:NNn \str_map_variable:cn
\texttt{\str_map_variable:NNn \{token list\} \{variable\} \{code\}}
Converts the \{token list\} to a \{string\} then stores each \{character\} in the \{string\} (including spaces) in turn in the (string or 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 \{character\} in the \{string\}, or its original value if the \{string\} is empty. See also \str_map_inline:Nn.

\str_map_break:
\texttt{\str_map_break:}
Used to terminate a \str_map_... function before all characters in the \{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{\TeX hackers 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.
Used to terminate a `\str_map_inline:Nn \l_my_str`
{\str_if_eq:nnT { #1 } { bingo }
\{ \str_map_break:n { <code> } \}
% Do something useful}

Use outside of a `\str_map_inline:` scenario leads to low level \TeX{ errors.

\textbf{\LaTeX{} hackers 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.

## 6 Working with the content of strings

`\str_use:N` Recover the content of a `(str var)` 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 `(str)` directly without an accessor function.

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

`\str_count_spaces:n` Leaves in the input stream the number of space characters in the string representation of `(token list)`, as an integer denotation. Of course, this function has no `\str_count_spaces:n` variant.
\texttt{\texttt{\str_head:N}} \* \str_head:n \{\langle \texttt{token list} \rangle \}
\str_head:c \* \str_head:n
\str_head:n
\str_head_ignore_spaces:n \*

Revised: 2015-09-18

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

\begin{verbatim}
\str_tail:N \*
\str_tail:c \*
\str_tail:n \*
\str_tail_ignore_spaces:n *

Revised: 2015-09-18
\end{verbatim}

Converts the \langle \texttt{token list} \rangle to a \langle \texttt{string} \rangle, 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{\str_tail:N} and \texttt{\str_tail:n} only trim that space, while \texttt{\str_tail_ignore_spaces:n} removes the first non-space character and any space before it. If the \langle \texttt{token list} \rangle is empty (or blank in the case of the \_ignore_spaces variant), then nothing is left on the input stream.

\begin{verbatim}
\str_item:nn \*
\str_item:nn \*
\str_item_ignore_spaces:nn *

Revised: 2015-09-18
\end{verbatim}

Converts the \langle \texttt{token list} \rangle to a \langle \texttt{string} \rangle, and leaves in the input stream the character in position \langle \texttt{integer expression} \rangle of the \langle \texttt{string} \rangle, starting at 1 for the first (left-most) character. In the case of \texttt{\str_item:Nn} and \texttt{\str_item:nn}, all characters including spaces are taken into account. The \texttt{\str_item_ignore_spaces:nn} function skips spaces when counting characters. If the \langle \texttt{integer expression} \rangle is negative, characters are counted from the end of the \langle \texttt{string} \rangle. Hence, \(-1\) is the right-most character, \textit{etc.}
\texttt{\textbackslash str\_range:nnn \{}\textit{token list}\} \{\textit{start index}\} \{\textit{end index}\}
\texttt{\textbackslash str\_range:nnn *
\texttt{\textbackslash str\_range:nnn *
\texttt{\textbackslash str\_range:nnn *
\texttt{\textbackslash str\_range\_ignore\_spaces:nnn *}

Converts the \textit{(token list)} to a \textit{(string)}, and leaves in the input stream the characters from the \textit{(start index)} to the \textit{(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 \textit{(start index)} and \textit{(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 \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 \( M \) 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}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ 2 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ -4 \} \{ -1 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ -2 \} \{ -1 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ 0 \} \{ -1 \} \}
\end{verbatim}

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

\begin{verbatim}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ 5 \} \{ 2 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdef \} \{ -1 \} \{ -4 \} \}
\end{verbatim}

both yield empty strings.

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

\begin{verbatim}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdefg \} \{ 2 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdefg \} \{ 2 \} \{ -3 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdefg \} \{ -6 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abcdefg \} \{ -6 \} \{ -3 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abc-defg \} \{ 2 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abc-defg \} \{ 2 \} \{ -3 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abc-defg \} \{ -6 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range:nnn \{} abc-defg \} \{ -6 \} \{ -3 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range\_ignore\_spaces:nnn \{} abcdefg \} \{ 2 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range\_ignore\_spaces:nnn \{} abcdefg \} \{ 2 \} \{ -3 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range\_ignore\_spaces:nnn \{} abcdefg \} \{ -6 \} \{ 5 \} \}
\texttt{iow\_term:x \{ \textbackslash str\_range\_ignore\_spaces:nnn \{} abcdefg \} \{ -6 \} \{ -3 \} \}
\end{verbatim}

68
\io_term:x { \str_range_ignore_spaces:nnn { abcd-efg } { 2 } { 5 } }
\io_term:x { \str_range_ignore_spaces:nnn { abcd-efg } { 2 } { -3 } }
\io_term:x { \str_range_ignore_spaces:nnn { abcd-efg } { -6 } { 5 } }
\io_term:x { \str_range_ignore_spaces:nnn { abcd-efg } { -6 } { -3 } }

will print four instances of bcde, four instances of bc e and eight instances of bcde.

7 String manipulation

\str_lowercase:n \str_lowercase:f \str_uppercase:n \str_uppercase:f

Converts the input \textit{tokens} to their string representation, as described for \tl_to_str:n, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file \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

\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 }
}

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

• Caseless comparisons: use \str_foldcase:n for this situation (case folding is distinct from lower casing).

• 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.

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

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

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by \str_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 \str_foldcase:n follows the “full” scheme defined by the Unicode Consortium (e.g. SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (i.e. I always folds to i and not to ĩ).

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

8 Viewing strings

\str_show:N \langle str var\rangle

Displays the content of the \langle str var\rangle on the terminal.

\str_log:N \langle str var\rangle

Writes the content of the \langle str var\rangle in the log file.
9 Constant token lists

\c_\_ampersand\_str
\c_\_atsign\_str
\c_\_backslash\_str
\c_\_left\_brace\_str
\c_\_right\_brace\_str
\c_\_circumflex\_str
\c_\_colon\_str
\c_\_dollar\_str
\c_\_hash\_str
\c_\_percent\_str
\c_\_tilde\_str
\c_\_underscore\_str

New: 2015-09-19

10 Scratch strings

\l_\_tmpa\_str
\l_\_tmpb\_str

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

\g_\_tmpa\_str
\g_\_tmpb\_str

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Part IX
The i3str-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.\footnote{Encodings and escapings will be added as they are requested.}
- 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.\footnote{Encodings and escapings will be added as they are requested.}

2 Conversion functions

\begin{verbatim}
\str_set_convert:Nnnn \str_gset_convert:Nnnn
\end{verbatim}
This function converts the \texttt{⟨string⟩} from the encoding given by \texttt{⟨name 1⟩} to the encoding given by \texttt{⟨name 2⟩}, and stores the result in the \texttt{⟨str var⟩}. Each \texttt{⟨name⟩} can have the form \texttt{⟨encoding⟩} or \texttt{⟨encoding⟩/⟨escaping⟩}, where the possible values of \texttt{⟨encoding⟩} and \texttt{⟨escaping⟩} are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty \texttt{⟨name⟩} indicates the use of “native” strings, 8-bit for pdf\TeX{}, and Unicode strings for the other two engines.

For example,

\begin{verbatim}
\str_set_convert:Nnnn \lfoo_str { Hello! } { } { utf16/hex }
\end{verbatim}
results in the variable \texttt{\lfoo_str} holding the string FEFF00480065006C006C0021.
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 \texttt{⟨string⟩} is not valid according to the \texttt{⟨escaping 1⟩} and \texttt{⟨encoding 1⟩}, or if it cannot be reencoded in the \texttt{⟨encoding 2⟩} and \texttt{⟨escaping 2⟩} (for instance, if a character does not exist in the \texttt{⟨encoding 2⟩}). 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 \texttt{⟨encoding 2⟩}, or an encoding-specific replacement character, or the question mark character.
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\_set\_convert:NnnnTF \{str var\} \{\langle string\rangle\} \{(name 1)\} \{(name 2)\} \{(true code)\} \{(false code)\}}

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

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

### 4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In X\TeX/Lu\TeX, would it be better to use the \texttt{\^\^\^\^\^\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{\textbackslash 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{\textbackslash 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 \^!\^*\^+-./\^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 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{align*}
\texttt{\seq_new:N} & \quad \texttt{\seq_new:N \langle \textit{sequence} \rangle} \\
\texttt{\seq_new:c} & \quad \texttt{\seq_new:c} \\
\texttt{\seq_clear:N} & \quad \texttt{\seq_clear:N \langle \textit{sequence} \rangle} \\
\texttt{\seq_gclear:N} & \quad \texttt{\seq_gclear:N} \\
\texttt{\seq_clear_new:N} & \quad \texttt{\seq_clear_new:N \langle \textit{sequence} \rangle} \\
\texttt{\seq_gclear_new:N} & \quad \texttt{\seq_gclear_new:N} \\
\texttt{\seq_set_eq:NN} & \quad \texttt{\seq_set_eq:NN \langle \textit{sequence}_1 \rangle \langle \textit{sequence}_2 \rangle} \\
\texttt{\seq_set_eq:Nn} & \quad \texttt{\seq_set_eq:Nn \langle \textit{sequence}_1 \rangle \langle \textit{sequence}_2 \rangle} \\
\texttt{\seq_gset_eq:NN} & \quad \texttt{\seq_gset_eq:NN \langle \textit{sequence}_1 \rangle \langle \textit{sequence}_2 \rangle} \\
\texttt{\seq_gset_eq:Nn} & \quad \texttt{\seq_gset_eq:Nn \langle \textit{sequence}_1 \rangle \langle \textit{sequence}_2 \rangle} \\
\texttt{\seq_set_from_clist:NN} & \quad \texttt{\seq_set_from_clist:NN \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_set_from_clist:cN} & \quad \texttt{\seq_set_from_clist:cN \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_set_from_clist:Nn} & \quad \texttt{\seq_set_from_clist:Nn \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_gset_from_clist:NN} & \quad \texttt{\seq_gset_from_clist:NN \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_gset_from_clist:cN} & \quad \texttt{\seq_gset_from_clist:cN \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_gset_from_clist:Nn} & \quad \texttt{\seq_gset_from_clist:Nn \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle} \\
\texttt{\seq_gset_from_clist:cn} & \quad \texttt{\seq_gset_from_clist:cn \langle \textit{sequence} \rangle \langle \textit{comma list} \rangle}
\end{align*}

\textit{New: 2014-07-17}

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

Clears all items from the \langle \textit{sequence} \rangle.

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

Sets the content of \langle \textit{sequence}_1 \rangle equal to that of \langle \textit{sequence}_2 \rangle.

Converts the data in the \langle \textit{comma list} \rangle into a \langle \textit{sequence} \rangle: the original \langle \textit{comma list} \rangle is unchanged.
\seq_const_from_clist:Nn
\seq_const_from_clist:cn

New: 2017-11-28

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

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

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

\seq_if_exist_p:N
\seq_if_exist:NTF
\seq_if_exist:c
\seq_if_exist:TF
\seq_if_exist_p:c
\seq_if_exist:N
\seq_if_exist:CT
\seq_if_exist:CT

New: 2012-03-03

2 Appending data to sequences

\seq_put_left:Nn
\seq_put_left:(NV|NV|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|NV|No|Nx|cn|cV|cv|co|cx)

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_put_right:Nn
\seq_put_right:(NV|NV|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|NV|No|Nx|cn|cV|cv|co|cx)

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, \textit{i.e.} setting the \(token\ list\ variable\) used with \texttt{\tl_set:Nn} and \textit{never} \texttt{\tl_gset:Nn}.
\seq_get_left:NN \seq_get_left:cN
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 \seq_get_right:cN
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 \seq_pop_left:cN
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 \seq_gpop_left:cN
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 \seq_pop_right:cN
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 \seq_gpop_right:cN
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 \seq_item:cn
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.

\TeX hackers note: 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_var
\seq_rand_item:c

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 \TeX.

\textbf{\TeXhackersnote}: 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 \texttt{\token_list \var \{ \texttt{true code} \} \{ \texttt{false code} \}}

If the \textit{sequence} is empty, leaves the \texttt{false code} in the input stream. The value of the \texttt{\token_list \var} 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 \texttt{\token_list \var} without removing it from the \textit{sequence}, then leaves the \texttt{true code} in the input stream. The \texttt{\token_list \var} is assigned locally.

\seq_get_left:NN \seq_get_left:cN

\seq_get_right:NNTF \texttt{\token_list \var \{ \texttt{true code} \} \{ \texttt{false code} \}}

If the \textit{sequence} is empty, leaves the \texttt{false code} in the input stream. The value of the \texttt{\token_list \var} 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 \texttt{\token_list \var} without removing it from the \textit{sequence}, then leaves the \texttt{true code} in the input stream. The \texttt{\token_list \var} is assigned locally.

\seq_get_right:NN \seq_get_right:cN

\seq_pop_left:NNTF \texttt{\token_list \var \{ \texttt{true code} \} \{ \texttt{false code} \}}

If the \textit{sequence} is empty, leaves the \texttt{false code} in the input stream. The value of the \texttt{\token_list \var} 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 \texttt{\token_list \var}, i.e. removes the item from the \textit{sequence}, then leaves the \texttt{true code} in the input stream. Both the \textit{sequence} and the \texttt{\token_list \var} are assigned locally.

\seq_pop_left:NN \seq_pop_left:cN

\seq_gpop_left:NNTF \texttt{\token_list \var \{ \texttt{true code} \} \{ \texttt{false code} \}}

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

\textbullet \textlangle \textit{sequence}\textrangle \textlangle \textit{item}\textrangle
\textlangle \textit{item}\textrangle comparison takes place on a token basis, as for \texttt{\textbackslash \texttt{tl_if_eq:nnTF}}.

\texttt{\textbackslash seq_sort:Nn} \texttt{(sequence)} \texttt{(comparison code)}
Sorts the items in the \texttt{\textlangle \textit{sequence}\textrangle} according to the \texttt{\textlangle \textit{comparison code}\textrangle}, and assigns the result to \texttt{\textlangle \textit{sequence}\textrangle}. The details of sorting comparison are described in Section 1.
\seq_shuffle:N (seq var)
Sets the \langle seq var \rangle to the result of placing the items of the \langle seq var \rangle in a random order. Each item is (roughly) as likely to end up in any given position.

**TeXhackers 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:N (sequence)
\seq_if_empty_p:c * \seq_if_empty:NTF (sequence) \{\langle true code \rangle\} \{\langle false code \rangle\}
Tests if the \langle sequence \rangle is empty (containing no items).

\seq_if_in:NnTF (sequence) \{\langle item \rangle\} \{\langle true code \rangle\} \{\langle false code \rangle\}
Tests if the \langle item \rangle is present in the \langle sequence \rangle.

7 Mapping to sequences

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.

\seq_map_function:NN ★ \seq_map_function:NN (sequence) \{\langle function \rangle\}
Applies \langle function \rangle to every \langle item \rangle stored in the \langle sequence \rangle. The \langle function \rangle will receive one argument for each iteration. The \langle items \rangle are returned from left to right. To pass further arguments to the \langle function \rangle, 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:Nn (sequence) \{\langle inline function \rangle\}
Applies \langle inline function \rangle to every \langle item \rangle stored within the \langle sequence \rangle. The \langle inline function \rangle should consist of code which will receive the \langle item \rangle as #1. The \langle items \rangle 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 \langle code \rangle receives each item in the \langle sequence \rangle 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 \langle sequence \rangle: for each item in \l_my_seq the function \prg_replicate:nn receives 2 and \langle item \rangle 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|cNn|ccn)

Stores each \langle item \rangle of the \langle sequence \rangle in turn in the (token list) \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 sequence \rangle, or its original value if the \langle sequence \rangle is empty. The \langle items \rangle are returned from left to right.

\seq_map_break:

Used to terminate a \seq_map... function before all entries in the \langle sequence \rangle 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 \{ \langle code \rangle \}

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

\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... scenario leads to low level \TeX errors.

\textbf{\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.

\seq_count:N \langle sequence \rangle

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 \langle seq var \rangle \{ \langle separator between two \rangle \} \{ \langle separator between more than two \rangle \} \{ \langle separator between final two \rangle \}

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 \{ \langle separator between two \rangle \} \{ \langle separator between more than two \rangle \} \{ \langle separator between final two \rangle \}

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{\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.

82
\texttt{\seq_use:Nn} \langle \text{seq var} \rangle \{ \langle \text{separator} \rangle \}

Places the contents of the \langle \text{seq var} \rangle in the input stream, with the \langle \text{separator} \rangle between the items. If the sequence has a single item, it is placed in the input stream with no \langle \text{separator} \rangle, 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 { ~and~ }
\end{verbatim}

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

\textbf{\TeX}hackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \langle \text{items} \rangle 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.

\texttt{\seq_get:NN} \langle \text{sequence} \rangle \langle \text{token list variable} \rangle

Reads the top item from a \langle sequence \rangle into 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 \texttt{\q_no_value}.

\texttt{\seq_pop:NN} \langle \text{sequence} \rangle \langle \text{token list variable} \rangle

Pops the top item from a \langle sequence \rangle into 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 \texttt{\q_no_value}.

\texttt{\seq_gpop:NN} \langle \text{sequence} \rangle \langle \text{token list variable} \rangle

Pops the top item from a \langle sequence \rangle into the \langle token list variable \rangle. The \langle sequence \rangle is modified globally, while 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 \texttt{\q_no_value}.

\texttt{\seq_get:NNTF} \langle \text{sequence} \rangle \langle \text{token list variable} \rangle \{ \langle \text{true code} \rangle \} \{ \langle \text{false code} \rangle \}

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

\begin{align*}
\text{\texttt{\textbackslash seq\_pop:NNTF}} & \quad \text{\texttt{\textbackslash seq\_pop:cNTF}} \\
\text{\texttt{\textbackslash seq\_gpop:NNTF}} & \quad \text{\texttt{\textbackslash seq\_gpop:cNTF}}
\end{align*}

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

\begin{align*}
\text{\texttt{\textbackslash seq\_push:Nn}} & \quad \text{\texttt{\textbackslash seq\_push:NV|Nv|No|Nx|cn|cV|cv|co|cx}} \\
\text{\texttt{\textbackslash seq\_gpush:NN}} & \quad \text{\texttt{\textbackslash seq\_gpush:NV|Nv|No|Nx|cn|cV|cv|co|cx}}
\end{align*}

Adds the ⟨item⟩ to the top of the ⟨sequence⟩.

## 10 Sequences as sets

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

Sets should not contain several occurrences of a given item. To make sure that a ⟨sequence variable⟩ only has distinct items, use \texttt{\textbackslash seq\_remove\_duplicates:N N ⟨sequence variable⟩}. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set ⟨seq var⟩ are straightforward. For instance, \texttt{\textbackslash seq\_count:N ⟨seq var⟩} expands to the number of items, while \texttt{\textbackslash seq\_if\_in:NnTF ⟨seq var⟩ ⟨{item}⟩} tests if the ⟨item⟩ is in the set.

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

\begin{align*}
\text{\texttt{\textbackslash seq\_if\_in:NnF ⟨seq var⟩ ⟨{item}⟩}} \\
\quad \text{\{ \texttt{\textbackslash seq\_put\_right:Nn ⟨seq var⟩ ⟨{item}⟩} \}}
\end{align*}

Removing an ⟨item⟩ from a set ⟨seq var⟩ can be done using \texttt{\textbackslash seq\_remove\_all:Nn},

\begin{align*}
\text{\texttt{\textbackslash seq\_remove\_all:Nn ⟨seq var⟩ ⟨{item}⟩}}
\end{align*}

The intersection of two sets ⟨seq var₁⟩ and ⟨seq var₂⟩ can be stored into ⟨seq var₃⟩ by collecting items of ⟨seq var₁⟩ which are in ⟨seq var₂⟩.
\seq_clear:N \seq_var_3
\seq_map_inline:Nn \seq_var_1
{
\seq_if_in:NnT \seq_var_2 {#1}
{ \seq_put_right:Nn \seq_var_3 {#1} }
}
The code as written here only works if \seq_var_3 is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence \l__\pkg\_internal_seq, then \seq_var_3 should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 through
\seq_concat:NNN \seq_var_3 \seq_var_1 \seq_var_2
\seq_remove_duplicates:N \seq_var_3
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__\pkg\_internal_seq, then the difference between \seq_var_2 and \seq_var_1, and finally concatenating the two differences to get the symmetric differences.
\seq_set_eq:NN \l__\pkg\_internal_seq \seq_var_1
\seq_map_inline:Nn \seq_var_2
{ \seq_remove_all:Nn \l__\pkg\_internal_seq {#1} }
\seq_set_eq:NN \seq_var_3 \seq_var_2
\seq_map_inline:Nn \seq_var_1
{ \seq_remove_all:Nn \seq_var_3 {#1} }
\seq_concat:NNN \seq_var_3 \seq_var_3 \l__\pkg\_internal_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 \LaTeX3-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 \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_seq  
\g_tmpb_seq  
New: 2012-04-26

12 Viewing sequences

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

\seq_log:N  \seq_log:N \langle sequence \rangle  
\seq_log:c
 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

\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 − followed by such a sequence, and 0 for zero. The ⟨integer expression⟩ should consist, after expansion, of +, −, *, /, (, ) and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- / 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 \(2^{31} - 1\), except in the case of scaling operations \(a*b/c\), for which \(a*b\) may be arbitrarily large;
- parentheses may not appear after unary + or −, namely placing +( or −( at the start of an expression or after +, −, *, / or ) leads to an error.

Each integer operand can be either an integer variable (with no need for \int_use:N) or an integer denotation. For example both

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

and

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

evaluate to −6 because \l_my_tl expands to the integer denotation 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 \exp_not:n and its variants have no effect while \exp_not:N may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate \int_eval:n. The result is not an ⟨internal integer⟩, and therefore requires suitable termination if used in a \TeX-style integer assignment.

As all \TeX integers, integer operands can also be dimension or skip variables, converted to integers in sp, or octal numbers given as ’ followed by digits other than 8 and 9, or hexadecimal numbers given as " 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 ’⟨char⟩.
\begin{verbatim}
\int_eval:w * \int_eval:w \langle \text{integer expression} \rangle
Evaluates the \langle \text{integer expression} \rangle 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 \texttt{\textbackslash \texttt{scan\_stop}}: it is removed, otherwise not. Spaces do \texttt{not} terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, \texttt{\int_eval:w 1\_u+1\_u9} expands to \texttt{29} since the digit \texttt{9} is not part of the expression.

\int_sign:n * \int_sign:n \{\langle \text{integer} \rangle\}
Evaluates the \langle \text{integer expression} \rangle then leaves 1 or 0 or –1 in the input stream according to the sign of the result.

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

\int_div_round:nn * \int_div_round:nn \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\}
Evaluates the two \langle \text{integer expressions} \rangle 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 \langle \text{integer expression} \rangle. The result is left in the input stream as an \langle \text{integer denotation} \rangle after two expansions.

\int_div_truncate:nn * \int_div_truncate:nn \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\}
Evaluates the two \langle \text{integer expressions} \rangle 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 \langle \text{integer denotation} \rangle after two expansions.

\int_max:nn \int_max:nn \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\}
\int_min:nn \int_min:nn \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\}
Evaluates the \langle \text{integer expressions} \rangle as described for \int_eval:n and leaves either the larger or smaller value in the input stream as an \langle \text{integer denotation} \rangle after two expansions.

\int_mod:nn \int_mod:nn \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\}
Evaluates the two \langle \text{integer expressions} \rangle 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 \{\langle \text{integer expression} \rangle\} \{\langle \text{integer expression} \rangle\} times \langle \text{integer expression} \rangle \rangle from \langle \text{integer expression} \rangle. Thus, the result has the same sign as \langle \text{integer expression} \rangle and its absolute value is strictly less than that of \langle \text{integer expression} \rangle. The result is left in the input stream as an \langle \text{integer denotation} \rangle after two expansions.
\end{verbatim}

2 Creating and initialising integers

\int_new:N \int_new:N \langle \text{integer} \rangle
Creates a new \langle \text{integer} \rangle or raises an error if the name is already taken. The declaration is global. The \langle \text{integer} \rangle is initially equal to 0.
\int_const:Nn \int_const:cn
Updated: 2011-10-22
\int_const:Nn (integer) \{(integer expression)\}
Creates a new constant \textit{(integer)} or raises an error if the name is already taken. The value of the \textit{(integer)} is set globally to the \textit{(integer expression)}.

\int_zero:N \int_zero:c
\int_gzero:N \int_gzero:c
\int_zero_new:N \int_zero_new:c
\int_gzero_new:N \int_gzero_new:c
New: 2011-10-22
\int_zero:N (integer)
Sets \textit{(integer)} to 0.

\int_zero_new:N (integer)
Ensures that the \textit{(integer)} exists globally by applying \int_new:N if necessary, then applies \int_(g)zero:N to leave the \textit{(integer)} set to zero.

\int_set_eq:NN \int_set_eq:c \int_gset_eq:NN \int_gset_eq:c
\int_if_exist_p:N \int_if_exist:p \int_if_exist:NTF \int_if_exist:cc
\int_if_exist_p:c \int_if_exist:NTF \int_if_exist:cc
New: 2011-10-22
\int_set_eq:NN (integer_1) (integer_2)
Sets the content of \textit{(integer_1)} equal to that of \textit{(integer_2)}.

\int_if_exist_p:N \int_if_exist:p \int_if_exist:NTF \int_if_exist:cc
Tests whether the \textit{(int)} is currently defined. This does not check that the \textit{(int)} really is an integer variable.

3 Setting and incrementing integers

\int_add:Nn \int_add:cn \int_gadd:Nn \int_gadd:cn
Updated: 2011-10-22
\int_add:Nn (integer) \{(integer expression)\}
Adds the result of the \textit{(integer expression)} to the current content of the \textit{(integer)}.

\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_decr:N (integer)
Decreases the value stored in \textit{(integer)} by 1.

\int_incr:N (integer)
Increases the value stored in \textit{(integer)} by 1.
\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}

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

\section{Integer expression conditionals}

\texttt{int_compare_p:nNn} * \texttt{int_compare:nNnTF} *

\texttt{int_compare_p:nNn} \{\textit{intexpr}_1\} \{\textit{relation}\} \{\textit{intexpr}_2\}

\texttt{int_compare:nNnTF} \{\textit{intexpr}_1\} \{\textit{relation}\} \{\textit{intexpr}_2\}

\{\textit{true code}\} \{\textit{false code}\}

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.
This function evaluates the \textit{integer expressions} as described for \texttt{\textbackslash int\_eval\_n} and compares consecutive result using the corresponding \textit{relation}, namely it compares \texttt{\textbackslash intexpr\_1} and \texttt{\textbackslash intexpr\_2} using the \texttt{\textbackslash relation\_1}, then \texttt{\textbackslash intexpr\_2} and \texttt{\textbackslash intexpr\_3} using the \texttt{\textbackslash relation\_2}, until finally comparing \texttt{\textbackslash intexpr\_N} and \texttt{\textbackslash intexpr\_N+1} using the \texttt{\textbackslash relation\_N}. The test yields \texttt{true} if all comparisons are \texttt{true}. Each \textit{integer expression} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \texttt{false}, then no other \textit{integer expression} is evaluated and no other comparison is performed. The \textit{relations} can be any of the following:

\begin{itemize}
  \item Equal \hspace{1cm} = \text{ or } ==
  \item Greater than or equal to \hspace{1cm} >=
  \item Greater than \hspace{1cm} >
  \item Less than or equal to \hspace{1cm} <=
  \item Less than \hspace{1cm} <
  \item Not equal \hspace{1cm} !=
\end{itemize}

This function is more flexible than \texttt{\textbackslash int\_compare\_nNnTF} but around 5 times slower.
This function evaluates the \( \text{test integer expression} \) and compares this in turn to each of the \( \text{integer expression cases} \). If the two are equal then the associated \( \text{code} \) is left in the input stream and other cases are discarded. If any of the cases are matched, the \( \text{true code} \) is also inserted into the input stream (after the code for the appropriate case), while if none match then the \( \text{false code} \) is inserted. The function \( \texttt{int\_case:nn} \), which does nothing if there is no match, is also available. For example

\[
\texttt{int\_case:nnF}\{ 2 \times 5 \}
\]

\[
\{ 5 \} \quad \{ \text{Small} \}
\{ 4 + 6 \} \quad \{ \text{Medium} \}
\{ -2 \times 10 \} \quad \{ \text{Negative} \}
\{ \text{No idea!} \}
\]

leaves “Medium” in the input stream.

---

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

### 6 Integer expression loops

\[
\texttt{int\_do\_until:nNnn} \quad \texttt{int\_do\_while:nNnn} \quad \texttt{int\_eval:n}\{\text{intexpr}_1\}\{relation\}\{\text{intexpr}_2\}\{\text{code}\}
\]

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

\[
\texttt{int\_do\_while:nNnn} \quad \texttt{int\_do\_while:nNnn}\{\text{intexpr}_1\}\{relation\}\{\text{intexpr}_2\}\{\text{code}\}
\]

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

93
\int_until_do:nNnn \{⟨intexpr1⟩\} ⟨relation⟩ {⟨intexpr2⟩} {⟨code⟩}
Evaluates the relationship between the two ⟨integer expressions⟩ as described for \int_compare:nNnTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is false. After the ⟨code⟩ has been processed by \TeX{} the test is repeated, and a loop occurs until the test is true.

\int_while_do:nNnn \{⟨intexpr1⟩\} ⟨relation⟩ {⟨intexpr2⟩} {⟨code⟩}
Evaluates the relationship between the two ⟨integer expressions⟩ as described for \int_compare:nNnTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is true. After the ⟨code⟩ has been processed by \TeX{} the test is repeated, and a loop occurs until the test is false.

\int_do_until:nn ⟨integer relation⟩ {⟨code⟩}
Places the ⟨code⟩ in the input stream for \TeX{} to process, and then evaluates the ⟨integer relation⟩ as described for \int_compare:nTF. If the test is false then the ⟨code⟩ is inserted into the input stream again and a loop occurs until the ⟨relation⟩ is true.

\int_do_while:nn ⟨integer relation⟩ {⟨code⟩}
Places the ⟨code⟩ in the input stream for \TeX{} to process, and then evaluates the ⟨integer relation⟩ as described for \int_compare:nTF. If the test is true then the ⟨code⟩ is inserted into the input stream again and a loop occurs until the ⟨relation⟩ is false.

\int_until_do:nn ⟨integer relation⟩ {⟨code⟩}
Evaluates the ⟨integer relation⟩ as described for \int_compare:nNnTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is false. After the ⟨code⟩ has been processed by \TeX{} the test is repeated, and a loop occurs until the test is true.

\int_while_do:nn ⟨integer relation⟩ {⟨code⟩}
Evaluates the ⟨integer relation⟩ as described for \int_compare:nNnTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is true. After the ⟨code⟩ has been processed by \TeX{} the test is repeated, and a loop occurs until the test is false.
7 Integer step functions

\int_step_function:nN \{\text{final value}\} \{\text{function}\}
\int_step_function:nnN \{\text{initial value}\} \{\text{final value}\} \{\text{function}\}
\int_step_function:nnnN \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{function}\}

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

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

would print

\begin{verbatim}
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
\end{verbatim}

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

\int_step_inline:nn \{\text{final value}\} \{\text{code}\}
\int_step_inline:nn \{\text{initial value}\} \{\text{final value}\} \{\text{code}\}
\int_step_inline:nn \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{code}\}

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

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

\int_step_variable:nN \{\text{final value}\} \{tl var\} \{\text{code}\}
\int_step_variable:nN \{\text{initial value}\} \{\text{final value}\} \{tl var\} \{\text{code}\}
\int_step_variable:nNn \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{tl var\} \{\text{code}\}

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

The functions \int_step_variable:nN and \int_step_variable:nNn both use a fixed \{step\} of 1, and in the case of \int_step_variable:nN the \{initial value\} 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.

\[ \text{\texttt{\textbackslash int\_to\_arabic:n \{integer expression\}}} \]
Places the value of the \( \text{\texttt{\textbackslash int\_to\_arabic:n \{integer expression\}}} \) in the input stream as digits, with category code 12 (other).

\[ \text{\texttt{\textbackslash int\_to\_alph:n \{integer expression\}}} \]
Evaluates the \( \text{\texttt{\textbackslash int\_to\_alph:n \{integer expression\}}} \) 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

\[ \text{\texttt{\textbackslash int\_to\_alph:n \{ 1 \}}} \]
places \texttt{a} in the input stream,

\[ \text{\texttt{\textbackslash int\_to\_alph:n \{ 26 \}}} \]
is represented as \texttt{z} and

\[ \text{\texttt{\textbackslash int\_to\_alph:n \{ 27 \}}} \]
is converted to \texttt{aa}. For conversions using other alphabets, use \texttt{\textbackslash int\_to\_symbols:nnn} to define an alphabet-specific function. The basic \texttt{\textbackslash int\_to\_alph:n} and \texttt{\textbackslash 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).

\[ \text{\texttt{\textbackslash int\_to\_symbols:nnn \{(integer expression) \{(total symbols)\} \{(value to symbol mapping)\}}} \]
This is the low-level function for conversion of an \( \text{\texttt{\textbackslash int\_to\_symbols:nnn \{(integer expression) \{(total symbols)\} \{(value to symbol mapping)\}}} \) into a symbolic form (often letters). The \( \text{\texttt{\textbackslash int\_to\_symbols:nnn \{(integer expression) \{(total symbols)\} \{(value to symbol mapping)\}}} \) available should be given as an integer expression. Values are actually converted to symbols according to the \( \text{\texttt{\textbackslash int\_to\_symbols:nnn \{(integer expression) \{(total symbols)\} \{(value to symbol mapping)\}}} \) pairs of entries, a number and the appropriate symbol. Thus the \texttt{\textbackslash int\_to\_alph:n} function is defined as

\[ \text{\texttt{\textbackslash cs\_new:Npn \textbackslash int\_to\_alph:n \#1 \{ \text{\texttt{\textbackslash int\_to\_symbols:nnn \{#1\} \{total symbols\}\{value to symbol mapping\}}} \}}} \]
\[ \text{\texttt{\textbackslash int\_to\_symbols:nnn \{(integer expression) \{(total symbols)\} \{(value to symbol mapping)\}}} \]

\[ \text{\texttt{\textbackslash cs\_new:Npn \textbackslash int\_to\_alph:n \#1 \{ \text{\texttt{\textbackslash int\_to\_symbols:nnn \{#1\} \{26\}\{\{1\}\{a\}\{2\}\{b\}\{\ldots\}\{26\}\{z\}\}}}}} \]

} \]
\int_to_bin:n \ (\text{integer expression})

Calculates the value of the \textit{integer expression} and places the binary representation of the result in the input stream.

\int_to_hex:n \ (\text{integer expression})

Calculates the value of the \textit{integer expression} and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for \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).

\int_to_oct:n \ (\text{integer expression})

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

\int_to_base:nn \ (\text{integer expression}) \ {\text{\langle base\rangle}}

Calculates the value of the \textit{integer expression} and converts it into the appropriate representation in the \textit{\langle base\rangle}; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for \texttt{int_to_base:n} and upper case ones for \texttt{int_to_Base:n}. The maximum \textit{\langle base\rangle} value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\texttt{\textbf{\LaTeX}hackers note:}\ This is a generic version of \texttt{int_to_bin:n}, \textit{etc.}

\int_to_roman:n \ (\text{integer expression})

Places the value of the \textit{integer expression} 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}.

\textbf{9 Converting from other formats to integers}

\int_from_alph:n \ \{\text{\langle letters\rangle}\}

Converts the \textit{\langle letters\rangle} into the integer (base 10) representation and leaves this in the input stream. The \textit{\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}.
\int_from_bin:n\star

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

\int_from_hex:n\star

Converting the \textit{hexadecimal number} \texttt{⟨hexadecimal number⟩} into the integer (base 10) representation and leaves this in the input stream. Digits greater than \texttt{9} may be represented in the \textit{hexadecimal number} by upper or lower case letters. The \textit{hexadecimal number} \texttt{⟨hexadecimal number⟩} is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and –. This is the inverse function of \texttt{\int_to_hex:n} and \texttt{\int_to_Hex:n}.

\int_from_oct:n\star

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

\int_from_roman:n\star

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

\int_from_base:nn\star

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

\int_rand:nn\star

Evaluating the two \textit{integer expressions} \texttt{⟨integer expressions⟩} and producing a pseudo-random number between the two (with bounds included). This is not available in older versions of X\TeX{}.

\int_rand:n\star

Evaluating the \textit{integer expression} \texttt{⟨integer expression⟩} then produces a pseudo-random number between \texttt{1} and the \texttt{⟨integer expression⟩} (included). This is not available in older versions of X\TeX{}.

10 Random integers
11 Viewing integers

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

\int_show:c \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
\langle integer expression \rangle
 Writes the value of the \langle integer \rangle in the log file.

\int_log:n \langle integer expression \rangle
 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

\c_max_int
The maximum value that can be stored as an integer.

\c_max_register_int
Maximum number of registers.

\c_max_char_int
Maximum character code completely supported by the engine.

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

\texttt{\textbackslash int\_value: w \{integer\}}
\texttt{\textbackslash int\_value: w \{integer denotation\} \{optional space\}}

Expands the following tokens until an \{integer\} 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 \{integer\} 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 \{octal digits\} or \{hexadecimal digits\} or \{character\}).

In this last case expansion stops once a non-digit is found; if that is a space it is removed
as in \texttt{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.

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

14 Primitive conditionals

\texttt{\if\_int\_compare: w \{integer\}_1 \{relation\} \{integer\}_2}
\texttt{\texttt{\{true code\}}}
\texttt{\else:}
\texttt{\{false code\}}
\texttt{\texttt{\fi:}}

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

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

\texttt{\if\_case: w \{integer\} \{case\}_0}
\texttt{\texttt{\or: \{case\}_1}}
\texttt{\texttt{\or: \ldots}}
\texttt{\else: \{default\}}
\texttt{\fi:}

Selects a case to execute based on the value of the \{integer\}. The first case ((\{case\}_0)) is
executed if \{integer\} is 0, the second ((\{case\}_1)) if the \{integer\} is 1, etc. The \{integer\}
may be a literal, a constant or an integer expression (\textit{e.g.} using \texttt{\int\_eval:n}).

\TeX{}hackers note: These are the \TeX{} primitives \texttt{\ifcase} and \texttt{\or.}
\if_int_odd:w \if_int_odd:w \tokens \optional \space
\true \code \else:
\true \code \fi:

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

\TeXhackers \note: This is the \TeX primitive \fodd.
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 \( \text{flag name} \) such as \texttt{str_missing}. The \( \text{flag name} \) 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 {\langle flag name\rangle}
\end{verbatim}

Creates a new flag with a name given by \( \text{flag name} \), or raises an error if the name is already taken. The \( \text{flag name} \) 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 {\langle flag name\rangle}
\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 {\langle flag name\rangle}
\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 {\langle flag name\rangle}
\end{verbatim}

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

\begin{verbatim}
\flag_log:n \flag_log:n {\langle flag name\rangle}
\end{verbatim}

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

\flag_if_exist:n \{flag name\}
\flag_if_exist:nTF
This function returns \texttt{true} if the \texttt{(flag name)} references a flag that has been defined previously, and \texttt{false} otherwise.

\flag_if_raised:n \{flag name\}
\flag_if_raised:nTF
This function returns \texttt{true} if the \texttt{(flag)} has non-zero height, and \texttt{false} if the \texttt{(flag)} has zero height.

\flag_height:n \{flag name\}
Expands to the height of the \texttt{(flag)} as an integer denotation.

\flag_raise:n \{flag name\}
The \texttt{(flag)}'s height is increased by 1 locally.
Part XIII

The \texttt{l3prg} package

Control structures

Conditional processing in \LaTeXe{}3 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 \textit{state} is returned. The states returned are \texttt{⟨true⟩} and \texttt{⟨false⟩}.

\LaTeXe{}3 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{\textbackslash 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{\textbackslash 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{\textbackslash cs\_if\_free:N\textbackslash TF} which also takes one argument (the \texttt{N}) and then executes either \texttt{true} or \texttt{false} depending on the result.

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

1 Defining a set of conditional functions

\begin{verbatim}
\prg_new_conditional:Npnn \prg_set_conditional:Npnn \prg_new_conditional:Nnn \prg_set_conditional:Nnn
\prg_new_protected_conditional:Npnn \prg_set_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \prg_set_protected_conditional:Nnn
\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 (\textit{cf.} \texttt{\textbackslash cs\_new:Npnn}) whereas the \texttt{set} versions do no check and perform assignments locally (\textit{cf.} \texttt{\textbackslash cs\_set:Npnn}). The conditionals created are dependant on the comma-separated list of \texttt{(conditions)}, which should be one or more of \texttt{p}, \texttt{T}, \texttt{F} and \texttt{TF}.

\begin{verbatim}
\prg_new_protected_conditional:Npnn \prg_set_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \prg_set_protected_conditional:Nnn
\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 (\textit{cf.} \texttt{\textbackslash cs\_new:Npnn}) whereas the \texttt{set} version do not (\textit{cf.} \texttt{\textbackslash cs\_set:Npnn}). The conditionals created are dependant on the comma-separated list of \texttt{(conditions)}, which should be one or more of \texttt{T}, \texttt{F} and \texttt{TF} (not \texttt{p}).

The conditionals are defined by \texttt{\textbackslash prg\_new\_conditional:Npnn} and friends as:
\( (\text{name})_p : \langle \text{arg spec} \rangle \) — 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.

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

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

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

The \( \langle \text{code} \rangle \) of the test may use \( \langle \text{parameters} \rangle \) as specified by the second argument to \texttt{\prg_set_conditional:Npnn}: this should match the \( \langle \text{argument specification} \rangle \) but this is not enforced. The \texttt{\Nnn} versions infer the number of arguments from the argument specification given (\textit{cf.} \texttt{\cs_new:Nn}, etc.). Within the \( \langle \text{code} \rangle \), the functions \texttt{\prg_return_true:} and \texttt{\prg_return_false:} are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

\begin{verbatim}
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{   \if_meaning:w \l_tmpa_tl #1
     \prg_return_true:
     \else:
       \if_meaning:w \l_tmpa_tl #2
         \prg_return_true:
         \else:
           \prg_return_false:
           \fi:
       \fi:
   }
\end{verbatim}

This defines the function \texttt{\foo_if_bar_p:NN}, \texttt{\foo_if_bar:NNTF} and \texttt{\foo_if_bar:NNT} but not \texttt{\foo_if_bar:NNF} (because \texttt{F} is missing from the \( \langle \text{conditions} \rangle \) list). The return statements take care of resolving the remaining \texttt{\else:} and \texttt{\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.

\begin{verbatim}
\prg_new_eq_conditional:NNn \prg_new_eq_conditional:NNn \prg_set_eq_conditional:NNn \langle \text{conditions} \rangle
\end{verbatim}

These functions copy a family of conditionals. The \texttt{\new} version checks for existing definitions (\textit{cf.} \texttt{\cs_new_eq:NN}) whereas the \texttt{\set} version does not (\textit{cf.} \texttt{\cs_set_eq:NN}). The conditionals copied are depended on the comma-separated list of \( \langle \text{conditions} \rangle \), which should be one or more of \texttt{p}, \texttt{T}, \texttt{F} and \texttt{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.

---

**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 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).

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

\begin{Verbatim}
\bool_new:N \bool_new:c
\end{Verbatim}

Creates a new \bool_ or raises an error if the name is already taken. The declaration is global. The \bool_ is initially false.
\bool_const:Nn
\bool_const:cn
New: 2017-11-28

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

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

\bool_set_eq:NN \bool_set_eq:cn(\text{cN}|c|cc) \bool_gset_eq:NN
\bool_gset_eq:(\text{cN}|c|cc)

\bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn
Updated: 2017-07-15

\bool_if:p:N \bool_if:p:c \bool_if:NTF \bool_if:TF
\bool_if:cf \bool_if:TF
Updated: 2017-07-15

\bool_show:N \bool_show:c
New: 2012-02-09
Updated: 2015-08-01

\bool_show:n
New: 2012-02-09
Updated: 2017-07-15

\bool_log:N \bool_log:c
New: 2014-08-22
Updated: 2015-08-03

\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 ⟨boolean⟩
Sets ⟨boolean⟩ logically false.

\bool_set_true:N ⟨boolean⟩
Sets ⟨boolean⟩ logically true.

\bool_set_eq:NN ⟨boolean⟩1 ⟨boolean⟩2
Sets ⟨boolean⟩1 to the current value of ⟨boolean⟩2.

\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 ⟨boolean⟩
\bool_if:p:c ⟨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 ⟨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 ⟨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.

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any \LTX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any \LTX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean \texttt{(true)} or \texttt{(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 \texttt{(true)} or \texttt{(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 \} \&\&
\begin{itemize}
  \item \int_compare_p:n \{ 2 = 3 \} ||
  \item \int_compare_p:n \{ 4 \leq 4 \} ||
  \item \str_if_eq_p:nn \{ abc \} \{ def \}
\end{itemize}
! \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.

\TeXhackers 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 \texttt{\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

\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.

\bool_if_p:n {⟨boolean expression⟩} \bool_if:nTF {⟨boolean expression⟩} {⟨true code⟩} {⟨false code⟩}
Tests the current truth of ⟨boolean expression⟩, and continues expansion based on this result. The ⟨boolean expression⟩ should consist of a series of predicates or boolean variables with the logical relationship between these defined using && ("And"), || ("Or"), ! ("Not") and parentheses. The logical Not applies to the next predicate or group.

\bool_lazy_all_p:n ★ \bool_lazy_all:nTF ★
\bool_lazy_all_p:n { {⟨boolexpr⟩1} {⟨boolexpr⟩2} ··· {⟨boolexpr⟩N} }
\bool_lazy_all:nTF { {⟨boolexpr⟩1} {⟨boolexpr⟩2} ··· {⟨boolexpr⟩N} } {⟨true code⟩} {⟨false code⟩}
Implements the “And” operation on the ⟨boolean expressions⟩, hence is true if all of them are true and false if any of them is false. Contrarily to the infix operator &&, only the ⟨boolean expressions⟩ which are needed to determine the result of \bool_lazy_all:nTF are evaluated. See also \bool_lazy_and:nnTF when there are only two ⟨boolean expressions⟩.

\bool_lazy_and_p:nn ★ \bool_lazy_and:nn★
\bool_lazy_and_p:nn { {⟨boolexpr⟩1} {⟨boolexpr⟩2} }
\bool_lazy_and:nnTF { {⟨boolexpr⟩1} {⟨boolexpr⟩2} } {⟨true code⟩} {⟨false code⟩}
Implements the “And” operation between two boolean expressions, hence is true if both are true. Contrarily to the infix operator &&, the ⟨boolexpr⟩2 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⟩.
\bool_lazy_or_p:n \* \bool_lazy_or:p:nn
\bool_lazy_or:n \* \bool_lazy_or:nn

Rev: 2015-11-15
Updated: 2017-07-15

\bool_lazy_any_p:n \* \bool_lazy_any:p:n \* \bool_lazy_any:n \* \bool_lazy_any:nn
\bool_lazy_any:nTF

Rev: 2015-11-15
Updated: 2017-07-15

\bool_while_do:cn \* \bool_while_do:Nn
\bool_until_do:cn \* \bool_until_do:Nn
\bool_do_while:cn \* \bool_do_while:Nn
\bool_do_until:cn \* \bool_do_until:Nn

Rev: 2015-11-15
Updated: 2017-07-15

\bool_xor:p:nn \* \bool_xor:nn
\bool_xor:nTF

Rev: 2015-11-15
Updated: 2017-07-15

\bool_not_p:n
\bool_not:n

Updated: 2017-07-15

\bool_xor:p:nn \* \bool_xor:p:nn \* \bool_xor:nTF
\bool_xor:nTF

Rev: 2018-06-09
Updated: 2017-07-15

\bool_do_until:Nn \* \bool_do_until:cn
\bool_until_do:Nn \* \bool_until_do:cn
\bool_while_do:Nn \* \bool_while_do:cn

Updated: 2017-07-15

\bool_lazy_or_p:nn { \{booleanexpr_1\} \{booleanexpr_2\} \ldots \{booleanexpr_N\} }
\bool_lazy_or:nTF { \{booleanexpr_1\} \{booleanexpr_2\} \ldots \{booleanexpr_N\} } \{\texttt{true code}\}
\bool_lazy_or:nnTF
\bool_lazy_or:nn
\bool_lazy_or:nTF
\bool_lazy_or:p:nn

Implementation of the “Or” operation on the \{boolean expressions\}, hence is \texttt{true} if any of them is \texttt{true} and \texttt{false} if all of them are \texttt{false}. Contrarily to the infix operator \texttt{||}, only the \{boolean expressions\} which are needed to determine the result of \bool_lazy_or:p:nn when there are only two \{boolean expressions\}.

\bool_lazy_any_p:n { \{booleanexpr_1\} \{booleanexpr_2\} \ldots \{booleanexpr_N\} }
\bool_lazy_any:nTF { \{booleanexpr_1\} \{booleanexpr_2\} \ldots \{booleanexpr_N\} } \{\texttt{true code}\}
\bool_lazy_any:p:n
\bool_lazy_any:n
\bool_lazy_any:nn
\bool_lazy_any:nTF

Implementation of the “Or” operation between two boolean expressions, hence is \texttt{true} if either one is \texttt{true}. Contrarily to the infix operator \texttt{||}, the \{booleanexpr_2\} is only evaluated if it is needed to determine the result of \bool_lazy_or:nTF. See also \bool_lazy_or:p:nn when there are more than two \{boolean expressions\}.

\bool_not_p:n \* \bool_not:n \* \bool_not:n { \{boolean expression\} }

Function version of \texttt{\!\{\{boolean expression\}\}} within a boolean expression.

\bool_xor_p:nn \* \bool_xor:p:nn \* \bool_xor:nTF
\bool_xor:nTF \* \bool_xor:nnTF
\bool_xor:p:nn \* \bool_xor:p:nn \* \bool_xor:nTF
\bool_xor:nTF \* \bool_xor:nnTF

Implementation of an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

\section{Logical loops}

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn \* \bool_do_until:cn \* \bool_do_until:Nn \{ \{code\} \}

Places the \{code\} in the input stream for \TeX{} to process, and then checks the logical value of the \{boolean\}. If it is \texttt{false} then the \{code\} is inserted into the input stream again and the process loops until the \{boolean\} is \texttt{true}.

\bool_do_while:Nn \* \bool_do_while:cn \* \bool_do_while:Nn \{ \{code\} \}

Places the \{code\} in the input stream for \TeX{} to process, and then checks the logical value of the \{boolean\}. If it is \texttt{true} then the \{code\} is inserted into the input stream again and the process loops until the \{boolean\} is \texttt{false}.

\bool_until_do:Nn \* \bool_until_do:cn \* \bool_until_do:Nn \{ \{code\} \}

This function firsts checks the logical value of the \{boolean\}. If it is \texttt{false} the \{code\} is placed in the input stream and expanded. After the completion of the \{code\} the truth of the \{boolean\} is re-evaluated. The process then loops until the \{boolean\} is \texttt{true}.

\bool_while_do:Nn \* \bool_while_do:cn \* \bool_while_do:Nn \{ \{code\} \}

This function firsts checks the logical value of the \{boolean\}. If it is \texttt{true} the \{code\} is placed in the input stream and expanded. After the completion of the \{code\} the truth of the \{boolean\} is re-evaluated. The process then loops until the \{boolean\} is \texttt{false}.
\bool_do_until:nn \bool_do_until:nn \{\text{boolean expression}\} \{\text{code}\}

Places the \{\text{code}\} in the input stream for \TeX{} to process, and then checks the logical
value of the \{\text{boolean expression}\} as described for \bool_if:nTF. If it is \text{false} then
the \{\text{code}\} is inserted into the input stream again and the process loops until the \{\text{boolean expression}\} evaluates to \text{true}.

\bool_do_while:nn \bool_do_while:nn \{\text{boolean expression}\} \{\text{code}\}

Places the \{\text{code}\} in the input stream for \TeX{} to process, and then checks the logical
value of the \{\text{boolean expression}\} as described for \bool_if:nTF. If it is \text{true} then the
\{\text{code}\} is inserted into the input stream again and the process loops until the \{\text{boolean expression}\} evaluates to \text{false}.

\bool_until_do:nn \bool_until_do:nn \{\text{boolean expression}\} \{\text{code}\}

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

\bool_while_do:nn \bool_while_do:nn \{\text{boolean expression}\} \{\text{code}\}

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

5 Producing multiple copies

\prg_replicate:nn \prg_replicate:nn \{\text{integer expression}\} \{\text{tokens}\}

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

6 Detecting \TeX{}’s mode

\mode_if_horizontal_p: \mode_if_horizontal_p:
\mode_if_horizontal:TF \{\text{true code}\} \{\text{false code}\}

Detected if \TeX{} is currently in horizontal mode.

\mode_if_inner_p: \mode_if_inner_p:
\mode_if_inner:TF \{\text{true code}\} \{\text{false code}\}

Detected if \TeX{} is currently in inner mode.

\mode_if_math_p: \mode_if_math_p:
\mode_if_math:TF \{\text{true code}\} \{\text{false code}\}

Detected if \TeX{} is currently in maths mode.
\mode_if_vertical_p: ⋆ \mode_if_vertical:TF ⋆ \mode_if_vertical:TF \{(true code)\} \{(false code)\}

Detects if \TeX is currently in vertical mode.

7 Primitive conditionals

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

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

\if_bool:N ⟨boolean⟩ ⟨true code⟩ \else: ⟨false code⟩ \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⟩_map_break: \{⟨code⟩\}

Used to mark the end of a recursion or mapping: the functions \(\langle type⟩\_map_break:\) and \(\langle type⟩\_map_break:n\) use this to break out of the loop (see \prg_map_break:Nn for how to set these up). After the loop ends, the \(⟨code⟩\) 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⟩_map_break: \{⟨user code⟩\}

\prg_break_point:Nn \langle type⟩_map_break: \{⟨ending code⟩\}

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

For types with mappings defined in the kernel, \(\langle type⟩\_map_break: \) and \(\langle type⟩\_map_break:n\) are defined as \prg_map_break:Nn \langle type⟩_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.
This copy of \texttt{\textit{prg\_do\_nothing}}: is used to mark the end of a fast short-term recursion: the function \texttt{\textit{prg\_break:n}} uses this to break out of the loop.

\texttt{\textit{prg\_break:n}} \texttt{\langle code \rangle} \ldots \texttt{\textit{prg\_break_point}}:

Breaks a recursion which has no \texttt{(ending code)} and which is not a user-breakable mapping (see for instance \texttt{\textit{prop\_get:Nn}}), and inserts the \texttt{(code)} in the input stream.

\section{Internal programming functions}

\texttt{\textit{group\_align\_safe\_begin}}: \texttt{\textit{group\_align\_safe\_begin}}:

\texttt{\textit{group\_align\_safe\_end}}: \texttt{\textit{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 \texttt{k} token inside \texttt{\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 \texttt{\peek\_after:Nw} would result in a forbidden comparison of the internal \texttt{\endtemplate} token, yielding a fatal error. Each \texttt{\textit{group\_align\_safe\_begin}}: must be matched by a \texttt{\textit{group\_align\_safe\_end}}; although this does not have to occur within the same function.
Part XIV
The l3sys package: System/runtime functions

1 The name of the job

\c_sys_jobname_str

Constant that gets the “job name” assigned when \TeX\ starts.

\TeXhackers 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

\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int

The date and time at which the current job was started: these are all reported as integers.

\TeXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

3 Engine

\sys_if_engine_luatex_p: \star
\sys_if_engine_luatex: TF \star
\sys_if_engine_pdfTeX_p: \star
\sys_if_engine_pdfTeX: TF \star
\sys_if_engine_ptex_p: \star
\sys_if_engine_ptex: TF \star
\sys_if_engine_uptex_p: \star
\sys_if_engine_uptex: TF \star
\sys_if_engine_xetex_p: \star
\sys_if_engine_xetex: TF \star

Conditional which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)p\TeX tests are for \epsilon-pTEX and \epsilon-up\TeX as expl3 requires the \epsilon-T\TeX extensions. Each conditional is true for exactly one supported engine. In particular, \sys_if_engine_ptex_p: is true for \epsilon-p\TeX but false for \epsilon-up\TeX.

\c_sys_engine_str

The current engine given as a lower case string: one of \texttt{luatex}, \texttt{pdftex}, \texttt{ptex}, \texttt{uptex} or \texttt{xetex}.
4 Output format

\sys_if_output_dvi_p: * \sys_if_output_dvi:TF \langle \text{true code} \rangle \{ \langle \text{false code} \rangle \}

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_p: * \sys_if_platform_unix:TF \langle \text{true code} \rangle \{ \langle \text{false code} \rangle \}

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 \{ \langle \text{integer expression} \rangle \}

Globally sets the seed for the engine’s pseudo-random number generator to the \langle integer expression \rangle. This random seed affects all \ldots_rand functions (such as \int_rand:mm 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{TeXhackers 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

New: 2019-09-20

Defines \texttt{(tl)} to the text returned by the \texttt{(shell command)}. The \texttt{(shell command)} is converted to a string using \texttt{tl_to_str:n}. Category codes may need to be set appropriately via the \texttt{(setup)} argument, which is run just before running the \texttt{(shell command)} (in a group). If shell escape is disabled, the \texttt{(tl var)} will be set to \texttt{\q_no_value} in the non-branching version. Note that quote characters (\texttt{"}) \emph{cannot} be used inside the \texttt{(shell command)}. The \texttt{\sys_get_shell:nnNTF} conditional returns \texttt{true} if the shell is available and no quote is detected, and \texttt{false} otherwise.

\c_sys_shell_escape_int

New: 2017-05-27

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

New: 2017-05-27

Performs a check for whether shell escape is enabled. This returns \texttt{true} if either of restricted or unrestricted shell escape is enabled.

\sys_if_shell_unrestricted_p: \sys_if_shell_unrestricted:TF

New: 2017-05-27

Performs a check for whether \textit{unrestricted} shell escape is enabled.

\sys_if_shell_restricted_p: \sys_if_shell_restricted:TF

New: 2017-05-27

Performs a check for whether \textit{restricted} shell escape is enabled. This returns \texttt{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 \texttt{\sys_if_shell::}.

\sys_shell_now:n \sys_shell_now:x

New: 2017-05-27

Execute \texttt{(tokens)} through shell escape immediately.

\sys_shell_shipout:n \sys_shell_shipout:x

New: 2017-05-27

Execute \texttt{(tokens)} through shell escape at shipout.
7.1 Loading configuration data

\sys_load_backend:n \{\langle backend\rangle\}

Loads the additional configuration file needed for backend support. If the \langle backend\rangle 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:

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} \ldots \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\},\{f\}} namely the five items \texttt{a}, \texttt{b}, \texttt{c-d}, \texttt{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_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 produce errors when their argument contains \{, \} or \# (assuming the usual \TeX{} category codes apply). 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: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.

118
\clist_const:Nn \clist_const:({list} | {var} | {cx})
\new: 2014-07-05

Creates a new constant {list} or raises an error if the name is already taken. The value of the {list} is set globally to the {comma list}.

\clist_clear:N \clist_clear:cn \clist_gclear:N \clist_gclear:cn

Clears all items from the {comma list}.

\clist_clear_new:N \clist_clear_new:cn \clist_gclear_new:N \clist_gclear_new:cn

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 \clist_gset_eq:NN \clist_gset_eq:cn
\new: 2014-07-17

Sets the content of {comma list} equal to that of {comma list2}.

\clist_set_from_seq:NN \clist_set_from_seq:cn \clist_gset_from_seq:NN \clist_gset_from_seq:cn
\new: 2014-07-17

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

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

Concatenates the content of {comma list} and {comma list2} together and saves the result in {comma list1}. The items in {comma list2} are placed at the left side of the new comma list.

\clist_if_exist_p:N \clist_if_exist_p:cn \clist_if_exist:NTF \clist_if_exist:cTT \new: 2012-03-03

Tests whether the {comma list} is currently defined. This does not check that the {comma list} really is a comma list.
# Adding data to comma lists

Sets \((\text{comma list})\) to contain the \((\text{items})\), 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 \((\text{tokens})\) as a single \((\text{item})\) even if the \((\text{tokens})\) contain commas or spaces, add a set of braces: 
\[
\clist_set:Nn \text{\{\{item\}_1,\ldots,\{item\}_n\}}
\]

Appends the \((\text{items})\) to the left of the \((\text{comma list})\). 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 \((\text{tokens})\) as a single \((\text{item})\) even if the \((\text{tokens})\) contain commas or spaces, add a set of braces: 
\[
\clist_put_left:Nn \text{\{\\{item\}_1,\ldots,\{item\}_n\}}
\]

Appends the \((\text{items})\) to the right of the \((\text{comma list})\). 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 \((\text{tokens})\) as a single \((\text{item})\) even if the \((\text{tokens})\) contain commas or spaces, add a set of braces: 
\[
\clist_put_right:Nn \text{\{\{item\}_1,\ldots,\{item\}_n\}}
\]

## 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 \{comma list\}
\clist_gremove_duplicates:N \clist_gremove_duplicates:c

Removes duplicate items from the \{comma list\}, leaving the left most copy of each item in the \{comma list\}. The \{item\} comparison takes place on a token basis, as for \tl_if_eq:nn(TF).

\TeXhackers note: This function iterates through every item in the \{comma list\} and does a comparison with the \{items\} already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the \{comma list\} contains \{, \}, or \# (assuming the usual \TeX\ category codes apply).

\clist_remove_all:NN \clist_remove_all:cn \clist_gremove_all:NN \clist_gremove_all:cn

Updated: 2011-09-06

\clist_remove_all:NN \{comma list\} \{\{item\}\}
\clist_gremove_all:NN \{comma list\} \{\{item\}\}

Removes every occurrence of \{item\} from the \{comma list\}. The \{item\} comparison takes place on a token basis, as for \tl_if_eq:nn(TF).

\TeXhackers note: The function may fail if the \{item\} contains \{, \}, or \# (assuming the usual \TeX\ category codes apply).

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

New: 2014-07-18

\clist_reverse:N \{comma list\}
\clist_greverse:N \{comma list\}

Reverses the order of items stored in the \{comma list\}.

\TeXhackers note: The result is returned within \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_reverse:n \clist_reverse:n \clist_greverse:n \clist_greverse:n

New: 2014-07-18

\clist_reverse:n \{\{comma list\}\}

Leaves the items in the \{comma list\} in the input stream in reverse order. Contrarily to other what is done for other \texttt{n}-type \{comma list\} arguments, braces and spaces are preserved by this process.

\TeXhackers note: The result is returned within \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 \clist_gsort:Nn \clist_gsort:cn

New: 2017-02-06

\clist_sort:Nn \{clist var\} \{\{comparison code\}\}
\clist_gsort:Nn \{clist var\} \{\{comparison code\}\}

Sorts the items in the \{clist var\} according to the \{comparison code\}, and assigns the result to \{clist var\}. 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:c \clist_if_empty_p:N (comma list) \{\{true code\}\} \{\{false code\}\}

Tests if the \texttt{(comma list)} is empty (containing no items).

\clist_if_empty_p:n \clist_if_empty_p:n \{\{comma list\}\}
\clist_if_empty:nTF \clist_if_empty:nTF \{\{comma list\}\} \{\{true code\}\} \{\{false code\}\}

Tests if the \texttt{(comma list)} is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list \{-,\-,\-\} (without outer braces) is empty, while \{-,\{\},\} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

\clist_if_in:NnTF \clist_if_in:NnTF \{\{comma list\}\} \{\{item\}\} \{\{true code\}\} \{\{false code\}\}
\clist_if_in:N \clist_if_in:N \{\{NV\}|\No|\cn|\cV|\co\} \{\{true code\}\} \{\{false code\}\}
\clist_if_in:nn \clist_if_in:nn \{\{nV\}|\no\} \{\{true code\}\} \{\{false code\}\}

Tests if the \texttt{(item)} is present in the \texttt{(comma list)}. In the case of an n-type \texttt{(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 \texttt{true}.

\TeX\hackers note: The function may fail if the \texttt{(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, \textit{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\}_1,\{b\}_2,\{c\}_3, then the arguments passed to the mapped function are ‘a’, ‘b\}_2’, 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 \clist_map_function:NN \clist_map_function:nN \clist_map_function:NN

Applies \texttt{(function)} to every \texttt{(item)} stored in the \texttt{(comma list)}. The \texttt{(function)} receives one argument for each iteration. The \texttt{(items)} are returned from left to right. The function \clist_map_inline:Nn is in general more efficient than \clist_map_function:NN.
Applies \textit{(inline function)} to every \textit{(item)} stored within the \textit{(comma list)}. The \textit{(inline function)} should consist of code which receives the \textit{(item)} as \#1. The \textit{(items)} are returned from left to right.

Stores each \textit{(item)} of the \textit{(comma list)} in turn in the (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{(item)} in the \textit{(comma list)}, or its original value if there were no \textit{(item)}. The \textit{(items)} are returned from left to right.

\textbf{\textsc{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.
\clist_map_break:n \{\textit{code}\}

Used to terminate a \clist_map_... function before all entries in the \textit{comma list} have been processed, inserting the \textit{(code)} after the mapping has ended. This normally takes place within a conditional statement, for example

\begin{verbatim}
\clist_map_inline:Nn \l_my_clist
{\str_if_eq:nnTF { #1 } { bingo } { \clist_map_break:n { <code> } }
{ % Do something useful }
}
\end{verbatim}

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

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

\clist_count:N \{\textit{comma list}\}

Leaves the number of items in the \textit{comma list} in the input stream as an \textit{integer denotation}. The total number of items in a \textit{comma list} includes those which are duplicates, i.e. every item in a \textit{comma list} is counted.

\section{Using the content of comma lists directly}

\clist_use:Nnnn \{\textit{clist var}\} {\{separator between two\}} {\{separator between more than two\}} {\{separator between final two\}}

Places the contents of the \textit{clist var} in the input stream, with the appropriate \textit{separator} between the items. Namely, if the comma list has more than two items, the \textit{separator between more than two} is placed between each pair of items except the last, for which the \textit{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 \textit{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,

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

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.

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

Places the contents of the \clist var in the input stream, with the \(\text{\texttt{separator}}\) between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error 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 \texttt{(\exp_not:n)}, which means that the \langle items \rangle do not expand further when appearing in an 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.

\begin{verbatim}
\clist_get:NN \clist_get:cn \clist_get:NN \clist_get:cN \clist_get:NN \clist_get:cN \clist_get:NN \clist_get:NNTF \clist_get:NNTF \clist_get:NNTF
\end{verbatim}

\begin{verbatim}
\clist_pop:NN \clist_pop:cn \clist_pop:NN \clist_pop:cn \clist_pop:NN \clist_pop:NNTF \clist_pop:NNTF \clist_pop:NNTF
\end{verbatim}

\begin{verbatim}
\clist_gpop:NN \clist_gpop:cn \clist_gpop:NN \clist_gpop:cn \clist_gpop:NN \clist_gpop:NNTF \clist_gpop:NNTF \clist_gpop:NNTF
\end{verbatim}

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:NNTN \clist_gpop:cNTF

\clist_gpop:NNTN \langle comma list \rangle \{ (token list variable) \{ (true code) \} \{ (false code) \} \}

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. The \langle comma list \rangle is modified globally, while the \langle token list variable \rangle is assigned locally.

\clist_push:Nn \langle comma list \rangle \{ (items) \}

\clist_push:Nn \langle comma list \rangle \{ (N|V|No|Nx|cn|cV|co|cx) \}

\clist_gpush:Nn \langle comma list \rangle \{ (N|V|No|Nx|cn|cV|co|cx) \}

Adds the \{ (items) \} to the top of the \langle comma list \rangle. Spaces are removed from both sides of each item as for any n-type comma list.

8 Using a single item

\clist_item:Nn \langle comma list \rangle \{ (integer expression) \}

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

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

\clist_rand_item:N \langle clist var \rangle
\clist_rand_item:n \langle comma list \rangle

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

9 Viewing comma lists

\clist_show:N \langle comma list \rangle
\clist_show:c

Displays the entries in the \langle comma list \rangle in the terminal.
\clist_show:n \clist_show:n \{(tokens)\}
Displays the entries in the comma list in the terminal.

\clist_log:N \clist_log:N \langle comma list \rangle
\clist_log:c
Writes the entries in the \langle comma list \rangle in the log file. See also \clist_show:N which displays the result in the terminal.

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

\char_set_active_eq:NW ⟨char⟩ ⟨function⟩

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

\char_set_active_eq:nN ⟨integer expression⟩ ⟨function⟩

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

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.

\TeXhackers note: Exactly two expansions are needed to produce the character.

\char_lowercase:N \char_uppercase:N \char_titlecase:N \char_foldcase:N \char_str_lowercase:N \char_str_uppercase:N \char_str_titlecase:N \char_str_foldcase:N

\\char_lowercase:N \langle char \rangle

Converts the ⟨char⟩ 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 ⟨char⟩ (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

\texttt{\char_set_catcode_escape:N \char_set_catcode_letter:N \langle \texttt{character} \rangle}\vspace{10pt}
\texttt{\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 \langle \texttt{integer expression} \rangle \char_set_catcode_other:N \%}
\texttt{\char_set_catcode_comment:n \char_set_catcode_invalid:n}\vspace{10pt}

Sets the category code of the \langle \texttt{character} \rangle to that indicated in the function name. Depending on the current category code of the \langle \texttt{token} \rangle the escape token may also be needed:

\texttt{\char_set_catcode_other:N \%}

The assignment is local.

\texttt{\char_set_catcode_escape:n \char_set_catcode_letter:n \langle \texttt{integer expression} \rangle}
\texttt{\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}\vspace{10pt}

Sets the category code of the \langle \texttt{character} \rangle which has character code as given by the \langle \texttt{integer expression} \rangle. This version can be used to set up characters which cannot otherwise be given (\textit{cf.} the \texttt{N}-type variants). The assignment is local.
These functions set the category code of the \emph{character} which has character code as given by the \emph{integer expression}. The first \emph{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\_\langle type \rangle} should be preferred, but there are cases where these lower-level functions may be useful.

Expands to the current category code of the \emph{character} with character code given by the \emph{integer expression}.

Displays the current category code of the \emph{character} with character code given by the \emph{integer expression} on the terminal.

Sets up the behaviour of the \emph{character} when found inside \texttt{\text_lowercase:n}, such that \emph{character}1 will be converted into \emph{character}2. The two \emph{characters} may be specified using an \emph{integer expression} for the character code concerned. This may include the \TeX\ \emph{\texttt{\char\_method}} for converting a single character into its character code:

\begin{verbatim}
\char_set_lccode:nn { \textsc{a} } { \textsc{A} } % Standard behaviour
\char_set_lccode:nn { \textsc{A} } { \textsc{a} + 32 }
\char_set_lccode:nn { 50 } { 60 }
\end{verbatim}

The setting applies within the current \TeX\ group.

Expands to the current lower case code of the \emph{character} with character code given by the \emph{integer expression}.

Displays the current lower case code of the \emph{character} with character code given by the \emph{integer expression} on the terminal.

Sets up the behaviour of the \emph{character} when found inside \texttt{\text_uppercase:n}, such that \emph{character}1 will be converted into \emph{character}2. The two \emph{characters} may be specified using an \emph{integer expression} for the character code concerned. This may include the \TeX\ \emph{\texttt{\char\_method}} for converting a single character into its character code:

\begin{verbatim}
\char_set_uccode:nn { \textsc{a} } { \textsc{A} } % Standard behaviour
\char_set_uccode:nn { \textsc{A} } { \textsc{a} - 32 }
\char_set_uccode:nn { 60 } { 50 }
\end{verbatim}

The setting applies within the current \TeX\ group.
\char_value_uccode:n \char_value_uccode:n {(integer expression)}
Expands to the current upper case code of the \langle character \rangle with character code given by the \langle integer expression \rangle.

\char_show_uccode:n \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.

\char_set_mathcode:nn\char_set_mathcode:nn {{intexpr_1}} {{intexpr_2}}
Updated: 2015-08-06
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.

\char_value_mathcode:n \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.

\char_show_mathcode:n \char_show_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.

\char_set_sfcode:nn\char_set_sfcode:nn {{intexpr_1}} {{intexpr_2}}
Updated: 2015-08-06
This function sets up the space factor for the \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.

\char_value_sfcode:n \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.

\char_show_sfcode:n \char_show_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.

\l_char_active_seq
New: 2012-01-23
Updated: 2015-11-11
Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category \langle active \rangle (catcode 13). Each entry in the sequence consists of a single escaped token, for example \textbackslash -. Active tokens should be added to the sequence when they are defined for general document use.

\l_char_special_seq
New: 2012-01-23
Updated: 2015-11-11
Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories \langle letter \rangle (catcode 11) or \langle other \rangle (catcode 12). Each entry in the sequence consists of a single escaped token, for example \textbackslash \ for the backslash or \textbackslash \ for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.
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.

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

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

\c_catcode_letter_token
\c_catcode_other_token

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

4 Converting tokens

\token_to_meaning:N \token_to_meaning:N (token)

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 \cs_set_nopar:Npn and token list variables defined using \tl_new:N are described as macros.

\TeXhackers note: This is the \TeX primitive \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 \TeX-type arguments.

\token_to_str:N \token_to_str:N (token)

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: \token_to_str:N is the \TeX primitive \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 \TeX-type arguments.
5 Token conditionals

\tokens_if_group_begin_p:N \token_if_group_begin_p:N \token_if_group_begin_NTF \token_if_group_begin:NTF \token_if_group_begin_p:N \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF

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

\tokens_if_group_end_p:N \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end_p:N \token_if_group_end:NTF

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

\tokens_if_math_toggle_p:N \token_if_math_toggle_p:N \token_if_math_toggle:NTF \token_if_math_toggle:NTF \token_if_math_toggle_p:N \token_if_math_toggle:NTF \token_if_math_toggle_p:N \token_if_math_toggle:NTF \token_if_math_toggle_p:N \token_if_math_toggle:NTF

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

\tokens_if_alignment_p:N \token_if_alignment_p:N \token_if_alignment:NTF \token_if_alignment:NTF \token_if_alignment_p:N \token_if_alignment:NTF \token_if_alignment_p:N \token_if_alignment:NTF \token_if_alignment_p:N \token_if_alignment:NTF \token_if_alignment_p:N \token_if_alignment:NTF \token_if_alignment_p:N \token_if_alignment:NTF

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

\tokens_if_parameter_p:N \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF \token_if_parameter_p:N \token_if_parameter:NTF

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

\tokens_if_math_superscript_p:N \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF \token_if_math_superscript_p:N \token_if_math_superscript:NTF

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

\tokens_if_math_subscript_p:N \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF \token_if_math_subscript_p:N \token_if_math_subscript:NTF

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

\tokens_if_space_p:N \token_if_space_p:N \token_if_space:NTF \token_if_space:NTF \token_if_space_p:N \token_if_space:NTF \token_if_space_p:N \token_if_space:NTF \token_if_space_p:N \token_if_space:NTF \token_if_space_p:N \token_if_space:NTF \token_if_space_p:N \token_if_space:NTF

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

134
Tests if \(\text{token}\) has the category code of a letter token.

Tests if \(\text{token}\) has the category code of an “other” token.

Tests if \(\text{token}\) has the category code of an active character.

Tests if the two \(\text{tokens}\) have the same character code.

Tests if the two \(\text{tokens}\) have the same meaning when expanded.

Tests if \(\text{token}\) is a \LaTeX\ macro.

Tests if \(\text{token}\) is a control sequence.

Tests if \(\text{token}\) is expandable. This test returns \(\text{false}\) for an undefined token.

Tests if \(\text{token}\) is a long macro.

Tests if the \(\text{token}\) is a protected macro: for a macro which is both protected and long this returns \text{false}.
Tests if the \langle token\rangle is a protected long macro.

Tests if the \langle token\rangle is defined to be a chardef.

\textbf{TeXhackers note:} Booleans, boxes and small integer constants are implemented as \texttt{chardefs}.

Tests if the \langle token\rangle is defined to be a mathchardef.

Tests if the \langle token\rangle is defined to be a dimension register.

Tests if the \langle token\rangle is defined to be a integer register.

\textbf{TeXhackers note:} Constant integers may be implemented as integer registers, \texttt{chardefs}, or \texttt{mathchardefs} depending on their value.

Tests if the \langle token\rangle is defined to be a muskip register.

Tests if the \langle token\rangle is defined to be a skip register.
Tests if the \langle token \rangle is defined to be a tok register (not used by \LaTeX3).

Tests if the \langle 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{\peekafter:Nw} is provided along with a family of predefined tests for common cases. As peeking ahead does not skip spaces the predefined tests include both a space-respecting and space-skipping version.

\begin{verbatim}
\peekafter:Nw (function) \langle token \rangle
\end{verbatim}

Locally sets the test variable \lpeek_token equal to \langle token \rangle (as an implicit token, not as a token list), and then expands the \langle function \rangle. The \langle token \rangle remains in the input stream as the next item after the \langle function \rangle. The \langle token \rangle here may be \texttt{,}, \texttt{,}, or \texttt{)} (assuming normal \TeX category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\begin{verbatim}
\peekgafter:Nw (function) \langle token \rangle
\end{verbatim}

Globally sets the test variable \gpeek_token equal to \langle token \rangle (as an implicit token, not as a token list), and then expands the \langle function \rangle. The \langle token \rangle remains in the input stream as the next item after the \langle function \rangle. The \langle token \rangle here may be \texttt{,}, \texttt{,}, or \texttt{)} (assuming normal \TeX category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\lpeek_token
Token set by \texttt{\peekafter:Nw} and available for testing as described above.

\gpeek_token
Token set by \texttt{\peekgafter:Nw} and available for testing as described above.

\peekcatcode:NTF \langle test token \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}
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{\tokenif_eq_catcode:NTF}). Spaces are respected 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 non-space \(\langle\text{token}\rangle\) 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:NTF}). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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 \(\langle\text{token}\rangle\) 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:NTF}). Spaces are respected by the test and the \(\langle\text{token}\rangle\) is removed from the input stream if the test is true. The function then places either the \(\langle\text{true code}\rangle\) or \(\langle\text{false code}\rangle\) in the input stream (as appropriate to the result of the test).

Tests if the next non-space \(\langle\text{token}\rangle\) in the input stream has the same character code as the \(\langle\text{test token}\rangle\) (as defined by the test \texttt{\textbackslash token\_if\_eq\_charcode:NTF}). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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 \(\langle\text{token}\rangle\) in the input stream has the same character code as the \(\langle\text{test token}\rangle\) (as defined by the test \texttt{\textbackslash token\_if\_eq\_charcode:NTF}). Spaces are respected by the test and the \(\langle\text{token}\rangle\) is removed from the input stream if the test is true. The function then places either the \(\langle\text{true code}\rangle\) or \(\langle\text{false code}\rangle\) in the input stream (as appropriate to the result of the test).
Tests if the next non-space \(\langle\text{token}\rangle\) in the input stream has the same character code as the \(\langle\text{test token}\rangle\) (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 \(\langle\text{token}\rangle\) is removed from the input stream if the test is true. The function then places either the \(\langle\text{true code}\rangle\) or \(\langle\text{false code}\rangle\) in the input stream (as appropriate to the result of the test).

Tests if the next \(\langle\text{token}\rangle\) in the input stream has the same meaning as the \(\langle\text{test token}\rangle\) (as defined by the test \texttt{token_if_eq_meaning:NNTF}). 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 \(\langle\text{token}\rangle\) in the input stream has the same meaning as the \(\langle\text{test token}\rangle\) (as defined by the test \texttt{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 \(\langle\text{token}\rangle\) is removed from the input stream if the test is true. The function then places either the \(\langle\text{true code}\rangle\) or \(\langle\text{false code}\rangle\) in the input stream (as appropriate to the result of the test).

Tests if the next non-space \(\langle\text{token}\rangle\) in the input stream has the same meaning as the \(\langle\text{test token}\rangle\) (as defined by the test \texttt{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 \(\langle\text{token}\rangle\) is removed from the input stream if the test is true. The function then places either the \(\langle\text{true code}\rangle\) or \(\langle\text{false code}\rangle\) in the input stream (as appropriate to the result of the test).
Tests if the next ⟨token⟩ in the input stream can be safely grabbed as an N-type argument. The test is ⟨false⟩ if the next ⟨token⟩ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in LaTeX3) and ⟨true⟩ in all other cases. Note that a ⟨true⟩ result ensures that the next ⟨token⟩ is a valid N-type argument. However, if the next ⟨token⟩ is for instance \c_space_token, the test takes the ⟨false⟩ branch, even though the next ⟨token⟩ is in fact a valid N-type argument. The ⟨token⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (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 TeX’s state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, \use:n is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

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

- Expanding \the\font results in a token that looks identical to the command that was used to select the current font (such as \tenrm) but it differs from it in shape.
- A “frozen” \relax, which differs from the primitive in shape (but has the same meaning), is inserted when the closing \fi of a conditional is encountered before the conditional is evaluated.
- Expanding \noexpand ⟨token⟩ (when the ⟨token⟩ is expandable) results in an internal token, displayed (temporarily) as \notexpanded: ⟨token⟩, whose shape coincides with the ⟨token⟩ and whose meaning differs from \relax.
- An \outer endtemplate: can be encountered when peeking ahead at the next token; this expands to another internal token, end of alignment template.
- Tricky programming might access a frozen \endwrite.

---
6In 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 \LaTeX\ 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 (\texttt{space}, often character code 32),
9. the letter (\texttt{letter}, such as A),
10. the character (\texttt{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 \LaTeX\ for most functions and some variables (\texttt{tl}, \texttt{fp}, \texttt{seq}, ...),
- a primitive such as \texttt{\def} or \texttt{\topmark}, used in \LaTeX\ for some functions,
- a register such as \texttt{\count123}, used in \LaTeX\ for the implementation of some variables (\texttt{int}, \texttt{dim}, ...),
- a constant integer such as \texttt{\char"56} or \texttt{\mathchar"121},
- a font selection command,
- undefined.

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

\langle properties \rangle \texttt{macro:} \langle parameters \rangle \rightarrow \langle replacement \rangle

141
where \(\langle \text{properties} \rangle\) is among \texttt{\textbackslash protected\textbackslash long\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

\LaTeX{}3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a ⟨key⟩ and an associated ⟨value⟩. The ⟨key⟩ and ⟨value⟩ may both be any ⟨balanced text⟩. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique ⟨key⟩: if an entry is added to a property list which already contains the ⟨key⟩ then the new entry overwrites the existing one. The ⟨keys⟩ are compared on a string basis, using the same method as \texttt{\textbackslash str_if_eq:n}.\texttt{nn}

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of input parsed by the \texttt{keys} module.

1 Creating and initialising property lists

\begin{verbatim}
\prop_new:N \prop_new:c
\prop_clear:N \prop_clear:c \prop_clear:N \prop_clear:c
\prop_clear_new:N \prop_clear_new:c \prop_clear_new:N \prop_clear_new:c
\prop_set_eq:NN \prop_set_eq:cn \prop_gset_eq:NN \prop_gset_eq:cc
\prop_set_from_keyval:Nn \prop_set_from_keyval:cn \prop_set_from_keyval:Nn \prop_set_from_keyval:cn
\end{verbatim}

\texttt{\prop_new:N \prop_new:c} \texttt{⟨property list⟩}

Creates a new ⟨property list⟩ or raises an error if the name is already taken. The declaration is global. The ⟨property list⟩ initially contains no entries.

\texttt{\prop_clear:N \prop_clear:c \prop_gclear:N \prop_gclear:c}

Clears all entries from the ⟨property list⟩.

\texttt{\prop_clear_new:N \prop_clear_new:c \prop_clear_new:N \prop_clear_new:c}

Ensures that the ⟨property list⟩ exists globally by applying \texttt{\prop_new:N} if necessary, then applies \texttt{\prop_(g)clear:N} to leave the list empty.

\texttt{\prop_set_eq:NN \prop_set_eq:cc \prop_gset_eq:cc}

Sets the content of ⟨property list1⟩ equal to that of ⟨property list2⟩.

\texttt{\prop_set_from_keyval:Nn \prop_set_from_keyval:cn \prop_set_from_keyval:Nn \prop_set_from_keyval:cn}

Sets ⟨prop var⟩ 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_const_from_keyval:cn
New: 2017-11-28
Updated: 2019-08-25

\prop_const_from_keyval:Nn \prop_const_from_keyval:cn

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_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnV|cno|cnx|cVn|cVV|con|coo)
Updated: 2012-07-09

\prop_put:Nnn \prop_put:
(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnV|cno|cnx|cVn|cVV|con|coo)

Adds an entry to the \property\ list which may be accessed using the \key\ and which has \value. Both the \key\ and \value\ may contain any \balanced\ text. The \key\ is stored after processing with \tl_to_str:n, meaning that category codes are ignored. If the \key\ is already present in the \property\ list, the existing entry is overwritten by the new \value.

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn

\prop_put_if_new:Nnn \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_get:(NVN|NoN|cnN|cVN|coN)
Updated: 2011-08-28

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

Recover 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
\prop_pop:(NoN|cnN|coN)
Updated: 2011-08-18

\prop_pop:NnN \prop_pop:
(NoN|cnN|coN)

Recover 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.

144
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 \texttt{\q_no_value}. 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 \texttt{prop_gpop:NnNTF}.

\prop_item:Nn \langle property list \rangle \{\langle key \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.

\textbf{\TeX} hackers note: This function is slower than the non-expandable analogue \texttt{prop_get:Nn}. The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \langle value \rangle does not expand further when appearing in an \texttt{x}-type argument expansion.

\prop_count:N \langle property list \rangle

Leaves the number of key–value pairs in the \langle property list \rangle in the input stream as an \langle integer denotation \rangle.

\textbf{4} Modifying property lists

\prop_remove:Nn \langle property list \rangle \{\langle key \rangle\}

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, \textit{i.e} there is no need to test for the existence of a key before deleting it.

\textbf{5} Property list conditionals

\prop_if_exist_p:N \langle property list \rangle
\prop_if_exist:NTF \langle property list \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}

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.

\prop_if_empty_p:N \langle property list \rangle
\prop_if_empty:NTF \langle property list \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}

Tests if the \langle property list \rangle is empty (containing no entries).
Tests if the ⟨key⟩ is present in the ⟨property list⟩, making the comparison using the method described by \texttt{str_if_eq:nnTF}.

\textbf{\textit{\LaTeX}hacker's note:} This function iterates through every key–value pair in the ⟨property list⟩ and is therefore slower than using the non-expandable \texttt{prop_get:NnNTF}.

### 6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

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 \langle function \rangle or \langle code \rangle discussed below remain in effect after the loop.

\prop_map_function:NN \langle property list \rangle \langle function \rangle

Applies \langle function \rangle to every \langle entry \rangle stored in the \langle property list \rangle. The \langle function \rangle receives two arguments for each iteration: the \langle key \rangle and associated \langle value \rangle. The order in which \langle entries \rangle are returned is not defined and should not be relied upon. To pass further arguments to the \langle function \rangle, see \prop_map_tokens:Nn.

\prop_map_inline:Nn \langle property list \rangle \{ \langle inline function \rangle \}

Applies \langle inline function \rangle to every \langle entry \rangle stored within the \langle property list \rangle. The \langle inline function \rangle should consist of code which receives the \langle key \rangle as \#1 and the \langle value \rangle as \#2. The order in which \langle entries \rangle are returned is not defined and should not be relied upon.

\prop_map_tokens:Nn \langle property list \rangle \{ \langle code \rangle \}

Analogue of \prop_map_function:NN which maps several tokens instead of a single function. The \langle code \rangle receives each key–value pair in the \langle property list \rangle 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 \langle key \rangle and the \langle value \rangle 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 \langle property list \rangle 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 \(\langle\text{code}\rangle\)

Used to terminate a \prop_map... function before all entries in the \langle\text{property list}\rangle have been processed, inserting the \langle\text{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\text{code}\rangle } 
     } {
     \% Do something useful
     }
}

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

**\TeX{}hackers note:** When the mapping is broken, additional tokens may be inserted before the \langle\text{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:c

Displays the entries in the \langle\text{property list}\rangle in the terminal.

\prop_log:N \prop_log:c

Writes the entries in the \langle\text{property list}\rangle in the log file.

---

### 9 Scratch property lists

\l_tmpa_prop \l_tmpb_prop

Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any \TeX{}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

Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any \TeX{}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

\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 \ forces an explicit space. Additionally, \, \, \, \ and \ can be used to produce the corresponding character.

Messages may be subdivided by one level using the / character. This is used within the message filtering system to allow for example the TeX 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.

\msg_new:nnnn \msg_new:nnn
\msg_set:nnnn \msg_set:nnn
\msg_gset:nnnn \msg_gset:nnn

\msg_new:nnnn \msg_new:nnn \msg_set:nnnn \msg_set:nnn

\msg_new:nnnn \msg_new:nnn \msg_set:nnnn \msg_set:nnn

Creates a (message) for a given (module). The message is defined to first give (text) and then (more text) if the user requests it. If no (more text) is available then a standard text is given instead. Within (text) and (more text) four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the (message) already exists.

Sets up the text for a (message) for a given (module). The message is defined to first give (text) and then (more text) if the user requests it. If no (more text) is available then a standard text is given instead. Within (text) and (more text) four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.
Tests whether the ⟨message⟩ for the ⟨module⟩ is currently defined.

## 2 Contextual information for messages

**\msg_line_context:**

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text on line.

**\msg_line_number:**

Prints the current line number when a message is given.

**\msg_fatal_text:n**

Produces the standard text

**Fatal Package ⟨module⟩ Error**

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the ⟨module⟩ to be included.

**\msg_critical_text:n**

Produces the standard text

**Critical Package ⟨module⟩ Error**

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the ⟨module⟩ to be included.

**\msg_error_text:n**

Produces the standard text

**Package ⟨module⟩ Error**

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the ⟨module⟩ to be included.

**\msg_warning_text:n**

Produces the standard text

**Package ⟨module⟩ Warning**

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the ⟨module⟩ to be included. The ⟨type⟩ of ⟨module⟩ may be adjusted: Package is the standard outcome: see \msg_module_type:n.
\textbackslash msg\_info\_text:n \{\textbackslash module\}\}

Produces the standard text:

\textbf{Package} \textit{\langle module\rangle} \textbf{Info}

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \langle module\rangle to be included. The \langle type\rangle of \langle module\rangle may be adjusted: Package is the standard outcome: see \textbackslash msg\_module\_type:n.

\textbackslash msg\_module\_name:n \{\textbackslash module\}\}

\textit{\langle module\rangle}

Expands to the public name of the \langle module\rangle as defined by \textbackslash g\_msg\_module\_name\_prop (or otherwise leaves the \langle module\rangle unchanged).

\textbackslash msg\_module\_type:n \{\textbackslash module\}\}

\textit{\langle module\rangle}

Expands to the description which applies to the \langle module\rangle, for example a Package or Class. The information here is defined in \textbackslash g\_msg\_module\_type\_prop, and will default to Package if an entry is not present.

\textbackslash msg\_see\_documentation\_text:n \{\textbackslash module\}\}

\textit{\langle module\rangle}

Produces the standard text

\textit{See the \langle module\rangle documentation for further information.}

This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \langle module\rangle to be included. The name of the \langle module\rangle may be altered by use of \textbackslash g\_msg\_module\_documentation\_prop

\textbackslash g\_msg\_module\_name\_prop

Provides a mapping between the module name used for messages, and that for documentation. For example, \LaTeX3 core messages are stored in the reserved \LaTeX\ tree, but are printed as \LaTeX3.

\textbackslash g\_msg\_module\_type\_prop

Provides a mapping between the module name used for messages, and that type of module. For example, for \LaTeX3 core messages, an empty entry is set here meaning that they are not described using the standard Package text.

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 (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 \endinput primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

Issues (module) error (message), passing (arg one) to (arg four) to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

Issues (module) warning (message), passing (arg one) to (arg four) to the text-creating functions. The warning text is added to the log file and the terminal, but the \TeX run is not interrupted.
Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The information text is added to the log file.

---

Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The information text is added to the log file: the output is briefer than `msg_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

```latex
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```latex
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```latex
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```latex
\msg_redirect_module:nnn { module } { error } { warning }
```
to alter only messages from that module, or even

\msg_redirect_name:nnn \{ module \} \{ my-message \} \{ warning \}

to target just one message. Redirection applies first to individual messages, then to
messages from one module and finally to messages of one class. Thus it is possible to
select out an individual message for special treatment even if the entire class is already
redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an
empty argument for the target class. Redirection to a missing class 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 \{ ⟨class one⟩⟩ \{ ⟨class two⟩⟩
Changes the behaviour of messages of ⟨class one⟩ so that they are processed using the
code for those of ⟨class two⟩.

\msg_redirect_module:nnn \{ ⟨module⟩⟩ \{ ⟨class one⟩⟩ \{ ⟨class two⟩⟩
Redirects message of ⟨class one⟩ for ⟨module⟩ to act as though they were from ⟨class
two⟩. Messages of ⟨class one⟩ from sources other than ⟨module⟩ are not affected by this
redirection. This function can be used to make some messages “silent” by default. For
example, all of the warning messages of ⟨module⟩ could be turned off with:

\msg_redirect_module:nnn \{ module \} \{ warning \} \{ none \}

\msg_redirect_name:nnn \{ ⟨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:nnn \{ module \} \{ annoying-message \} \{ none \}
Part XIX

The l3file package
File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix \file..., while others are used to work with files on a line by line basis and have prefix \ior... (reading) or \iow... (writing).

It is important to remember that when reading external files TeX 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 ⟨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 \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 LaTeX. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

Globally reserves the name of the ⟨stream⟩, either for reading or for writing as appropriate. The ⟨stream⟩ is not opened until the appropriate \..._open:Nn function is used. Attempting to use a ⟨stream⟩ which has not been opened is an error, and the ⟨stream⟩ will behave as the corresponding \c_term_....

\ior_open:Nn ⟨stream⟩
\ior_open:cn

Opens ⟨file name⟩ for reading using ⟨stream⟩ as the control sequence for file access. If the ⟨stream⟩ was already open it is closed before the new operation begins. The ⟨stream⟩ is available for access immediately and will remain allocated to ⟨file name⟩ until a \ior_-close:N instruction is given or the TeX run ends. If the file is not found, an error is raised.
\ior_open:NnTF
\ior_open:cnTF
\ior_close:N
\ior_close:c
\iow_open:Nn
\iow_open:cn
\iow_close:N
\iow_close:c
\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:
\ior_get:NN \ior_get:NN \ior_get:NNTF

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material read from the \langle stream \rangle 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 \% 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/uni2423b/uni2423c/uni2423}. Any blank line is converted to the token \texttt{\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{\par} tokens. In the non-branching version, where the \langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q_no_value}.

\textbf{\TeXHackers 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.

\ior_str_get:NN \ior_str_get:NNTF

Function that reads one line from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material is read from the \langle stream \rangle 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 \langle token list variable \rangle being empty. Unlike \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 \langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q_no_value}.

\textbf{\TeXHackers note:} This protected macro is a wrapper around the \eTeX primitive \texttt{\textbackslash readline}. Regardless of settings, \eTeX 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 \ior_str_get:NN.

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.
\ior_map_inline:Nn
Applies the \langle inline function \rangle to each set of \langle lines \rangle obtained by calling \ior_get:NN until reaching the end of the file. \TeX ignores any trailing new-line marker from the file it reads. The \langle inline function \rangle should consist of code which receives the \langle line \rangle as \#1.

\ior_str_map_inline:Nn
Applies the \langle inline function \rangle to every \langle line \rangle in the \langle stream \rangle. The material is read from the \langle stream \rangle as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The \langle inline function \rangle should consist of code which receives the \langle line \rangle as \#1. Note that \TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \TeX also ignores any trailing new-line marker from the file it reads.

\ior_map_variable:NNn
For each set of \langle lines \rangle obtained by calling \ior_get:NN until reaching the end of the file, stores the \langle lines \rangle in the \langle tl var \rangle then 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 set of \langle lines \rangle, or its original value if the \langle stream \rangle is empty. \TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_map_inline:Nn.

\ior_str_map_variable:NNn
For each \langle line \rangle in the \langle stream \rangle, stores the \langle line \rangle in the \langle variable \rangle then applies the \langle code \rangle. The material is read from the \langle stream \rangle as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). 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 line \rangle, or its original value if the \langle stream \rangle is empty. Note that \TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \TeX also ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_str_map_inline:Nn.

\ior_map_break:
Used to terminate a \ior_map_... function before all lines from the \langle stream \rangle have been processed. 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: }
{ % Do something useful }
}

Use outside of a \ior_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.

159
\ior_map_break:n \ior_map_break:n \{\langle code\rangle\}

Used to terminate a \ior_map_inline:Nn 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_inline:Nn 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\rangle\} \{\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 \texttt{true} value if the \langle stream\rangle is not open.

1.2 Writing to files

\iow_now:Nn \iow_now:Nn \{\langle tokens\rangle\}
\iow_now:Nn \iow_now:Nn \langle tokens\rangle \{\langle tokens\rangle\}
\iow_now:Nn \iow_now:Nn \langle tokens\rangle

This function writes \langle tokens\rangle to the specified \langle stream\rangle immediately \textit{(i.e. the write operation is called on expansion of \iow_now:Nn)}.

\iow_log:n \iow_log:n \{\langle tokens\rangle\}
\iow_log:n \iow_log:n \{\langle tokens\rangle\}
\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\}
\iow_term:n \iow_term:n \{\langle tokens\rangle\}
\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.
\texttt{\textbackslash iow\textunderscore shipout}\texttt{:Nn} \texttt{\textbackslash iow\textunderscore shipout}\texttt{:Nnx|cn|cx}

This function writes \langle tokens\rangle to the specified \langle stream\rangle when the current page is finalised (\textit{i.e.} at shipout). The x-type variants expand the \langle tokens\rangle at the point where the function is used but \textit{not} when the resulting tokens are written to the \langle stream\rangle (\textit{cf.} \texttt{\textbackslash iow\textunderscore shipout\_x:Nn}).

\textbf{\LaTeXhackers note:} When using \texttt{expl3} with a format other than \texttt{\LaTeX{}}, new line characters inserted using \texttt{\textbackslash iow\textunderscore newline}: or using the line-wrapping code \texttt{\textbackslash iow\textunderscore wrap:nnnN} are not recognized in the argument of \texttt{\textbackslash iow\textunderscore shipout}:Nn. This may lead to the insertion of additional unwanted line-breaks.

\texttt{\textbackslash iow\textunderscore shipout}\texttt{:x:Nn} \texttt{\textbackslash iow\textunderscore shipout}\texttt{:x}(Nx|cn|cx)

This function writes \langle tokens\rangle to the specified \langle stream\rangle when the current page is finalised (\textit{i.e.} at shipout). The \langle 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{\LaTeXhackers note:} This is a wrapper around the \LaTeX{} primitive \texttt{\textbackslash write}. When using \texttt{expl3} with a format other than \texttt{\LaTeX{}}, new line characters inserted using \texttt{\textbackslash iow\textunderscore newline}: or using the line-wrapping code \texttt{\textbackslash iow\textunderscore wrap:nnnN} are not recognized in the argument of \texttt{\textbackslash iow\textunderscore shipout}:Nn. This may lead to the insertion of additional unwanted line-breaks.

\texttt{\textbackslash iow\textunderscore char}\texttt{:N} \texttt{\textbackslash iow\textunderscore char}\texttt{:N (\textbackslash char)}

Inserts \langle char\rangle into the output stream. Useful when trying to write difficult characters such as \textbackslash{}, \{$\}$, \textit{etc.} in messages, for example:

\texttt{\textbackslash iow\textunderscore now}\texttt{:Nx \texttt{\textbackslash g\textunderscore my\textunderscore iow} \{(\texttt{\textbackslash iow\textunderscore char}\texttt{:N} \texttt{\langle text\rangle \texttt{\textbackslash iow\textunderscore char}\texttt{:N})\}\}}

The function has no effect if writing is taking place without expansion (\textit{e.g.} in the second argument of \texttt{\textbackslash iow\textunderscore now}:Nn).

\texttt{\textbackslash iow\textunderscore newline:} \texttt{\textbackslash iow\textunderscore newline:}

Function to add a new line within the \langle 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 \texttt{\textbackslash iow\textunderscore now}:Nn).

\textbf{\LaTeXhackers note:} When using \texttt{expl3} with a format other than \texttt{\LaTeX{}}, the character inserted by \texttt{\textbackslash iow\textunderscore newline:} is not recognized by \texttt{\LaTeX{}}, which may lead to the insertion of additional unwanted line-breaks. This issue only affects \texttt{\textbackslash iow\textunderscore shipout}:Nn, \texttt{\textbackslash iow\textunderscore shipout\_x:Nn} and direct uses of primitive operations.
1.3 Wrapping lines in output

\input{io/wrap}
\newcommand{\iow_wrap}{\ifnum\iow_line_count_int<31\relax
This function wraps the \textit{text} to a fixed number of characters per line. At the start
of each line which is wrapped, the \textit{run-on text} is inserted. The line character count
targeted is the value of \texttt{\iow_line_count_int} minus the number of characters in the
\textit{run-on text} for all lines except the first, for which the target number of characters is
simply \texttt{\iow_line_count_int} since there is no run-on text. The \textit{text} and \textit{run-on text} are exhaustively expanded by the function, with the following substitutions:

- \texttt{\textbackslash \ or \ \iownewline}: may be used to force a new line,
- \texttt{\textbackslash \ } may be used to represent a forced space (for example after a control sequence),
- \texttt{\#}, \texttt{\%}, \texttt{\{}, \texttt{\}} may be used to represent the corresponding character,
- \texttt{\iowallowbreak}: may be used to allow a line-break without inserting a space
  (this is experimental),
- \texttt{\iowindent}: may be used to indent a part of the \textit{text} (not the \textit{run-on text}).

Additional functions may be added to the wrapping by using the \texttt{\setup}, which is
executed before the wrapping takes place: this may include overriding the substitutions
listed.

Any expandable material in the \textit{text} which is not to be expanded on wrapping
should be converted to a string using \texttt{\token_to_str:N}, \texttt{\tl_to_str:n}, \texttt{\tl_to_str:N},
etc.

The result of the wrapping operation is passed as a braced argument to the
\texttt{\function}, which is typically a wrapper around a write operation. The output of \texttt{\iow_wrap}
\texttt{\iowlinecount} \texttt{(i.e. the argument passed to the \texttt{\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.

\textbf{\TeX{}hackers note:} Internally, \texttt{\iow_wrap} carries out an x-type expansion on the
\textit{text} to expand it. This is done in such a way that \texttt{\exp_not:N} or \texttt{\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 \textit{text}.

\input{io/indent}
\newcommand{\iowindent}{\ifnum\iow_line_count_int<31\relax
In the first argument of \texttt{\iow_wrap} (for instance in messages), indents \textit{text} by
four spaces. This function does not cause a line break, and only affects lines which start
within the scope of the \textit{text}. In case the indented \textit{text} should appear on separate lines
from the surrounding text, use \texttt{\textbackslash \ } to force line breaks.

\input{iow_line_count_int}
\newcommand{\iowlinecount}{\texttt{\iow_line_count_int}
The maximum number of characters in a line to be written by the \texttt{\iow_wrap} 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 \LaTeX3-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_ior} \texttt{\g_tmpb_ior}

New: 2017-12-11

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

\texttt{\c_log_iow} \texttt{\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 \LaTeX3-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_iow} \texttt{\g_tmpb_iow}

New: 2017-12-11

1.5 Primitive conditionals

\texttt{\if_eof:w \if_eof:w (stream)}
\texttt{\false_code}
\texttt{\else: \false_code}
\texttt{\fi:}

Tests if the \texttt{(stream)} returns “end of file”, which is true for non-existent files. The \texttt{\else:} branch is optional.

\TeXhackers note: This is the \TeX primitive \texttt{\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 (\textit{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 \texttt{(name)} and \texttt{(ext)} parts together make up the file name, thus the \texttt{(name)} part may be thought of as the “job name” for the current file. Note that \TeX does not provide information on the \texttt{(ext)} part for the main (top level) file and that this file always has an empty \texttt{(dir)} component. Also, the \texttt{(name)} here will be equal to \texttt{\c_sys_jobname_str}, which may be different from the real file name (if set using \texttt{--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.

\TeXhackers note: When working as a package in \LaTeX, expl3 will automatically append the current \input@path to the set of values from \l_file_search_path_seq.

\file_if_exist:nTF \{\langle file name\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}

Searches for \langle file name\rangle using the current \TeX search path and the additional paths controlled by \l_file_search_path_seq.

\file_get:nnN \{\langle filename\rangle\} \{\langle setup\rangle\} \{\langle tl\rangle\}
\file_get:nnNTF \{\langle filename\rangle\} \{\langle setup\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}

Defines \langle tl\rangle to the contents of \langle filename\rangle. Category codes may need to be set appropriately via the \langle setup\rangle argument. The non-branching version sets the \langle tl\rangle to \q_no_value if the file is not found. The branching version runs the \langle true code\rangle after the assignment to \langle tl\rangle if the file is found, and \langle false code\rangle otherwise.

\file_get_full_name:nN \{\langle file name\rangle\}
\file_get_full_name:VN
\file_get_full_name:nNTF \{\langle file name\rangle\}
\file_get_full_name:VNTF

Searches for \langle file name\rangle in the path as detailed for \file_if_exist:nTF, and if found sets the \langle tl var\rangle the fully-qualified name of the file, i.e. the path and file name. This includes an extension .tex when the given \langle file name\rangle has no extension but the file found has that extension. In the non-branching version, the \langle tl var\rangle will be set to \q_no_value in the case that the file does not exist.

\file_full_name:n \{\langle file name\rangle\}
\file_full_name:V \star

Searches for \langle file name\rangle in the path as detailed for \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 .tex when the given \langle file name\rangle 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 \texttt{(full name)} and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The \texttt{(dir)}: everything up to the last / (path separator) in the \texttt{(file path)}. As with system \TeX\ variables and related functions, the \texttt{(dir)} does \textit{not} include the trailing / unless it points to the root directory. If there is no path (only a file name), \texttt{(dir)} is empty.

- The \texttt{(name)}: everything after the last / up to the last ., where both of those characters are optional. The \texttt{(name)} may contain multiple . characters. It is empty if \texttt{(full name)} consists only of a directory name.

- The \texttt{(ext)}: everything after the last . (including the dot). The \texttt{(ext)} is empty if there is no . after the last /.

This function does not expand the \texttt{(full name)} before turning it to a string. It assume that the \texttt{(full name)} either contains no quote (”) characters or is surrounded by a pair of quotes.

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 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 \texttt{(start index)} and \texttt{(end index)} values work as described for \texttt{\str\_range:nnn}.

Sets the \texttt{(tl var)} to the result of applying \texttt{\file\_mdfive\_hash:nnn} to the \texttt{(file)}. If the file is not found, the \texttt{(tl var)} will be set to \texttt{\q\_no\_value}.
\file_size:n \{(file name)\}

Searches for \(\text{(file name)}\) using the current TeX search path and the additional paths controlled by \texttt{\l_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)\} \{tl var\}

Sets the \(\text{(tl var)}\) to the result of applying \file_size:n to the \(\text{(file)}\). If the file is not found, the \(\text{(tl var)}\) will be set to \texttt{\q_no_value}. This is not available in older versions of XeTeX.

\file_timestamp:n \{(file name)\}

Searches for \(\text{(file name)}\) using the current TeX search path and the additional paths controlled by \texttt{\l_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 \texttt{D:(year)(month)(day)(hour)(minute)(second)(offset)}, where the latter may be \texttt{Z} (UTC) or \texttt{(+}-\texttt{minus})\texttt{hours}'\texttt{minutes}'. When the file is not found, the result of expansion is empty. This is not available in older versions of XeTeX.

\file_get_timestamp:nN \{(file name)\} \{tl var\}

Sets the \(\text{(tl var)}\) to the result of applying \file_timestamp:n to the \(\text{(file)}\). If the file is not found, the \(\text{(tl var)}\) will be set to \texttt{\q_no_value}. This is not available in older versions of XeTeX.

\file_compare_timestamp:p:nNn \{(file-1)\} \{comparator\} \{(file-2)\} \{\text{(true code)}\} \{\text{(false code)}\}

\file_compare_timestamp_p:nNn \star \file_compare_timestamp:nNn \star

Compares the file stamps on the two \texttt{(files)} as indicated by the \texttt{(comparator)}, and inserts either the \texttt{(true code)} or \texttt{(false case)} 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 XeTeX.

\file_input:n \{(file name)\}

Updated: 2017-06-26

Searches for \(\text{(file name)}\) in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional \LaTeX{} source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.
\file_if_exist_input:n \{file name\} \{false code\}

Searches for \textit{file name} using the current \TeX{} search path and the additional paths controlled by \texttt{\file_path_include:n}. If found then reads in the file as additional \LaTeX{} source as described for \texttt{\file_input:n}, otherwise inserts \textit{false code}. Note that these functions do not raise an error if the file is not found, in contrast to \texttt{\file_input:n}.

\file_input_stop: \file_input_stop:

Ends the reading of a file started by \texttt{\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:n(nm)} 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!

\file_show_list: \file_show_list:
\file_log_list: \file_log_list:

These functions list all files loaded by \LaTeX{} commands that populate \texttt{@filelist} or by \texttt{\file_input:n}. While \texttt{\file_show_list:} displays the list in the terminal, \texttt{\file_log_list:} outputs it to the log file only.
Part XX

The l3skip package

Dimensions and skips

\LaTeX3 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: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 0pt.

\begin{verbatim}
\dim_const:Nn \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 \texttt{dimension} is set globally to the \texttt{dimension expression}.

\begin{verbatim}
\dim_zero:N \dim_zero:N \dim_zero:c \dim_gzero:N \dim_gzero:c
\end{verbatim}

Sets \texttt{dimension} to 0pt.

\begin{verbatim}
\dim_zero_new:N \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\_gzero:N} to leave the \texttt{dimension} set to zero.

\begin{verbatim}
\dim_if_exist_p:N \dim_if_exist_p:N \dim_if_exist_p:c \dim_if_exist_p:cn \dim_if_exist:N \dim_if_exist:N \dim_if_exist:NTF \dim_if_exist:NTF \dim_if_exist:c \dim_if_exist:NTF \dim_if_exist:NTF
\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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_add:Nn</td>
<td>Adds the result of the (\text{dimension expression}) to the current content of the (\text{dimension}).</td>
</tr>
<tr>
<td>\dim_add:cn</td>
<td></td>
</tr>
<tr>
<td>\dim_gadd:Nn</td>
<td></td>
</tr>
<tr>
<td>\dim_gadd:cn</td>
<td></td>
</tr>
</tbody>
</table>

Updated: 2011-10-22

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_set:Nn</td>
<td>Sets (\text{dimension}) to the value of (\text{dimension expression}), which must evaluate to a length with units.</td>
</tr>
<tr>
<td>\dim_set:cn</td>
<td></td>
</tr>
<tr>
<td>\dim_gset:Nn</td>
<td></td>
</tr>
<tr>
<td>\dim_gset:cn</td>
<td></td>
</tr>
</tbody>
</table>

Updated: 2011-10-22

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_set_eq:NN</td>
<td>Sets the content of (\text{dimension}_1) equal to that of (\text{dimension}_2).</td>
</tr>
<tr>
<td>\dim_set_eq:cN</td>
<td></td>
</tr>
<tr>
<td>\dim_set_eq:Nc</td>
<td></td>
</tr>
<tr>
<td>\dim_set_eq:cc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_sub:Nn</td>
<td>Subtracts the result of the (\text{dimension expression}) from the current content of the (\text{dimension}).</td>
</tr>
<tr>
<td>\dim_sub:cn</td>
<td></td>
</tr>
<tr>
<td>\dim_gsub:Nn</td>
<td></td>
</tr>
<tr>
<td>\dim_gsub:cn</td>
<td></td>
</tr>
</tbody>
</table>

Updated: 2011-10-22

## 3 Utilities for dimension calculations

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_abs:n</td>
<td>Converts the (\text{dimexpr}) to its absolute value, leaving the result in the input stream as a (\text{dimension denotation}).</td>
</tr>
<tr>
<td>\dim_abs:n</td>
<td></td>
</tr>
</tbody>
</table>

Updated: 2012-09-26

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\dim_max:nn</td>
<td>Evaluates the two (\text{dimension expressions}) and leaves either the maximum or minimum value in the input stream as appropriate, as a (\text{dimension denotation}).</td>
</tr>
<tr>
<td>\dim_max:nn</td>
<td></td>
</tr>
<tr>
<td>\dim_min:nn</td>
<td></td>
</tr>
<tr>
<td>\dim_min:nn</td>
<td></td>
</tr>
</tbody>
</table>

New: 2012-09-09

Updated: 2012-09-26
\dim_ratio:nn \{dimexpr_1\} \{dimexpr_2\}

 Parses the two \langle dimension expressions \rangle and converts the ratio of the two to a form suitable for use inside a \langle 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:nNNn \{dimexpr_1\} \{relation\} \{dimexpr_2\}
\dim_compare:nNnTF \{dimexpr_1\} \{relation\} \{dimexpr_2\}
\{\text{true code}\} \{\text{false code}\}

 This function first evaluates each of the \langle dimension expressions \rangle as described for \dim_eval:n. The two results are then compared using the \langle relation\rangle:

\begin{align*}
\text{Equal} &= \quad = \\
\text{Greater than} &= > \\
\text{Less than} &= <
\end{align*}

 This function is less flexible than \dim_compare:nTF but around 5 times faster.
\dim_compare_p:n \dim_compare:nTF

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 \texttt{dimexpr_1} and \texttt{dimexpr_2} using the \textit{relation_1}, then \texttt{dimexpr_2} and \texttt{dimexpr_3} using the \textit{relation_2}, until finally comparing \texttt{dimexpr_N} and \texttt{dimexpr_{N+1}} using the \textit{relation_N}. 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 \hspace{1cm} = \hspace{1cm} \texttt{or} \hspace{1cm} ==
  \item Greater than or equal to \hspace{1cm} \geq
  \item Greater than \hspace{1cm} \gt
  \item Less than or equal to \hspace{1cm} \leq
  \item Less than \hspace{1cm} \lt
  \item Not equal \hspace{1cm} \neq
\end{itemize}

This function is more flexible than \texttt{\dim_compare:nNnTF} but around 5 times slower.
This function evaluates the \( \text{test dimension expression} \) and compares this in turn to each of the \( \text{dimension expression cases} \). If the two are equal then the associated \( \text{code} \) is left in the input stream and other cases are discarded. If any of the cases are matched, the \( \text{true code} \) is also inserted into the input stream (after the code for the appropriate case), while if none match then the \( \text{false code} \) is inserted. The function \( \texttt{dim\_case:nn} \) 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_1) \} \{ (dimexpr_2) \} \{ (relation) \} \{ (code) \}
\end{verbatim}

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

\begin{verbatim}
\dim_do_while:nNnn \{ (dimexpr_1) \} \{ (relation) \} \{ (dimexpr_2) \} \{ (code) \}
\end{verbatim}

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

\begin{verbatim}
\dim_until_do:nNnn \{ (dimexpr_1) \} \{ (relation) \} \{ (dimexpr_2) \} \{ (code) \}
\end{verbatim}

Evaluates the relationship between the two \( \text{dimension expressions} \) as described for \( \texttt{dim\_compare:nNnTF} \), and then places the \( \text{code} \) in the input stream if the \( \text{relation} \) is \texttt{false}. After the \( \text{code} \) has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}. 

172
\texttt{\textbackslash dim\_while\_do:nNnn} \footnote{\textit{\textbackslash dim\_while\_do:nNnn} \{(dimexpr_1)\} \{(relation)\} \{(dimexpr_2)\} \{(code)\}}

Evaluates the relationship between the two \textit{dimension expressions} as described for \texttt{\textbackslash dim\_compare:nNnT}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \textit{true}. After the \textit{(code)} has been processed by \TeX{}, the test is repeated, and a loop occurs until the test is \textit{false}.

\texttt{\textbackslash dim\_do\_until:nn} \footnote{\textit{\textbackslash dim\_do\_until:nn} \{(dimension relation)\} \{(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 dim\_compare:nNnT}. If the test is \textit{false} then the \textit{(code)} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \textit{true}.

\texttt{\textbackslash dim\_while\_do:nn} \footnote{\textit{\textbackslash dim\_while\_do:nn} \{(dimension relation)\} \{(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 dim\_compare:nNnT}. If the test is \textit{true} then the \textit{(code)} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \textit{false}.

\texttt{\textbackslash until\_do:nn} \footnote{\textit{\textbackslash until\_do:nn} \{(dimension relation)\} \{(code)\}}

Evaluates the \textit{dimension relation} as described for \texttt{\textbackslash dim\_compare:nNnT}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \textit{false}. After the \textit{(code)} has been processed by \TeX{}, the test is repeated, and a loop occurs until the test is \textit{true}.

\texttt{\textbackslash dim\_do\_do:nn} \footnote{\textit{\textbackslash dim\_do\_do:nn} \{(dimension relation)\} \{(code)\}}

Evaluates the \textit{dimension relation} as described for \texttt{\textbackslash dim\_compare:nNnT}, and then places the \textit{(code)} in the input stream if the \textit{(relation)} is \textit{true}. After the \textit{(code)} has been processed by \TeX{}, the test is repeated, and a loop occurs until the test is \textit{false}.

\section{Dimension step functions}

\texttt{\textbackslash dim\_step\_function:nnnN} \footnote{\textit{\textbackslash dim\_step\_function:nnnN} \{(initial value)\} \{(step)\} \{(final value)\} \{(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 dim\_step\_inline:nnn} \footnote{\textit{\textbackslash dim\_step\_inline:nnn} \{(initial value)\} \{(step)\} \{(final value)\} \{(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}).
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 the \textit{tl var} defined as the current \textit{value}. Thus the \textit{code} should make use of the \textit{tl var}.

7 Using \texttt{dim} expressions and variables

\texttt{dim_eval:n \{\textit{dimension expression}\}}

Evaluates the \textit{dimension expression}, expanding any dimensions and token list variables within the \textit{expression} 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 \textit{dimension denotation} 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 \textit{internal dimension}.

\texttt{dim_sign:n \{\textit{dimexpr}\}}

Evaluates the \textit{dimexpr} then leaves 1 or 0 or –1 in the input stream according to the sign of the result.

\texttt{dim_use:N \{\textit{dimension}\}}\texttt{dim_use:c}

Recover the content of a \textit{dimension} 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 \textit{dimension} is required (such as in the argument of \texttt{dim_eval:n}).

\TeXhackers note: \texttt{dim_use:N} is the \TeX primitive \texttt{the}: this is one of several \LaTeX names for this primitive.

\texttt{dim_to_decimal:n \{\textit{dimexpr}\}}

Evaluates the \textit{dimension expression}, and leaves the result, expressed in points (pt) in the input stream, with \textit{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

\texttt{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.
\dim_to_decimal_in_bp:n  \dim_to_decimal_in_bp:n \{\langle \text{dimexpr} \rangle\}

Evaluates the \langle \text{dimension expression} \rangle, and leaves the result, expressed in big points (bp) in the input stream, with \textit{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  \dim_to_decimal_in_sp:n \{\langle \text{dimexpr} \rangle\}

Evaluates the \langle \text{dimension expression} \rangle, and leaves the result, expressed in scaled points (sp) in the input stream, with \textit{no units}. The result is necessarily an integer.

\dim_to_decimal_in_unit:nn  \dim_to_decimal_in_unit:nn \{\langle \text{dimexpr}_1 \rangle\} \{\langle \text{dimexpr}_2 \rangle\}

Evaluates the \langle \text{dimension expressions} \rangle, and leaves the value of \langle \text{dimexpr}_1 \rangle, expressed in a unit given by \langle \text{dimexpr}_2 \rangle, 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 \varepsilon-\TeX primitives, which is required internally by \dim_to_decimal_in_unit:nn.

\dim_to_fp:n  \dim_to_fp:n \{\langle \text{dimexpr} \rangle\}

Expands to an internal floating point number equal to the value of the \langle \text{dimexpr} \rangle 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 \texttt{dim} variables

\dim_show:N  \dim_show:N \{\langle \text{dimension} \rangle\}

\dim_show:N
\dim_show:c

Displays the value of the \langle \text{dimension} \rangle on the terminal.
\dim_show:n \dim_show:n \{\langle\text{dimension expression}\rangle\}

Displays the result of evaluating the \textit{\langle\text{dimension expression}\rangle} on the terminal.

\dim_log:N \dim_log:c \dim_log:n \dim_log:n \{\langle\text{dimension expression}\rangle\}

Writes the value of the \textit{\langle\text{dimension}\rangle} in the log file.

\dim_log:n \dim_log:n \{\langle\text{dimension expression}\rangle\}

Writes the result of evaluating the \textit{\langle\text{dimension expression}\rangle} in the log file.

\section{Constant dimensions}

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

\c_max_dim\c_zero_dim

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

\section{Scratch dimensions}

\l_tmpa_dim \l_tmpb_dim \g_tmpa_dim \g_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.

\section{Creating and initialising \texttt{skip} variables}

\skip_new:N\skip_new:n \skip_new:c \{\langle\text{skip}\rangle\}

Creates a new \texttt{\langle\text{skip}\rangle} or raises an error if the name is already taken. The declaration is global. The \texttt{\langle\text{skip}\rangle} 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:NTF  \skip_if_exist:N  \skip_if_exist:cTF
\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

12 Setting skip variables

\skip_zero:N \skip \{ \langle \text{skip expression} \rangle \}
Sets \langle \text{skip} \rangle to 0 pt.

\skip_zero_new:N \skip \{ \langle \text{skip} \rangle \}
Ensures that the \langle \text{skip} \rangle exists globally by applying \texttt{\skip_new:N} if necessary, then applies \texttt{\skip_(g)zero:N} to leave the \langle \text{skip} \rangle set to zero.

\skip_if_exist_p:N \skip \{ \langle \text{skip} \rangle \}
\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.

\skip_add:Nn \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 \{ \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} \rangle_1 \langle \text{skip} \rangle_2
Sets the content of \langle \text{skip} \rangle_1 equal to that of \langle \text{skip} \rangle_2.

\skip_sub:Nn \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 {⟨\textit{skipexpr}_1⟩} {⟨\textit{skipexpr}_2⟩}
\skip_if_eq:nnTF {⟨\textit{skipexpr}_1⟩} {⟨\textit{skipexpr}_2⟩} {⟨\text{true code}⟩} {⟨\text{false code}⟩}

This function first evaluates each of the \textit{(skip expressions)} as described for \texttt{\skip_eval:n}. The two results are then compared for exact equality, \textit{i.e.} both the fixed and rubber components must be the same for the test to be true.

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

Evaluates the \textit{(skip expression)} as described for \texttt{\skip_eval:n}, and then tests if all of its components are finite.

14 Using \textit{skip} expressions and variables

\texttt{\skip_eval:n} {⟨\textit{skip expression}⟩}

Evaluates the \textit{(skip expression)}, expanding any skips and token list variables within the \textit{(expression)} to their content (without requiring \texttt{\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 \textit{(glue denotation)} after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \TeX-style assignment as it is \textit{not} an \textit{(internal glue)}.

\texttt{\skip_use:N} \texttt{⟨\textit{skip}⟩}

Recovers the content of a \textit{(skip)} 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 \textit{(dimension)} or \textit{(skip)} is required (such as in the argument of \texttt{\skip_eval:n}).

\TeXhacks note: \texttt{\skip_use:N} is the \TeX primitive \texttt{\the}: this is one of several \LaTeX names for this primitive.

15 Viewing \textit{skip} variables

\texttt{\skip_show:N} \texttt{⟨\textit{skip}⟩}

Displays the value of the \textit{(skip)} on the terminal.

\texttt{\skip_show:N} {⟨\textit{skip expression}⟩}

Displays the result of evaluating the \textit{(skip expression)} on the terminal.
\skip_log:N \skip_log:c

New: 2014-08-22
Updated: 2015-08-03

\skip_log:n

New: 2014-08-22
Updated: 2015-08-07

16 Constant skips

\c_max_skip

Updated: 2012-11-02

\c_zero_skip

Updated: 2012-11-01

17 Scratch skips

\l_tmpa_skip \l_tmpb_skip

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

\g_tmpa_skip \g_tmpb_skip

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

18 Inserting skips into the output

\skip_horizontal:N \skip_horizontal:c \skip_horizontal:n

Updated: 2011-10-22

\skip_horizontal:N \skip_horizontal:c \skip_horizontal:n

\skip_horizontal:n \{\langle skip expression\rangle\}

Inserts a horizontal \langle skip \rangle into the current list. The argument can also be a \langle dim \rangle.

\TeXhackers note: \skip_horizontal:N is the \TeX primitive \hskip renamed.
\skip_vertical:N  \skip_vertical:c  \skip_vertical:n

Inserts a vertical \textit{skip} into the current list. The argument can also be a \textit{dim}.

\textit{\LaTeX}hackers note: \skip_vertical:N is the \LaTeX{} primitive \textbackslash{skip} renamed.

19 Creating and initialising \texttt{muskip} variables

\texttt{\muskip_new:N} \texttt{\muskip_new:c}

Creates a new \texttt{muskip} or raises an error if the name is already taken. The declaration is global. The \texttt{muskip} is initially equal to 0 \texttt{mu}.

\texttt{\muskip_const:Nn} \texttt{\muskip_const:cn}

Creates a new constant \texttt{muskip} or raises an error if the name is already taken. The value of the \texttt{muskip} is set globally to the \texttt{muskip expression}.

\texttt{\skip_zero:N} \texttt{\skip_zero:c}

\texttt{\skip_zero_new:N} \texttt{\skip_zero_new:c}

\texttt{\skip_zero_gnew:N} \texttt{\skip_zero_gnew:c}

Sets \texttt{muskip} to 0 \texttt{mu}.

\texttt{\muskip_if_exist_p:N} \texttt{\muskip_if_exist_p:c} \texttt{\muskip_if_exist:NTF} \texttt{\muskip_if_exist:c:TF}

Tests whether the \texttt{muskip} is currently defined. This does not check that the \texttt{muskip} really is a muskip variable.

20 Setting \texttt{muskip} variables

\texttt{\muskip_add:Nn} \texttt{\muskip_add:cn} \texttt{\muskip_gadd:Nn} \texttt{\muskip_gadd:cn}

Adds the result of the \texttt{muskip expression} to the current content of the \texttt{muskip}. 

180
\muskip_set:Nn \muskip_set:cn \muskip_gset:Nn \muskip_gset:cn

Updated: 2011-10-22

\muskip_set_eq:NN \muskip_set_eq:cn \muskip_gset_eq:NN

\muskip_sub:Nn \muskip_sub:cn \muskip_gsub:Nn \muskip_gsub:cn

Updated: 2011-10-22

\muskip_use:N \muskip_use:c

Updated: 2011-10-22

\muskip_eval:n \muskip_eval:n

Updated: 2011-10-22

\muskip_show:N \muskip_show:c

Updated: 2015-08-03

\textbf{21 Using muskip expressions and variables}

\textbf{22 Viewing muskip variables}
\muskip_show:n \muskip_show:n {⟨muskip expression⟩}
Displays the result of evaluating the ⟨muskip expression⟩ on the terminal.

\muskip_log:N \muskip_log:c \muskip_log:n \muskip_log:n
\muskip_log:n {⟨muskip⟩}
Writes the value of the ⟨muskip⟩ in the log file.

\muskip_log:n {⟨muskip expression⟩}
Writes the result of evaluating the ⟨muskip expression⟩ in the log file.

\c_max_muskip\c_zero_muskip
The maximum value that can be stored as a muskip, with no stretch nor shrink component.
A zero length as a muskip, with no stretch nor shrink component.

\l_tmpa_muskip \l_tmpb_muskip \g_tmpa_muskip \g_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.

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
\25 Primitive conditional
\if_dim:w \if_dim:w (dimen₁) ⟨relation⟩ (dimen₂)
⟨true code⟩
\else:
 ⟨false⟩
\fi:
Compare two dimensions. The ⟨relation⟩ is one of <, = or > with category code 12.

\TeXhackers note: This is the \TeX primitive \ifdim.
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 \langle module\rangle \langle keyval list\rangle

Parses the `\langle keyval list\rangle` and defines the keys listed there for `\langle module\rangle`. The `\langle module\rangle` name is treated as a string. In practice the `\langle module\rangle` should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The `\langle keyval list\rangle` 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 `\langle key\rangle`, which when used may be supplied with a `\langle value\rangle`. 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 }
{
}

184
keyname .value_required:n = true,
keyname .code:n = Some-code-using-#1
}

Note that with the exception of the special .undefine: property, all key properties define the key within the current \TeX{} scope.

\begin{itemize}
  \item \code{bool_set:N} = \code{boolean}
  \item \code{bool_set:c}
  \item \code{bool_gset:N}
  \item \code{bool_gset:c}
\end{itemize}

Updated: 2013-07-08

\begin{itemize}
  \item \code{bool_set_inverse:N} = \code{boolean}
  \item \code{bool_set_inverse:c}
  \item \code{bool_gset_inverse:N}
  \item \code{bool_gset_inverse:c}
\end{itemize}

Updated: 2013-07-08

\begin{itemize}
  \item \code{choice:}
\end{itemize}

Updated: 2013-07-10

\begin{itemize}
  \item \code{choices:nn} = \code{choices} \{\code{code}\}
\end{itemize}

Updated: 2013-07-10

\begin{itemize}
  \item \code{clist_set:N} = \code{comma list variable}
  \item \code{clist_set:c}
  \item \code{clist_gset:N}
  \item \code{clist_gset:c}
\end{itemize}

Updated: 2013-07-10

\begin{itemize}
  \item \code{code:n} = \code{code}
\end{itemize}

Updated: 2013-07-10

\begin{itemize}
  \item \code{cs_set:Np}
  \item \code{cs_set:cp}
  \item \code{cs_set_protected:Np}
  \item \code{cs_set_protected:cp}
  \item \code{cs_gset:Np}
  \item \code{cs_gset:cp}
  \item \code{cs_gset_protected:Np}
  \item \code{cs_gset_protected:cp}
\end{itemize}

New: 2020-01-11

\begin{itemize}
  \item \code{key} .bool_set:N = \code{boolean}
\end{itemize}

Defines \code{key} to set \code{boolean} to \code{value} (which must be either \code{true} or \code{false}). If the \code{variable} does not exist, it will be created globally at the point that the key is set up.

\begin{itemize}
  \item \code{key} .bool_set_inverse:N = \code{boolean}
\end{itemize}

Defines \code{key} to set \code{boolean} to the logical inverse of \code{value} (which must be either \code{true} or \code{false}). If the \code{boolean} does not exist, it will be created globally at the point that the key is set up.

\begin{itemize}
  \item \code{key} .choice:
\end{itemize}

Sets \code{key} to act as a choice key. Each valid choice for \code{key} must then be created, as discussed in section 3.

\begin{itemize}
  \item \code{key} .choices:nn = \code{choices} \{\code{code}\}
\end{itemize}

Sets \code{key} to act as a choice key, and defines a series \code{choices} which are implemented using the \code{code}. Inside \code{code}, \code{l_keys_choice_tl} will be the name of the choice made, and \code{l_keys_choice_int} will be the position of the choice in the list of \code{choices} (indexed from 1). Choices are discussed in detail in section 3.

\begin{itemize}
  \item \code{key} .clist_set:N = \code{comma list variable}
\end{itemize}

Defines \code{key} to set \code{comma list variable} to \code{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{itemize}
  \item \code{key} .code:n = \code{code}
\end{itemize}

Stores the \code{code} for execution when \code{key} is used. The \code{code} can include one parameter (#1), which will be the \code{value} given for the \code{key}.

\begin{itemize}
  \item \code{key} .cs_set:Np = \code{control sequence} \{\code{arg. spec.}\}
\end{itemize}

Defines \code{key} to set \code{control sequence} to have \code{arg. spec.} and replacement text \code{value}.
(key) .default:n = { (default) }

Creates a (default) value for (key), which is used if no value is given. This will be used
if only the key name is given, but not if a blank (value) is given:

\keys_define:nn { mymodule }
  { 
    key .code:n = Hello-\#1, 
    key .default:n = World 
  }
\keys_set:nn { mymodule } 
  { 
    key = Fred, \% Prints 'Hello Fred'
    key, \% Prints 'Hello World'
    key = , \% Prints 'Hello '
  }

The default does not affect keys where values are required or forbidden. Thus a required
value cannot be supplied by a default value, and giving a default value for a key which
cannot take a value does not trigger an error.

(key) .dim_set:N = (dimension)

Defines (key) to set (dimension) to (value) (which must a dimension 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.

(key) .fp_set:N = (floating point)

Defines (key) to set (floating point) to (value) (which must a floating point expression).
If the variable does not exist, it 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.

(key) .groups:n = { (groups) }

Defines (key) as belonging to the (groups) declared. Groups provide a “secondary axis”
for selectively setting keys, and are described in Section 6.

(key) .inherit:n = { (parents) }

Specifies that the (key) path should inherit the keys listed as (parents). For example,
after setting

\keys_define:nn { foo } { test .code:n = \tl_show:n \{#1\} }
\keys_define:nn { } { bar .inherit:n = foo }

setting

\keys_set:nn { bar } { test = a }

will be equivalent to

\keys_set:nn { foo } { test = a }

186
### .initial:n
Initialises the ⟨key⟩ with the ⟨value⟩, equivalent to
\keys_set:nn {⟨module⟩} { ⟨key⟩ = ⟨value⟩ }

### .int_set:N
Defines ⟨key⟩ to set ⟨integer⟩ to ⟨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.

### .meta:n
Makes ⟨key⟩ a meta-key, which will set ⟨keyval list⟩ in one go. The ⟨keyval list⟩ can refer as #1 to the value given at the time the ⟨key⟩ is used (or, if no value is given, the ⟨key⟩’s default value).

### .meta:nn
Makes ⟨key⟩ a meta-key, which will set ⟨keyval list⟩ in one go using the ⟨path⟩ in place of the current one. The ⟨keyval list⟩ can refer as #1 to the value given at the time the ⟨key⟩ is used (or, if no value is given, the ⟨key⟩’s default value).

### .multichoice:
Sets ⟨key⟩ to act as a multiple choice key. Each valid choice for ⟨key⟩ must then be created, as discussed in section 3.

### .multichoices:nn
Sets ⟨key⟩ to act as a multiple choice key, and defines a series ⟨choices⟩ which are implemented using the ⟨code⟩. Inside ⟨code⟩, \l_keys_choice_tl will be the name of the choice made, and \l_keys_choice_int will be the position of the choice in the list of ⟨choices⟩ (indexed from 1). Choices are discussed in detail in section 3.

### .muskip_set:N
Defines ⟨key⟩ to set ⟨muskip⟩ to ⟨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.

### .prop_put:N
Defines ⟨key⟩ to put the ⟨value⟩ onto the ⟨property list⟩ stored under the ⟨key⟩. If the variable does not exist, it is created globally at the point that the key is set up.
\skip_set:N \skip_set:c \skip_gset:N \skip_gset:c

Defines \key{\skip_set:N} to set \skip to \value{\skip} (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.

\tl_set:N \tl_set:c \tl_gset:N \tl_gset:c

Defines \key{\tl_set:N} to set \token list variable to \value{\tl_set:N}. If the variable does not exist, it is created globally at the point that the key is set up.

\tl_set_x:N \tl_set_x:c \tl_gset_x:N \tl_gset_x:c

Defines \key{\tl_set_x:N} to set \token list variable to \value{\tl_set_x:N}, which will be subjected to an \xtype expansion (\text{i.e.} using \textbackslash \tl_set:Nx). If the variable does not exist, it is created globally at the point that the key is set up.

\define:\undefined:

Removes the definition of the \key{\define:\undefined:} within the current scope.

\value_forbidden:n \value_forbidden:c \value_required:n \value_required:c

Specifies that \key{\value_forbidden:n} cannot receive a \value{\value_forbidden:n} when used. If a \value{\value_forbidden:n} is given then an error will be issued. Setting the property \texttt{false} cancels the restriction.

\value_required:n \value_required:c

Specifies that \key{\value_required:n} must receive a \value{\value_required:n} when used. If a \value{\value_required:n} is not given then an error will be issued. Setting the property \texttt{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:

\keys_define:nn { mymodule / subgroup }
{ key .code:n = code }

or to the key name:

\keys_define:nn { mymodule }
{ subgroup / key .code:n = code }

As illustrated, the best choice of token for sub-dividing keys in this way is / . This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name \texttt{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 \l3keys\system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the .\choice:\ property:

\keys_define:nn { mymodule }
{ key .\choice: }

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the .\choices:nn property.

\keys_define:nn { mymodule }
{ key .\choices:nn =
{ choice-a, choice-b, choice-c }
{ You\-gave\-choice\-'\tl_use:N \l_keys_choice_tl',~
which\-is\-in\-position\-\int_use:N \l_keys_choice_int \c_space_tl
in\-the\-list. }
}

The index \l_keys_choice_int in the list of choices starts at 1.

\l_keys_choice_int \l_keys_choice_tl

Inside the code block for a choice generated using .\choices:nn, the variables \l_keys_choice_tl and \l_keys_choice_int are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using .\code:n, the value passed to the key (i.e. the choice name) is also available as #1.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the .\choice: property of a key, then manually defining sub-keys.

\keys_define:nn { mymodule }
{ key .\choice:,
 key / choice-a .\code:n = code-a,
 key / choice-b .\code:n = code-b,
 key / choice-c .\code:n = code-c,
}

It is possible to mix the two methods, but manually-created choices should not use \l_keys_choice_tl or \l_keys_choice_int. These variables do not have defined
behaviour when used outside of code created using \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 define as sub-keys. Thus both

\begin{verbatim}
\keys_define:nn { mymodule }
{  
  key .multichoices:nn =  
  { choice-a, choice-b, choice-c }  
  {  
    You-gave-choice-\texttt{\tl_use:N \l_keys_choice_tl},~\ 
    which-is-in-position-\ 
    \int_use:N \l_keys_choice_int \texttt{\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.

4 Setting keys

\keys_set:nn \keys_set:\{\langle module\rangle\} \{\langle keyval list\rangle\}

 Parses the \langle keyval list\rangle, and sets those keys which are defined for \langle module\rangle. The behaviour on finding an unknown key can be set by defining a special unknown key: this is illustrated later.

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.

5 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.

```latex
\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

```latex
\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

```latex
\keys_define:nn { mymodule }
{  
  key-one .code:n = { \my_func:n {#1} },  
  key-one .groups:n = { first },  
  key-two .tl_set:N = \l_my_a_tl ,  
```
key-two .groups:n = { first },
key-three .tl_set:N = \l_my_b_tl,
key-three .groups:n = { second },
key-four .fp_set:N = \l_my_a_fp,
}

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 {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}
\keys_set_filter:nnnN {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}{
  \langle tl \rangle
}
\keys_set_filter:nnnnN {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}{
  \langle root \rangle
}{
  \langle tl \rangle
}

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 {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}
\keys_set_groups:nnnN {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}{
  \langle root \rangle
}
\keys_set_groups:nnnnN {
  \langle module \rangle
}{
  \langle groups \rangle
}{
  \langle keyval list \rangle
}{
  \langle root \rangle
}{
  \langle root \rangle
}

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_p:nnTF *
\keys_if_exist_p:nn *
\keys_if_exist_p:nnTF *
\keys_if_exist_p:nn *
\keys_if_exist_p: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 \textit{choice} is defined for the \textit{key} within the \textit{module}, i.e. if any code has been defined for \textit{key}/\textit{choice}. The test is \textit{false} if the \textit{key} itself is not defined.

\begin{verbatim}
\keys_if_choice_exist_p:nnn *
\keys_if_choice_exist:nnnTF *
\end{verbatim}

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 \textit{key–value list} into \textit{keys} and associated \textit{values}. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces 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 \{ function₁ \} \{ function₂ \} \{ key–value list \}

Parses the \{ key–value list \} into a series of \{ keys \} and associated \{ values \}, or keys alone
(if no \{ value \} was given). \{ function₁ \} should take one argument, while \{ function₂ \}
should absorb two arguments. After \keyval_parse:NNn has parsed the \{ key–value list \},
\{ function₁ \} is used to process keys given with no value and \{ function₂ \} 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 noth-
ing) 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.

\TeX hackers note: The result is returned within \exp_not:n, which means that the con-
verted input stream does not expand further when appearing in an \x-type or \e-type argument
expansion.
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 $2^{30} - 1$ (i.e. one power lower than the usual \texttt{\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
\intarray_new: \langle intarray var \rangle \{ \langle size \rangle \}\n\end{verbatim}

Evaluates the integer expression $\langle size \rangle$ and allocates an \texttt{intarray 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:N * \intarray_count:c *
\intarray_count: \langle intarray var \rangle
\end{verbatim}

Expands to the number of entries in the \texttt{intarray variable}. Contrarily to \texttt{\seq_-count:N} this is performed in constant time.

\begin{verbatim}
\intarray_gset:Nnn \intarray_gset:cn
\intarray_gset: \langle intarray var \rangle \{ \langle position \rangle \} \{ \langle value \rangle \}
\end{verbatim}

Stores the result of evaluating the integer expression $\langle value \rangle$ into the \texttt{intarray variable} at the (integer expression) $\langle position \rangle$. If the $\langle position \rangle$ is not between 1 and the $\intarray_count:N$, or the $\langle value \rangle$’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
\intarray_const_from_clist: \langle intarray var \rangle \{ \texttt{intexpr clist} \}
\end{verbatim}

Creates a new constant \texttt{intarray variable} or raises an error if the name is already taken. The \texttt{intarray 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:cn
\intarray_gzero: \langle intarray var \rangle
\end{verbatim}

Sets all entries of the \texttt{intarray variable} to zero. Assignments are always global.
\texttt{\intarray_item:Nn} \{\texttt{intarray var}\} \{\texttt{position}\}

Expands to the integer entry stored at the \{integer expression\} \{position\} in the \{integer array variable\}. If the \{position\} is not between 1 and the \texttt{\intarray_count:N}, an error occurs.

\texttt{\intarray_rand_item:N} \{\texttt{intarray var}\}

Selects a pseudo-random item of the \{integer array\}. If the \{integer array\} is empty, produce an error.

\texttt{\intarray_show:N} \{\texttt{intarray var}\}
\texttt{\intarray_log:N} \{\texttt{intarray var}\}

Displays the items in the \{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 \( \sin^d x \), \( \cos^d x \), \( \tan^d x \), \( \cot^d x \), \( \sec^d x \), \( \csc^d x \) expecting their arguments in degrees.
- Inverse trigonometric functions: \( \arcsin x \), \( \arccos x \), \( \arctan x \), \( \arccot x \), \( \arcsec x \), \( \arccsc x \) giving a result in radians, and \( \arcsin^d x \), \( \arccos^d x \), \( \arctan^d x \), \( \arccot^d x \), \( \arcsec^d x \), \( \arccsc^d x \) giving a result in degrees.
- Hyperbolic functions and their inverse functions: \( \sinh x \), \( \cosh x \), \( \tanh x \), \( \coth x \), \( \sech x \), \( \csch x \), and \( \text{asinh} x \), \( \text{acosh} x \), \( \text{atanh} x \), \( \text{acoth} x \), \( \text{asech} x \), \( \text{acsch} x \).
- Extrema: \( \max(x_1, x_2, \ldots) \), \( \min(x_1, x_2, \ldots) \), \( \abs(x) \).
- 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 \times \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

\fp_new:N \texttt{\fp \_new:N} (\texttt{fp var})

Creates a new (\texttt{fp var}) or raises an error if the name is already taken. The declaration is global. The (\texttt{fp var}) is initially \texttt{+0}.

\fp_const:Nn \texttt{\fp \_const:Nn} (\texttt{fp var}) \{\texttt{\langle floating point expression\rangle}\}

Creates a new constant (\texttt{fp var}) or raises an error if the name is already taken. The (\texttt{fp var}) is set globally equal to the result of evaluating the (\texttt{floating point expression}).

\fp_zero:N \texttt{\fp \_zero:N} (\texttt{fp var})

Sets the (\texttt{fp var}) to \texttt{+0}.
2 Setting floating point variables

\fp_set:Nn \fp_var \{\text{floating point expression}\}
Sets \fp_var equal to the result of computing the \text{floating point expression}.

\fp_set_eq:NN \fp_var_1 \fp_var_2
Sets the floating point variable \fp_var_1 equal to the current value of \fp_var_2.

\fp_add:Nn \fp_var \{\text{floating point expression}\}
Adds the result of computing the \text{floating point expression} to the \fp_var. This also applies if \fp_var and \text{floating point expression} evaluate to tuples of the same size.

\fp_sub:Nn \fp_var \{\text{floating point expression}\}
Subtracts the result of computing the \text{floating point expression} from the \fp_var. This also applies if \fp_var and \text{floating point expression} evaluate to tuples of the same size.

3 Using floating points

\fp_eval:n \{\text{floating point expression}\}
Evaluates the \text{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 \((\fp_1, \fp_2, \ldots, \fp_n)\) if \(n > 1\) and \((\fp_1)\) or () for fewer items. This function is identical to \fp_to_decimal:n.
\fp_sign:n \{⟨fpexpr⟩\}  
Evaluates the ⟨fpexpr⟩ and leaves its sign in the input stream using \fp_eval:n \{sign(⟨result⟩)\}: +1 for positive numbers and for +∞, −1 for negative numbers and for −∞, ±0 for ±0. If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is 0.

\fp_decimal:N \{⟨fp expr⟩\}  
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_decimal:n and they are combined as (⟨fp1⟩, ⟨fp2⟩, ..., ⟨fpn⟩) if n > 1 and (⟨fp1⟩) or () for fewer items.

\fp_to_dim:N \{⟨fp expr⟩\}  
Evaluates the ⟨floating point expression⟩ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range \([-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]\) of valid \TeX{} dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values ±∞ and NaN, trigger an “invalid operation” exception.

\fp_to_int:N \{⟨fp expr⟩\}  
Evaluates the ⟨floating point expression⟩, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range \([-2^{31} + 1, 2^{31} - 1]\) of valid \TeX{} integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values ±∞ and NaN, trigger an “invalid operation” exception.

\fp_to_scientific:N \{⟨fp expr⟩\}  
Evaluates the ⟨floating point expression⟩ and expresses the result in scientific notation:  
\langle optional -$\langle$digit$\rangle$,$\langle$15 digits$\rangle$e$\langle$optional sign$\rangle$$\langle$exponent$\rangle$\rangle  
The leading ⟨digit⟩ is non-zero except in the case of ±0. The values ±∞ and NaN trigger an “invalid operation” exception. Normal category codes apply: thus the e is category code 11 (a letter). For a tuple, each item is converted using \fp_to_scientific:n and they are combined as (⟨fp1⟩,$\cup$⟨fp2⟩,$\cup$...⟨fpn⟩) if n > 1 and (⟨fp1⟩) or () for fewer items.

201
\fp_to_tl:N \fp_to_tl:c \fp_to_tl:n

Evaluates the \textit{(floating point expression)} and expresses the result in (almost) the shortest possible form. Numbers in the ranges \((0, 10^{-3})\) and \([10^{10}, \infty)\) are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from \texttt{fp_to_scientific:n}). Numbers in the range \([10^{-3}, 10^{10})\) are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see \texttt{fp_to_decimal:n}). Negative numbers start with \texttt{-}. The special values \(\pm 0\), \(\pm \infty\) and \texttt{NaN} are rendered as \(0\), \(-0\), \(\texttt{inf}\), \(-\texttt{inf}\), and \texttt{nan} respectively. Normal category codes apply and thus \texttt{inf} or \texttt{nan}, if produced, are made up of letters. For a tuple, each item is converted using \texttt{fp_to_tl: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. This function is identical to \texttt{fp_to_decimal:n}.

\fp_use:N \fp_use:c

Inserts the value of the \texttt{(fp var)} into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values \(\pm 0\) and \texttt{NaN} trigger an “invalid operation” exception. For a tuple, each item is converted using \texttt{fp_to_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. This function is identical to \texttt{fp_to_decimal:n}.

\section{Floating point conditionals}

\fp_if_exist_p:N \fp_if_exist_p:c \fp_if_exist:N \fp_if_exist_p:N \fp_if_exist_p:c

Tests whether the \texttt{(fp var)} is currently defined. This does not check that the \texttt{(fp var)} really is a floating point variable.

\fp_compare_p:nNn \fp_compare_p:nNF \fp_compare_p:nN \fp_compare_p:nNF

Compares the \texttt{(fpexpr1)} and the \texttt{(fpexpr2)}, and returns \texttt{true} if the \texttt{(relation)} is obeyed. Two floating points \(x\) and \(y\) may obey four mutually exclusive relations: \(x < y\), \(x = y\), \(x > y\), or \(x ? y\) (“not ordered”). The last case occurs exactly if one or both operands is \texttt{NaN} or is a tuple, unless they are equal tuples. Note that a \texttt{NaN} is distinct from any value, even another \texttt{NaN}, hence \(x = x\) is not true for a \texttt{NaN}. To test if a value is \texttt{NaN}, compare it to an arbitrary number with the “not ordered” relation.

\texttt{\fp_compare:nNnTF \{<value>\} ? \{0\} \{\} % <value> is nan \{\} % <value> is not nan}

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 \(\texttt{false}\) (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}.
\texttt{\bf \texttt{fp_compare_p:n}} *
\texttt{\bf \texttt{fp_compare:nTF}} *

Updated: 2013-12-14

\texttt{\bf \texttt{fp_compare_p:n}}
\begin{verbatim}
{ (fpexpr₁) \texttt{\langle relation₁ \rangle} \\
  \ldots \\
  (fpexprₙ) \texttt{\langle relationₙ \rangle} \\
  (fpexpr_{n+1}) }
\end{verbatim}

\texttt{\bf \texttt{fp_compare:nTF}}
\begin{verbatim}
{ (fpexpr₁) \texttt{\langle relation₁ \rangle} \\
  \ldots \\
  (fpexprₙ) \texttt{\langle relationₙ \rangle} \\
  (fpexpr_{n+1}) }
\end{verbatim}

\begin{verbatim}
\texttt{\langle true code \rangle} \texttt{\{\langle false code \rangle\}}
\end{verbatim}

Evaluates the \texttt{\langle floating point expressions \rangle} as described for \texttt{\bf \texttt{fp_eval:n}} and compares consecutive result using the corresponding \texttt{\langle relation \rangle}, namely it compares \texttt{\langle fpexpr₁ \rangle} and \texttt{\langle fpexpr₂ \rangle} using the \texttt{\langle relation₁ \rangle}, then \texttt{\langle fpexpr₂ \rangle} and \texttt{\langle fpexpr₃ \rangle} using the \texttt{\langle relation₂ \rangle}, until finally comparing \texttt{\langle fpexprₙ \rangle} and \texttt{\langle fpexpr_{n+1} \rangle} using the \texttt{\langle relationₙ \rangle}. The test yields \texttt{true} if all comparisons are \texttt{true}. Each \texttt{\langle floating point expression \rangle} is evaluated only once.

Contrarily to \texttt{\bf \texttt{int_compare:nTF}}, all \texttt{\langle floating point expressions \rangle} are computed, even if one comparison is \texttt{false}. Two floating points \texttt{x} and \texttt{y} may obey four mutually exclusive relations: \texttt{x < y}, \texttt{x = y}, \texttt{x > y}, or \texttt{x 
ot= 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{\langle relation \rangle} can be any (non-empty) combination of \texttt{<}, \texttt{=}, \texttt{>} and \texttt{?}, plus an optional leading \texttt{!} (which negates the \texttt{\langle relation \rangle}), with the restriction that the \texttt{\langle relation \rangle} may not start with \texttt{?}, as this symbol has a different meaning (in combination with \texttt{:}) within floating point expressions. The comparison \texttt{x \langle relation \rangle y} is then \texttt{true} if the \texttt{\langle relation \rangle} does not start with \texttt{!} and the actual relation \texttt{<}, \texttt{=}, \texttt{>} or \texttt{?} between \texttt{x} and \texttt{y} appears within the \texttt{\langle relation \rangle}, or on the contrary if the \texttt{\langle relation \rangle} starts with \texttt{!} and the relation between \texttt{x} and \texttt{y} does not appear within the \texttt{\langle relation \rangle}. Common choices of \texttt{\langle relation \rangle} include \texttt{\geq} (greater or equal), \texttt{\neq} (not equal), \texttt{\not=} (comparable).

This function is more flexible than \texttt{\bf \texttt{fp_compare:nNnTF}} and only slightly slower.

5 Floating point expression loops

\texttt{\bf \texttt{fp_do_until:nNnn}} *
\begin{verbatim}
\texttt{\bf \texttt{fp_do_until:nNnn}} \texttt{(fpexpr₁)} \texttt{\langle relation \rangle} \texttt{(fpexpr₂)} \texttt{\langle code \rangle}
\end{verbatim}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\bf \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{\langle relation \rangle} is \texttt{true}.

\texttt{\bf \texttt{fp_do_while:nNnn}} *
\begin{verbatim}
\texttt{\bf \texttt{fp_do_while:nNnn}} \texttt{(fpexpr₁)} \texttt{\langle relation \rangle} \texttt{(fpexpr₂)} \texttt{\langle code \rangle}
\end{verbatim}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\bf \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{\langle relation \rangle} is \texttt{false}.

203
\fp_until_do:nNnn \star \fp_until_do:nNnn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF, and then places the (code) in the input stream if the (relation) is false. After the (code) has been processed by \TeX the test is repeated, and a loop occurs until the test is true.

\fp_while_do:nNnn \star \fp_while_do:nNnn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF, and then places the (code) in the input stream if the (relation) is true. After the (code) has been processed by \TeX the test is repeated, and a loop occurs until the test is false.

\fp_do_until:nn \star \fp_do_until:nn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Places the (code) in the input stream for \TeX to process, and then evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF. If the test is false then the (code) is inserted into the input stream again and a loop occurs until the (relation) is true.

\fp_do_while:nn \star \fp_do_while:nn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Places the (code) in the input stream for \TeX to process, and then evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF. If the test is true then the (code) is inserted into the input stream again and a loop occurs until the (relation) is false.

\fp_until_do:nn \star \fp_until_do:nn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF, and then places the (code) in the input stream if the (relation) is false. After the (code) has been processed by \TeX the test is repeated, and a loop occurs until the test is true.

\fp_while_do:nn \star \fp_while_do:nn \{ \fpexpr \} \{ \relation \} \{ \fpexpr \} \{ \code \}

Evaluates the relationship between the two (floating point expressions) as described for \fp_compare:nNnTF, and then places the (code) in the input stream if the (relation) is true. After the (code) has been processed by \TeX the test is repeated, and a loop occurs until the test is false.
\texttt{\textbackslash fp\_step\_function:nnnN} \textbackslash  \texttt{\textbackslash fp\_step\_function:nnnc} \textbackslash  

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}). 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 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{itemize}
\item [I saw 1.0]
\item [I saw 1.1]
\item [I saw 1.2]
\item [I saw 1.3]
\item [I saw 1.4]
\item [I saw 1.5]
\end{itemize}

\textbf{\textbackslash \texttt{\textbackslash \textbackslash \texttt{\textbackslash FP\_Hackers 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.

\texttt{\textbackslash fp\_step\_inline:nnn} \textbackslash  \texttt{\textbackslash fp\_step\_variable:nnnn} \textbackslash  

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 \textit{#1} replaced by the current \textit{value}. Thus the \textit{code} should define a function of one argument (\textit{#1}).

\section{Some useful constants, and scratch variables}

\begin{verbatim}
\texttt{\textbackslash c\_zero\_fp} \texttt{\textbackslash c\_minus\_zero\_fp}
\end{verbatim}

Zero, with either sign.

\begin{verbatim}
\texttt{\textbackslash c\_one\_fp}
\end{verbatim}

One as an \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, \(e = \exp(1)\).

The value of \(\pi\). This can be input directly in a floating point expression as \texttt{pi}.

The value of \(1^\circ\) 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.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as \(0 / 0\), or \(10^{10000}\). The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- **Overflow** occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in \(\pm \infty\).

- **Underflow** occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in \(\pm 0\).

- **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 (\textit{e.g.}, \texttt{\textbackslash fp\_to\_dim:n}).

- **Division by zero** occurs when dividing a non-zero number by 0, or when evaluating functions at poles, \textit{e.g.}, \texttt{ln}(0) or \texttt{cot}(0). This results in \(\pm \infty\).
(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 \LaTeX3.

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 \langle\texttt{exception}\rangle \texttt{(overflow, underflow, invalid\_operation or division\_by\_zero)} within the current group are treated as \langle\texttt{trap type}\rangle, which can be

- \texttt{none}: the \langle\texttt{exception}\rangle will be entirely ignored, and leave no trace;
- \texttt{flag}: the \langle\texttt{exception}\rangle will turn the corresponding flag on when it occurs;
- \texttt{error}: additionally, the \langle\texttt{exception}\rangle will halt the \TeX{} run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

Flags denoting the occurrence of various floating-point exceptions.

\begin{verbatim}
flag\_fp\_overflow
flag\_fp\_underflow
flag\_fp\_invalid\_operation
flag\_fp\_division\_by\_zero
\end{verbatim}

8 Viewing floating points

\begin{verbatim}
\fp\_show:N \langlefp\ var\rangle
\fp\_show:c \langle\texttt{floating point expression}\rangle
\fp\_show:n
\end{verbatim}

Evaluates the \langle\texttt{floating point expression}\rangle and displays the result in the terminal.

\begin{verbatim}
\fp\_log:N \langlefp\ var\rangle
\fp\_log:c \langle\texttt{floating point expression}\rangle
\fp\_log:n
\end{verbatim}

Evaluates the \langle\texttt{floating point expression}\rangle 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 $e$ or $E$, 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 $\c_e_fp$ (which is faster).

Special numbers are input as follows:

- $\mathbf{inf}$ represents $+\infty$, and can be preceded by any $\langle \text{sign} \rangle$, yielding $\pm \infty$ as appropriate.
- $\mathbf{nan}$ 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 $\\text{\texttt{\textbackslash inf\textbackslash y}}$, $\\text{\texttt{\textbackslash pi}}$, or $\sin$ 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.

208
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 (2\(\pi\)) when neither factor is in parentheses.
- Binary \(*\) and \(/\), implicit multiplication by juxtaposition with parentheses (for instance 3(4+5)).
- Binary + and −.
- Comparisons \(\geq\), \(!=\), \(<?\), 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

\[
1/2\pi = 1/(2\pi), \\
1/2\pi(\pi + \pi) = (2\pi)^{-1}(\pi + \pi) \approx 1, \\
\sin2\pi = \sin(2)\pi \neq 0, \\
2 \cdot 2\text{max}(3, 5) = 2^2 \text{max}(3, 5) = 20, \\
1\text{in}/1\text{cm} = (1\text{in})/(1\text{cm}) = 2.54.
\]

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 \texttt{false} if it is \(\pm 0\), and \texttt{true} otherwise, including when it is \texttt{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 \texttt{NaN} result.
The ternary operator `?:` results in `<operand2>` if `<operand1>` is true (not ±0), and `<operand3>` if `<operand1>` is false (±0). All three `<operands>` are evaluated in all cases; they may be tuples. The operator is right associative, hence

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle ? \langle operand2 \rangle : \langle operand3 \rangle \}}} \\
\text{The ternary operator `?:` results in `<operand2>` if `<operand1>` is true (not ±0), and `<operand3>` if `<operand1>` is false (±0). All three `<operands>` are evaluated in all cases; they may be tuples. The operator is right associative, hence}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\{}} \\
1 + 3 > 4 \ ? \ 1 : \\
2 + 4 > 5 \ ? \ 2 : \\
3 + 5 > 6 \ ? \ 3 : 4 \\
\text{\texttt{\}}} \\
\text{first tests whether 1 + 3 > 4; since this isn’t true, the branch following : is taken, and 2 + 4 > 5 is compared; since this is true, the branch before : is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle || \langle operand2 \rangle \}}} \\
\text{If `<operand1>` is true (not ±0), use that value, otherwise the value of `<operand2>`}. \text{Both `<operands>` are evaluated in all cases; they may be tuples. In `<operand1> || <operand2>` || ... || `<operands>n`, the first true (nonzero) `<operand>` is used and if all are zero the last one (±0) is used.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle && \langle operand2 \rangle \}}} \\
\text{If `<operand1>` is false (equal to ±0), use that value, otherwise the value of `<operand2>`}. \text{Both `<operands>` are evaluated in all cases; they may be tuples. In `<operand1> && <operand2>` && ... && `<operands>n`, the first false (±0) `<operand>` is used and if none is zero the last one is used.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\langle operand1 \rangle \langle relation1 \rangle}} \\
\text{\ldots} \\
\text{\langle operandN \rangle \langle relationN \rangle} \\
\text{\langle operandN+1 \rangle} \\
\text{\}}} \\
\text{Each `<relation>` consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by !, and may not start with `?`. This evaluates to +1 if all comparisons `<operandi> <relationi>` `<operandi>1` are true, and +0 otherwise. All `<operands>` are evaluated (once) in all cases. See \texttt{\textbackslash fp_compare:nTF} for details.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle + \langle operand2 \rangle \}}} \\
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle - \langle operand2 \rangle \}}} \\
\text{Computes the sum or the difference of its two `<operands>`. The “invalid operation” exception occurs for \infty - \infty. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle \langle relation1 \rangle}} \\
\text{\ldots} \\
\text{\langle operandN \rangle \langle relationN \rangle} \\
\text{\langle operandN+1 \rangle} \\
\text{\}}} \\
\text{Each `<relation>` consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by !, and may not start with `?`. This evaluates to +1 if all comparisons `<operandi> <relationi>` `<operandi>1` are true, and +0 otherwise. All `<operands>` are evaluated (once) in all cases. See \texttt{\textbackslash fp_compare:nTF} for details.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle + \langle operand2 \rangle \}}} \\
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle - \langle operand2 \rangle \}}} \\
\text{Computes the sum or the difference of its two `<operands>`. The “invalid operation” exception occurs for \infty - \infty. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle \langle relation1 \rangle}} \\
\text{\ldots} \\
\text{\langle operandN \rangle \langle relationN \rangle} \\
\text{\langle operandN+1 \rangle} \\
\text{\}}} \\
\text{Each `<relation>` consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by !, and may not start with `?`. This evaluates to +1 if all comparisons `<operandi> <relationi>` `<operandi>1` are true, and +0 otherwise. All `<operands>` are evaluated (once) in all cases. See \texttt{\textbackslash fp_compare:nTF} for details.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle + \langle operand2 \rangle \}}} \\
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle - \langle operand2 \rangle \}}} \\
\text{Computes the sum or the difference of its two `<operands>`. The “invalid operation” exception occurs for \infty - \infty. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.}
\end{array}
\]

\[
\begin{array}{l}
\text{\texttt{\textbackslash fp_eval:n \{ \langle operand1 \rangle \langle relation1 \rangle}} \\
\text{\ldots} \\
\text{\langle operandN \rangle \langle relationN \rangle} \\
\text{\langle operandN+1 \rangle} \\
\text{\}}} \\
\text{Each `<relation>` consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by !, and may not start with `?`. This evaluates to +1 if all comparisons `<operandi> <relationi>` `<operandi>1` are true, and +0 otherwise. All `<operands>` are evaluated (once) in all cases. See \texttt{\textbackslash fp_compare:nTF} for details.}
\end{array}
\]
\[ \text{ Computes the product or the ratio of its two operands. The “invalid operation” exception occurs for } \infty/\infty, \, 0/0, \text{ or } 0 * \infty. \text{ “Division by zero” occurs when dividing a finite non-zero number by } \pm 0. \text{ “Underflow” and “overflow” occur when appropriate. When } \langle \text{operand}_1 \rangle \text{ is a tuple and } \langle \text{operand}_2 \rangle \text{ is a floating point number, each item of } \langle \text{operand}_1 \rangle \text{ is multiplied or divided by } \langle \text{operand}_2 \rangle. \text{ Other combinations yield an “invalid operation” exception and a NaN result.} \]

\[ \text{ The unary } + \text{ does nothing, the unary } - \text{ changes the sign of the } \langle \text{operand} \rangle \text{ (for a tuple, of all its components), and } ! \langle \text{operand} \rangle \text{ evaluates to 1 if } \langle \text{operand} \rangle \text{ is false (is } \pm 0) \text{ and 0 otherwise (this is the not boolean function). Those operations never raise exceptions.} \]

\[ \text{ Raises } \langle \text{operand}_1 \rangle \text{ to the power } \langle \text{operand}_2 \rangle. \text{ This operation is right associative, hence } 2 \rightarrow 2 \rightarrow 3 \rightarrow 256. \text{ If } \langle \text{operand}_1 \rangle \text{ is negative or } -0 \text{ then: the result’s sign is } + \text{ if the } \langle \text{operand}_2 \rangle \text{ is infinite and } (-1)^p \text{ if the } \langle \text{operand}_2 \rangle \text{ is } p/q \text{ with } p, q \text{ integers; the result is } +0 \text{ if } \abs{\langle \text{operand}_1 \rangle}^\langle \text{operand}_2 \rangle \text{ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising } \pm 0 \text{ to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.} \]

\[ \text{ Computes the absolute value of the } \langle \text{fpexpr} \rangle. \text{ If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also } \text{\texttt{\textbackslash fp_abs:n}.} \]

\[ \text{ Computes the exponential of the } \langle \text{fpexpr} \rangle. \text{ “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.} \]

\[ \text{ Computes the factorial of the } \langle \text{fpexpr} \rangle. \text{ If the } \langle \text{fpexpr} \rangle \text{ is an integer between } -0 \text{ and } 3248 \text{ included, the result is finite and correctly rounded. Larger positive integers give } +\infty \text{ with “overflow”, while } \text{\texttt{\textbackslash fact}(+\infty) = +\infty} \text{ and } \text{\texttt{\textbackslash fact}(nan) = nan} \text{ with no exception. All other inputs give NaN with the “invalid operation” exception.} \]

\[ \text{ Computes the natural logarithm of the } \langle \text{fpexpr} \rangle. \text{ Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for } \ln(-0). \text{ “Division by zero” occurs when evaluating } \ln(+0) = -\infty. \text{ “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.} \]
\texttt{logb} \quad \texttt{\textbackslash fp\_eval:n \{ logb ( \textbackslash{fpexpr} ) \} }

Determines the exponent of the \langle fpexpr \rangle, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\log_{10}(0) = -\infty$. Other special values are $\log_{10}(\pm\infty) = +\infty$ and $\log_{10}(\text{NaN}) = \text{NaN}$. If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is NaN.

\texttt{max} \quad \texttt{\textbackslash fp\_eval:n \{ max ( \textbackslash{fpexpr}_1 , \textbackslash{fpexpr}_2 , \ldots ) \} }

\texttt{min} \quad \texttt{\textbackslash fp\_eval:n \{ min ( \textbackslash{fpexpr}_1 , \textbackslash{fpexpr}_2 , \ldots ) \} }

Evaluates each \langle fpexpr \rangle and computes the largest (smallest) of those. If any of the \langle fpexpr \rangle 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{\textbackslash fp\_eval:n \{ round ( \textbackslash{fpexpr} ) \} }

\texttt{\textbackslash fp\_eval:n \{ round ( \textbackslash{fpexpr}_1 , \textbackslash{fpexpr}_2 ) \} }

\texttt{\textbackslash fp\_eval:n \{ round ( \textbackslash{fpexpr}_1 , \textbackslash{fpexpr}_2 , \textbackslash{fpexpr}_3 ) \} }

Only round accepts a third argument. Evaluates \langle fpexpr_1 \rangle = x and \langle fpexpr_2 \rangle = n and \langle fpexpr_3 \rangle = t then rounds $x$ to $n$ places. If $n$ is an integer, this rounds $x$ to a multiple of $10^{-n}$; if $n = +\infty$, this always yields $x$; if $n = -\infty$, this yields one of $\pm 0$, $\pm \infty$, or NaN; if $n = \text{NaN}$, this yields NaN; if $n$ is neither $\pm \infty$ nor an integer, then an “invalid operation” exception is raised. When \langle fpexpr_2 \rangle is omitted, $n = 0$, i.e., \langle fpexpr_1 \rangle is rounded to an integer. The rounding direction depends on the function.

- **round** yields the multiple of $10^{-n}$ closest to $x$, with ties ($x$ half-way between two such multiples) rounded as follows. If $t$ is NaN (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if $t$ is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”).

- **floor** yields the largest multiple of $10^{-n}$ smaller or equal to $x$ (“round towards negative infinity”);

- **ceil** yields the smallest multiple of $10^{-n}$ greater or equal to $x$ (“round towards positive infinity”);

- **trunc** yields a multiple of $10^{-n}$ with the same sign as $x$ and with the largest absolute value less than that of $x$ (“round towards zero”).

“Overflow” occurs if $x$ is finite and the result is infinite (this can only happen if \langle fpexpr_2 \rangle $< -9984$). If any operand is a tuple, “invalid operation” occurs.

\texttt{\textbackslash fp\_eval:n \{ sign ( \textbackslash{fpexpr} ) \} }

Evaluates the \langle fpexpr \rangle and determines its sign: $+1$ for positive numbers and $+\infty$, $-1$ for negative numbers and for $-\infty$, $\pm 0$ for $\pm 0$, and NaN for NaN. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.
\fp_eval:n { \sin( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \cos( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \tan( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \cot( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \csc( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \sec( \langle \text{fexpr} \rangle ) } \\

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \langle \text{fexpr} \rangle given in radians. For arguments given in degrees, see \text{sind}, \text{cosd}, etc. Note that since \pi is irrational, \sin(8\pi) is not quite zero, while its analogue \text{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.

Updated: 2013-11-17

\fp_eval:n { \sind( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \cosd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \tand( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \cotd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \cscd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \secd( \langle \text{fexpr} \rangle ) } \\

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \langle \text{fexpr} \rangle given in degrees. For arguments given in radians, see \text{sin}, \text{cos}, etc. Note that since \pi is irrational, \sin(8\pi) is not quite zero, while its analogue \text{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.

New: 2013-11-02

\fp_eval:n { \asind( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \acosd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \acscd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \asecd( \langle \text{fexpr} \rangle ) } \\

Computes the arcsine, arccosine, arccosecant, or arccosecant of the \langle \text{fexpr} \rangle and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for \text{asin} and \text{acsc} and $[0, \pi]$ for \text{acos} and \text{asec}. For a result in degrees, use \text{asind}, etc. If the argument of \text{asin} or \text{acos} lies outside the range $[-1,1]$, or the argument of \text{acsc} or \text{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.

New: 2013-11-02

\fp_eval:n { \asind( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \acosd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \acscd( \langle \text{fexpr} \rangle ) } \\
\fp_eval:n { \asecd( \langle \text{fexpr} \rangle ) } \\

Computes the arcsine, arccosine, arccosecant, or arccosecant of the \langle \text{fexpr} \rangle and returns the result in degrees, in the range $[-90, 90]$ for \text{asin} and \text{acsc} and $[0, 180]$ for \text{acos} and \text{asec}. For a result in radians, use \text{asind}, etc. If the argument of \text{asin} or \text{acos} lies outside the range $[-1,1]$, or the argument of \text{acsc} or \text{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.

New: 2013-11-02
\begin{verbatim}
atan \fp_eval:n \{ atan( \fpexpr ) \}
acot \fp_eval:n \{ acot( \fpexpr ) \}
\fp_eval:n \{ acot( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ atand( \fpexpr ) \}
\fp_eval:n \{ atand( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ acotd( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ acotd( \fpexpr_1 , \fpexpr_2 ) \}
\end{verbatim}

Those functions yield an angle in radians: \texttt{atand} and \texttt{acotd} are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the \texttt{\langle fexpr \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 \((\texttt{\langle fexpr_2 \rangle}, \texttt{\langle fexpr_1 \rangle})\): this is the arctangent of \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\), possibly shifted by \(\pi\) depending on the signs of \(\langle fexpr_1 \rangle\) and \(\langle fexpr_2 \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\texttt{\langle fexpr_1 \rangle}, \texttt{\langle fexpr_2 \rangle})\), equal to the arccotangent of \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\), possibly shifted by \(\pi\). Both two-argument functions take values in the wider range \([-\pi, \pi]\). The ratio \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), or when both yield \(\pm \infty\), the resulting angle is one of \\{\pm \pi/4, \pm 3\pi/4\} depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

\begin{verbatim}
atan \fp_eval:n \{ atan( \fpexpr ) \}
acot \fp_eval:n \{ acot( \fpexpr ) \}
\fp_eval:n \{ acot( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ atand( \fpexpr ) \}
\fp_eval:n \{ atand( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ acotd( \fpexpr_1 , \fpexpr_2 ) \}
\fp_eval:n \{ acotd( \fpexpr_1 , \fpexpr_2 ) \}
\end{verbatim}

Those functions yield an angle in degrees: \texttt{atand} and \texttt{acotd} are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the \texttt{\langle fexpr \rangle}: arctangent takes values in the range \([-90, 90]\), and arccotangent in the range \([0, 180]\). The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates \((\texttt{\langle fexpr_2 \rangle}, \texttt{\langle fexpr_1 \rangle})\): this is the arctangent of \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\), possibly shifted by \(180\) depending on the signs of \(\langle fexpr_1 \rangle\) and \(\langle fexpr_2 \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\texttt{\langle fexpr_1 \rangle}, \texttt{\langle fexpr_2 \rangle})\), equal to the arccotangent of \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\), possibly shifted by \(180\). Both two-argument functions take values in the wider range \([-180, 180]\). The ratio \(\langle fexpr_1 \rangle / \langle fexpr_2 \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), or when both yield \(\pm \infty\), the resulting angle is one of \\{\pm 45, \pm 135\} depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

\begin{verbatim}
\sqrt \fp_eval:n \{ sqrt( \fpexpr ) \}
\end{verbatim}

Computes the square root of the \texttt{\langle fexpr \rangle}. The “invalid operation” is raised when the \texttt{\langle fexpr \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 \(\sqrt{NaN} = NaN\).
\fp_eval:n { rand() }\nopagebreak

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 \(\LaTeX\). The random seed can be queried using \texttt{\sys_rand_seed:} and set using \texttt{\sys_gset_rand_seed:n}.\nopagebreak

\textbf{\texttt{\LaTeX}hackers note:} This is based on pseudo-random numbers provided by the engine's primitive \texttt{\pdfuniformdeviate} in \texttt{pdfTeX}, \texttt{\psTeX}, \texttt{\upTeX} and \texttt{\uniformdeviate} in \texttt{LuaTeX} and \texttt{\XeTeX}. 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{\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.\nopagebreak

\fp_eval:n { randint(⟨fpexpr⟩) } \nopagebreak
\fp_eval:n { randint(⟨fpexpr⟩, ⟨fpexpr⟩) } \nopagebreak

Produces a pseudo-random integer between 1 and \(⟨\text{fpexpr}⟩\) or between \(⟨\text{fpexpr}_1⟩\) and \(⟨\text{fpexpr}_2⟩\) inclusive. The bounds must be integers in the range \((-10^{16}, 10^{16})\) and the first must be smaller or equal to the second. See \texttt{rand} for important comments on how these pseudo-random numbers are generated.\nopagebreak

\begin{itemize}
\item \texttt{\inf}\nopagebreak
\item \texttt{\nan}\nopagebreak
\end{itemize}

The special values \(+\infty\), \(-\infty\), and \texttt{NaN} are represented as \texttt{inf}, \texttt{-inf} and \texttt{nan} (see \texttt{\c_-inf_fp}, \texttt{\c_minus_inf_fp} and \texttt{\c_nan_fp}).\nopagebreak

\begin{itemize}
\item \texttt{\pi}\nopagebreak
\end{itemize}

The value of \(\pi\) (see \texttt{\c_pi_fp}).\nopagebreak

\begin{itemize}
\item \texttt{\deg}\nopagebreak
\end{itemize}

The value of \(1^\circ\) in radians (see \texttt{\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 \texttt{1} and \texttt{+0}.

\begin{verbatim}
\fp_abs:n {⟨floating point expression⟩}
\end{verbatim}

Evaluates the \texttt{⟨floating point expression⟩} as described for \texttt{\fp_eval:n} and leaves the absolute value of the result in the input stream. If the argument is ±\infty, \texttt{NaN} or a tuple, “invalid operation” occurs. Within floating point expressions, \texttt{abs()} can be used; it accepts ±\infty and \texttt{NaN} as arguments.

\begin{verbatim}
\fp_max:nn {⟨fp expression 1⟩} {⟨fp expression 2⟩}
\end{verbatim}

Evaluates the \texttt{⟨floating point expressions⟩} as described for \texttt{\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{\texttt{fp_format}:nn \{\texttt{fpexpr}\} \{\texttt{format}\}}, but what should \{\texttt{format}\} be? More general pretty printing?
• Add \texttt{and}, \texttt{or}, \texttt{xor}? Perhaps under the names \texttt{all}, \texttt{any}, and \texttt{xor}?
• Add $\log(x,b)$ 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 !), 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{\texttt{fp_if_nan}:nTF}?
• Support keyword arguments?
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 ?, !\leq?, \&\&, |\mid, and \texttt{!}.
• Logarithms of numbers very close to 1 are inaccurate.
• When rounding towards $-\infty$, \texttt{\texttt{dim_to_fp}:n \{0pt\}} should return $-0$, not $+0$.
• The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
• 0e9999999999 gives a \TeX{} “number too large” error.
• Subnormals are not implemented.
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 \texttt{parse} auxiliaries for each operation were set up in the corresponding module, rather than centralizing in \texttt{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 \texttt{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 $c = \frac{2000}{\lfloor 200x \rfloor + 1} \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of $t$. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.

• Improve notations in the explanations of the division algorithm (\texttt{l3fp-basics}).

• Understand and document \texttt{\_\_fp\_basics\_pack\_weird\_low:NNNNw} and \texttt{\_\_fp\_-basics\_pack\_weird\_high:NNNNNNNNw} 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 \texttt{TeX} fp packages, the international standards,…

• Also take into account the “inexact” exception?

• Support multi-character prefix operators (\texttt{e.g., @} 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.

\lfparray_new:Nn \fparray_new:Nn \lfparray var \{size\}

Evaluates the integer expression \langle size \rangle and allocates a \langle floating point array variable \rangle with that number of \langle zero \rangle entries. The variable name should start with \g_ because assignments are always global.

\lfparray_count:N \fparray_count:N \lfparray var

Expands to the number of entries in the \langle floating point array variable \rangle. This is performed in constant time.

\lfparray_gset:Nnn \fparray_gset:Nnn \lfparray var \{position\} \{value\}

Stores the result of evaluating the floating point expression \langle value \rangle into the \langle floating point array variable \rangle at the \langle integer expression \rangle \langle position \rangle. If the \langle position \rangle is not between 1 and the \lfparray_count:N, an error occurs. Assignments are always global.

\lfparray_gzero:N \fparray_gzero:N \lfparray var

Sets all entries of the \langle floating point array variable \rangle to +0. Assignments are always global.

\lfparray_item:Nn \fparray_item:Nn \lfparray var \{position\}

\lfparray_item_to_tl:Nn \fparray_item_to_tl:Nn \lfparray var \{position\}

Applies \lfp_use:N or \lfp_to_tl:N (respectively) to the floating point entry stored at the \langle integer expression \rangle \langle position \rangle in the \langle floating point array variable \rangle. If the \langle position \rangle is not between 1 and the \lfparray_count:N, an error occurs.
Part XXV

The l3sort package

Sorting functions

1 Controlling sorting

\l3\TeX{} 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 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 comparison code \rangle consisting only of \texttt{\sort_return_swapped} reverses the list (in a fairly inefficient way).

\texttt{\texttt{T\TeX{}hackers note:}} The current implementation is limited to sorting approximately 20000 items (40000 in Lu\TeX{}), depending on what other packages are loaded.

Internally, the code from \texttt{l3sort} stores items in \texttt{\toks} registers allocated locally. Thus, the \langle 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}
\seq_sort:Nn \seq_var
{ \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\ldots} 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 \l3tl-analysis package: Analysing token lists

1 \l3tl-analysis documentation

This module mostly provides internal functions for use in the \l3regex module. However, it provides as a side effect a user debugging function, very similar to the \ShowTokens macro from the ted package.

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

Displays to the terminal the detailed decomposition of the \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.

\tl_analysis_map_inline:nn \{\langle token list \rangle\} \{\langle inline function \rangle\}

Applies the \langle inline function \rangle to each individual \langle token \rangle in the \langle token list \rangle. The \langle inline function \rangle receives three arguments:

- \langle tokens \rangle, which both o-expand and x-expand to the \langle token \rangle. The detailed form of \langle token \rangle may change in later releases.

- \langle char code \rangle, a decimal representation of the character code of the token, –1 if it is a control sequence (with \langle catcode \rangle 0).

- \langle catcode \rangle, a capital hexadecimal digit which denotes the category code of the \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 \langle inline function \rangle remain in effect after the loop.
Part XXVII

The \texttt{l3regex} package: Regular expressions in \TeX

The \texttt{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\textemdash cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
\end{verbatim}
the token list variable \texttt{\l_my_tl} holds the text \textquote{This\textemdash cat.}, where the first occurrence of \textquote{at} was replaced by \textquote{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 \texttt{\w} sequence represents any \textquote{word} character, and + indicates that the \texttt{\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, \texttt{\0} denotes the full match (here, a word). The command \texttt{\emph} is inserted using \texttt{\c{emph}}, and its argument \texttt{\0} is put between braces \texttt{\cB\{} and \texttt{\cE\}}.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using \texttt{\regex_const:Nn}. For example,
\begin{verbatim}
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (~BE).* \cE. }
\end{verbatim}
stores in \texttt{\c_foo_regex} a regular expression which matches the starting marker for an environment: \textquote{\begin}, followed by a begin-group token (\texttt{\cB.}), then any number of tokens which are neither begin-group nor end-group character tokens (\texttt{~BE}.*), ending with an end-group token (\texttt{\cE.}). As explained in the next section, the parentheses \textquote{capture} the result of \texttt{~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 \texttt{\regex_show:n} to these regular expressions.

- \texttt{Cat} matches the word \textquote{Cat} capitalized in this way, but also matches the beginning of the word \textquote{Cattle}: use \texttt{\bCat\b} to match a complete word only.
- \texttt{[abc]} matches one letter among \textquote{a}, \textquote{b}, \textquote{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 *) 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|m|ex|m|cm|pt|in|cm)\._\* matches an explicit dimension with any unit that \TeX{} knows, where (?i) means to treat lowercase and uppercase letters identically.

• \([\+\-]\)*((?i)nan|inf|\(\d+\|\d*\.|\d+\)\.*((?i)pt|in|m|ex|m|cm|pt|in|m)\._\* 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:mnN when the goal is to extract matches or submatches in a finer way than with \regex_extract_all:mnN.

While it is impossible for a regular expression to match only integer expressions, \([\+\-]\)*\(\d+\|\d*\.|\d+\)\.*((?i)pt|in|m|ex|m|cm|pt|in|m|cm)\._\* 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 \(, \), ?, .);

• 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.

223
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, \texttt{\%abc\%} matches
the characters \texttt{abc\%} (with arbitrary category codes), but does not match the control
sequence \texttt{\%abc} followed by a percent character. Matching control sequences can be done
using the \texttt{\{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.

\texttt{\textbackslash x\{hh\ldots\}} Character with hex code hh\ldots

\texttt{\textbackslash xhh} Character with hex code hh.

\texttt{\textbackslash a} Alarm (hex 07).

\texttt{\textbackslash e} Escape (hex 1B).

\texttt{\textbackslash f} Form-feed (hex 0C).

\texttt{\textbackslash n} New line (hex 0A).

\texttt{\textbackslash r} Carriage return (hex 0D).

\texttt{\textbackslash t} Horizontal tab (hex 09).

Character types.

. A single period matches any token.

\texttt{\textbackslash d} Any decimal digit.

\texttt{\textbackslash h} Any horizontal space character, equivalent to \texttt{[\textbackslash \textasciicircum~I]}: space and tab.

\texttt{\textbackslash s} Any space character, equivalent to \texttt{[\textbackslash \textasciicircum~I\textbackslash~J\textbackslant~L\textbackslant~M]}.

\texttt{\textbackslash v} Any vertical space character, equivalent to \texttt{[\textbackslant~J\textbackslant~K\textbackslant~L\textbackslant~M]}.

Note that \texttt{\textbackslant~K}
is a vertical space, but not a space, for compatibility with Perl.

\texttt{\textbackslash w} Any word character, \textit{i.e.}, alphanumerics and underscore, equivalent to the explicit
class \texttt{[A-Za-z0-9_]}.  

\texttt{\textbackslash D} Any token not matched by \texttt{\textbackslash d}.

\texttt{\textbackslash H} Any token not matched by \texttt{\textbackslash h}.

\texttt{\textbackslash N} Any token other than the \texttt{\textbackslash n} character (hex 0A).

\texttt{\textbackslash S} Any token not matched by \texttt{\textbackslash s}.

\texttt{\textbackslash V} Any token not matched by \texttt{\textbackslash v}.

\texttt{\textbackslash W} Any token not matched by \texttt{\textbackslash w}.

Of those, ., \texttt{\textbackslash D}, \texttt{\textbackslash H}, \texttt{\textbackslash N}, \texttt{\textbackslash S}, \texttt{\textbackslash W}, and \texttt{\textbackslash W} match arbitrary control sequences.

Character classes match exactly one token in the subject.

\texttt{[\ldots\ldots]} Positive character class. Matches any of the specified tokens.

224
[^...\b] 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 ⟨name⟩, which can be alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, or 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.

\$, \Z 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.

(\ldots) Capturing group.

(?:\ldots) 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\{\begin{regex}\}\ A control sequence whose csname matches the \begin{regex}, anchored at the beginning and end, so that \c\begin{begin} matches exactly \begin{begin} and nothing else.

\c\X 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 CBEMTPUDSLOA). For instance, \c\{[A-Z]\d\} matches uppercase letters and digits of category code letter, \c\C matches any control sequence, and \c\O(abc) matches \textit{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 CBEMTPUDSLOA). For instance, \c\{LSO\}(...) matches two tokens of category letter, space, or other.

\c\{−XYZ\} Applies to the next object and prevents it from matching any token with category \X, \Y, or \Z (each being any of 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\{L0\}[A-F]] matches what \TeX\ considers as hexadecimal digits, namely digits with category other, or uppercase letters from \textit{A} to \textit{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\c\{\O\\}cd) matches \textit{abcd} where all characters are of category letter, except \texttt{*} 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, \u{\langle tl var name\rangle} matches the exact contents of the token list \langle tl var\rangle. Within a \c{...} 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:nn { 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:nnN { (. \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 (\ldots); similarly for \2, \ldots, \9 and \g{\ldots} (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 T\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 \s have category code 10, while spaces inserted through \x20 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_new:N \regex_new:N
\regex_set:Nn \regex_gset:Nn \regex_const:Nn
\regex_new:N \regex_new:N
\regex_set:Nn \regex_gset:Nn \regex_const:Nn
\regex_new:N \regex_set:Nn \regex_gset:Nn \regex_const:Nn
\regex_show:n \regex_show:N \regex_show:N
\regex_show:n \regex_show:N
\regex_match:nnTF \regex_match:nnTF \regex_match:nnTF
\regex_match:nnTF \regex_match:nnTF
\regex_match:nnTF \regex_match:nnTF

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_\texttt{(g)set:Nn}.

\regex_match:nnTF \regex_match:nnTF \regex_match:nnTF \regex_match:nnTF
\regex_match:nnTF \regex_match:nnTF \regex_match:nnTF
\regex_match:nnTF \regex_match:nnTF
\regex_match:nnTF \regex_match:nnTF

leaves \texttt{TRUE} then \texttt{FALSE} in the input stream.
\regex_count:nnN { (regex) } { (token list) } (int var)

Sets (int var) within the current \TeX{} group level equal to the number of times (regular expression) appears in (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) } { abbababcbb } \l_foo_int
\end{verbatim}

results in \l_foo_int taking the value 5.

5 Submatch extraction

\regex_extract_once:nnN { (regex) } { (token list) } (seq var)
\regex_extract_once:nnNTF { (regex) } { (token list) } (seq var) { (true code) } { (false code) }

Finds the first match of the (regular expression) in the (token list). If it exists, the match is stored as the first item of the (seq var), and further items are the contents of capturing groups, in the order of their opening parenthesis. The (seq var) is assigned locally. If there is no match, the (seq var) is cleared. The testing versions insert the (true code) into the input stream if a match was found, and the (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 !!!0. Thus, \l_foo_seq contains as a result the items {LaTeX!!!}, {La}, and {!!!}, and the 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.

\regex_extract_all:nnN { (regex) } { (token list) } (seq var)
\regex_extract_all:nnNTF { (regex) } { (token list) } (seq var) { (true code) } { (false code) }

Finds all matches of the (regular expression) in the (token list), and stores all the submatch information in a single sequence (concatenating the results of multiple \regex_extract_once:nnN calls). The (seq var) is assigned locally. If there is no match, the (seq var) is cleared. The testing versions insert the (true code) into the input stream if a match was found, and the (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 true branch is left in the input stream.
Splits the \langle token list \rangle into a sequence of parts, delimited by matches of the \langle regular expression \rangle. If the \langle 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 seq var \rangle is local. If no match is found the resulting \langle seq var \rangle has the \langle token list \rangle as its sole item. If the \langle regular expression \rangle matches the empty token list, then the \langle token list \rangle is split into single tokens. The testing versions insert the \langle true code \rangle into the input stream if a match was found, and the \langle false code \rangle otherwise. For example, after
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
\{ true \} \{ false \}
the sequence \l_path_seq contains the items \{the\}, \{path\}, \{for\}, \{this\}, and \{file.tex\}, and the true branch is left in the input stream.

6 Replacement

Searches for the \langle regular expression \rangle in the \langle token list \rangle and replaces the first match with the \langle replacement \rangle. The result is assigned locally to \langle tl var \rangle. In the \langle replacement \rangle, \\0 represents the full match, \\1 represent the contents of the first capturing group, \\2 of the second, etc.

\regex_replace_all:nnNTF { / } { \the/path/for/this/file.tex } \l_path_seq
\{ true \} \{ false \}

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 \c_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 \c_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 \c_regex_balance_tl to build the function \c_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 \c_regex_action_free:n.
- Optimize the use of \c_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 pdfTeX.

• 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 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: T\TeX{} programmers have lived so far without any need for named macro parameters.

The following features of 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 catastrophic 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 algorithm, 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 \x{...} 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: Xe\TeX and Lua\TeX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.
Part XXVIII
The \texttt{l3box} package

Boxes

There are three kinds of box operations: horizontal mode denoted with prefix \texttt{\hbox{...}}, vertical mode with prefix \texttt{\vbox{...}}, and the generic operations working in both modes with prefix \texttt{\box{...}}.

1 Creating and initialising boxes

\begin{verbatim}
\box_new:N \box_new:c
\box_clear:N \box_clear:c \box_gclear:N \box_gclear:c
\box_gclear_new:N \box_gclear_new:c
\box_set_eq:NN \box_set_eq:N \box_set_eq:cn \box_set_eq:cc
\box_gset_eq:NN \box_gset_eq:N \box_gset_eq:cn \box_gset_eq:cc
\box_if_exist_p:N \box_if_exist:NTF \box_if_exist:TF \box_if_exist:c
\end{verbatim}

\texttt{\box_new:N \box_new:c} \texttt{(box)}

Creates a new \texttt{(box)} or raises an error if the name is already taken. The declaration is global. The \texttt{(box)} is initially void.

\texttt{\box_clear:N \box_clear:c \box_gclear:N \box_gclear:c} \texttt{(box)}

Clears the content of the \texttt{(box)} by setting the box equal to \texttt{c_empty_box}.

\texttt{\box_gclear_new:N \box_gclear_new:c \box_gclear_new:c} \texttt{(box)}

Ensures that the \texttt{(box)} exists globally by applying \texttt{\box_new:N} if necessary, then applies \texttt{\box_(g)clear:N} to leave the \texttt{(box)} empty.

\texttt{\box_set_eq:NN \box_set_eq:N \box_set_eq:cn \box_set_eq:cc} \texttt{(box1) (box2)}

Sets the content of \texttt{(box1)} equal to that of \texttt{(box2)}.

\texttt{\box_if_exist_p:N \box_if_exist:NTF \box_if_exist:TF \box_if_exist:c \box_if_exist:TF}\texttt{\texttt{\texttt{\texttt{(box)}}}} \texttt{\{true code\} \{false code\}}

Tests whether the \texttt{(box)} is currently defined. This does not check that the \texttt{(box)} really is a box.

2 Using boxes

\begin{verbatim}
\box_use:N \box_use:c
\end{verbatim}

\texttt{\box_use:N \box_use:c} \texttt{(box)}

Inserts the current content of the \texttt{(box)} onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

\textbf{TeX hackers note}: This is the \TeX{} primitive \texttt{\copy{}}.

\newpage

235
\texttt{\box_move_right:nn} \{\texttt{dimexpr}\} \{\texttt{box function}\}

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} \{\texttt{box}\} or a “raw” box specification such as \texttt{vbox:n \{ xyz \}}.

\texttt{\box_move_left:nn}

\texttt{\box_move_up:nn} \{\texttt{dimexpr}\} \{\texttt{box function}\}

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} \{\texttt{box}\} or a “raw” box specification such as \texttt{vbox:n \{ xyz \}}.

3 Measuring and setting box dimensions

\texttt{\box_dp:N} \{\texttt{box}\}

Calculates the depth (below the baseline) of the \texttt{\box} in a form suitable for use in a \texttt{dimension expression}.

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

\texttt{\box_dp:c}

\texttt{\box ht:N} \{\texttt{box}\}

Calculates the height (above the baseline) of the \texttt{\box} in a form suitable for use in a \texttt{dimension expression}.

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

\texttt{\box ht:c}

\texttt{\box wd:N} \{\texttt{box}\}

Calculates the width of the \texttt{\box} in a form suitable for use in a \texttt{dimension expression}.

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

\texttt{\box_set_dp:Nn} \{\texttt{box}\} \{\texttt{dimension expression}\}

Set the depth (below the baseline) of the \texttt{\box} to the value of the \{\texttt{dimension expression}\}.

\texttt{\box_set_dp:cn}

\texttt{\box_gset_dp:Nn}

\texttt{\box_gset_dp:cn}

Updated: 2019-01-22

\texttt{\box_set_ht:Nn}

\texttt{\box_set_ht:cn}

\texttt{\box_gset_ht:Nn}

\texttt{\box_gset_ht:cn}

Updated: 2019-01-22
\texttt{\box_set_wd:Nn} \texttt{\box_set_wd:cn} \texttt{\box_gset_wd:Nn} \texttt{\box_gset_wd:cn}

Updated: 2019-01-22

4 Box conditionals

\texttt{\box_if_empty_p:N \box_if_empty_p:c \box_if_empty:NTF \box_if_empty:cTF}

Tests if \texttt{(box)} is a empty (equal to \texttt{\c_empty_box}).

\texttt{\box_if_horizontal_p:N \box_if_horizontal_p:c \box_if_horizontal:NTF \box_if_horizontal:cTF}

Tests if \texttt{(box)} is a horizontal box.

\texttt{\box_if_vertical_p:N \box_if_vertical_p:c \box_if_vertical:NTF \box_if_vertical:cTF}

Tests if \texttt{(box)} is a vertical box.

5 The last box inserted

\texttt{\box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c}

\texttt{\box_set_to_last:NTF \box_set_to_last:cTF}

Sets the \texttt{(box)} equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the \texttt{(box)} is always void as it is not possible to recover the last added item.

6 Constant boxes

\texttt{\c_empty_box}

Updated: 2012-11-04

\textbf{\TeX}hackers note: At the \TeX{} level this is a void box.
7 Scratch boxes

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.

\l_tmpa_box
\l_tmpb_box
Updated: 2012-11-04

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.

\g_tmpa_box
\g_tmpb_box

8 Viewing box contents

\box_show:N \box_show:N (box)
Shows full details of the content of the \langle box \rangle in the terminal.

\box_log:N \box_log:N (box)
Writes full details of the content of the \langle box \rangle to the log.

\box_log:N (box) \box_log:N \{ (intexpr_1) \} \{ (intexpr_2) \}
Writes the contents of \langle box \rangle to the log, showing the first \langle intexpr_1 \rangle items of the box, and descending into \langle intexpr_2 \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

\bbox:n \bbox:n \langle contents \rangle
Typesets the \langle contents \rangle into a horizontal box of natural width and then includes this box in the current list for typesetting.
\hbox_to_wd:nn \hbox_to_wd:nn \hbox_to_zer:nn \hbox_set:Nn \hbox_set_to_wd:Nnn \hbox_overlap_right:n \hbox_overlap_left:n \hbox_set:Nw \hbox_set_to_wd:Nnw \hbox_unpack:N

Typesets the \langle contents \rangle into a horizontal box of width \langle dimexpr \rangle and then includes this box in the current list for typesetting.

Typesets the \langle contents \rangle into a horizontal box of zero width and then includes this box in the current list for typesetting.

Typesets the \langle contents \rangle at natural width and then stores the result inside the \langle box \rangle.

Typesets the \langle contents \rangle to the width given by the \langle dimexpr \rangle and then stores the result inside the \langle box \rangle.

Typesets the \langle contents \rangle into a horizontal box of zero width such that material protrudes to the right of the insertion point.

Typesets the \langle contents \rangle into a horizontal box of zero width such that material protrudes to the left of the insertion point.

Typesets the \langle contents \rangle at natural width and then stores the result inside the \langle box \rangle. In contrast to \hbox_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.

Typesets the \langle contents \rangle to the width given by the \langle dimexpr \rangle and then stores the result inside the \langle box \rangle. In contrast to \hbox_set_to_wd:Nnn 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.

Unpacks the content of the horizontal \langle box \rangle, retaining any stretching or shrinking applied when the \langle box \rangle was set.

\TeXhackers note: This is the \TeX primitive \unhcopy.
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 \{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 \{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 \{\dimexpr\} \{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 \{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 \{box\} \{contents\}

Typesets the \textit{contents} at natural height and then stores the result inside the \textit{box}.

\vbox_set_top:Nn \{box\} \{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 \{box\} \{\dimexpr\} \{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:cw \vbox_set_to_ht_end:
\vbox_gset:Nw \vbox_gset:cw \vbox_gset_end:

Updated: 2017-04-05

\vbox_set_to_ht:Nnw \vbox_set_to_ht:dimexpr \vbox_set_end:
Typesets the \langle contents \rangle to the height given by the \langle dimexpr \rangle and then stores the result inside the \langle box \rangle. In contrast to \vbox_set_to_ht:Nnn 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_split_to_ht:NNn \vbox_set_split_to_ht:dimexpr \vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:dimexpr \vbox_gset_split_to_ht:NNn \vbox_gset_split_to_ht:dimexpr

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 \textit{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
the first use of \texttt{\texttt{\textsc{box}} \texttt{show:N}} will show an entirely cleared (void) box, and the second will show the letter \texttt{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.

\texttt{\texttt{\textsc{box}} \texttt{use_drop:N}} \langle \texttt{box} \rangle

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{\TeXhacksnote:} This is the \texttt{\textsc{box}} primitive.

\texttt{\texttt{\textsc{box}} \texttt{set_eq_drop:NN}} \langle \texttt{box}_1 \rangle \langle \texttt{box}_2 \rangle

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.

\texttt{\texttt{\textsc{box}} \texttt{gset_eq_drop:NN}} \langle \texttt{box}_1 \rangle \langle \texttt{box}_2 \rangle

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.

\texttt{\texttt{\textsc{hbox}} \texttt{unpack_drop:N}} \langle \texttt{box} \rangle

Unpacks the content of the horizontal \texttt{\langle box \rangle}, retaining any stretching or shrinking applied when the \langle box \rangle was set. The original \langle box \rangle is then dropped.

\textbf{\TeXhacksnote:} This is the \TeX primitive \texttt{\texttt{unhbox}}.

\texttt{\texttt{\textsc{vbox}} \texttt{unpack_drop:N}} \langle \texttt{box} \rangle

Unpacks the content of the vertical \texttt{\langle box \rangle}, retaining any stretching or shrinking applied when the \langle box \rangle was set. The original \langle box \rangle is then dropped.

\textbf{\TeXhacksnote:} This is the \TeX primitive \texttt{\texttt{unvbox}}.

13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in \TeX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.
Resizes the `<box>` to fit within the given `<x-size>` (horizontally) and `<y-size>` (vertically); both of the sizes are dimension expressions. The `<y-size>` is the height only: it does not include any depth. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the resizing is applied. The final size of the `<box>` is the smaller of `{<x-size>}` and `{<y-size>}`, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-size>` results in the `<box>` having a depth dependent on the height of the original and *vice versa.*

Resizes the `<box>` to fit within the given `<x-size>` (horizontally) and `<y-size>` (vertically); both of the sizes are dimension expressions. The `<y-size>` is the total vertical size (height plus depth). The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the resizing is applied. The final size of the `<box>` is the smaller of `{<x-size>}` and `{<y-size>}`, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-size>` results in the `<box>` having a depth dependent on the height of the original and *vice versa.*

Resizes the `<box>` to `<y-size>` (vertically), scaling the horizontal size by the same amount; `<y-size>` is a dimension expression. The `<y-size>` is the height only: it does not include any depth. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the resizing is applied. A negative `<y-size>` causes the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-size>` results in the `<box>` having a depth dependent on the height of the original and *vice versa.*
Resizes the \langle box \rangle to \langle y-size \rangle (vertically), scaling the horizontal size by the same amount: \langle y-size \rangle is a dimension expression. The \langle y-size \rangle is the total vertical size (height plus depth). The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the resizing is applied. A negative \langle y-size \rangle causes the material in the \langle box \rangle to be reversed in direction, but the reference point of the \langle box \rangle is unchanged. Thus a negative \langle y-size \rangle results in the \langle box \rangle having a depth dependent on the height of the original and vice versa.

Resizes the \langle box \rangle to \langle x-size \rangle (horizontally), scaling the vertical size by the same amount: \langle x-size \rangle is a dimension expression. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the resizing is applied. A negative \langle x-size \rangle causes the material in the \langle box \rangle to be reversed in direction, but the reference point of the \langle box \rangle is unchanged. Thus a negative \langle x-size \rangle results in the \langle box \rangle having a depth dependent on the height of the original and vice versa.

Resizes the \langle box \rangle to \langle x-size \rangle (horizontally) and \langle y-size \rangle (vertically): both of the sizes are dimension expressions. The \langle y-size \rangle is the height only and does not include any depth. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the resizing is applied. Negative sizes cause the material in the \langle box \rangle to be reversed in direction, but the reference point of the \langle box \rangle is unchanged. Thus a negative \langle y-size \rangle results in the \langle box \rangle having a depth dependent on the height of the original and vice versa.

Resizes the \langle box \rangle to \langle x-size \rangle (horizontally) and \langle y-size \rangle (vertically): both of the sizes are dimension expressions. The \langle y-size \rangle is the total vertical size (height plus depth). The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the resizing is applied. Negative sizes cause the material in the \langle box \rangle to be reversed in direction, but the reference point of the \langle box \rangle is unchanged. Thus a negative \langle y-size \rangle results in the \langle box \rangle having a depth dependent on the height of the original and vice versa.
\texttt{\texttt{\boxrotate:Nn}} \texttt{\boxrotate:cn} \texttt{\boxgrotate:Nn} \texttt{\boxgrotate:cn}

Rotates the \texttt{\langle box\rangle} by \texttt{\langle angle\rangle} (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{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the rotation is applied.

\texttt{\texttt{\boxscale:Nnn}} \texttt{\boxscale:cn} \texttt{\boxgscale:Nnn} \texttt{\boxgscale:cnn}

Scales the \texttt{\langle box\rangle} by factors \texttt{\langle x-scale\rangle} and \texttt{\langle y-scale\rangle} in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the scaling is applied. Negative scalings cause the material in the \texttt{\langle box\rangle} to be reversed in direction, but the reference point of the \texttt{\langle box\rangle} is unchanged. Thus a negative \texttt{\langle y-scale\rangle} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and vice versa.

14 Primitive box conditionals

\texttt{\if_hbox:N \ast} \texttt{\if_hbox:N \langle box\rangle}
\texttt{\langle true code\rangle}
\texttt{\else:}
\texttt{\langle false code\rangle}
\texttt{\fi:}
Tests is \texttt{\langle box\rangle} is a horizontal box.

\textbf{\texttt{\texttt{T}e\texttt{X}hackers note:} This is the \texttt{\TeX} primitive \texttt{\ifhbox}.}

\texttt{\if_vbox:N \ast} \texttt{\if_vbox:N \langle box\rangle}
\texttt{\langle true code\rangle}
\texttt{\else:}
\texttt{\langle false code\rangle}
\texttt{\fi:}
Tests is \texttt{\langle box\rangle} is a vertical box.

\textbf{\texttt{\texttt{T}e\texttt{X}hackers note:} This is the \texttt{\TeX} primitive \texttt{\ifvbox}.}

\texttt{\if_box_empty:N \ast} \texttt{\if_box_empty:N \langle box\rangle}
\texttt{\langle true code\rangle}
\texttt{\else:}
\texttt{\langle false code\rangle}
\texttt{\fi:}
Tests is \texttt{\langle box\rangle} is an empty (void) box.

\textbf{\texttt{\texttt{T}e\texttt{X}hackers note:} This is the \texttt{\TeX} primitive \texttt{\ifvoid}.}
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

Clears the content of the \texttt{coffin}.

\coffin_set_eq:NN \coffin_set_eq: \coffin_gset_eq:NN \coffin_gset_eq:(Nc|cN|cc)
New: 2011-08-17
Updated: 2019-01-21

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:N \coffin_if_exist:NTF \coffin_if_exist:cTF
New: 2012-06-20

Tests whether the \texttt{coffin} is currently defined.

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

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 \textit{material} in horizontal mode, storing the result in the \textit{coffin}. The standard poles for the \textit{coffin} are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn \vcoffin_set:cnn \vcoffin_gset:Nnn \vcoffin_gset:cnn

Tylesets the \textit{material} in vertical mode constrained to the given \textit{width} and stores the result in the \textit{coffin}. The standard poles for the \textit{coffin} are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn \coffin_set_horizontal_pole:cnw \coffin_gset_horizontal_pole:Nnn \coffin_gset_horizontal_pole:cnw

Sets the \textit{pole} to run horizontally through the \textit{coffin}. The \textit{pole} is placed at the \textit{offset} from the bottom edge of the bounding box of the \textit{coffin}. The \textit{offset} should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn \coffin_set_vertical_pole:cnw \coffin_gset_vertical_pole:Nnn \coffin_gset_vertical_pole:cnw

Sets the \textit{pole} to run vertically through the \textit{coffin}. The \textit{pole} is placed at the \textit{offset} from the left-hand edge of the bounding box of the \textit{coffin}. The \textit{offset} should be given as a dimension expression.
3 Coffin affine transformations

Resized the \(\langle\text{coffin}\rangle\) to \(\langle\text{width}\rangle\) and \(\langle\text{total-height}\rangle\), both of which should be given as dimension expressions.

Rotates the \(\langle\text{coffin}\rangle\) by the given \(\langle\text{angle}\rangle\) (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.

Scales the \(\langle\text{coffin}\rangle\) by a factors \(\langle\text{x-scale}\rangle\) and \(\langle\text{y-scale}\rangle\) in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

4 Joining and using coffins

This function attaches \(\langle\text{coffin}_2\rangle\) to \(\langle\text{coffin}_1\rangle\) such that the bounding box of \(\langle\text{coffin}_1\rangle\) is not altered, i.e. \(\langle\text{coffin}_2\rangle\) can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating \(\langle\text{handle}_1\rangle\), the point of intersection of \(\langle\text{coffin}_1\text{-pole}_1\rangle\) and \(\langle\text{coffin}_1\text{-pole}_2\rangle\), and \(\langle\text{handle}_2\rangle\), the point of intersection of \(\langle\text{coffin}_2\text{-pole}_1\rangle\) and \(\langle\text{coffin}_2\text{-pole}_2\rangle\). \(\langle\text{coffin}_2\rangle\) is then attached to \(\langle\text{coffin}_1\rangle\) such that the relationship between \(\langle\text{handle}_1\rangle\) and \(\langle\text{handle}_2\rangle\) is described by the \(\langle\text{x-offset}\rangle\) and \(\langle\text{y-offset}\rangle\). The two offsets should be given as dimension expressions.

This function joins \(\langle\text{coffin}_2\rangle\) to \(\langle\text{coffin}_1\rangle\) such that the bounding box of \(\langle\text{coffin}_1\rangle\) 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 \(\langle\text{handle}_1\rangle\), the point of intersection of \(\langle\text{coffin}_1\text{-pole}_1\rangle\) and \(\langle\text{coffin}_1\text{-pole}_2\rangle\), and \(\langle\text{handle}_2\rangle\), the point of intersection of \(\langle\text{coffin}_2\text{-pole}_1\rangle\) and \(\langle\text{coffin}_2\text{-pole}_2\rangle\). \(\langle\text{coffin}_2\rangle\) is then attached to \(\langle\text{coffin}_1\rangle\) such that the relationship between \(\langle\text{handle}_1\rangle\) and \(\langle\text{handle}_2\rangle\) is described by the \(\langle\text{x-offset}\rangle\) and \(\langle\text{y-offset}\rangle\). The two offsets should be given as dimension expressions.
Typesetting is carried out by first calculating \( \text{handle} \), the point of intersection of \( \text{pole}_1 \) and \( \text{pole}_2 \). The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the \( \text{handle} \) is described by the \( x\text{-offset} \) and \( y\text{-offset} \). The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

5 Measuring coffins

\( \text{coffin}\_dp:N \)\( \text{coffin} \)
Calculates the depth (below the baseline) of the \( \text{coffin} \) in a form suitable for use in a \( \langle \text{dimension expression} \rangle \).

\( \text{coffin}\_ht:N \)\( \text{coffin} \)
Calculates the height (above the baseline) of the \( \text{coffin} \) in a form suitable for use in a \( \langle \text{dimension expression} \rangle \).

\( \text{coffin}\_wd:N \)\( \text{coffin} \)
Calculates the width of the \( \text{coffin} \) in a form suitable for use in a \( \langle \text{dimension expression} \rangle \).

6 Coffin diagnostics

\( \text{coffin}\_display\_handles:Nn \)\( \text{coffin} \)\( \langle \text{color} \rangle \)
This function first calculates the intersections between all of the \( \langle \text{poles} \rangle \) of the \( \text{coffin} \) 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.

\( \text{coffin}\_mark\_handle:Nnnn \)\( \text{coffin} \)\( \langle \text{pole}_1 \rangle \)\( \langle \text{pole}_2 \rangle \)\( \langle \text{color} \rangle \)
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.

\( \text{coffin}\_show\_structure:N \)\( \text{coffin} \)
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 (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 \LaTeX{}3. 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 \color_group_begin: ... \color_group_end: group.
Part XXXI

The l3luatex package:
LuaTEX-specific functions

The LuaTEX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pdfTeX, upTeX or XeTeX these raise an error: use \sys_if_engine_luatex:T to avoid this. Details on using Lua with the LuaTEX engine are given in the LuaTEX manual.

1 Breaking out to Lua

\lua_now:n \{⟨token list⟩\}

The ⟨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 ⟨Lua input⟩ is passed to the Lua interpreter for processing. Each \lua_now:n block is treated by Lua as a separate chunk. The Lua interpreter executes the ⟨Lua input⟩ immediately, and in an expandable manner.

\TeX hackers note: \lua_now:e is a macro wrapper around \directlua: when LuaTEX is in use two expansions are required to yield the result of the Lua code.

\lua_shipout:n \{⟨token list⟩\}

The ⟨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 ⟨Lua input⟩ is passed to the Lua interpreter when the current page is finalised (i.e. at shipout). Each \lua_shipout:n block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨Lua input⟩ during the page-building routine: no TeX expansion of the ⟨Lua input⟩ will occur at this stage.

In the case of the \lua_shipout_e:n version the input is fully expanded by TeX in an e-type manner during the shipout operation.

\TeX hackers note: At a TeX level, the ⟨Lua input⟩ is stored as a “whatsit”.

\lua_escape:n \{⟨token list⟩\}

Converts the ⟨token list⟩ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to \n and \r, respectively.

\TeX hackers note: \lua_escape:e is a macro wrapper around \luaescapestring: when 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.

\texttt{l3kernel}\texttt{.charcat} \texttt{(charcode), (catcode)}

Constructs a character of \texttt{charcode} and \texttt{catcode} and returns the result to \TeX.

\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.

\texttt{l3kernel.filedump} \texttt{(file), (offset), (length)}

Returns the uppercase hexadecimal representation of the content of the \texttt{file} read as bytes. If the \texttt{length} is given, only this part of the file is returned; similarly, one may specify the \texttt{offset} from the start of the file. If the \texttt{length} is not given, the entire file is read starting at the \texttt{offset}.

\texttt{l3kernel.filemdfivesum} \texttt{(file)}

Returns the 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{file} is not found, nothing is returned with \texttt{no error raised}.

\texttt{l3kernel.filemoddate} \texttt{(file)}

Returns the date/time of last modification of the \texttt{file} in the format

\begin{equation}
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
\end{equation}

where the latter may be \texttt{Z} (UTC) or \texttt{plus-minus}\texttt{hours}'\texttt{minutes}'. If the \texttt{file} is not found, nothing is returned with \texttt{no error raised}.

\texttt{l3kernelfilesize} \texttt{(file)}

Returns the size of the \texttt{file} in bytes. If the \texttt{file} is not found, nothing is returned with \texttt{no error raised}.

\texttt{l3kernel.resettimer}()

Resets the timer used by \texttt{l3kernel.elapsedtime}.

\texttt{l3kernel.shellescape} \texttt{(cmd)}

Executes the \texttt{cmd} and prints to the log as for \pdfTeX.

\texttt{l3kernel.strcmp} \texttt{(str one), (str two)}

Compares the two strings and returns 0 to \TeX if the two are identical.
Part XXXII

The 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 ⟨text⟩ are normalized and become { and }, respectively.

1.1 Expanding text

\textexpand:n \langle \text \rangle

Takes user input ⟨text⟩ 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 \l_text_math_delims_tl or as the argument to commands listed in \l_text_math_arg_tl). Commands which are neither engine- nor \LaTeX protected are expanded exhaustively. Any commands listed in \l_text_expand_exclude_tl, \l_text_accents_tl and \l_text_letterlike_tl are excluded from expansion.

\textdeclareexpandequivalent:Nn \langle cmd \rangle \{ \langle replacement \rangle \}

\textdeclareexpandequivalent:cn

Declares that the ⟨replacement⟩ tokens should be used whenever the ⟨cmd⟩ (a single token) is encountered. The ⟨replacement⟩ tokens should be expandable.
1.2 Case changing

\text_lowercase:n * \text_uppercase:n \{(tokens)\}  
\text_uppercase:nn \{(language)\} \{(tokens)\}  
\text_titlecase:n * \text_titlecase_first:n * \text_lowercase:nn * \text_uppercase:nn * \text_titlecase:nn * \text_titlecase_first:nn * \text_uppercase:nn

Takes user input (\text) first applies \textexpand, 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 \text_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 \textupper case variant does not attempt any case changing at all after the first letter has been processed.

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 }\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\TeX or Lua\TeX 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, T2 and LGR font encodings. Thus for example ā can be case-changed using pdf\TeX. For \TeX 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. Since there is a T1 slot for the großes Eszett in T1, this tailoring is available with pdf\TeX as well as in the Unicode \TeX engines.
- 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\TeX.

For titlecasing, note that there are two functions available. The function \text_titlecase:n applies (broadly) uppercasing to the first letter of the input, then lowercasing the rest. The \text_titlecase_first variant applies (broadly) uppercasing to the first letter of the input, then lowercasing the rest.
1.3 Removing formatting from text

\text_purify:n \{\text\}

Takes user input \text{} and expands as described for \text_expand:n, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in \text_math_delims_tl or as the argument to commands listed in \text_math_arg_tl) is left contained in a pair of $ delimiters. Non-expandable functions present in the \text{} must either have a defined equivalent (see \text_declare_purify_equivalent:Nn) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

\text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nx

Declares that the \text{} tokens should be used whenever the \text{cmd} (a single token) is encountered. The \text{} tokens should be expandable.

1.4 Control variables

\text{accents_tl}

Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the LaTeX package.)

\text{letterlike_tl}

Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the LaTeX package.)

\text{math_arg_tl}

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 \text_math_delims_tl but for a command rather than paired delimiters.

\text{math_delims_tl}

Lists pairs of tokens which delimit (in-line) math mode content; such content may be excluded from processing.

\text{case_exclude_arg_tl}

Lists commands which are excluded from case changing.

\text{expand_exclude_tl}

Lists commands which are excluded from expansion.

\text{titlecase_check_letter_bool}

Controls how the start of titlecasing is handled: when \text{true}, the first letter in text is considered. The standard setting is \text{true}.

258
Part XXXIV

The \texttt{l3legacy} package

Interfaces to legacy concepts

There are a small number of \TeX{} or \TeX{}\LaTeX{}\玉 concepts which are not used in \texttt{expl3} code but which need to be manipulated when working as a \TeX{}\LaTeX{}\玉 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{}\玉/\plain{} \TeX{} conditional (generated by \texttt{\newif}) if \texttt{true} or \texttt{false} and branches accordingly. The \langle\textit{name}\rangle of the conditional should \textit{omit} the leading \texttt{if}. 

259
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} (\exp\texttt{l3}) while still being experimental.

\textit{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 small additions to the kernel. We encourage programmers to test them out and report back on the \LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the \LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are not necessarily stable and we may have to adjust them!

2 Additions to \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}

\texttt{\box_clip:N \{box\}} Clips the \{box\} in the output so that only material inside the bounding box is displayed in the output. The updated \{box\} is an hbox, irrespective of the nature of the \{box\} before the clipping is applied.

\textbf{These functions require the \LaTeXX native drivers: they do not work with the \LaTeX \texttt{2e} graphics drivers!}

\textbf{\LaTeXX 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.

260
\box_set_trim:Nnnnn \box_set_trim:cnmnn \box_gset_trim:Nnnnn \box_gset_trim:cnmnn

Rev: 2019-01-23

Adjusts the bounding box of the \langle box \rangle \langle left \rangle is removed from the left-hand edge of the bounding box, \langle right \rangle from the right-hand edge and so forth. All adjustments are \langle dimension expressions \rangle. Material outside of the bounding box is still displayed in the output unless \box_clip:N is subsequently applied. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle 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

Rev: 2019-01-23

Adjusts the bounding box of the \langle box \rangle such that it has lower-left co-ordinates \langle llx \rangle, \langle lly \rangle and upper-right co-ordinates \langle urx \rangle, \langle ury \rangle. All four co-ordinate positions are \langle dimension expressions \rangle. Material outside of the bounding box is still displayed in the output unless \box_clip:N is subsequently applied. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the viewport operation is applied.

3 Additions to \texttt{l3expan}

\exp_args_generate:n \exp_args_generate:n \{\langle variant argument specifiers \rangle\}

Rev: 2018-04-04
Updated: 2019-02-08

Defines \exp_args:N\langle variant \rangle functions for each \langle variant \rangle given in the comma list \{\langle variant argument specifiers \rangle\}. Each \langle variant \rangle 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 \langle variant \rangle. This is only useful for cases where \csGenerate_variant:Nn is not applicable.

4 Additions to \texttt{l3fp}

\fp_if_nan:n \fp_if_nan:n \{\langle fpexpr \rangle\}

Rev: 2019-08-25

Evaluates the \langle fpexpr \rangle and tests whether the result is exactly NaN. The test returns false for any other result, even a tuple containing NaN.

5 Additions to \texttt{l3file}

\iow_allow_break: \iow_allow_break:

Rev: 2018-12-29

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.
\ior_get_term:nN \ior_str_get_term:nN

New: 2019-03-23

\ior_get_term:nN \ior_str_get_term:nN \langle prompt \rangle \langle token list variable \rangle

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 \rangle variable.
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. \iow_term:n.
Where the \langle prompt \rangle is given, it will appear in the terminal followed by an =, e.g.
prompt=

\ior_shell_open:Nn \ior_shell_open:nN \langle stream \rangle \langle{shell command} \rangle

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

\flag_raise_if_clear:n \star \flag_raise_if_clear:n \langle{flag name} \rangle

Ensures the \langle flag \rangle is raised by making its height at least 1, locally.

7 Additions to l3intarray

\intarray_gset_rand:Nnn \intarray_gset_rand:cnn \intarray_gset_rand:Nn \intarray_gset_rand:cn

New: 2018-05-05

\intarray_gset_rand:Nnn \intarray_gset_rand:cnn \langle intarray var \rangle \langle{minimum} \rangle \langle{maximum} \rangle
\intarray_gset_rand:Nn \intarray_gset_rand:cn \langle intarray var \rangle \langle{maximum} \rangle

Evaluates the integer expressions \langle minimum \rangle and \langle maximum \rangle then sets each entry (independent-
ly) 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

\intarray_to_clist:N \star \intarray_to_clist:N \langle intarray \rangle

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 \texttt{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 \texttt{\textbackslash 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 \textbackslash{} or \textbackslash\{} 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 upfront. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\begin{verbatim}
\msg\_expandable\_error:nnnnnn * \msg\_expandable\_error:nnfff * \msg\_expandable\_error:nnnn * \msg\_expandable\_error:nnf * \msg\_expandable\_error:nnf * \msg\_expandable\_error:nnnnn * \msg\_expandable\_error:nnn * \msg\_expandable\_error:nnf * \msg\_expandable\_error:nnf * \msg\_expandable\_error:nnn * \msg\_expandable\_error:nnn * \msg\_expandable\_error:nnf *
\end{verbatim}

New: 2015-08-06
Updated: 2019-02-28

Issues an “Undefined error” message from \LaTeX\ itself using the undefined control sequence \texttt{\textbackslash{}::\textbackslash{}error} then prints “! \langle module\rangle:\langle error\ message\rangle\textbackslash{}, 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.

\begin{verbatim}
\msg\_show\_eval:NN \msg\_log\_eval:NN
\end{verbatim}

New: 2017-12-04

\begin{verbatim}
\msg\_show\_eval:NN \langle function\rangle\ \langle expression\rangle
\end{verbatim}

Shows or logs the \langle expression\rangle (turned into a string), an equal sign, and the result of applying the \langle function\rangle to the \langle expression\rangle (with I-expansion). For instance, if the \langle function\rangle is \texttt{\textbackslash{}int\_eval:n} and the \langle expression\rangle is 1+2 then this logs $1+2=3$.

\begin{verbatim}
\msg\_show:nnnnnn \msg\_show:nnxxxx \msg\_show:nnnnn \msg\_show:nnxxx \msg\_show:nnnn \msg\_show:nnxx \msg\_show:nnx \msg\_show:nn
\end{verbatim}

New: 2017-12-04

Issues \langle module\rangle information \langle message\rangle, passing \langle arg\ one\rangle to \langle arg\ four\rangle to the text-creating functions. The information text is shown on the terminal and the \LaTeX\ run is interrupted in a manner similar to \texttt{\textbackslash{}tl\_show:n}. This is used in conjunction with \texttt{\msg\_show\_item:n} and similar functions to print complex variable contents completely. If the formatted text does not contain \texttt{\textgreater\textgreater} at the start of a line, an additional line \texttt{\textgreater\textgreater\textless\textless} will be put at the end. In addition, a final period is added if not present.
Used in the text of messages for `\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.

**9 Additions to l3prg**

`\bool_set_inverse:N` \texttt{\langle boolean \rangle}

Toggles the \texttt{\langle boolean \rangle} from \texttt{true} to \texttt{false} and conversely: sets it to the inverse of its current value.

```
\bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_false:n \star \bool_case_false:n \star \bool_case_false:n
\bool_case_true:n \star \bool_case_true:n \star \bool_case_true:n
\bool_case_false:n \star \bool_case_false:n \star \bool_case_false:n
```

Rev: 2018-05-10

```
\bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_false:n \star \bool_case_false:n \star \bool_case_false:n
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
\bool_case_true:nTF \star \bool_case_true:nTF
```

Rev: 2019-02-10

Evaluates in turn each of the \texttt{\langle boolean expression cases \rangle} until the first one that evaluates to \texttt{true} or to \texttt{false}, for \texttt{\bool_case_true:n} and \texttt{\bool_case_false:n}, respectively. The \texttt{\langle code \rangle} associated to this first case is left in the input stream, followed by the \texttt{\langle true code \rangle}, and other cases are discarded. If none of the cases match then only the \texttt{\langle false code \rangle} is inserted. The functions \texttt{\bool_case_true:n} and \texttt{\bool_case_false:n}, which do nothing if there is no match, are also available. For example

```
\bool_case_true:nF
{ \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }{ Fits }
{ \int_compare_p:n { \l__mypkg_total_int >= 10 } }{ Many }
{ \l__mypkg_special_bool }{ Special }
{ No idea! }
```

leaves “`Fits`” or “`Many`” or “`Special`” or “`No idea!`” in the input stream, in a way similar to some other language’s “`if ... elseif ... elseif ... else ...`”.
10 Additions to \textit{l3prop}

\texttt{\prop_rand_key_value:N * \prop_rand_key_value:c *}
\texttt{Rev: 2016-12-06}

\prop_rand_key_value:N \texttt{(prop var)}

Selects a pseudo-random key–value pair from the \texttt{\langle property list \rangle} and returns \texttt{\langle key \rangle} and \texttt{\langle value \rangle}. If the \texttt{\langle property list \rangle} is empty the result is empty. This is not available in older versions of X\TeXX.

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

\begin{itemize}
\item \texttt{\prop_rand_key_value:N * \prop_rand_key_value:c *}
\end{itemize}

11 Additions to \textit{l3seq}

\texttt{\seq_mapthread_function:NNN * \seq_mapthread_function:NNN \texttt{(seq1) (seq2) \langle function \rangle}}

Applies \texttt{\langle function \rangle} to every pair of items \texttt{\langle seq1-item \rangle}–\texttt{\langle seq2-item \rangle} from the two sequences, returning items from both sequences from left to right. The \texttt{\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).

\texttt{\seq_set_filter:NNn \seq_gset_filter:NNn}

Evaluates the \texttt{\langle inline bool expr \rangle} for every \texttt{\langle item \rangle} stored within the \texttt{\langle sequence2 \rangle}. The \texttt{\langle inline bool expr \rangle} receives the \texttt{\langle item \rangle} as \texttt{#1}. The sequence of all \texttt{\langle items \rangle} for which the \texttt{\langle inline bool expr \rangle} evaluated to \texttt{true} is assigned to \texttt{\langle sequence1 \rangle}.

\textbf{\texthacks:} Contrarily to other mapping functions, \texttt{\seq_map_break:} cannot be used in this function, and would lead to low-level \texttt{\TeX} errors.

\texttt{\seq_set_map:NNn \seq_gset_map:NNn}

Applies \texttt{\langle inline function \rangle} to every \texttt{\langle item \rangle} stored within the \texttt{\langle sequence2 \rangle}. The \texttt{\langle inline function \rangle} should consist of code which will receive the \texttt{\langle item \rangle} as \texttt{#1}. The sequence resulting from \texttt{x-expanding} \texttt{\langle inline function \rangle} applied to each \texttt{\langle item \rangle} is assigned to \texttt{\langle sequence1 \rangle}. As such, the code in \texttt{\langle inline function \rangle} should be expandable.

\textbf{\texthacks:} Contrarily to other mapping functions, \texttt{\seq_map_break:} cannot be used in this function, and would lead to low-level \texttt{\TeX} errors.
Sets the \texttt{seq} equal to a sequence whose items are obtained by x-expanding \texttt{loop code} \texttt{function}. This expansion must result in successive calls to the \texttt{function} with no nonexpandable tokens in between. More precisely the \texttt{function} is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The \texttt{loop code} must be expandable; it can be for example \texttt{\tl_map_function:NN \tl var} or \texttt{\clist_map_function:nN \{clist\}} or \texttt{\int_step_function:nnnN \{initial value\} \{step\} \{final value\}}.

\newpage

Sets the \texttt{seq} equal to a sequence whose items are obtained by x-expanding \texttt{loop code} applied to a \texttt{function} derived from the \texttt{inline code}. A \texttt{function} is defined, that takes one argument, x-expands the \texttt{inline code} with that argument as \#1, then adds appropriate separators to turn the result into an item of the sequence. The x-expansion of \texttt{loop code} \texttt{function} must result in successive calls to the \texttt{function} with no nonexpandable tokens in between. The \texttt{loop code} must be expandable; it can be for example \texttt{\tl_map_function:NN \tl var} or \texttt{\clist_map_function:nN \{clist\}} or \texttt{\int_step_function:nnnN \{initial value\} \{step\} \{final value\}}, but not the analogous “inline” mappings.

Applies \texttt{function} to every entry in the \texttt{sequence variable}. The \texttt{function} should have signature \texttt{nn}. It receives two arguments for each iteration: the \texttt{index} (namely 1 for the first entry, then 2 and so on) and the \texttt{item}.

Applies \texttt{inline function} to every entry in the \texttt{sequence variable}. The \texttt{inline function} should consist of code which receives the \texttt{index} (namely 1 for the first entry, then 2 and so on) as \#1 and the \texttt{item} as \#2.
12 Additions to \texttt{l3sys}

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 \texttt{\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 e-\texttt{p\TeX} version.

\[ \texttt{p}(\text{major}).\langle \text{revision}\rangle-u(\text{major}).(\text{minor})-\langle e\text{\TeX}\rangle \]

where the \texttt{u} part is only present for \texttt{up\TeX}.

\texttt{c_sys_engine_version_str}

\begin{verbatim}
New: 2018-05-02
\end{verbatim}

\texttt{\sys_if_rand_exist_p: \sys_if_rand_exist:TF {\langle true code\rangle} {\langle false code\rangle}}

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}.

\texttt{\sys_if_rand_exist_p: \sys_if_rand_exist:TF}
Leaves in the input stream the items from the \textlangle start index\textrangle to the \textlangle end index\textrangle inclusive, using the same indexing as \texttt{l_range:nnn}. Spaces are ignored. Regardless of whether items appear with or without braces in the \textlangle token list\textrangle, the \texttt{l_range_braced:nnn} function wraps each item in braces, while \texttt{l_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 \texttt{\{b\}{c\}{d\}{e\}{}f}, \texttt{\{c\}{d\}{e\}{}f\}{}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 \texttt{bcde{}}, \texttt{cde{}f}, \texttt{e{}f}, and an empty line to the terminal. Because braces are removed, the result of \texttt{l_range_unbraced:nnn} may have a different number of items as for \texttt{l_range:nnn} or \texttt{l_range_braced:nnn}. In cases where preserving spaces is important, consider the slower function \texttt{l_range:nnn}.

\textbf{TEXhackers note:} The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \textlangle item\textrangle 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 \textlangle tl var\textrangle 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 \texttt{l3tl} functions. Until \texttt{tl_build_end:N \{tl var\}} is called, applying any function from \texttt{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 \textlangle tl var\textrangle respectively.

Clears the \textlangle tl var\textrangle 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 \textlangle tl var\textrangle respectively.
\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_gput_right:Nn \tl_build_gput_right:Nx 

\tl_build_put_left:Nn (\tl var) \{\tokens\} \tl_build_put_right:Nn (\tl var) \{\tokens\} 

Adds \tokens to the left or right side of the current contents of \tl var. The \tl var must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The put and gput functions must be used for local and global \tl var respectively. The right functions are about twice faster than the left functions.

\tl_build_get:N \tl_build_get:NN 

\tl_build_get:N (\tl var_1) (\tl var_2) 

Stores the contents of the \tl var_1 in the \tl var_2. The \tl var_1 must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The \tl var_2 is a “normal” token list variable, assigned locally using \tl_set:Nn.

\tl_build_end:N \tl_build_gend:N 

\tl_build_end:N (\tl var) 

\tl_build_gend:N (\tl var) 

Gets the contents of \tl var and stores that into the \tl var using \tl_set:Nn. The \tl var must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The end and gend functions must be used for local and global \tl var respectively. These functions completely remove the setup code that enabled \tl var to be used for other \tl_build_... functions.

## 14 Additions to l3token

\c_catcode_active_space_tl 

\c_catcode_active_space_tl 

\char_to_utfviii_bytes:n \star 

\char_to_utfviii_bytes:n \{\codepoint\} 

Converts the (Unicode) \codepoint 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.

\char_to_nfd:N \star 

\char_to_nfd:N \{\char\} 

Converts the \char to the Unicode Normalization Form Canonical Decomposition. The category code of the generated character is the same as the \char. 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{\token_if_eq_catcode:NNTF} or \texttt{\token_if_eq_charcode:NNTF} or \texttt{\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{\c_group_begin_token} and \texttt{\c_group_end_token}, and when spaces (including \texttt{\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 A { \iwow_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}

Removes explicit and implicit space tokens (category code 10 and character code 32) from the input stream, then inserts \textit{\texttt{\langle code\rangle}}.

\paragraph*{Part XXXVI}

\section*{Implementation}

\subsection*{1 l3bootstrap implementation}

\paragraph*{1.1 Format-specific code}

The very first thing to do is to bootstrap the in\TeX{} system so that everything else will actually work. \TeX{} does not start with some pretty basic character codes set up.

\begin{verbatim}
\*{\texttt{\langle in\TeX{} | package\rangle}}
\*{\texttt{\langle @@=kernel\rangle}}
\end{verbatim}

Tab characters should not show up in the code, but to be on the safe side.

\begin{verbatim}
\*{\texttt{\langle in\TeX{}\rangle}}
\*{\texttt{\catcode \{ = 1 \}}
\*{\texttt{\catcode \} = 2 \}}
\*{\texttt{\catcode \# = 6 \}}
\*{\texttt{\catcode \^ = 7 \}}
\*{\texttt{\langle /in\TeX{}\rangle}}
\end{verbatim}

\paragraph*{270}
For LuaTeX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

Depending on the versions available, the \LaTeX format may not have the raw \Umath primitive names available. We fix that globally: it should cause no issues. Older Lua\TeX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start \U this should be reasonably safe.

### 1.2 The \texttt{pdfstrcmp} primitive in \texttt{Xe\TeX}

Only pdf\TeX has a primitive called \texttt{pdfstrcmp}. The \texttt{Xe\TeX} version is just \texttt{strcmp}, so there is some shuffling to do. As this is still a real primitive, using the pdf\TeX name is “safe”.

### 1.3 Loading support Lua code

When Lua\TeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in \texttt{l3luatex} and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.
In package mode for Lua\TeX\ we make sure the basic support is loaded: this is only necessary in plain.

\begin{verbatim}
\expandafter\ifx\csname directlua\endcsname\relax
\else
\ifnum\luatexversion<95 
\else
\input{ltluatex}\
\fi
\langle
\*
pkg
\endgroup
\expandafter\ifx\csname newcatcodetable\endcsname\relax
\input{ltluatex}\
\fi
\langle
\directlua{require("expl3")}\
\end{verbatim}

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

\begin{verbatim}
\ifnum 0%
\directlua{if status.ini_version then
     tex.write("1")
   end
}>0 
\everyjob\expandafter{%
   \the\expandafter\everyjob
   \csname\detokenize{lua_now:n}\endcsname{require("expl3")}\
}\%
\fi
\fi
\fi
\end{verbatim}

\section{Engine requirements}

The code currently requires e-\TeX\ and functionality equivalent to \texttt{\pdfstrcmp}, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

\begin{verbatim}
\begin{verbatim}
\begingroup
\def\next\endgroup
\def\ShortText{Required primitives not found}\
\def\LongText{LaTeX3 requires the e-\TeX\ primitives and additional functionality as described in the README file.\LineBreak These are available in the engines\LineBreak - pdfTeX v1.40\LineBreak - XeTeX v0.99992\LineBreak - LuaTeX v0.95\LineBreak - e-(u)pTeX mid-2012\LineBreak or later.\LineBreak}
\end{verbatim}
\ifnum0%
\fi
\end{verbatim}
For pdfTeX and XeTeX the '-etex' command-line switch is also needed.

Format building will abort!

Loading of expl3 will abort!
1.5 Extending allocators

In format mode, allocating registers is handled by l3alloc. However, in package mode it’s much safer to rely on more general code. For example, the ability to extend \TeX’s allocation routine to allow for \TeX{} has been around since 1997 in the etex package.

Loading this support is delayed until here as we are now sure that the \TeX{} extensions and \pdfstrcmp or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For \LaTeX{} we need to make sure that the extended pool is being used: expl3 uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the etex package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that \reserveinserts is \outer and so is accessed here by csname. In earlier versions, loading etex was done directly and so \reserveinserts appeared in the code: this then required a \relax after \RequirePackage to prevent an error with “unsafe” definitions as seen for example with capoptions. The optional loading here is done using a group and \ifx test as we are not quite in the position to have a single name for \pdfstrcmp just yet.

1.6 Character data

\TeX{} needs various pieces of data to be set about characters, in particular which ones to treat as letters and which \lccode values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for \LaTeX{} hyphenation patterns can be read anywhere, other engines have to build them into the format and so we must do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (In\i\TeX{} category codes and primitive availability and must therefore loaded very early. This has a knock-on
effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For Xe\TeX{} and Lua\TeX{}, which are natively Unicode engines, simply load the Unicode data.

\begin{verbatim}
(*initex*)
\ifdefined\Umathcode
\input load-unicode-data %
\input load-unicode-math-classes %
\else
\endgroup
\end{verbatim}

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

\begin{verbatim}
\begingroup
\def\temp{%
  \ifnum\count0>\count2 
    \global\lccode\count0 = \count0%
    \global\uccode\count0 = \numexpr\count0 - "20\relax
    \advance\count0 by 1%
    \expandafter\temp
  \fi
}
\count0 = "A0 %
\count2 = "BC %
\temp
\count0 = "E0 %
\count2 = "FF %
\temp
\endgroup
\end{verbatim}

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an \sfcode of 999. (The characters A–Z are set up correctly by ini\TeX{}.)

\begin{verbatim}
\def\temp{%
  \ifnum\count0>\count2 
    \global\lccode\count0 = \numexpr\count0 + "20\relax
    \global\uccode\count0 = \count0%
    \global\sfcode\count0 = 999%
    \advance\count0 by 1%
    \expandafter\temp
  \fi
}
\count0 = "80 %
\count2 = "9C %
\temp
\count0 = "C0 %
\count2 = "DF %
\temp
\end{verbatim}

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

\begin{verbatim}
\global\lccode`\^Y = `\^Y%
\end{verbatim}

275
1.7 The \LaTeX3 code environment

The code environment is now set up.

\ExplSyntaxOff

Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied \ExplSyntaxOff becomes a “do nothing” command until \ExplSyntaxOn is used. For format mode, there is no need to save category codes so that step is skipped.

\protected\def\ExplSyntaxOff{}\%
\protected\edef\ExplSyntaxOff{\%
\protected\def\ExplSyntaxOff{}\%
catcode 9 = \the\catcode 9\relax
\catcode 32 = \the\catcode 32\relax
\catcode 34 = \the\catcode 34\relax
\catcode 38 = \the\catcode 38\relax
\catcode 58 = \the\catcode 58\relax
\catcode 94 = \the\catcode 94\relax
\catcode 95 = \the\catcode 95\relax
\catcode 124 = \the\catcode 124\relax
\catcode 126 = \the\catcode 126\relax
\endlinechar = \the\endlinechar\relax
\chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
\}%
\langle\package\rangle

(End definition for \ExplSyntaxOff. This function is documented on page 7.)

The code environment is now set up.

\catcode 9 = 9\relax
\catcode 32 = 9\relax
\catcode 34 = 12\relax
\catcode 38 = 4\relax
\catcode 58 = 11\relax
\catcode 94 = 7\relax
\catcode 95 = 11\relax
\catcode 124 = 12\relax
\catcode 126 = 10\relax
\endlinechar = 32\relax
The status for experimental code syntax: this is on at present.

\chardef\l__kernel_expl_bool = 1

(End definition for \l__kernel_expl_bool.)

The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

\protected \def \ExplSyntaxOn
{
\bool_if:NF \l__kernel_expl_bool
{
\cs_set_protected:Npx \ExplSyntaxOff
{
\char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
\char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
\char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
\char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
\char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
\char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
\char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
\char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
\char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
\tex_endlinechar:D = \tex_the:D \tex_endlinechar:D \scan_stop:
\bool_set_false:N \l__kernel_expl_bool
\cs_set_protected:Npn \ExplSyntaxOff { }
}
\char_set_catcode:ignore:n { 9 } % tab
\char_set_catcode:ignore:n { 32 } % space
\char_set_catcode_other:n { 34 } % double quote
\char_set_catcode_alignment:n { 38 } % ampersand
\char_set_catcode_letter:n { 58 } % colon
\char_set_catcode_math_superscript:n { 94 } % circumflex
\char_set_catcode_letter:n { 95 } % underscore
\char_set_catcode_other:n { 124 } % pipe
\char_set_catcode_space:n { 126 } % tilde
\tex_endlinechar:D = 32 \scan_stop:
\bool_set_true:N \l__kernel_expl_bool
}
\cs_set_protected:Npx \ExplSyntaxOff
}
\l__kernel_expl_bool

(End definition for \ExplSyntaxOn. This function is documented on page 7.)

⟨/initex | package⟩

2 l3names implementation

⟨/initex | package⟩

The prefix here is kernel. A few places need @@ to be left as is; this is obtained as

⟨@@=kernel⟩

277
The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

The \texttt{\let} primitive is renamed by hand first as it is essential for the entire process to follow. This also uses \texttt{\global}, as that way we avoid leaving an unneeded csname in the hash table.

```
\let \tex_global:D \global
\let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps \texttt{\__kernel_primitive:NN} trapped.

```
\begingroup
\__kernel_primitive:NN
\__kernel_primitive:NN
\__kernel_primitive:NN
\__kernel_primitive:NN
```

A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
\long \def \__kernel_primitive:NN #1#2
{
  \tex_global:D \tex_let:D #2 \#1
  \langle \textit{initex} \rangle
  \tex_global:D \tex_let:D #1 \tex_undefined:D
  \langle \textit{/initex} \rangle
}
```

(End definition for \texttt{\__kernel_primitive:NN}.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
\langle /initex | package \rangle
\langle /initex | names | package \rangle
```

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

```
\__kernel_primitive:NN \ tex_space:D
\__kernel_primitive:NN \ / \ tex_italiccorrection:D
\__kernel_primitive:NN \ - \ tex_hyphen:D
```

Now all the other primitives.

```
\__kernel_primitive:NN \ \above \ tex_above:D
\__kernel_primitive:NN \ \abovedisplayshortskip \ tex_abovedisplayshortskip:D
\__kernel_primitive:NN \ \abovedisplayskip \ tex_abovedisplayskip:D
\__kernel_primitive:NN \ \abovewithdelims \ tex_abovewithdelims:D
\__kernel_primitive:NN \ \accent \ tex_accent:D
\__kernel_primitive:NN \ \adjdemerits \ tex_adjdemerits:D
\__kernel_primitive:NN \ \advance \ tex_advance:D
\__kernel_primitive:NN \ \afterassignment \ tex_afterassignment:D
\__kernel_primitive:NN \ \aftergroup \ tex_aftergroup:D
\__kernel_primitive:NN \ \atop \ tex_atop:D
\__kernel_primitive:NN \ \atopwithdelims \ tex_atopwithdelims:D
\__kernel_primitive:NN \ \badness \ tex_badness:D
\__kernel_primitive:NN \ \baselineskip \ tex_baselineskip:D
\__kernel_primitive:NN \ \batchmode \ tex_batchmode:D
\__kernel_primitive:NN \ \begingroup \ tex_begingroup:D
```

278
\__kernel_primitive:NN \ignorespaces \tex_ignorespaces:D
\__kernel_primitive:NN \immediate \tex_immediate:D
\__kernel_primitive:NN \indent \tex_indent:D
\__kernel_primitive:NN \input \tex_input:D
\__kernel_primitive:NN \inputlineno \tex_inputlineno:D
\__kernel_primitive:NN \insert \tex_insert:D
\__kernel_primitive:NN \insertpenalties \tex_insertpenalties:D
\__kernel_primitive:NN \interlinepenalty \tex_interlinepenalty:D
\__kernel_primitive:NN \jobname \tex_jobname:D
\__kernel_primitive:NN \kern \tex_kern:D
\__kernel_primitive:NN \language \tex_language:D
\__kernel_primitive:NN \lastbox \tex_lastbox:D
\__kernel_primitive:NN \lastkern \tex_lastkern:D
\__kernel_primitive:NN \lastpenalty \tex_lastpenalty:D
\__kernel_primitive:NN \lastskip \tex_lastskip:D
\__kernel_primitive:NN \lccode \tex_lccode:D
\__kernel_primitive:NN \leaders \tex_leaders:D
\__kernel_primitive:NN \left \tex_left:D
\__kernel_primitive:NN \lefthyphenmin \tex_lefthyphenmin:D
\__kernel_primitive:NN \leftskip \tex_leftskip:D
\__kernel_primitive:NN \leqno \tex_leqno:D
\__kernel_primitive:NN \let \tex_let:D
\__kernel_primitive:NN \limits \tex_limits:D
\__kernel_primitive:NN \linepenalty \tex_linepenalty:D
\__kernel_primitive:NN \lineskip \tex_lineskip:D
\__kernel_primitive:NN \lineskiplimit \tex_lineskiplimit:D
\__kernel_primitive:NN \long \tex_long:D
\__kernel_primitive:NN \looseness \tex_looseness:D
\__kernel_primitive:NN \lower \tex_lower:D
\__kernel_primitive:NN \lowercase \tex_lowercase:D
\__kernel_primitive:NN \mag \tex_mag:D
\__kernel_primitive:NN \mark \tex_mark:D
\__kernel_primitive:NN \mathaccent \tex_mathaccent:D
\__kernel_primitive:NN \mathbin \tex_mathbin:D
\__kernel_primitive:NN \mathchar \tex_mathchar:D
\__kernel_primitive:NN \mathchardef \tex_mathchardef:D
\__kernel_primitive:NN \mathchoice \tex_mathchoice:D
\__kernel_primitive:NN \mathclose \tex_mathclose:D
\__kernel_primitive:NN \mathcode \tex_mathcode:D
\__kernel_primitive:NN \mathinner \tex_mathinner:D
\__kernel_primitive:NN \mathop \tex_mathop:D
\__kernel_primitive:NN \mathopen \tex_mathopen:D
\__kernel_primitive:NN \mathord \tex_mathord:D
\__kernel_primitive:NN \mathpunct \tex_mathpunct:D
\__kernel_primitive:NN \mathrel \tex_mathrel:D
\__kernel_primitive:NN \mathsurround \tex_mathsurround:D
\__kernel_primitive:NN \maxdeadcycles \tex_maxdeadcycles:D
\__kernel_primitive:NN \maxdepth \tex_maxdepth:D
\__kernel_primitive:NN \meaning \tex_meaning:D
\__kernel_primitive:NN \medmuskip \tex_medmuskip:D
\__kernel_primitive:NN \message \tex_message:D
\__kernel_primitive:NN \mkern \tex_mkern:D
\__kernel_primitive:NN \month \tex_month:D
\__kernel_primitive:NN \moveleft \tex_moveleft:D
\__kernel_primitive:NN \right \text{right:D}
\__kernel_primitive:NN \rightskip \text{rightskip:D}
\__kernel_primitive:NN \righthyphenmin \text{righthyphenmin:D}
\__kernel_primitive:NN \scriptfont \text{scriptfont:D}
\__kernel_primitive:NN \scriptscriptfont \text{scriptscriptfont:D}
\__kernel_primitive:NN \scriptscriptstyle \text{scriptscriptstyle:D}
\__kernel_primitive:NN \scriptspace \text{scriptspace:D}
\__kernel_primitive:NN \scriptstyle \text{scriptstyle:D}
\__kernel_primitive:NN \scrollmode \text{scrollmode:D}
\__kernel_primitive:NN \setbox \text{setbox:D}
\__kernel_primitive:NN \setlanguage \text{setlanguage:D}
\__kernel_primitive:NN \sfcode \text{sfcode:D}
\__kernel_primitive:NN \shipout \text{shipout:D}
\__kernel_primitive:NN \show \text{show:D}
\__kernel_primitive:NN \showbox \text{showbox:D}
\__kernel_primitive:NN \showboxbreadth \text{showboxbreadth:D}
\__kernel_primitive:NN \showboxdepth \text{showboxdepth:D}
\__kernel_primitive:NN \showlists \text{showlists:D}
\__kernel_primitive:NN \showthe \text{showthe:D}
\__kernel_primitive:NN \skewchar \text{skewchar:D}
\__kernel_primitive:NN \span \text{span:D}
\__kernel_primitive:NN \special \text{special:D}
\__kernel_primitive:NN \splitbotmark \text{splitbotmark:D}
\__kernel_primitive:NN \splitfirstmark \text{splitfirstmark:D}
\__kernel_primitive:NN \splitmaxdepth \text{splitmaxdepth:D}
\__kernel_primitive:NN \splittopskip \text{splittopskip:D}
\__kernel_primitive:NN \string \text{string:D}
\__kernel_primitive:NN \tabskip \text{tabskip:D}
\__kernel_primitive:NN \textfont \textfont:D}
\__kernel_primitive:NN \textstyle \textstyle:D}
\__kernel_primitive:NN \time \text{time:D}
\__kernel_primitive:NN \topmark \text{topmark:D}
\__kernel_primitive:NN \toupskip \text{toupskip:D}
\__kernel_primitive:NN \tracingcommands \text{tracingcommands:D}
\__kernel_primitive:NN \tracinglostchars \text{tracinglostchars:D}
\__kernel_primitive:NN \tracingmacros \text{tracingmacros:D}
\__kernel_primitive:NN \tracingonline \text{tracingonline:D}
\__kernel_primitive:NN \tracingoutput \text{tracingoutput:D}
\__kernel_primitive:NN \tracingpages \text{tracingpages:D}
\__kernel_primitive:NN \tracingparagraphs \text{tracingparagraphs:D}
\__kernel_primitive:NN \tracingrestores \text{tracingrestores:D}
\__kernel_primitive:NN \tracingstats \text{tracingstats:D}
\__kernel_primitive:NN \uccode \text{uccode:D}
Primitives introduced by \TeX.

\begin{itemize}
\item \texttt{\textbf{\textbackslash beginL}}
\item \texttt{\textbf{\textbackslash beginR}}
\item \texttt{\textbf{\textbackslash botmarks}}
\item \texttt{\textbf{\textbackslash currentgrouplevel}}
\item \texttt{\textbf{\textbackslash currentgroupstyle}}
\item \texttt{\textbf{\textbackslash currentifbranch}}
\item \texttt{\textbf{\textbackslash currentiflevel}}
\item \texttt{\textbf{\textbackslash currentifstyle}}
\item \texttt{\textbf{\textbackslash detokenize}}
\item \texttt{\textbf{\textbackslash dimexpr}}
\item \texttt{\textbf{\textbackslash displaywidowpenalties}}
\item \texttt{\textbf{\textbackslash endL}}
\item \texttt{\textbf{\textbackslash endR}}
\item \texttt{\textbf{\textbackslash eTeXrevision}}
\item \texttt{\textbf{\textbackslash eTeXversion}}
\item \texttt{\textbf{\textbackslash everyeof}}
\item \texttt{\textbf{\textbackslash firstmarks}}
\item \texttt{\textbf{\textbackslash fontchardp}}
\item \texttt{\textbf{\textbackslash fontcharht}}
\end{itemize}

Primitives introduced by \TeX.\TeX.
Post-ε-Ti\TeX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdf\TeX which directly relate to PDF output: these are copied with the names unchanged.

\__kernel_primitive:NN \fontcharic \tex_fontcharic:D
\__kernel_primitive:NN \fontcharwd \tex_fontcharwd:D
\__kernel_primitive:NN \glueexpr \tex_glueexpr:D
\__kernel_primitive:NN \glueshrink \tex_glueshrink:D
\__kernel_primitive:NN \glueshrinkorder \tex_glueshrinkorder:D
\__kernel_primitive:NN \gluestretch \tex_gluestretch:D
\__kernel_primitive:NN \gluestretchorder \tex_gluestretchorder:D
\__kernel_primitive:NN \gluetomu \tex_gluetomu:D
\__kernel_primitive:NN \ifcsname \tex_ifcsname:D
\__kernel_primitive:NN \ifdefined \tex_ifdefined:D
\__kernel_primitive:NN \iffontchar \tex_iffontchar:D
\__kernel_primitive:NN \interactionmode \tex_interactionmode:D
\__kernel_primitive:NN \interlinepenalties \tex_interlinepenalties:D
\__kernel_primitive:NN \lastlinefit \tex_lastlinefit:D
\__kernel_primitive:NN \lastnodetype \tex_lastnodetype:D
\__kernel_primitive:NN \marks \tex_marks:D
\__kernel_primitive:NN \middle \tex_middle:D
\__kernel_primitive:NN \muexpr \tex_muexpr:D
\__kernel_primitive:NN \mutoglue \tex_mutoglue:D
\__kernel_primitive:NN \numexpr \tex_numexpr:D
\__kernel_primitive:NN \pagediscards \tex_pagediscards:D
\__kernel_primitive:NN \parshapedimen \tex_parshapedimen:D
\__kernel_primitive:NN \parshepindent \tex_parshepindent:D
\__kernel_primitive:NN \parshapelength \tex_parshapelength:D
\__kernel_primitive:NN \preshapebackend \tex_preshapebackend:D
\__kernel_primitive:NN \preshapebackendspacing \tex_preshapebackendspacing:D
\__kernel_primitive:NN \preshapespacing \tex_preshapespacing:D
\__kernel_primitive:NN \predisplaydirection \tex_predisplaydirection:D
\__kernel_primitive:NN \protected \tex_protected:D
\__kernel_primitive:NN \readline \tex_readline:D
\__kernel_primitive:NN \savinghyphcodes \tex_savinghyphcodes:D
\__kernel_primitive:NN \savingydiscards \tex_savingydiscards:D
\__kernel_primitive:NN \scantokens \tex_scantokens:D
\__kernel_primitive:NN \showgroups \tex_showgroups:D
\__kernel_primitive:NN \showifs \tex_showifs:D
\__kernel_primitive:NN \splitbotmarks \tex_splitbotmarks:D
\__kernel_primitive:NN \splitdiscards \tex_splitdiscards:D
\__kernel_primitive:NN \splitfirstmarks \tex_splitfirstmarks:D
\__kernel_primitive:NN \TeXeTeXstate \tex_TeXXeTeXstate:D
\__kernel_primitive:NN \topmarks \tex_topmarks:D
\__kernel_primitive:NN \tracingassigns \tex_tracingassigns:D
\__kernel_primitive:NN \tracinggroups \tex_tracinggroups:D
\__kernel_primitive:NN \tracingifs \tex_trampoline:D
\__kernel_primitive:NN \tracingnesting \tex_tracingnesting:D
\__kernel_primitive:NN \unexpanded \tex_unexpanded:D
\__kernel_primitive:NN \unexpanded \tex_unexpanded:D
\__kernel_primitive:NN \widopenalties \tex_widopenalties:D
\__kernel_primitive:NN \pdfannot \tex_pdfannot:D
\__kernel_primitive:NN \pdfcatalog \tex_pdfcatalog:D
\__kernel_primitive:NN \pdfcompresslevel \tex_pdfcompresslevel:D
These are not related to PDF output and either already appear in other engines without the \texttt{pdf} prefix, or might reasonably do so at some future stage. We therefore drop the leading \texttt{pdf} here.
The version primitives are not related to PDF mode but are pdftex-specific, so again are carried forward unchanged.

These ones appear in pdfTeX but don’t have pdf in the name at all: no decisions to make.

Post pdftex primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

XeTeX-specific primitives. Note that XeTeX’s \strcmp is handled earlier and is “rolled up” into \pdfstrcmp. A few cross-compatibility names which lack the pdf of the original are handled later.
Primitives from pdfTeX that \textXETeX renames: also helps with Lua\textXETeX.

\begin{verbatim}
\__kernel_primitive:NN \creationdate \tex_creationdate:D
\__kernel_primitive:NN \elapsedtime \tex_elapsedtime:D
\__kernel_primitive:NN \filedump \tex_filedump:D
\__kernel_primitive:NN \filesize \tex_filesize:D
\__kernel_primitive:NN \mdfivesum \tex_mdfivesum:D
\__kernel_primitive:NN \ifprimitive \tex_ifprimitive:D
\__kernel_primitive:NN \primitive \tex_primitive:D
\__kernel_primitive:NN \resettimer \tex_resettimer:D
\__kernelPrimitive:NN \shellescape \tex_shellescape:D
\end{verbatim}

Primitives from Lua\textXETeX, some of which have been ported back to \textXETeX.

\begin{verbatim}
\__kernelPrimitive:NN \alignmark \tex_alignmark:D
\__kernelPrimitive:NN \aligntab \tex_aligntab:D
\__kernelPrimitive:NN \attribute \tex_attribute:D
\__kernelPrimitive:NN \attributedef \tex_attributedef:D
\__kernelPrimitive:NN \automaticdiscretionary \tex_automaticdiscretionary:D
\__kernelPrimitive:NN \automatichyphenmode \tex_automatichyphenmode:D
\__kernelPrimitive:NN \automatichyphenpenalty \tex_automatichyphenpenalty:D
\__kernelPrimitive:NN \begincsname \tex_begincsname:D
\__kernelPrimitive:NN \bodydir \tex_bodydir:D
\__kernelPrimitive:NN \bodydirection \tex_bodydirection:D
\__kernelPrimitive:NN \boxdir \tex_boxdir:D
\__kernelPrimitive:NN \boxdirection \tex_boxdirection:D
\__kernelPrimitive:NN \breakafterdirmode \tex_breakafterdirmode:D
\__kernelPrimitive:NN \catcodetable \tex_catcodetable:D
\__kernelPrimitive:NN \clearmarks \tex_clearmarks:D
\__kernelPrimitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
\__kernelPrimitive:NN \crampedscriptscriptstyle \tex_crampedscriptscriptstyle:D
\__kernelPrimitive:NN \crampedscriptstyle \tex_crampedscriptstyle:D
\__kernelPrimitive:NN \crampedtextstyle \tex_crampedtextstyle:D
\__kernelPrimitive:NN \csstring \tex_csstring:D
\__kernelPrimitive:NN \directlua \tex_directlua:D
\__kernelPrimitive:NN \dviextension \tex_dviextension:D
\__kernelPrimitive:NN \dvifeedback \tex_dvifeedback:D
\__kernelPrimitive:NN \dvivariable \tex_dvivariable:D
\__kernelPrimitive:NN \eTeXglueshrinkorder \tex_eTeXglueshrinkorder:D
\__kernelPrimitive:NN \eTeXgluestretchorder \tex_eTeXgluestretchorder:D
\__kernelPrimitive:NN \etoksapp \tex_etoksapp:D
\__kernelPrimitive:NN \etokspre \tex_etokspre:D
\__kernelPrimitive:NN \exceptionpenalty \tex_exceptionpenalty:D
\__kernelPrimitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
\__kernelPrimitive:NN \firstvalidlanguage \tex_firstvalidlanguage:D
\__kernelPrimitive:NN \fontid \tex_fontid:D
\__kernelPrimitive:NN \hjcode \tex_hjcode:D
\__kernelPrimitive:NN \hpack \tex_hpack:D
\__kernelPrimitive:NN \hyphenationbounds \tex_hyphenationbounds:D
\end{verbatim}
Primitives from pdfTeX that LuaTeX renames.

\__kernel_primitive:NN \adjustspacing \text_adjustspacing:D
\__kernel_primitive:NN \adjustspacing\text_adjustspacing:D
\__kernel_primitive:NN \draftmode \text_draftmode:D
\__kernel_primitive:NN \draftmode\text_draftmode:D
\__kernel_primitive:NN \expandglyphsinfont \text_fontexpand:D
\__kernel_primitive:NN \ifabsdim \text_ifabsdim:D
\__kernel_primitive:NN \ifabsdim\text_ifabsdim:D
\__kernel_primitive:NN \ignoreligaturesinfont \text_ignoreligaturesinfont:D
\__kernel_primitive:NN \ignoreligaturesinfont\text_ignoreligaturesinfont:D
\__kernel_primitive:NN \lastsavedboxresourceindex \text_lastsavedboxresourceindex
\text_lastsavedboxresourceindex:D
\__kernel_primitive:NN \lastsavedimageresourceindex \text_lastsavedimageresourceindex
\text_lastsavedimageresourceindex:D
\__kernel_primitive:NN \lastsavedimageresourcepages \text_lastsavedimageresourcepages
\text_lastsavedimageresourcepages:D
\__kernel_primitive:NN \lastxpos \text_lastxpos:D
\__kernel_primitive:NN \lastypos \text_lastypos:D
\__kernel_primitive:NN \normaldeviate \text_normaldeviate:D
\__kernel_primitive:NN \outputmode \text_outputmode:D
\__kernel_primitive:NN \pageheight \text_pageheight:D
\__kernel_primitive:NN \pagewidth \text_pagewidth:D
\__kernel_primitive:NN \protrudechars \text_protrudechars:D
\__kernel_primitive:NN \protrudechars\text_protrudechars:D
\__kernel_primitive:NN \randomseed \text_randomseed:D
\text_randomseed:D

Primitives from pdfTeX that LuaTeX renames.

\__kernel_primitive:NN \adjustspacing \text_adjustspacing:D
\__kernel_primitive:NN \copyfont \text_copyfont:D
\__kernel_primitive:NN \draftmode \text_draftmode:D
\__kernel_primitive:NN \draftmode\text_draftmode:D
\__kernel_primitive:NN \expandglyphsinfont \text_fontexpand:D
\__kernel_primitive:NN \ifabsdim \text_ifabsdim:D
\__kernel_primitive:NN \ifabsdim\text_ifabsdim:D
\__kernel_primitive:NN \ignoreligaturesinfont \text_ignoreligaturesinfont:D
\__kernel_primitive:NN \ignoreligaturesinfont\text_ignoreligaturesinfont:D
\__kernel_primitive:NN \lastsavedboxresourceindex \text_lastsavedboxresourceindex
\text_lastsavedboxresourceindex:D
\__kernel_primitive:NN \lastsavedimageresourceindex \text_lastsavedimageresourceindex
\text_lastsavedimageresourceindex:D
\__kernel_primitive:NN \lastsavedimageresourcepages \text_lastsavedimageresourcepages
\text_lastsavedimageresourcepages:D
\__kernel_primitive:NN \lastxpos \text_lastxpos:D
\__kernel_primitive:NN \lastypos \text_lastypos:D
\__kernel_primitive:NN \normaldeviate \text_normaldeviate:D
\__kernel_primitive:NN \outputmode \text_outputmode:D
\__kernel_primitive:NN \pageheight \text_pageheight:D
\__kernel_primitive:NN \pagewidth \text_pagewidth:D
\__kernel_primitive:NN \protrudechars \text_protrudechars:D
\__kernel_primitive:NN \protrudechars\text_protrudechars:D
\__kernel_primitive:NN \randomseed \text_randomseed:D
\text_randomseed:D
The set of Unicode math primitives were introduced by \texttt{Xe\TeX} and \texttt{Lua\TeX} in a somewhat complex fashion: a few first as \texttt{\LaTeX}... which were then renamed with \texttt{Lua\TeX} having a lot more. These names now all start \texttt{\U}... and mainly \texttt{\Umath}....
Primitives from p\LaTeX.

\begin{verbatim}
1191 \__kernel_primitive:NN \Umathunderdelimitervgap
1192 \tex_Umathunderdelimitervgap:D
1193 \__kernel_primitive:NN \Unosubscript \tex_Unosubscript:D
1194 \__kernel_primitive:NN \Unosuperscript \tex_Unosuperscript:D
1195 \__kernel_primitive:NN \Uoverdelimiter \tex_Uoverdelimiter:D
1196 \__kernel_primitive:NN \Uradical \tex_Uradical:D
1197 \__kernel_primitive:NN \Uroot \tex_Uroot:D
1198 \__kernel_primitive:NN \Uskewed \tex_Uskewed:D
1199 \__kernel_primitive:NN \Uskewedwithdelims \tex_Uskewedwithdelims:D
1200 \__kernel_primitive:NN \Ustack \tex_Ustack:D
1201 \__kernel_primitive:NN \Ustartdisplaymath \tex_Ustartdisplaymath:D
1202 \__kernel_primitive:NN \Ustartmath \tex_Ustartmath:D
1203 \__kernel_primitive:NN \Ustopdisplaymath \tex_Ustopdisplaymath:D
1204 \__kernel_primitive:NN \Ustopmath \tex_Ustopmath:D
1205 \__kernel_primitive:NN \Usubscript \tex_Usubscript:D
1206 \__kernel_primitive:NN \Usuperscript \tex_Usuperscript:D
1207 \__kernel_primitive:NN \Uunderdelimiter \tex_Uunderdelimiter:D
1208 \__kernel_primitive:NN \Uvextensible \tex_Uvextensible:D

Primitives from p\LaTeX.

1209 \__kernel_primitive:NN \autospacing \tex_autospacing:D
1210 \__kernel_primitive:NN \autoxspacing \tex_autoxspacing:D
1211 \__kernel_primitive:NN \currentcjktoken \tex_currentcjktoken:D
1212 \__kernel_primitive:NN \currentspacingmode \tex_currentspacingmode:D
1213 \__kernel_primitive:NN \currentxspacingmode \tex_currentxspacingmode:D
1214 \__kernel_primitive:NN \disinhibitglue \tex_disinhibitglue:D
1215 \__kernel_primitive:NN \dtou \tex_dtou:D
1216 \__kernel_primitive:NN \epTeXinputencoding \tex_epTeXinputencoding:D
1217 \__kernel_primitive:NN \epTeXversion \tex_epTeXversion:D
1218 \__kernel_primitive:NN \euc \tex_euc:D
1219 \__kernel_primitive:NN \hfi \tex_hfi:D
1220 \__kernel_primitive:NN \ifdbox \tex_ifdbox:D
1221 \__kernel_primitive:NN \ifddir \tex_ifddir:D
1222 \__kernel_primitive:NN \ifjfont \tex_ifjfont:D
1223 \__kernel_primitive:NN \iftbox \tex_iftbox:D
1224 \__kernel_primitive:NN \iftfont \tex_iftfont:D
1225 \__kernel_primitive:NN \iftext \tex_iftext:D
1226 \__kernel_primitive:NN \jcharwidowpenalty \tex_jcharwidowpenalty:D
1227 \__kernel_primitive:NN \jfam \tex_jfam:D
1228 \__kernel_primitive:NN \jfont \tex_jfont:D
1229 \__kernel_primitive:NN \jis \tex_jis:D
1229 \__kernel_primitive:NN \kanjiskip \tex_kanjiskip:D
1229 \__kernel_primitive:NN \kansuji \tex_kansuji:D
1229 \__kernel_primitive:NN \kansujichar \tex_kansujichar:D
1229 \__kernel_primitive:NN \kcatcode \tex_kcatcode:D
1229 \__kernel_primitive:NN \kuten \tex_kuten:D
1229 \__kernel_primitive:NN \lastnodechar \tex_lastnodechar:D
1229 \__kernel_primitive:NN \lastnodesubtype \tex_lastnodesubtype:D
1229 \__kernel_primitive:NN \noautospacing \tex_noautospacing:D

296
\end{verbatim}

Primitives from upTEX.

Omega primitives provided by upTEX (listed separately mainly to allow understanding of their source).

End of the “just the names” part of the source.
The \texttt{shipout} primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do not delay action to the \AtBeginDocument hook. (We cannot use \texttt{primitive} as that doesn’t allow us to make a direct copy of the primitive itself.) As we know that \LaTeX{} is in use, we use its \texttt{@tfor} loop here.

Some tidying up is needed for \texttt{(pdf)tracingfonts}. Newer Lua\TeX{} has this simply as \texttt{tracingfonts}, but that is overwritten by the \LaTeX{} kernel. So any spurious definition has to be removed, then the real version saved either from the pdf\TeX{} name or from Lua\TeX{}. In the latter case, we leave \texttt{@tracingfonts} available: this might be useful and almost all \LaTeX{} users will have expl3 loaded by fontsymb. (We follow the usual kernel convention that \texttt{@@} is used for saved primitives.)
That is also true for the LaTeX primitives under \LaTeX (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

Which also covers those slightly odd ones.
Only pdfTeX and LuaTeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

A few packages do unfortunate things to date-related primitives.
Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that \texttt{\textunderscore pdf\textunderscore tex\textunderscore version:D} is a marker for pdfTeX alone: useful in engine-dependent code later.

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using \texttt{\end} as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

2.1 Deprecated functions

Older versions of expl3 divided up primitives by “source”: that becomes very tricky with multiple parallel engine developments, so has been dropped. To cover the transition, we provide the older names here for a limited period (until the end of 2019).

To allow \texttt{\debug\textunderscore on:n \{(deprecation)\}} to work we save the list of primitives into \texttt{\_\_kernel\_primitives}:
\textbegingroup:D
\tex_long:D \textdef:D \use_ii:nn #1#2 {#2}
\tex_long:D \textdef:D \use_none:n #1 { }
\tex_long:D \textdef:D \__kernel_primitive:NN #1#2
{\tex_ifdefined:D #1}
\tex_expandafter:D \use_ii:nn \texfi:D
\use_none:n { \texglobal:D \texlet:D \#2 \#1 }
\}
\tex_xdef:D \__kernel_primitives:
{\tex_unexpanded:D}
{ \tex_unexpanded:D}
\__kernel_primitive:NN \beginL \etex_beginL:D
\__kernel_primitive:NN \beginR \etex_beginR:D
\__kernel_primitive:NN \botmarks \etex_botmarks:D
\__kernel_primitive:NN \clubpenalties \etex_clubpenalties:D
\__kernel_primitive:NN \currentgrouplevel \etex_currentgrouplevel:D
\__kernel_primitive:NN \currentgrouptype \etex_currentgrouptype:D
\__kernel_primitive:NN \currentifbranch \etex_currentifbranch:D
\__kernel_primitive:NN \currentiflevel \etex_currentiflevel:D
\__kernel_primitive:NN \currentiftype \etex_currentiftype:D
\__kernel_primitive:NN \detokenize \etex_detokenize:D
\__kernel_primitive:NN \dimexpr \etex_dimexpr:D
\__kernel_primitive:NN \displaywidowpenalties
\etex_displaywidowpenalties:D
\__kernel_primitive:NN \endL \etex_endL:D
\__kernel_primitive:NN \endR \etex_endR:D
\__kernel_primitive:NN \eTeXrevision \etex_eTeXrevision:D
\__kernel_primitive:NN \eTeXversion \etex_eTeXversion:D
\__kernel_primitive:NN \everyeof \etex_everyeof:D
\__kernel_primitive:NN \firstmarks \etex_firstmarks:D
\__kernel_primitive:NN \fontchardp \etex_fontchardp:D
\__kernel_primitive:NN \fontcharht \etex_fontcharht:D
\__kernel_primitive:NN \fontcharic \etex_fontcharic:D
\__kernel_primitive:NN \fontcharwd \etex_fontcharwd:D
\__kernel_primitive:NN \glueexpr \etex_glueexpr:D
\__kernel_primitive:NN \glueexpr \etex_glueexpr:D
\__kernel_primitive:NN \gluestretch \etex_gluestretch:D
\__kernel_primitive:NN \gluestretch \etex_gluestretch:D
\__kernel_primitive:NN \gluestretchorder \etex_gluestretchorder:D
\__kernel_primitive:NN \gluetomu \etex_gluetomu:D
\__kernel_primitive:NN \ifcsname \etex_ifcsname:D
\__kernel_primitive:NN \ifdefined \etex_ifdefined:D
\__kernel_primitive:NN \iffontchar \etex_iffontchar:D
\__kernel_primitive:NN \interactionmode \etex_interactionmode:D
\__kernel_primitive:NN \interlinepenalties \etex_interlinepenalties:D
\__kernel_primitive:NN \lastlinefit \etex_lastlinefit:D
\__kernel_primitive:NN \lastnodetype \etex_lastnodetype:D
\__kernel_primitive:NN \marks \etex_marks:D
\__kernel_primitive:NN \middle \etex_middle:D
\__kernel_primitive:NN \muexpr \etex_muexpr:D
3 Internal kernel functions

\__kernel chk cs exist:N \<cs>
\__kernel chk cs exist:C\<cs>
This function is only created if debugging is enabled. It checks that \<cs> exists according to the criteria for \cs_if_exist_p:N, and if not raises a kernel-level error.

\__kernel chk defined:NT \<variable> \{(true code)\}
If \<variable> is not defined (according to \cs_if_exist:NTF), this triggers an error, otherwise the \{true code\} is run.
This function is only created if debugging is enabled. By default it is equivalent to \use_i:i:nnnn. When expression checking is enabled, it leaves in the input stream the result of \tex_the:D (eval) \tex_expr:D \tex_relax:D after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the \textbf{(caller)}. For instance \textbf{(eval)} can be \textbf{\_\_int_eval:w} and \textbf{(caller)} can be \textbf{\int_eval:n} or \textbf{\int_set:Nn}. The argument \textbf{(convert)} is empty except for mu expressions where it is \textbf{\tex_mutoglue:D}, used for internal purposes.

Evaluates the number of \textbf{(args)} and leaves the \textbf{(follow-on)} code followed by a brace group containing the required number of primitive parameter markers (#1, etc.). If the number of \textbf{(args)} is outside the range [0,9], the \textbf{(false code)} is inserted instead of the \textbf{(follow-on)}.

Stores both an \textbf{(error)} and \textbf{(working)} definition for given material such that they can be exchanged by \debug_on: and \debug_off:.

Carries out expansion on the \textbf{(expandable tokens)} before preventing further expansion of the \textbf{(content)}, i.e. allow it to be generated in some way.

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

Expands the \textbf{(name)} as per \textbf{\_\_kernel_file_name_sanitize:nN} then produces an error message indicating that this file was not found.

For converting a \textbf{(name)} to a string where active characters are treated as strings.

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the \TeX kernel is necessary.

Expands to the result of adding the three \textbf{(integers)} (which must be suitable input for \textbf{\int_eval:w}), avoiding intermediate overflow. Overflow occurs only if the overall result is outside \([-2^{31}+1,2^{31}−1]\]. The \textbf{(integers)} may be of the form \textbf{\int_eval:w}... \textbf{\scan_stop}: but may be evaluated more than once.
\_\_kernel_iow_with:Nn

\_\_kernel_iow_with:Nn \{integer\} \{value\} \{code\}

If the \{integer\} is equal to the \{value\} then this function simply runs the \{code\}. Otherwise it saves the current value of the \{integer\}, sets it to the \{value\}, runs the \{code\}, and restores the \{integer\} to its former value. This is used to ensure that the newlinechar is 10 when writing to a stream, which lets \low_newline: work, and that \errorcontextlines is –1 when displaying a message.

\_\_kernel_msg_new:nnn

\_\_kernel_msg_new:nnn \{module\} \{message\} \{text\} \{more text\}

Creates a kernel \{message\} for a given \{module\}. The message is defined to first give \{text\} and then \{more text\} if the user requests it. If no \{more text\} is available then a standard text is given instead. Within \{text\} and \{more text\} four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used. An error is raised if the \{message\} already exists.

\_\_kernel_msg_set:nnn

\_\_kernel_msg_set:nnn \{module\} \{message\} \{text\} \{more text\}

Sets up the text for a kernel \{message\} for a given \{module\}. The message is defined to first give \{text\} and then \{more text\} if the user requests it. If no \{more text\} is available then a standard text is given instead. Within \{text\} and \{more text\} four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used.

\_\_kernel_msg_fatal:nnnn

\_\_kernel_msg_fatal:nnnn \{module\} \{message\} \{arg one\} \{arg two\} \{arg three\} \{arg four\}

Issues kernel \{module\} error \{message\}, passing \{arg one\} to \{arg four\} to the text-creating functions. After issuing a fatal error the \TeX{} run halts. Cannot be redirected.

\_\_kernel_msg_error:nnnn

\_\_kernel_msg_error:nnnn \{module\} \{message\} \{arg one\} \{arg two\} \{arg three\} \{arg four\}

Issues kernel \{module\} error \{message\}, passing \{arg one\} to \{arg four\} to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.
\_kernel_msg_warning:nnnnn
\_kernel_msg_warning:nnxxxx
\_kernel_msg_warning:nnnnn
\_kernel_msg_warning:nnnn
\_kernel_msg_warning:nnx
\_kernel_msg_warning:nn

Issues kernel \textit{(module)} warning \textit{(message)}, passing \textit{(arg one)} to \textit{(arg four)} to the text-creating functions. The warning text is added to the log file, but the \LaTeX{} run is not interrupted.

\_kernel_msg_info:nnnnn
\_kernel_msg_info:nnxxxx
\_kernel_msg_info:nnnnn
\_kernel_msg_info:nnnn
\_kernel_msg_info:nnx
\_kernel_msg_info:nn

Issues kernel \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.

\_kernel_msg_expandable_error:nnnnnn
\_kernel_msg_expandable_error:nnxxxx
\_kernel_msg_expandable_error:nnnnn
\_kernel_msg_expandable_error:nnnn
\_kernel_msg_expandable_error:nnx
\_kernel_msg_expandable_error:nn

Issues an error, passing \textit{(arg one)} to \textit{(arg four)} to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.

\texttt{\_g\_kernel_prg\_map\_int}

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions \texttt{\(\text{\langle type\rangle}_\text{map}_1:w\)}, \texttt{\(\text{\langle type\rangle}_\text{map}_2:w\)}, etc., labelled by \texttt{\_g\_kernel_prg\_map\_int} hold functions to be mapped over various list datatypes in inline and variable mappings.

(End definition for \texttt{\_g\_kernel_prg\_map\_int}.)
\_\_kernel\_quark\_new\_test:N \_\_kernel\_quark\_new\_test:N \langle name \rangle:(arg spec)

Defines a quark-test function \langle name \rangle:(arg spec) which tests if its argument is \q\_\_\langle namespace\rangle\_\_recursion\_tail, then acts accordingly, as described below for each possible \langle arg spec \rangle.

The \langle namespace \rangle is determined as the first (nonempty) \_\_delimited word in \langle name \rangle and is used internally in the definition of auxiliaries. The function \_\_kernel\_quark\_new\_test:N does not define the \q\_\_\langle namespace\rangle\_\_recursion\_tail and \q\_\_\langle namespace\rangle\_\_recursion\_stop quarks. They should be manually defined with \quark\_new:N.

There are 6 different types of quark-test functions. Which one is defined depends on the \langle arg spec \rangle, which must be one of the options listed now. Four of them are modeled after \quark\_if\_recursion\_tail:\(N|n) and \quark\_if\_recursion\_tail\_do:\(N|n)n.

\n defines \langle name \rangle:n such that it checks if \#1 contains only \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so consumes all tokens up to \q\_\_\langle namespace\rangle\_\_recursion\_stop (c.f. \quark\_if\_recursion\_tail\_stop:n).

\nn defines \langle name \rangle:nn such that it checks if \#1 contains only \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so consumes all tokens up to \q\_\_\langle namespace\rangle\_\_recursion\_stop, then executes the code \#2 after that (c.f. \quark\_if\_recursion\_tail\_stop\_do:nn).

\N defines \langle name \rangle:N such that it checks if \#1 is \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so consumes all tokens up to \q\_\_\langle namespace\rangle\_\_recursion\_stop (c.f. \quark\_if\_recursion\_tail\_stop:N).

\nnN defines \langle name \rangle:nnN such that it checks if \#1 is \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so consumes all tokens up to \q\_\_\langle namespace\rangle\_\_recursion\_stop, then executes the code \#2 after that (c.f. \quark\_if\_recursion\_tail\_stop\_do:nnN).

The last two are modeled after \quark\_if\_recursion\_tail\_break:\(n|N)N, and in those cases the quark \q\_\_\langle namespace\rangle\_\_recursion\_stop is not used (and thus needs not be defined).

\nN defines \langle name \rangle:nN such that it checks if \#1 contains only \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so uses the \langle type\rangle\_map\_break: function \#2.

\NN defines \langle name \rangle:NN such that it checks if \#1 is \q\_\_\langle namespace\rangle\_\_recursion\_tail, and if so uses the \langle type\rangle\_map\_break: function \#2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.
\_kernel_quark_new_conditional:Nn \_kernel_quark_new_conditional:Nn
\_\langle namespace\rangle\_quark_if\_\langle name\rangle:\langle arg spec\rangle \{\langle conditions\rangle\}

Defines a collection of quark conditionals that test if their argument is the quark \q_\langle namespace\rangle\_\langle name\rangle and perform suitable actions. The \langle conditions\rangle are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each \langle condition\rangle in the list, as described for \prg_new_conditional:Npnn. The conditionals are defined using \prg_new_conditional:Npnn, so that their name is obtained by adding p, T, F, or TF to the base name \_\langle namespace\rangle\_quark_if\_\langle name\rangle:\langle arg spec\rangle.

The first argument of \_kernel_quark_new_conditional:Nn must contain \quark_if_ and :, as these markers are used to determine the \langle name\rangle of the quark \q_\langle namespace\rangle\_\langle name\rangle to be tested. This quark should be manually defined with \quark_new:N, as \_kernel_quark_new_conditional:Nn does not define it.

The function \_kernel_quark_new_conditional:Nn can define 2 different types of quark conditionals. Which one is defined depends on the \langle arg spec\rangle, which must be one of the following options, modeled after \quark_if_nil:(N|n)(TF).

n defines \_\langle namespace\rangle\_quark_if\_\langle name\rangle:n(TF) such that it checks if \#1 contains only \q_\langle namespace\rangle\_\langle name\rangle, and executes the proper conditional branch.

N defines \_\langle namespace\rangle\_quark_if\_\langle name\rangle:N(TF) such that it checks if \#1 is \q_\langle namespace\rangle\_\langle name\rangle, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

\c\_kernel_randint_max_int

Maximal allowed argument to \_kernel_randint:n. Equal to \textit{2}^{17} - 1.

(End definition for \c\_kernel_randint_max_int.)

\_kernel_randint:n

\_kernel_randint:n \{\langle max\rangle\}

Used in an integer expression this gives a pseudo-random number between 1 and \langle max\rangle included. One must have \langle max\rangle \leq \textit{2}^{17} - 1. The \langle max\rangle must be suitable for \int_value:w (and any \int_eval:w must be terminated by \scan_stop: or equivalent).

\_kernel_randint:nn

\_kernel_randint:nn \{\langle min\rangle\} \{\langle max\rangle\}

Used in an integer expression this gives a pseudo-random number between \langle min\rangle and \langle max\rangle included. The \langle min\rangle and \langle max\rangle must be suitable for \int_value:w (and any \int_eval:w must be terminated by \scan_stop: or equivalent). For small ranges \( R = \langle max\rangle - \langle min\rangle + 1 \leq \textit{2}^{17} - 1, \langle min\rangle - 1 + \_kernel_randint:n\{R\} \) is faster.

\_kernel_register_show:N
\_kernel_register_show:c

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

\_kernel_register_log:N
\_kernel_register_log:c

Used to write the contents of a \TeX{} register to the log file in a form similar to \_kernel_register_show:N.
\__kernel_str_to_other:n \__kernel_str_to_other:n {\langle token list\rangle}

Converts the \langle token list\rangle to a \langle other string\rangle, where spaces have category code “other”. This function can be \texttt{f}-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

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

Same behaviour \__kernel_str_to_other:n but only restricted-expandable. It takes a time linear in the character count of the string.

\__kernel_tl_to_str:w \__kernel_tl_to_str:w {\langle expandable tokens\rangle} {\langle tokens\rangle}

Carries out expansion on the \langle expandable tokens\rangle before conversion of the \langle tokens\rangle to a string as describe for \texttt{tl_to_str:n}. Typically, the \langle expandable tokens\rangle will alter the nature of the \langle tokens\rangle, i.e. allow it to be generated in some way. This function requires only a single expansion.

4 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

\__kernel_backend_literal:n \__kernel_backend_literal:n {\langle content\rangle}
\__kernel_backend_literal:n {\langle content\rangle}

\__kernel_backend_literal_postscript:n \__kernel_backend_literal_postscript:n {\langle PostScript\rangle}
\__kernel_backend_literal_postscript:x

\__kernel_backend_literal_pdf:n \__kernel_backend_literal_pdf:n {\langle PDF instructions\rangle}
\__kernel_backend_literal_pdf:x

\__kernel_backend_literal_svg:n \__kernel_backend_literal_svg:n {\langle SVG instructions\rangle}
\__kernel_backend_literal_svg:x

\__kernel_backend_postscript:n \__kernel_backend_postscript:n {\langle PostScript\rangle}
\__kernel_backend_postscript:x

Adds the \langle content\rangle literally to the current vertical list as a whatsit. The nature of the \langle content\rangle will depend on the backend in use.

Adds the \langle PostScript\rangle literally to the current vertical list as a whatsit. No positioning is applied.

Adds the \langle PDF instructions\rangle literally to the current vertical list as a whatsit. No positioning is applied.

Adds the \langle SVG instructions\rangle literally to the current vertical list as a whatsit. No positioning is applied.

Adds the \langle PostScript\rangle to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.
Arranges to align the PostScript and DVI current positions and scales.

Creates a scope for instructions at the backend level.

Applies the \langle\text{matrix}\rangle to the current transformation matrix.

Specifies whether to write headers for the backend.

The color stack used in pdf\TeX{} and Lua\TeX{} for the main color.

\section{\texttt{l3basics} implementation}

\subsection{Renaming some TeX primitives (again)}

Having given all the \TeX{} primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.\footnote{This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the \texttt{\tex\ldots:D} name in the cases where no good alternative exists.}

Then some conditionals.

(\textit{End definition for} \texttt{\if_true:} and others. \textit{These functions are documented on page 23.})

\TeX{} lets us detect some if its modes.
Building csnames and testing if control sequences exist.

\ifcs_exist:N \tex_let:D \ifcs_exist:N \tex_ifdefined:D
\ifcs_exist:w \tex_let:D \ifcs_exist:w \tex_ifcsname:D
\cs:w \tex_let:D \cs:w \tex_csname:D
\cs_end: \tex_let:D \cs_end: \tex_endcsname:D

Examining a control sequence or token.

\token_to_meaning:N \tex_let:D \token_to_meaning:N \tex_meaning:D
\cs_meaning:N \tex_let:D \cs_meaning:N \tex_meaning:D

Making strings.

\tl_to_str:n \tex_let:D \tl_to_str:n \tex_detokenize:D
\token_to_str:N \tex_let:D \token_to_str:N \tex_string:D
\__kernel_tl_to_str:w \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D

Adding material after the end of a group.

\group_insert_after:N \tex_let:D \group_insert_after:N \tex_aftergroup:D

For integers.

\__int_to_roman:w \tex_let:D \__int_to_roman:w \tex_romannumeral:D

Adding material after the end of a group.

\group_insert_after:N \tex_let:D \group_insert_after:N \tex_aftergroup:D
Discussed in \texttt{l3expan}, but needed much earlier.

\begin{verbatim}
\tokentotestringc \tokentotestring \csmeaningc
\end{verbatim}

A small number of variants defined by hand. Some of the necessary functions (\texttt{use_i:nn}, \texttt{use_ii:nn}, and \texttt{exp_args:NNc}) are not defined at that point yet, but will be defined before those variants are used. The \texttt{cs_meaning:c} command must check for an undefined control sequence to avoid defining it mistakenly.

\begin{verbatim}
\tokentotestringc \tokentotestring \csmeaningc
\end{verbatim}

\subsection{5.2 Defining some constants}

\texttt{\c_zero_int}  We need the constant \texttt{\c_zero_int} which is used by some functions in the \texttt{l3alloc} module. The rest are defined in the \texttt{l3int} module – at least for the ones that can be defined with \texttt{\tex_chardef:D} or \texttt{\tex_mathchardef:D}. For other constants the \texttt{l3int} module is required but it can’t be used until the allocation has been set up properly!

\begin{verbatim}
\tex_chardef:D \c_zero_int = 0
\end{verbatim}

\texttt{\c_max_register_int} This is here as this particular integer is needed both in package mode and to bootstrap \texttt{l3alloc}, and is documented in \texttt{l3int}. Lua\TeX and those which contain parts of the Omega extensions have more registers available than \texttt{\epsilon-TEX}.

\begin{verbatim}
\texifdef:D \texluatexversion:D \texchardef:D \c_max_register_int = 65535
\texelse:D
\texifdef:D \texomathchardef:D
\texomathchardef:D \c_max_register_int = 65535
\texelse:D
\texchardef:D \c_max_register_int = 32767
\texfi:D
\end{verbatim}

\section*{5.2 Defining some constants}

\texttt{\c_zero_int}  We need the constant \texttt{\c_zero_int} which is used by some functions in the \texttt{l3alloc} module. The rest are defined in the \texttt{l3int} module – at least for the ones that can be defined with \texttt{\tex_chardef:D} or \texttt{\tex_mathchardef:D}. For other constants the \texttt{l3int} module is required but it can’t be used until the allocation has been set up properly!

\begin{verbatim}
\tex_chardef:D \c_zero_int = 0
\end{verbatim}

\texttt{\c_max_register_int} This is here as this particular integer is needed both in package mode and to bootstrap \texttt{l3alloc}, and is documented in \texttt{l3int}. Lua\TeX and those which contain parts of the Omega extensions have more registers available than \texttt{\epsilon-TEX}.

\begin{verbatim}
\texifdef:D \texluatexversion:D \texchardef:D \c_max_register_int = 65535
\texelse:D
\texifdef:D \texomathchardef:D
\texomathchardef:D \c_max_register_int = 65535
\texelse:D
\texchardef:D \c_max_register_int = 32767
\texfi:D
\end{verbatim}
5.3 Defining functions

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

All assignment functions in \LaTeX{}3 should be naturally protected; after all, the \TeX{} primitives for assignments are and it can be a cause of problems if others aren’t.

Global versions of the above functions.

(End definition for \cs_set{nopar}{Npn} and others. These functions are documented on page 11.)

5.4 Selecting tokens

Scratch token list variable for l3expan, used by \use:x, used in defining conditionals. We don’t use \tl methods because l3basics is loaded earlier.

(End definition for \l_{_{\text{exp internal tl}}}.)

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

(End definition for \use:c. This function is documented on page 16.)
\texttt{\use:x} Fully expands its argument and passes it to the input stream. Uses the reserved \texttt{\_\_\_exp_internal_tl} which we’ve set up above.
\begin{verbatim}
\cs_set_protected:Npm \use:x #1
\{
 \cs_set_nopar:Npx \l__exp_internal_tl {#1}
 \l__exp_internal_tl
\}
\end{verbatim}
(End definition for \texttt{\use:x}. This function is documented on page 20.)
\begin{verbatim}
\end{verbatim}
\texttt{\use:e} In non-LuaLaTeXengines older than 2019, \texttt{\expanded} is emulated.
\begin{verbatim}
\cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
\tex_ifdefined:D \tex_expanded:D \tex_else:D
\cs_set:Npn \use:e #1 { \exp_args:Ne \use:n {#1} }
\tex_fi:D
\end{verbatim}
(End definition for \texttt{\use:e}. This function is documented on page 20.)
\begin{verbatim}
\end{verbatim}
\texttt{\use:n} These macros grab their arguments and return them back to the input (with outer braces removed).
\texttt{\use:nnn} The equivalent to \LaTeXε’s \texttt{@firstoftwo} and \texttt{@secondoftwo}.
\begin{verbatim}
\cs_set:Npn \use:n #1#2 {#1}
\cs_set:Npn \use:nn #1#2 {#1#2}
\cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
\cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
\end{verbatim}
(End definition for \texttt{\use:n} and \texttt{\use:nn}. These functions are documented on page 19.)
\begin{verbatim}
\end{verbatim}
\texttt{\use:i:nn} We also need something for picking up arguments from a longer list.
\texttt{\use:nnn} The equivalent to \LaTeXε’s \texttt{@firstoftwo} and \texttt{@secondoftwo}.
\begin{verbatim}
\cs_set:Npn \use:i:nn #1#2 #1 {#1}
\cs_set:Npn \use:ii:nn #1#2 #1 {#1#2}
\cs_set:Npn \use:iii:nn #1#2#3 {#1#2#3}
\cs_set:Npn \use:iv:nnnn #1#2#3#4 {#1#2#3#4}
\end{verbatim}
(End definition for \texttt{\use:i:nn} and \texttt{\use:ii:nn}. These functions are documented on page 19.)
\begin{verbatim}
\end{verbatim}
\texttt{\use:nnn} The equivalent to \LaTeXε’s \texttt{@firstoftwo} and \texttt{@secondoftwo}.
\begin{verbatim}
\cs_set:Npn \use:i:nnn #1#2#3 #1 {#1}
\cs_set:Npn \use:ii:nnn #1#2#3 #2 {#1#2}
\cs_set:Npn \use:iii:nnnn #1#2#3#4 #1 {#1#2#3}
\cs_set:Npn \use:iv:nnnnn #1#2#3#4#5 {#1#2#3#4}
\end{verbatim}
(End definition for \texttt{\use:i:nnn} and others. These functions are documented on page 19.)
\begin{verbatim}
\end{verbatim}
\texttt{\use:ii:nn}
Functions that gobble everything until they see either \( \texttt{\_q\_nil} \), \( \texttt{\_q\_stop} \), or \( \texttt{\_q\_recursion\_stop} \), respectively.

\begin{verbatim}
\cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
\cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
\cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
\end{verbatim}

(End definition for \use_none_delimit_by_q_nil:w, \use_none_delimit_by_q_stop:w, and \use_none_delimit_by_q_recursion_stop:w. These functions are documented on page 21.)

Functions that gobble everything until they see either \( \texttt{\_q\_nil} \), \( \texttt{\_q\_stop} \), or \( \texttt{\_q\_recursion\_stop} \), respectively.

\begin{verbatim}
\cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
\end{verbatim}

(End definition for \use_i_delimit_by_q_nil:nw, \use_i_delimit_by_q_stop:nw, and \use_i_delimit_by_q_recursion_stop:nw. These functions are documented on page 21.)

\subsection{Gobbling tokens from input}

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of \texttt{n}'s following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of \use_none:nnnnn, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

\begin{verbatim}
\cs_set:Npn \use_none:n #1 { }
\cs_set:Npn \use_none:nn #1#2 { }
\cs_set:Npn \use_none:nnn #1#2#3 { }
\cs_set:Npn \use_none:nnnn #1#2#3#4 { }
\cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
\cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
\cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
\cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
\cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8#9 { }
\end{verbatim}

(End definition for \use_none:n and others. These functions are documented on page 20.)

\subsection{Debugging and patching later definitions}

\begin{verbatim}
\__kernel_if_debug:TF A more meaningful test of whether debugging is enabled than messing up with guards.
\end{verbatim}

We can also more easily change the logic in one place then. This is needed primarily for deprecations.

\begin{verbatim}
\cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
\end{verbatim}

(End definition for \__kernel_if_debug:TF.)
\debug_on:n  Stubs.
\debug_off:n
\debug_suspend:
\debug_resume:

Some commands were more recently deprecated and not yet removed; only make these
into errors if the user requests it. This relies on two token lists, filled up in l3deprecation.

\__kernel_deprecation_code:nn
    \g__debug_deprecation_on_tl
    \g__debug_deprecation_off_tl

5.7 Conditional processing and definitions

Underneath any predicate function (_p) or other conditional forms (TF, etc.) is a
built-in logic saying that it after all of the testing and processing must return the (state)
this leaves \TeX in. Therefore, a simple user interface could be something like
\if_meaning:w #1#2
    \prg_return_true:
\else:
    \if_meaning:w #1#3
        \prg_return_true:
    \else:
        \prg_return_false:
    \fi:
\fi:

Usually, a \TeX programmer would have to insert a number of \exp_after:wN\s to ensure
the state value is returned at exactly the point where the last conditional is finished.
However, that obscures the code and forces the \TeX programmer to prove that he/she
knows the $2^n - 1$ table. We therefore provide the simpler interface.
The idea here is that \texttt{\exp:w} expands fully any \texttt{\else:} and \texttt{\fi:} that are waiting to be discarded, before reaching the \texttt{\exp_end:} which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

\begin{verbatim}
\cs_set:Npn \prg_return_true: { \exp_after:wN \use_i:nn \exp:w }
\cs_set:Npn \prg_return_false: { \exp_after:wN \use_ii:nn \exp:w}
\end{verbatim}

An extended state space could be implemented by including a more elaborate function in place of \texttt{\use_i:nn/\use_ii:nn}. Provided two arguments are absorbed then the code would work.

\textit{(End definition for \texttt{\prg_return_true:} and \texttt{\prg_return_false:]. These functions are documented on page 106.)}

Private version of \texttt{use_none_delimit_by_q_recursion_stop:w}.

\begin{verbatim}
\cs_set_protected:Npn \__prg_use_none_delimit_by_q_recursion_stop:w #1 \q__prg_recursion_stop { }
\end{verbatim}

\textit{(End definition for \texttt{\__prg_use_none_delimit_by_q_recursion_stop:w.})}

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those \texttt{Npnn} type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed \{\texttt{name}\} \{\texttt{signature}\} \{\texttt{boolean}\} \{\texttt{set or new}\} \{\texttt{maybe protected}\} \{\texttt{parameters}\} \{\texttt{TF,\ldots}\} \{\texttt{code}\} to the auxiliary function responsible for defining all conditionals. Note that \texttt{e} stands for expandable and \texttt{p} for protected.

\begin{verbatim}
\cs_set_protected:Npn \prg_set_conditional:Npnn \cs_set:Npn e \__prg_generate_conditional_parm:NNNpnn \cs_split_function:N #3
\end{verbatim}

\textit{(End definition for \texttt{\prg_set_conditional:Npnn and others. These functions are documented on page 104.)}

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary
generates the parameter text from the number of letters in the signature. Then feed
\{(name)\} \{(signature)\} \{boolean\} \{(set or new)\} \{\{maybe protected\}\} \{(parameters)\}
\{TF,\ldots\} \{(code)\} to the auxiliary function responsible for defining all conditionals. If
the \{signature\} has more than 9 letters, the definition is aborted since \TeX\ macros have
at most 9 arguments. The erroneous case where the function name contains no colon is
captured later.

\begin{verbatim}
\cs_set_protected:Npn \prg_set_conditional:Nnn
\{ \__prg_generate_conditional_count:NNNnn \cs_set:Npn e \}
\cs_set_protected:Npn \prg_new_conditional:Nnn
\{ \__prg_generate_conditional_count:NNNnn \cs_new:Npn e \}
\cs_set_protected:Npn \prg_set_protected_conditional:Nnn
\{ \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p \}
\cs_set_protected:Npn \prg_new_protected_conditional:Nnn
\{ \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p \}
\cs_set_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
\{ \use:x \{ \__prg_generate_conditional_count:nnNNNnn \cs_split_function:N #3 \}
#1 #2 \}
\cs_set_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
\{ \use:x \{ \__prg_generate_conditional:nnNNNnn \cs_split_function:N #8 \}
\cs_set_protected:Nnn \__prg_generate_conditional:NNnnnnNw
\__prg_generate_conditional_test:w
\__prg_generate_conditional_fast:nw
\end{verbatim}

The workhorse here is going through a list of desired forms, i.e., p, TF, T and F. The first
three arguments come from splitting up the base form of the conditional, which gives the
name, signature and a boolean to signal whether or not there was a colon in the name.
In the absence of a colon, we throw an error and don’t define any conditional. The fourth
and fifth arguments build up the defining function. The sixth is the parameters to use
(possibly empty), the seventh is the list of forms to define, the eighth is the replacement
text which we will augment when defining the forms. The use of \texttt{\tl_to_str:n} makes
the later loop more robust.

A large number of our low-level conditionals look like \{(code)\} \texttt{\prg_{return\_true}}:
\texttt{\else: \prg_{return\_false}}: \texttt{\fi}: so we optimize this special case by calling \texttt{\_\_prg_{generate\_conditional\_fast:nw}} \{(code)\}. This passes \texttt{\use_i:nn} instead of \texttt{\use_i_ii:nnn} to functions such as \texttt{\_\_prg_{generate\_p\_form:wNnnnnn}}.
Looping through the list of desired forms. First are six arguments and seventh is the
form. Use the form to call the correct type. If the form does not exist, the \texttt{\use:c}
construction results in \texttt{\relax}, and the error message is displayed (unless the form is
empty, to allow for \texttt{\{T, , F\}}), then \texttt{\use:none:nnnnnn} cleans up. Otherwise, the
error message is removed by the variant form.

How to generate the various forms. Those functions take the following arguments: 1:
junk, 2: \texttt{\cs_set:Npn} or similar, 3: \texttt{p} (for protected conditionals) or \texttt{e}, 4: function
name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by \texttt{\__prg_-_generate_conditional_fast:nw}), 8: \texttt{\use_i_ii:nmm} or \texttt{\use_i_i:n} (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present
after `\exp_end::` notice the construction of the different variants relies on this, and that the TF and F variants will be slightly faster than the T version. The p form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, `#7` has an extra `\if...`. To optimize a bit further we could replace `\exp_after:wN \use_ii:nnn` and similar by a single macro similar to `\__prg_p_true:w`. The drawback is that if the T or F arguments are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```latex
\cs_set_protected:Npn \__prg_generate_p_form:wNNnnnnN
\#1 \s__prg_stop #2#3#4#5#6#7#8
{     \if_meaning:w e #3
  \exp_after:wN \use_i:nn
  \else:
  \exp_after:wN \use_ii:nn
  \fi:
  \begin{verbatim}
    \#8
    { \exp_args:Nc #2 { #4 _p: #5 } #6 }
    { { #7 \exp_end: \c_true_bool \c_false_bool } }
    { #7 \__prg_p_true:w \fi: \c_false_bool }
  }
  \end{verbatim}

  \begin{verbatim}
    \__kernel_msg_error:nnx { kernel } { protected-predicate }
    \{ \token_to_str:c { #4 _p: #5 } \}
  \end{verbatim}

\end{verbatim}
\cs_set_protected:Npn \__prg_generate_T_form:wNNnnnnN
\#1 \s__prg_stop #2#3#4#5#6#7#8
{ #8
  \begin{verbatim}
    \exp_args:Nc #2 { #4 _p: #5 } #6 
    { { #7 \exp_end: \c_true_bool \c_false_bool } } 
    { #7 \__prg_p_true:w \fi: \c_false_bool }
  \end{verbatim}

\begin{verbatim}
  { \token_to_str:c { #4 _p: #5 } }
\end{verbatim}

\end{verbatim}
\cs_set_protected:Npn \__prg_generate_F_form:wNNnnnnN
\#1 \s__prg_stop #2#3#4#5#6#7#8
{ #8
  \begin{verbatim}
    \exp_args:Nc #2 { #4 _p: #5 } #6 
    { { #7 \exp_end: \c_true_bool \c_false_bool } } 
    { #7 \__prg_p_true:w \fi: \c_false_bool }
  \end{verbatim}

\begin{verbatim}
  { \token_to_str:c { #4 _p: #5 } }
\end{verbatim}

\end{verbatim}
\cs_set:Npn \__prg_generate_TF_form:wNNnnnnN
\#1 \s__prg_stop #2#3#4#5#6#7#8
{ #8
  \begin{verbatim}
    \exp_args:Nc #2 { #4 _p: #5 } #6 
    { { #7 \exp_end: \c_true_bool \c_false_bool } } 
    { #7 \__prg_p_true:w \fi: \c_false_bool }
  \end{verbatim}

\begin{verbatim}
  \token_to_str:c { #4 _p: #5 }
\end{verbatim}

\end{verbatim}
\end{verbatim}
\cs_set:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
```
(End definition for `\__prg_generate_p_form:wNNnnnnN` and others.)
The setting-equal functions. Split both functions and feed \{(name_1)\} \{(signature_1)\} \{boolean_1\} \{(name_2)\} \{(signature_2)\} \{boolean_2\} \{copying function\} \{conditions\} , \q__prg_recursion_tail , \q__prg_recursion_stop to a first auxiliary.

\cs_set_protected:Npn \prg_set_eq_conditional:NNn { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
\cs_set_protected:Npn \prg_new_eq_conditional:NNn { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
\cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4 { \use:x \exp_not:N \__prg_set_eq_conditional:nnNnnNNw \cs_split_function:N #2 \cs_split_function:N #3 \exp_not:N #1 \tl_to_str:n {#4} \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop } }

Split the function to be defined, and setup a manual clist loop over argument \#6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don’t know how to build conditionals, hence abort. Call the looping macro, with arguments \{(name_1)\} \{(signature_1)\} \{(name_2)\} \{(signature_2)\} \{copying function\} and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

\cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6 { \if_meaning:w \c_false_bool #3 \__kernel_msg_error:nnx { kernel } { missing-colon } { \token_to_str:c {#1} } \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w \fi: \if_meaning:w \c_false_bool #6 \__kernel_msg_error:nnx { kernel } { missing-colon } { \token_to_str:c {#4} } \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w \fi: \__prg_set_eq_conditional_loop:nnnnNw #1 \#2 \#4 \#5 }
\cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 , { \if_meaning:w \c_false_bool #3 \__kernel_msg_error:nxx }

(End definition for \prg_set_eq_conditional:NNn, \prg_new_eq_conditional:NNn, and \__prg_set_eq_conditional:NNNn. These functions are documented on page 105.)
\begin{verbatim}
{ kernel } { conditional-form-unknown }
{#6} { \token_to_str:c { #1 : #2 } }

\use_none:nnnnnn
\s__prg_stop
#5 {#1} {#2} {#3} {#4}
\__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5

\cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
{ #2 { #3 _p : #4 } { #5 _p : #6 } }
\cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
{ #2 { #3 : #4 TF } { #5 : #6 TF } }
\cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
{ #2 { #3 : #4 T } { #5 : #6 T } }
\cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
{ #2 { #3 : #4 F } { #5 : #6 F } }
\end{verbatim}

(End definition for \__prg_set_eq_conditional:nnNnnNNw and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using \if:w but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

\c_true_bool \c_false_bool

Here are the canonical boolean values.
\begin{verbatim}
\tex_chardef:D \c_true_bool = 1 ~
\tex_chardef:D \c_false_bool = 0 ~
\end{verbatim}

(End definition for \c_true_bool and \c_false_bool. These variables are documented on page 22.)

\section{Dissecting a control sequence}

\begin{verbatim}
\__cs_count_signature:N \__cs_count_signature:N {function}
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). The \langle number \rangle of tokens in the \langle signature \rangle is then left in the input stream. If there was no \langle signature \rangle then the result is the marker value −1.

\__cs_get_function_name:N * \__cs_get_function_name:N {function}
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). The \langle name \rangle is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

\__cs_get_function_signature:N * \__cs_get_function_signature:N {function}
Splits the \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). The \langle signature \rangle is then left in the input stream made up of tokens with category code 12 (other).
\end{verbatim}
Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the \escapechar is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from \token_to_str:N \a. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should not be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, \token_to_str:N \N yields the escape character itself and a space. The character codes are different, thus the \if:w test is false, and TEx reads \__cs_to_str:N after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial \tex_romannumeral:D. The second case is that the escape character is not printable. Then the \if:w test is unfinished after reading a the space from \token_to_str:N \N, and the auxiliary \__cs_to_str:w is expanded, feeding - as a second character for the test; the test is false, and TEx skips to \fi:, then performs \token_to_str:N, and stops the \tex_romannumeral:D with \c_zero_int. The last case is that the escape character is itself a space. In this case, the \if:w test is true, and the auxiliary \__cs_to_str:w comes into play, inserting -\int_value:w, which expands \c_zero_int to the character 0. The initial \tex_romannumeral:D then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of \tex_romannumeral:D. In all three cases, \cs_to_str:N takes two expansion steps to be fully expanded.

We implement the expansion scheme using \tex_romannumeral:D terminating it with \c_zero_int rather than using \exp:w and \exp_end:w as we normally do. The reason is that the code heavily depends on terminating the expansion with \c_zero_int so we make this dependency explicit.

If speed is a concern we could use \csstring in LuaTEx. For the empty csname that primitive gives an empty result while the current \cs_to_str:N gives incorrect results in all engines (this is impossible to fix without huge performance hit).
This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean \( (true) \) or \( (false) \) is returned with \( (true) \) for when there is a colon in the function and \( (false) \) if there is not.

We cannot use \: directly as it has the wrong category code so an \texttt{x}-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding \texttt{\cs_to_str:N} twice. If the function contained a colon, the auxiliary takes as \#1 the function name, delimited by the first colon, then the signature \#2, delimited by \texttt{\s__cs_mark}, then \texttt{\c_true_bool} as \#3, and \#4 cleans up until \texttt{\s__cs_stop}. Otherwise, the \#1 contains the function name and \texttt{\s__cs_mark} \texttt{\c_true_bool} \#2 is empty, \#3 is \texttt{\c_false_bool}, and \#4 cleans up. The second auxiliary trims the trailing \texttt{\s__cs_mark} from the function name if present (that is, if the original function had no colon).

5.9 Exist or free

A control sequence is said to \textit{exist} (to be used) if has an entry in the hash table and its meaning is different from the primitive \texttt{\relax} token. A control sequence is said to be \textit{free} (to be defined) if it does not already exist.

Two versions for checking existence. For the \texttt{N} form we firstly check for \texttt{\scan_stop} and then if it is in the hash table. There is no problem when inputting something like \texttt{\else} or \texttt{\fi} as \texttt{\relax} will only ever skip input in case the token tested against is \texttt{\scan_stop}.

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

This function is documented on page 17.
For the $c$ form we firstly check if it is in the hash table and then for $\text{scan_stop}$: so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

\begin{verbatim}
\prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF } {
    \if_cs_exist:w #1 \cs_end:
    \exp_after:wN \use_i:nn
    \else:
    \exp_after:wN \use_i:nn
    \fi:
    {
        \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_false:
        \else:
        \prg_return_true:
        \fi:
    }
}
\end{verbatim}

(End definition for $\text{cs_if_exist:NTF}$. This function is documented on page 22.)

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

The logical reversal of the above.

\begin{verbatim}
\prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF } {
    \if_meaning:w #1 \scan_stop:
    \prg_return_true:
    \else:
    \if_cs_exist:N #1
    \prg_return_false:
    \else:
    \prg_return_true:
    \fi:
} \fi:
\end{verbatim}

\begin{verbatim}
\prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF } {
    \if_cs_exist:w #1 \cs_end:
    \exp_after:wN \use_i:nn
    \else:
    \exp_after:wN \use_i:nn
    \fi:
    {
        \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_true:
        \else:
        \prg_return_false:
    }
}
\end{verbatim}
The \cs_if_exist_use:... functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the \c variants, we are careful not to put the control sequence in the hash table if it does not exist. In \LaTeX{} we could use the \lastnamedcs primitive.

\begin{verbatim}
\cs_set:Npn \cs_if_exist_use:NTF #1#2
\{ \cs_if_exist:NTF #1 { #1 #2 } \}
\cs_set:Npn \cs_if_exist_use:NF #1
\{ \cs_if_exist:NTF #1 { #1 } \}
\cs_set:Npn \cs_if_exist_use:NT #1 #2
\{ \cs_if_exist:NTF #1 { #1 #2 } { } \}
\cs_set:Npn \cs_if_exist_use:N #1
\{ \cs_if_exist:NTF #1 { #1 } { } \}
\cs_set:Npn \cs_if_exist_use:cTF #1#2
\{ \cs_if_exist:cTF {#1} { \use:c {#1} #2 } \}
\cs_set:Npn \cs_if_exist_use:cF #1
\{ \cs_if_exist:cTF {#1} { \use:c {#1} } \}
\cs_set:Npn \cs_if_exist_use:cT #1#2
\{ \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } \}
\cs_set:Npn \cs_if_exist_use:c #1
\{ \cs_if_exist:cTF {#1} { \use:c {#1} } { } \}
\end{verbatim}

(\textit{End definition for} \texttt{\cs_if_free:NTF. This function is documented on page 22.})

5.10 Preliminaries for new functions

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

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

\begin{verbatim}
\__kernel_msg_error:nnxx
\__kernel_msg_error:nnx
\__kernel_msg_error:nn
\end{verbatim}

If an internal error occurs before \LaTeX{} has loaded \l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the \newlinechar is needed, to turn \hfill into a proper line break in plain \TeX.

\begin{verbatim}
\cs_set_protected:Npn \__kernel_msg_error:nnxx #1#2#3#4
\{ \tex_newlinechar:D = \'\^^J \scan_stop:
\tex_errmessage:D
\{ !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!lassen!!!!!!!!!!!!!!! \^^J
Argh,-internal-LaTeX3-error! \"^J "^J
Module - #1, - message-name="#2": \"^J "^J
Arguments-'#3'-and-'#4' \"^J "^J
\}
\end{verbatim}

(\textit{End definition for} \texttt{\cs_if_exist_use:NTF. This function is documented on page 16.})
This is one for The LaTeX Project: bailing out

\end{D}

\cs_set_protected:Npn \__kernel_msg_error:nnx #1#2#3
{ \__kernel_msg_error:nnxx {#1} {#2} {#3} { } }
\cs_set_protected:Npn \__kernel_msg_error:nn #1#2
{ \__kernel_msg_error:nnxx {#1} {#2} { } { } }

(End definition for \__kernel_msg_error:nnxx, \__kernel_msg_error:nnx, and \__kernel_msg_error:nn.)

\msg_line_context: Another one from l3msg which will be altered later.
\cs_set:Npn \msg_line_context:
{ on-line \tex_the:D \tex_inputlineno:D }

(End definition for \msg_line_context:. This function is documented on page 151.)

\iow_log:x \iow_term:x We define a routine to write only to the log file. And a similar one for writing to both
the log file and the terminal. These will be redefined later by l3io.
\cs_set_protected:Npn \iow_log:x
{ \tex_immediate:D \tex_write:D -1 }
\cs_set_protected:Npn \iow_term:x
{ \tex_immediate:D \tex_write:D 16 }

(End definition for \iow_log:n. This function is documented on page 160.)

\__kernel_chk_if_free_cs:N \__kernel_chk_if_free_cs:c
This command is called by \cs_new_nopar:Npn and \cs_new_eq:NN etc. to make sure
that the argument sequence is not already in use. If it is, an error is signalled. It checks
if ⟨csname⟩ is undefined or \scan_stop:. Otherwise an error message is issued. We have
to make sure we don’t put the argument into the conditional processing since it may be
an \if... type function!
\cs_set_protected:Npn \__kernel_chk_if_free_cs:N #1
{ \cs_if_free:NF #1
{ \__kernel_chk_if_free_cs:N #2 #1
}
}
\cs_set_protected:Npn \__kernel_chk_if_free_cs:c
{ \exp_args:Nc \__kernel_chk_if_free_cs:N }

(End definition for \__kernel_chk_if_free_cs:N.)

5.11 Defining new functions
\__cs_tmp:w
Function which check that the control sequence is free before defining it.
\cs_set:Npn \__cs_tmp:w #1#2
{ \cs_set_protected:Npn \__cs_tmp:w #1 #2 #1 #1
}

337
Like \cs_set_nopar:Npn and \cs_new_nopar:Npn, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module). Global versions are also provided.

\cs_set:cpn⟨string⟩⟨rep-text⟩ turns ⟨string⟩ into a csname and then assigns ⟨rep-text⟩ to it by using \cs_set_nopar:Npn. This means that there might be a parameter string between the two arguments.

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

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

(End definition for \cs_new_nopar:Npn and others. These functions are documented on page 11.)

(End definition for \cs_set_nopar:Npn. This function is documented on page 11.)

(End definition for \cs_set_protected_nopar:Npn. This function is documented on page 12.)
Variants of the \cs_set_protected:Npn versions which make a csname out of the first arguments. We may also do this globally.

\__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
\__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
\__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
\__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
\__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

(End definition for \cs_set_protected:Npn. This function is documented on page 11.)

5.12 Copying definitions

These macros allow us to copy the definition of a control sequence to another control sequence. The = sign allows us to define funny char tokens like \texttt{=} itself or \texttt{/uni2423} with this function. For the definition of \c_space_char{} to work we need the \texttt{~} after the \texttt{=}.

\cs_set_eq:NN \cs_set_eq:cN \cs_set_eq:Nc \cs_set_eq:cc
\cs_gset_eq:NN \cs_gset_eq:cN \cs_gset_eq:Nc \cs_gset_eq:cc
\cs_new_eq:NN \cs_new_eq:cN \cs_new_eq:Nc \cs_new_eq:cc

(End definition for \cs_set_eq:NN, \cs_gset_eq:NN, and \cs_new_eq:NN. These functions are documented on page 15.)

5.13 Undefining functions

\cs_undefine:N \cs_undefine:c

The following function is used to free the main memory from the definition of some function that isn’t in use any longer. The c variant is careful not to add the control sequence to the hash table if it isn’t there yet, and it also avoids nesting \TeX{} conditionals in case \texttt{#1} is unbalanced in this matter.
5.14 Generating parameter text from argument count

\textsc{\LaTeX}3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text \{#1...#n\}, where \texttt{n} is the result of evaluating the second argument (as described in \texttt{\int_eval:n}). If the second argument gives a result outside the range [0,9], the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

\begin{verbatim}
\cs_set_protected:Npn \__kernel_cs_parm_from_arg_count:nnF #1#2
\{\exp_args:Nx \__cs_parm_from_arg_count_test:nnF {#1}
\if_case:w \int_eval:n {#2}
\or: { ##1 }
\or: { ##1##2 }
\or: { ##1##2##3 }
\or: { ##1##2##3##4 }
\or: { ##1##2##3##4##5 }
\or: { ##1##2##3##4##5##6 }
\or: { ##1##2##3##4##5##6##7 }
\or: { ##1##2##3##4##5##6##7##8 }
\or: { ##1##2##3##4##5##6##7##8##9 }
\else: { \c_false_bool }
\fi:
\}
\cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
{\if_meaning:w \c_false_bool #1 \exp_after:wN \use_ii:nn \else: \exp_after:wN \use_i:nn \fi:
{ #2 {#1} }
}
\end{verbatim}

(End definition for \__kernel_cs_parm_from_arg_count:nnF and \__cs_parm_from_arg_count_test:nnF.)
5.15 Defining functions from a given number of arguments

Counting the number of tokens in the signature, i.e., the number of arguments the function should take. Since this is not used in any time-critical function, we simply use $\texttt{tl\_count:n}$ if there is a signature, otherwise $-1$ arguments to signal an error. We need a variant form right away:

```latex
\cs_new:Npn \__cs_count_signature:N #1
\{ \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } \}
```

```latex
\cs_new:Npn \__cs_count_signature:n #1
\{ \int_eval:n { \__cs_count_signature:nnN #1 } \}
```

```latex
\cs_new:Npn \__cs_count_signature:nnN #1#2#3
\{ \if_meaning:w \c_true_bool #3 \tl_count:n {#2} \else: -1 \fi: \}
```

```latex
\cs_new:Npn \__cs_count_signature:c
\{ \exp_args:Nc \__cs_count_signature:N \}
```

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TEX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```latex
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
\{ \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3} \}
```

```latex
\cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
\{ \exp_args:Nc \cs_generate_from_arg_count:NNnn \}
```

```latex
\cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
\{ \exp_args:NNc \cs_generate_from_arg_count:NNnn \}
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```latex
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnnn #1#2#3#4#5
\cs_new_protected:Npn \cs_generate_from_arg_count:chnn
\cs_new_protected:Npn \cs_generate_from_arg_count:Ncnnn
\{ \exp_args:Nc \cs_generate_from_arg_count:NNn \}
```

```latex
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnnn #1#2#3#4#5
\cs_new_protected:Npn \cs_generate_from_arg_count:chnnn
\cs_new_protected:Npn \cs_generate_from_arg_count:Ncnnn
\{ \exp_args:Nc \cs_generate_from_arg_count:NNnn \}
```

(End definition for $\__cs_count_signature:N$, $\__cs_count_signature:n$, and $\__cs_count_signature:nnN$.)

(End definition for $\cs_generate_from_arg_count:NNnn$, $\cs_generate_from_arg_count:chnn$, and $\cs_generate_from_arg_count:Ncnn$.)

(End definition for $\cs_generate_from_arg_count:NNnnn$, $\cs_generate_from_arg_count:chnnn$, and $\cs_generate_from_arg_count:Ncnnn$. This function is documented on page 14.)
5.16 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, \texttt{\cs_set:Nn \foo_bar:nn \{#1,#2\}}.

We want to define \texttt{\cs_set:Nn} as

\begin{verbatim}
\cs_set:Nn \cs_set:Nx \cs_set_nopar:Nn \cs_set_protected:Nn \cs_set_protected_nopar:Nn
\cs_gset:Nn \cs_gset:Nx \cs_gset_nopar:Nn \cs_gset_protected:Nn \cs_gset_protected_nopar:Nn
\cs_new:Nn \cs_new:Nx \cs_new_nopar:Nn \cs_new_protected:Nn \cs_new_protected_nopar:Nn
\cs_gset_protected:Nn \cs_gset_protected_nopar:Nn \cs_gset_nopar:Nn \cs_gset:Nn
\cs_set_protected:Npn \cs_set:Nn #1#2
{ \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn { \@@_count_signature:N #1 } {#2} }
\end{verbatim}

In short, to define \texttt{\cs_set:Nn} we need just use \texttt{\cs_set:Npn}, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

\begin{verbatim}
\cs_set:Npn \__cs_tmp:w #1#2#3
{ \cs_new_protected:cpx { cs_ #1 : #2 } \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
{ \use:x \__cs_generate_from_signature:nnNNNn \cs_split_function:N #2
\end{verbatim}


\begin{verbatim}
\cs_set:Npn \__cs_generate_from_signature:NNn #1#2
{ \bool_if:NTF #3
\str_if_eq:eeF { }
\tl_map_function:nN \{#2\} \__cs_generate_from_signature:n
{ \_kernel_msg_error:nmx \{ kernel \} \{ non-base-function \}
\token_to_str:N \#5
}\cs_generate_from_arg_count:NNnn \#5 \#6
\{ \_kernel_msg_error:nmx \{ kernel \} \{ missing-colon \}
\token_to_str:N \#5\}
\}
\}
\cs_new:Npn \__cs_generate_from_signature:n #1
{ \_cs_generate_from_arg_count:NNnn \#1 #2}
\end{verbatim}

342
Then we define the 24 variants beginning with \texttt{N}.

The 24 \texttt{c} variants simply use \texttt{\exp_args:Nc}.

\begin{verbatim}
\end{verbatim}

(End definition for \texttt{\cs_set:Nn} and others. These functions are documented on page 13.)
5.17 Checking control sequence equality

Check if two control sequences are identical.

\begin{verbatim}
\cs_if_eq_p:NN \prg_new_conditional:Nnn \cs_if_eq:NN #1 #2 { p, T, F, TF }
\cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:cNF { \exp_args:NNc \cs_if_eq:NNF }
\cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:ccF { \exp_args:NNcc \cs_if_eq:NNF }
\end{verbatim}

(End definition for \cs_set:Nn. This function is documented on page 13.)

5.18 Diagnostic functions

\begin{verbatim}
\__kernel_chk_defined:NT \__kernel_msg_error:nnx { kernel } { variable-not-defined } { \token_to_str:N #1 }
\end{verbatim}

(End definition for \__kernel_chk_defined:NT. This function is documented on page 22.)
Simply using the `\show` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `\langle variable \rangle = \langle value \rangle`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```latex
\cs_new_protected:Npn \__kernel_register_show:N { \__kernel_register_show_aux:NN \tl_show:n }
\cs_new_protected:Npn \__kernel_register_show:c { \exp_args:Nc \__kernel_register_show:N }
\cs_new_protected:Npn \__kernel_register_log:N { \__kernel_register_show_aux:NN \tl_log:n }
\cs_new_protected:Npn \__kernel_register_log:c { \exp_args:Nc \__kernel_register_log:N }
\cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2 { \__kernel_chk_defined:NT #2 { \exp_args:No \__kernel_register_show_aux:nNN { \tex_the:D #2 } #2 #1 }
\cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3 { \exp_args:No #3 { \token_to_str:N #2 = \cs_meaning:N #2 } }
```

(End definition for `\__kernel_register_show:N` and others.)

Some control sequences have a very long name or meaning. Thus, simply using \TeX’s primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by `x`-expansion. The `\cs_show:N` and `\cs_log:N` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```latex
\cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
\cs_new_protected:Npn \cs_show:c { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
\cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
\cs_new_protected:Npn \cs_log:c { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
\cs_new_protected:Npn \__kernel_show:NN #1#2 { \group_begin:
\int_set:Nn \tex_escapechar:D { ‘\ } \exp_args:NNx \group_end:
\token_to_str:N #2 = \cs_meaning:N #2 }
```

(End definition for `\cs_show:N`, `\cs_log:N`, and `\__kernel_show:NN`. These functions are documented on page 16.)
5.19 Decomposing a macro definition

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn’t a macro, the token \scan_stop: is returned instead.

```latex
\use:x
\{ \exp_not:n { \cs_new:Npn \__kernel_prefix_arg_replacement:wN #1 } \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 } \}
\cs_new:Npn \cs_prefix_spec:N #1
\{ \token_if_macro:NTF #1
\{ \exp_after:wN \__kernel_prefix_arg_replacement:wN \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn \}
\{ \scan_stop: \}
\cs_new:Npn \cs_argument_spec:N #1
\{ \token_if_macro:NTF #1
\{ \exp_after:wN \__kernel_prefix_arg_replacement:wN \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn \}
\{ \scan_stop: \}
\cs_new:Npn \cs_replacement_spec:N #1
\{ \token_if_macro:NTF #1
\{ \exp_after:wN \__kernel_prefix_arg_replacement:wN \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn \}
\{ \scan_stop: \}
\cs_new:Npn \prg_do_nothing: { }
\{ \scan_stop: \}
```

(End definition for \cs_prefix_spec:N and others. These functions are documented on page 18.)

5.20 Doing nothing functions

\prg_do_nothing: This does not fit anywhere else!

\cs_new:Npn \prg_do_nothing: { }

(End definition for \prg_do_nothing:. This function is documented on page 9.)
5.21 Breaking out of mapping functions

In inline mappings, the nesting level must be reset at the end of the mapping, even when
the user decides to break out. This is done by putting the code that must be performed
as an argument of \_\_\_prg_break_point:Nn. The breaking functions are then defined to
jump to that point and perform the argument of \_\_\_prg_break_point:Nn, before the
user’s code (if any). There is a check that we close the correct loop, otherwise we continue
breaking.

\cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
\cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
\#5
\if_meaning:w #1 #4
\exp_after:wN \use_iii:nnn
\fi:
\prg_map_break:Nn #1 {#2}

(End definition for \prg_break_point:Nn and \prg_map_break:Nn. These functions are documented on
page 112.)

\prg_break_point:
\prg_break:
\prg_break:n
Very simple analogues of \prg_break_point:Nn and \prg_map_break:Nn, for use in fast
short-term recursions which are not mappings, do not need to support nesting, and in
which nothing has to be done at the end of the loop.

\cs_new_protected:Npn \mode_leave_vertical: {
\if_mode_vertical:
\exp_after:wN \tex_indent:D
\fi:
}

(End definition for \prg_break_point:, \prg_break:, and \prg_break:n. These functions are docu-
mented on page 113.)

5.22 Starting a paragraph

\mode_leave_vertical: The approach here is different to that used by \TeX{}2\epsilon{} or plain \TeX{}, which unbox a
void box to force horizontal mode. That inserts the \everypar{} tokens before the re-
inserted unboxing tokens. The approach here uses either the \quitvmode primitive or the
equivalent protected macro. In vertical mode, the \indent primitive is inserted: this
will switch to horizontal mode and insert \everypar tokens and nothing else. Unlike the
\TeX{}2\epsilon{} version, the availability of \TeX{} means using a mode test can be done at for
example the start of an \halign.

\cs_new_protected:Npn \mode_leave_vertical: {
\if_mode_vertical:
\exp_after:wN \tex_indent:D
\fi:
}

(End definition for \mode_leave_vertical:. This function is documented on page 24.)
6 l3expan implementation

\l__exp_internal_tl The \exp_ module has its private variable to temporarily store the result of x-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End definition for \l__exp_internal_tl.)

\exp_after:wN \exp_not:N \exp_not:n These are defined in l3basics, as they are needed “early”. This is just a reminder of that fact!

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 33.)

6.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless x is used. (Any version of x is going to have to use one of the \TeX3 names for \cs_set:Npx at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 6.8. In section 6.2 some common cases are coded by a more direct method for efficiency, typically using calls to \exp_after:wN.

\l__exp_internal_tl This scratch token list variable is defined in l3basics.

(End definition for \l__exp_internal_tl.)

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

An argument manipulator \::(Z) always has signature #1:::#2#3 where #1 holds the remaining argument manipulations to be performed, ::#2 serves as an end marker for the list of manipulations, #2 is the carried over result of the previous expansion steps and #3 is the argument about to be processed. One exception to this rule is ::p, which has to grab an argument delimited by a left brace.

\__exp_arg_next:nnn \__exp_arg_next:Nnn #1 is the result of an expansion step, #2 is the remaining argument manipulations and #3 is the current result of the expansion chain. This auxiliary function moves #1 back after #3 in the input stream and checks if any expansion is left to be done by calling #2. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the c of the final argument manipulation variants does not require a set of braces.

\cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 :: { #3 {#1} } }
\cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 :: { #3 #1 } }

(End definition for \__exp_arg_next:nnn and \__exp_arg_next:Nnn.)

::: The end marker is just another name for the identity function.

\cs_new:Npn \::: #1 {#1}

(End definition for :::. This function is documented on page 37.)
This function is used to skip an argument that doesn’t need to be expanded.
\cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 \{#3\} } }
(End definition for \::n. This function is documented on page 37.)

This function is used to skip an argument that consists of a single token and doesn’t need to be expanded.
\cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
(End definition for \::N. This function is documented on page 37.)

This function is used to skip an argument that is delimited by a left brace and doesn’t need to be expanded. It is not wrapped in braces in the result.
\cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
(End definition for \::p. This function is documented on page 37.)

This function is used to skip an argument that is turned into a control sequence without expansion.
\cs_new:Npn \::c #1 \::: #2#3
\cs_new:Npn \::c #1 \::: #2#3 { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
(End definition for \::c. This function is documented on page 37.)

This function is used to expand an argument once.
\cs_new:Npn \::o #1 \::: #2#3
\cs_new:Npn \::o #1 \::: #2#3 { \exp_args:Ne \__exp_arg_next:nnn {#3} {#1} {#2} }
(End definition for \::o. This function is documented on page 37.)

With the \textit{expanded} primitive available, just expand. Otherwise defer to \texttt{exp_args:Ne} implemented later.
\cs_if_exist:NTF \tex_expanded:D
\cs_new:Npn \::e #1 \::: #2#3
\cs_new:Npn \::e #1 \::: #2#3 { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
\cs_new:Npn \::e #1 \::: #2#3
\cs_new:Npn \::e #1 \::: #2#3 { \exp_args:Ne \__exp_arg_next:nnn {#3} {#1} {#2} }
(End definition for \::e. This function is documented on page 37.)

This function is used to expand a token list until the first unexpandable token is found. This is achieved through \texttt{exp:w exp_end_continue_f:w} that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce \texttt{exp_stop_f:} to mark such an end-of-expansion marker. For example, \texttt{f-expanding cs_set_eq:Nc \aaa \{ b \l_tmpa_tl b \} where \l_tmpa_tl contains the characters lur gives \texttt{tex_let:D \aaa = \blurb} which then turns out to start with the non-expandable token \texttt{tex_let:D}. Since the expansion of \texttt{exp:w exp_end_continue_f:w} is empty, we wind up with a fully expanded list, only \LaTeX{} has not tried to execute any of
the non-expandable tokens. This is what differentiates this function from the \texttt{x} argument type.

\begin{verbatim}
\cs_new:Npn \::f #1 \::: #2#3
\exp_after:wN \__exp_arg_next:nnn
\exp_after:wN \{ \exp:w \exp_end_continue_f:w #3 \}
\{#1\} \{#2\}
\}
\use:nn \{ \cs_new_eq:NN \exp_stop_f: \}
\end{verbatim}

(End definition for \texttt{\::f} and \texttt{\exp_stop_f}. These functions are documented on page 37.)

\texttt{\::x} This function is used to expand an argument fully. We build in the expansion of \_\_\_\_\_\_\_exp_arg_next:nnn.

\begin{verbatim}
\cs_new_protected:Npn \::x #1 \::: #2#3
\cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
\l__exp_internal_tl
\}
\end{verbatim}

(End definition for \texttt{\::x}. This function is documented on page 37.)

\texttt{\::v} These functions return the value of a register, i.e., one of \texttt{tl}, \texttt{clist}, \texttt{int}, \texttt{skip}, \texttt{dim}, \texttt{muskip}, or built-in \TeX{} register. The \texttt{V} version expects a single token whereas \texttt{v} like \texttt{c} creates a csname from its argument given in braces and then evaluates it as if it was a \texttt{V}. The \texttt{\exp:w} sets off an expansion similar to an \texttt{f}-type expansion, which we terminate using \texttt{\exp_end:}. The argument is returned in braces.

\begin{verbatim}
\cs_new:Npn \::V #1 \::: #2#3
\exp_after:wN \__exp_arg_next:nnn
\exp_after:wN \{ \exp:w \__exp_eval_register:N #3 \}
\{#1\} \{#2\}
\}
\cs_new:Npn \::v #1 \::: #2#3
\exp_after:wN \__exp_arg_next:nnn
\exp_after:wN \{ \exp:w \__exp_eval_register:c {#3} \}
\{#1\} \{#2\}
\}
\end{verbatim}

(End definition for \texttt{\::v} and \texttt{\::V}. These functions are documented on page 37.)

\texttt{\__exp_eval_register:N} \texttt{\__exp_eval_register:c} \texttt{\__exp_eval_error_msg:w} This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in \TeX{} register such as \texttt{\count}. For the \TeX{} registers we have to utilize a \texttt{\the} whereas for the macros we merely have to expand them once. The trick is to find out when to use \texttt{\the} and when not to. What we want here is to find out whether the token expands to something else when hit with \texttt{\exp_after:wN}. The technique is to compare the meaning of the token in question when it has been prefixed with \texttt{\exp_not:N} and the token itself. If it is a macro, the prefixed \texttt{\exp_not:N} temporarily turns it into the primitive \texttt{\scan_stop:}.

\begin{verbatim}
\cs_new:Npn \__exp_eval_register:N #1
\exp_after:wN \if_meaning:w \exp_not:N #1 #1
\exp_after:wN \__exp_arg_next:nnn
\exp_after:wN \{ \exp:w \__exp_eval_register:N #3 \}
\{#1\} \{#2\}
\}
\cs_new:Npn \__exp_eval_register:c
\exp_after:wN \if_meaning:w \exp_not:N \l__exp_internal_tl
\exp_after:wN \{ \exp:w \__exp_eval_register:c {#3} \}
\{#1\} \{#2\}
\}
\end{verbatim}

(End definition for \texttt{\__exp_eval_register:N} and \texttt{\__exp_eval_register:c}. These functions are documented on page 37.)
If the token was not a macro it may be a malformed variable from a \cexp\ expansion in which case it is equal to the primitive \scan_stop:. In that case we throw an error. We could let \TeX do it for us but that would result in the rather obscure

\[ \text{! You can’t use ‘relax’ after ‘the.} \]

which while quite true doesn’t give many hints as to what actually went wrong. We provide something more sensible.

\[
\begin{align*}
\text{\if_meaning:w } & \text{\scan_stop: } #1 \\
\_\_\_\_\text{exp_eval_error_msg:w} \\
\text{\fi:}
\end{align*}
\]

The next bit requires some explanation. The function must be initiated by \exp:w and we want to terminate this expansion chain by inserting the \exp_end: token. However, we have to expand the register #1 before we do that. If it is a \TeX\ register, we need to execute the sequence \exp_after:wN \exp_end: \tex_the:D #1 and if it is a macro we need to execute \exp_after:wN \exp_end: #1. We therefore issue the longer of the two sequences and if the register is a macro, we remove the \tex_the:D.

\[
\begin{align*}
\text{\else:} \\
\exp_after:wN \use_i_ii:nnn \\
\text{\fi:} \\
\exp_after:wN \exp_end: \tex_the:D #1 \\
\}
\end{align*}
\]

\cs_new:Npn \_\_\_\_\text{exp_eval_register:c} #1
\{
\exp_after:wN \_\_\_\_\text{exp_eval_register:N} \cs:w #1 \cs_end: \}

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

\[ \text{! Undefined control sequence.} \]
\[ \langle \text{argument} \rangle \ \text{LaTeX3 error:} \]
\[ \text{Erroneous variable used!} \]
\[ \text{1.55 \text{l_tilent:}Nv \text{l_tmpa_tl }\{\text{undefined_tl}\} \]
\[ \]
\( \text{End definition for } \_\_\_\_\text{exp_eval_register:N} \text{ and } \_\_\_\_\text{exp_eval_error_msg:w}. \)

6.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

\[
\begin{align*}
\text{\exp_args:Nc} \\
\text{\exp_args:cc}
\end{align*}
\]

\( \text{End definition for } \exp_args:Nc \text{ and } \exp_args:cc. \) \( \text{These functions are documented on page 29.} \)
Here are the functions that turn their argument into csnames but are expandable.

\begin{verbatim}
\exp_args:NNo \exp_args:NNNo \exp_args:Nccc
{ \exp_after:wN \exp_after:wN \cs:w \exp_after:wN \cs_end: 
\end{verbatim}
(End definition for \exp_args:NNo, \exp_args:NNNo, and \exp_args:Nccc. These functions are documented on page 30.)

\exp_args:Nf \exp_args:Nv \exp_args:NV
\begin{verbatim}
\exp_after:wN \exp_after:wN \exp:w \exp_end_continue_f:w \exp:w \__exp_eval_register:c \exp:w \__exp_eval_register:N
\end{verbatim}
(End definition for \exp_args:Nf, \exp_args:Nv, and \exp_args:NV. These functions are documented on page 30.)

\exp_args:Nf
\begin{verbatim}
\cs_if_exist:NTF \tex_expanded:D
{ \exp_after:wN \exp_after:wN \cs:w \exp_after:wN \cs_end: 
\end{verbatim}
(End definition for \exp_args:Nf. This function is documented on page 30.)

\exp_args:Nf
\exp_args:Nv
\begin{verbatim}
{ \exp_after:wN \exp_after:wN \exp:w \exp_end_continue_f:w
\end{verbatim}
\exp_args:Nf
\exp_args:Nv
\begin{verbatim}
{ \exp_after:wN \exp_after:wN \exp:w \exp_end_continue_f:w
\end{verbatim}
\exp_args:Nf
\exp_args:Nv
\begin{verbatim}
{ \exp_after:wN \exp_after:wN \exp:w \exp_end_continue_f:w
\end{verbatim}
Some more hand-tuned function with three arguments. If we forced that an o argument always has braces, we could implement \exp_args:Nco with less tokens and only two arguments.

\cs_new:Npn \exp_args:NNV #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
\end{verbatim}
\cs_new:Npn \exp_args:NNv #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
\end{verbatim}
\cs_if_exist:NTF \tex_expanded:D
\begin{verbatim}
    \cs_new:Npn \exp_args:NNe #1#2#3
    \begin{verbatim}
        \exp_after:wN #1 \exp_after:wN #2 \tex_expanded:D { {#3} }
    \end{verbatim}
\end{verbatim}
\cs_new:Npn \exp_args:NNf #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
\end{verbatim}
\cs_new:Npn \exp_args:Nco #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
\end{verbatim}
\cs_new:Npn \exp_args:NcV #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
\end{verbatim}
\cs_new:Npn \exp_args:Ncf #1#2#3
\begin{verbatim}
    \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
\end{verbatim}
(End definition for \exp_args:NNV and others. These functions are documented on page 31.)

\exp_args:NNNV
\exp_args:NcNc
\exp_args:NcNo
\exp_args:Ncco

A few more that we can hand-tune.

\exp_args:Nx

(End definition for \exp_args:NNNV and others. These functions are documented on page 32.)

\exp_args:Nx

(End definition for \exp_args:Nx. This function is documented on page 31.)

354
6.3 Last-unbraced versions

There are a few places where the last argument needs to be available unbraced. First some helper macros.

\begin{verbatim}
\cs_new:Npn \__exp_arg_last_unbraced:nn #1 #2 { #2 #1 }
\cs_new:Npn \::o_unbraced ::: #1 #2 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
\cs_new:Npn \::V_unbraced ::: #1 #2 {
\exp_after:wN \__exp_arg_last_unbraced:nn
\exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1} }
\cs_new:Npn \::v_unbraced ::: #1 #2 {
\exp_after:wN \__exp_arg_last_unbraced:nn
\exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1} }
\cs_if_exist:NTF \tex_expanded:D {
\cs_new:Npn \::e_unbraced ::: #1 #2 { \tex_expanded:D { \exp_not:n {#1} #2 } }
} {
\cs_new:Npn \::e_unbraced ::: #1 #2 { \exp:w \__exp_e:nn {#2} {#1} }
}
\cs_new:Npn \::f_unbraced ::: #1 #2 {
\exp_after:wN \__exp_arg_last_unbraced:nn
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1} }
\cs_new_protected:Npn \::x_unbraced ::: #1 #2 {
\cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
\l__exp_internal_tl }
\end{verbatim}

(End definition for \_\_exp_arg_last_unbraced:nn and others. These functions are documented on page 37.)

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

\begin{verbatim}
\exp_last_unbraced:No \exp_last_unbraced:NV \exp_last_unbraced:Ne \exp_last_unbraced:Nf \exp_last_unbraced:NNo \exp_last_unbraced:NNV \exp_last_unbraced:NfNo \exp_last_unbraced:NNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo \exp_last_unbraced:NfNo
\end{verbatim}

355
If \#2 is a single token then this can be implemented as

\begin{verbatim}
cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
  { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
\end{verbatim}

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that \#2 can be multiple tokens.

\begin{verbatim}
cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
  { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
  { \exp_after:wN #3 #2 #1 }
\end{verbatim}

(End definition for \exp_last_two_unbraced:Noo and \__exp_last_two_unbraced:noN. This function is documented on page 33.)

6.4 Preventing expansion

At the kernel level, we need the primitive behaviour to allow expansion before the brace group.

\begin{verbatim}
cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D
\end{verbatim}

(End definition for \__kernel_exp_not:w.)

\begin{verbatim}
cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
cs_if_exist:NTF \tex_expanded:D
  { \cs_new:Npn \exp_not:e #1
    { \__kernel_exp_not:w \exp_after:wN \exp:w \exp_end_continue_f:w #1 }
  }
cs_new:Npn \exp_not:f #1
  { \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN \exp:w \_\exp_eval_register:N #1 }
cs_new:Npn \exp_not:V #1
  { \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN \exp:w \__exp_eval_register:N #1 }
cs_new:Npn \exp_not:v #1
  { \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN \exp:w \__exp_eval_register:c {#1} }
\end{verbatim}

(End definition for \exp_not:c and others. These functions are documented on page 34.)
6.5 Controlled expansion

To trigger a sequence of “arbitrarily” many expansions we need a method to invoke \TeX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in \TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that \texttt{\the\romannumeral:D} expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: \texttt{\exp:w} is set equal to \texttt{\the\romannumeral:D} in \texttt{l3basics}. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops \texttt{\the\romannumeral:D}’s search for a number. Again, the definition of \texttt{\exp_end:} as the integer constant zero is in \texttt{l3basics}.

(Note that according to our specification all tokens we expand initiated by \texttt{\exp:w} are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an \texttt{f}-type expansion we provide the alphabetic constant ‘\texttt{^^@}’ that also represents 0 but this time \TeX’s syntax for a ⟨number⟩ continues searching for an optional space (and it continues expansion doing that) — see \TeX\book page 269 for details.

\begin{lstlisting}[language=TeX]
\ifcsexist:N ^^@
  \cs_new:Npn ^^@ { \__kernel_msg_expandable_error:nn { kernel } { bad-exp-end-f } }
\fi:
\end{lstlisting}

The same but grabbing an argument to remove spaces and braces.

\begin{lstlisting}[language=TeX]
\cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
\group_end:
\end{lstlisting}

(End definition for \texttt{\exp:w} and others. These functions are documented on page 96.)

6.6 Emulating \texttt{e}-type expansion

When the \texttt{\the\exp:D} primitive is available it is used to implement \texttt{e}-type expansion; otherwise we emulate it.

\begin{lstlisting}[language=TeX]
\cs_if_exist:NF \tex_expanded:D
  { \__exp_e:nn \__exp_e_end:nn
    \__exp_e:nn
    \__exp_e_end:nn

  Repeatedly expand tokens, keeping track of fully-expanded tokens in the second argument to \texttt{\__exp_e:nn}; this function eventually calls \texttt{\__exp_e_end:nn} to leave \texttt{\exp_end:} in the input stream, followed by the result of the expansion. There are many special cases:

\end{lstlisting}
spaces, brace groups, \noexpand, \unexpanded, \the, \primitive. While we use brace tricks \if_false: { \fi:, the expansion of this function is always triggered by \exp:w so brace balance is eventually restored after that is hit with a single step of expansion. Otherwise we could not nest e-type expansions within each other.

\cs_new:Npn \__exp_e:nn #1
\{ \if_false: { \fi:
 \tl_if_head_is_N_type:NTF {#1}
 \{ \__exp_e:N \}
 \}
 \tl_if_head_is_group:NTF {#1}
 \{ \__exp_e_group:n \}
 \}
 \tl_if_empty:NTF {#1}
 \{ \exp_after:wN \__exp_e_end:nn \}
 \exp_after:wN \__exp_e_space:nn
\exp_after:wN { \if_false: } \fi:
\}
\cs_new:Npn \__exp_e_end:nn #1#2 { \exp_end: #2 }

(End definition for \__exp_e:nn and \__exp_e_end:nn.)

\__exp_e_space:nn For an explicit space character, remove it by f-expansion and put it in the (future) output.

\cs_new:Npn \__exp_e_space:nn #1#2
\{ \exp_args:Nf \__exp_e:nn {#1} { #2 ~ } \}

(End definition for \__exp_e_space:nn.)

\__exp_e_group:n For a group, expand its contents, wrap it in two pairs of braces, and call \__exp_e_put:nn. This function places the first item (the double-brace wrapped result) into the output. Importantly, \tl_head:n works even if the input contains quarks.

\cs_new:Npn \__exp_e_group:n #1
\{ \exp_after:wN \__exp_e_put:nn
 \exp_after:wN \{ \exp_after:wN \{ \exp_after:wN \{ \exp:w \if_false: } \fi: \__exp_e:nn {#1} { } \}
\}
\cs_new:Npn \__exp_e_put:nn #1
\{ \exp_args:No \exp_args:No \__exp_e_put:nnn
 \{ \tl_head:n {#1} \} {#1} \}
\cs_new:Npn \__exp_e_put:nnn #1#2#3
\{ \exp_args:No \__exp_e:nn \{ \use_none:n #2 \} {#3 #1} \}

(End definition for \__exp_e_group:n, \__exp_e_put:nn, and \__exp_e_put:nnn.)
For an $N$-type token, call \__exp_e:Nnn with arguments the ⟨first token⟩, the remaining tokens to expand and what’s already been expanded. If the ⟨first token⟩ is non-expandable, including \protected ⟨long or not⟩ macros, it is put in the result by \__exp_e_protected:Nnn. The four special primitives \unexpanded, \noexpand, \the, \primitive are detected; otherwise the token is expanded by \__exp_e_expandable:Nnn.

We don’t try hard to make sensible error recovery since the error recovery of \tex_primitive:D when followed by something else than a primitive depends on the engine. The only valid case is when what follows is $N$-type. Then distinguish special primitives \unexpanded, \noexpand, \the, \primitive from other primitives. In the “other” case, the only reasonable way to check if the primitive that follows \tex_primitive:D is expandable is to expand and compare the before-expansion and after-expansion results. If they coincide then probably the primitive is non-expandable and should be put in the output together with \tex_primitive:D (one can cook up contrived counter-examples where the true \expanded would have an infinite loop), and otherwise one should continue expanding.
The \noexpand primitive has no effect when followed by a token that is not N-type; otherwise \__exp_e_put:nn can grab the next token and put it in the result unchanged.

(End definition for \__exp_e_primitive:Nnn and others.)
The \texttt{\texttt{\unexpanded}} primitive expands and ignores any space, \texttt{\texttt{\scan_stop:}}, or token affected by \texttt{\exp_not:N}, then expects a brace group. Since we only support brace-balanced token lists it is impossible to support the case where the argument of \texttt{\unexpanded} starts with an implicit brace. Even though we want to expand and ignore spaces we cannot blindly \texttt{f-expand} because tokens affected by \texttt{\exp_not:N} should discarded without being expanded further.

As usual distinguish four cases: brace group (the normal case, where we just put the item in the result), space (just \texttt{f-expand} to remove the space), empty (an error), or N-type \texttt{⟨token⟩}. In the last case call \texttt{\texttt{\_exp_e_unexpanded:nN}} triggered by an \texttt{f}-expansion. Having a non-expandable \texttt{⟨token⟩} after \texttt{\unexpanded} is an error (we recover by passing {} to \texttt{\unexpanded}; this is different from \TeX because the error recovery of \texttt{\unexpanded} changes the balance of braces), unless that \texttt{⟨token⟩} is \texttt{\scan_stop:} or a space (recall that we don’t implement the case of an implicit begin-group token). An expandable \texttt{⟨token⟩} is instead expanded, unless it is \texttt{nexpand}. The latter primitive can be followed by an expandable N-type token (removed), by a non-expandable one (kept and later causing an error), by a space (removed by \texttt{f}-expansion), or by a brace group or nothing (later causing an error).

```latex
\cs_new:Npn \_exp_e_unexpanded:Nnn #1 { \_exp_e_unexpanded:nn }
\cs_new:Npn \_exp_e_unexpanded:nn #1 #2 #3
{ \tl_if_head_is_N_type:nTF {#1} }
{ \exp_args:Nf \_exp_e_unexpanded:nn }
{ \_exp_e_unexpanded:N {#1} #1 }
{ \_exp_e_put:nn }
{ \_exp_e_empty:nTF {#1} }
{ \_kernel_msg_expansible_error:nn }
{ \_exp_e_end:nn }
{ \exp_args:Nf \_exp_e_unexpanded:nn }
{[#1]}
}
\cs_new:Npn \_exp_e_unexpanded:nN #1#2
{ \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \exp_after:wN \use_i:nn
\else:
\exp_after:wN \use_ii:nn
\fi:
{ \token_if_eq_catcode:NNTF #2 \c_space_token }
{ \exp_stop_f: }
{
Finally implement \textit{the}. Followed by anything other than an $N$-type \texttt{<token>} this causes an error (we just let TeX make one), otherwise we test the \texttt{<token>}. If the \texttt{<token>} is expandable, expand it. Otherwise it could be any kind of register, or things like \texttt{numexpr}, so there is no way to deal with all cases. Thankfully, only \texttt{toks} data needs to be protected from expansion since everything else gives a string of characters. If the \texttt{<token>} is \texttt{toks} we find a number and unpack using the \texttt{the_toks} functions. If it is a token register we unpack it in a brace group and call \texttt{\_\_exp_e_put:nn} to move it to the result. Otherwise we unpack and continue expanding (useless but safe) since it is basically impossible to have a handle on where the result of \texttt{the} ends.

(End definition for \texttt{\_\_exp_e_unexpanded:Nnn} and others.)
\exp_after:wN \__exp_e_the_toks:n
\exp_after:wN { \int_value:w \if_false: } \fi:
\else:
  \__exp_e_if_toks_register:NTF #1
  { \exp_after:wN \__exp_e_the_toks_reg:N }
  {
    \exp_after:wN \__exp_e:nn \exp_after:wN { \tex_the:D \if_false: } \fi: }
\exp_after:wN #1
\fi:
\}
\exp_after:wN \__exp_e_if_toks_register:NTF { \exp:w \if_false: } \fi:
\exp_after:wN \exp_end: #1

\cs_new:Npn \__exp_e_the_toks_reg:N #1
{\exp_after:wN \__exp_e_put:nnn { \tex_the:D \tex_toks:D #1 } { ? #2 }}

\cs_new:Npn \__exp_e_the_toks:wnn #1; #2
{\exp_args:No \__exp_e_put:nnn \exp_after:wN { \exp_after:wN { \tex_the:D \if_false: } \fi: #1 } }

\cs_new:Npn \__exp_e_the_toks:n #1
{\tl_if_head_is_N_type:nTF {#1}
{ \exp_after:wN \__exp_e_the_toks:N \if_false: { \fi: #1 } }
{ ; {#1} }}

\cs_new:Npn \__exp_e_the_toks:N #1
{\if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
  \exp_after:wN \use_i:nn
\else:
  \exp_after:wN \use_ii:nn
\fi:
{ }

(End definition for \__exp_e_the:Nnn, \__exp_e_the:N, and \__exp_e_the_toks_reg:N.)

\__exp_e_the_toks:wnn \__exp_e_the_toks:n \__exp_e_the_toks:N

The calling function has applied \int_value:w so we collect digits with \__exp_e_the_toks:n (which gets the token list as an argument) and \__exp_e_the_toks:N (which gets the first token in case it is N-type). The digits are themselves collected into an \int_value:w argument to \__exp_e_the_toks:wnn. Then that function unpacks the \toks⟨number⟩ into the result. We include ? because \__exp_e_put:nnn removes one item from its second argument. Note that our approach is rather crude: in cases like \the\toks12-34 the first \int_value:w removes the space and we will incorrectly unpack the \the\toks1234.

\cs_new:Npn \__exp_e_the_toks:wnn #1; #2
{\exp_args:No \__exp_e_put:nnn
  { \tex_the:D \tex_toks:D #1 } { ? #2 }}
We need to detect both \texttt{\toks} registers like \texttt{\toks@} in \LaTeX and parameters such as \texttt{\everypar}, as the result of unpacking the register should not expand further. Registers are found by \texttt{\token_if_toks_register:NTF} by inspecting the meaning. The list of parameters is finite so we just use a \texttt{\cs_if_exist:cTF} test to look up in a table. We abuse \texttt{\cs_to_str:N}'s ability to remove a leading escape character whatever it is.

\begin{verbatim}
\prg_new_conditional:Npnn \__exp_e_if_toks_register:N #1 { TF } { \token_if_toks_register:NTF #1 { \prg_return_true: } { \cs_if_exist:cTF { \__exp_e_the_ \exp_after:wN \cs_to_str:N \token_to_meaning:N #1: } { \prg_return_true: } { \prg_return_false: } }
\end{verbatim}

(End definition for \texttt{\__exp_e_the_toks:wnn, \__exp_e_the_toks:n, and \__exp_e_the_toks:N}.)

We are done emulating \texttt{e}-type argument expansion when \texttt{\expanded} is unavailable.

6.7 Defining function variants

\begin{verbatim}
\end{verbatim}

(End definition for \texttt{\__exp_e_if_toks_register:NTF} and others.)

We are done emulating \texttt{e}-type argument expansion when \texttt{\expanded} is unavailable.

\}
\s__cs_mark  Internal scan marks. No \l3quark yet, so do things by hand.
\s__cs_stop

\cs_new_eq:NN \s__cs_mark \scan_stop:
\cs_new_eq:NN \s__cs_stop \scan_stop:

(End definition for \s__cs_mark and \s__cs_stop.)

\__cs_use_none_delimit_by_s_stop:w
\__cs_use_i_delimit_by_s_stop:nw
\__cs_use_none_delimit_by_q_recursion_stop:w

\cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
\cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
\cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w #1 \q__cs_recursion_stop { }

(End definition for \__cs_use_none_delimit_by_s_stop:w, \__cs_use_i_delimit_by_s_stop:nw, and \__cs_use_none_delimit_by_q_recursion_stop:w)

\__cs_generate_variant:Nn  #1: Base form of a function; e.g., \tl_set:Nn
\__cs_generate_variant:cn  #2: One or more variant argument specifiers; e.g., {Nx,c,cx}

After making sure that the base form exists, test whether it is protected or not and define \__cs_tmp:w as either \cs_new:Npx or \cs_new_protected:Npx, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

\cs_new_protected:Npn \cs_generate_variant:Nn #1#2
{ \__cs_generate_variant:N #1
  \use:x
  { \__cs_generate_variant:nnNN \cs_split_function:N #1
    \exp_not:N #1
    \tl_to_str:n {#2} ,
    \exp_not:N \scan_stop: ,
    \exp_not:N \q__cs_recursion_stop
  }
}

(End definition for \__cs_generate_variant:Nn. This function is documented on page 27.)

\__cs_generate_variant:N
\__cs_generate_variant:ww
\__cs_generate_variant:wwNw

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the else: branch is safe because non-expandable primitives cannot be \TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then protected appears in the meaning before the fist occurrence of macro. The \ww auxiliary removes everything in the meaning string after the first ma. We use ma rather than the full macro because the meaning of the \firstmark primitive (and
four others) can contain an arbitrary string after a leading \texttt{firstmark}:. Then, look for \texttt{pr} in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, \texttt{\long}, \texttt{\protected}, \texttt{\protected\long}, \texttt{\first}, \texttt{\top}, \texttt{\bot}, \texttt{\splittop}, or \texttt{\splitbot}, with \texttt{\} replaced by the appropriate escape character. If \texttt{pr} appears in the part before \texttt{ma}, the first \texttt{_\cs_mark} is taken as an argument of the \texttt{wwNw} auxiliary, and #3 is \texttt{\cs_new_protected:Npx}, otherwise it is \texttt{\cs_new:Npx}.

\begin{verbatim}
\cs_new_protected:Npx \__cs_generate_variant:N #1
\{
  \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
  \exp_not:N \exp_not:N #1 #1
  \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
  \exp_not:N \else:
  \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
  \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
  \s__cs_mark
  \s__cs_mark \cs_new_protected:Npx
  \tl_to_str:n { pr }
  \s__cs_mark \cs_new:Npx
  \s__cs_stop
  \exp_not:N \fi:
\}
\exp_last_unbraced:NNNNo
\cs_new_protected:Npn \__cs_generate_variant:wwNN #1 #2 #3 #4
\{ \__cs_generate_variant:Nnnw #4 {#1}{#2}
\exp_last_unbraced:NNNNo
\cs_new_protected:Npn \__cs_generate_variant:wwNN #1 #2 #3 #4 #5 #6 #7
\{ \__cs_generate_variant:Nnnw #7 {#1}{#2}{#3}{#4}
\end{verbatim}

\begin{verbatim}
\__cs_generate_variant:nnNN #1 : Base name.
\#2 : Base signature.
\#3 : Boolean.
\#4 : Base function.

If the boolean is \texttt{\c_false_bool}, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as \#4 for efficiency.
\begin{verbatim}
\cs_new_protected:Npm \__cs_generate_variant:nnNN #1\#2\#3\#4
\{
  \if_meaning:w \c_false_bool \#3
  \_\kernel_msg_error:nnx { kernel } { missing-colon }
  \{ \token_to_str:c {#1} \}
  \exp_after:wN \_\cs_use_none_delimit_by_q_recursion_stop:w
  \fi:
  \_\cs_generate_variant:Nnmw #4 \#1\#2
\}
\end{verbatim}

\begin{verbatim}
\_\cs_generate_variant:Nnmw #1 : Base function.
\#2 : Base name.
\end{verbatim}
First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of \(l\) letters and the last \(k - l\) letters of the base signature (of length \(k\)). For example, for a base function \(\texttt{prop\_put:Nnn}\) which needs a \(cV\) variant form, we want the new signature to be \(cVn\).

There are further subtleties:

- In \(\texttt{cs\_generate\_variant:Nn \ foo:mmTF \{xxTF}\}\), we must define \(\texttt{foo:mmTF}\) using \(\exp\_args:Nxx\), rather than a hypothetical \(\exp\_args:NxxTF\). Thus, we wish to trim a common trailing part from the base signature and the variant signature.

- In \(\texttt{cs\_generate\_variant:Nn \ foo:on\{ox\}\}\), the function \(\texttt{foo:ox}\) must be defined using \(\exp\_args:Nnx\), not \(\exp\_args:Nox\), to avoid double \(o\) expansion.

- Lastly, \(\texttt{cs\_generate\_variant:Nn \ foo:on \{xn\}\}\) must trigger an error, because we do not have a means to replace \(o\)-expansion by \(x\)-expansion. More generally, we can only convert \(N\) to \(c\), or convert \(n\) to \(V\), \(v\), \(o\), \(f\), \(x\).

All this boils down to a few rules. Only \(n\) and \(N\)-type arguments can be replaced by \(\texttt{cs\_generate\_variant:Nn}\). Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to \(n\) except for \(N\) and \(p\)-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within \(x\)-expansion. The result is given to \(\texttt{\_\_cs\_generate\_variant:WWNN}\) (defined later) in the form \(\langle \text{processed variant signature} \rangle \s__cs\_mark \langle \text{errors} \rangle \s__cs\_stop \langle \text{base function} \rangle \langle \text{new function} \rangle\). If all went well, \(\langle \text{errors} \rangle\) is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after \#3 and after the following brace group. Those are ignored by \TeX{} when fetching the last argument for \(\texttt{\_\_cs\_generate\_variant\_loop:nwwNNn}\), but can be used as a delimiter for \(\texttt{\_\_cs\_generate\_variant\_loop\_end:nwwNNnn}\).

```latex
\cs_new_protected:Npn \_\_cs\_generate\_variant:Nnnw \#1\#2\#3\#4 , \\
{ \\
\if_meaning:w \scan_stop: \#4 \\
\exp_after:wN \_\_cs\_use\_none\_delimit\_by\_q\_recursion\_stop:w \\
\fi: \\
\use:x \\
\exp_not:N \_\_cs\_generate\_variant:WWNN \\
\_\_cs\_generate\_variant\_loop:nWWN \{ \} \\
\_\_cs\_generate\_variant\_loop\_end:nwwNNnn \\
\s__cs\_mark \\
\#3 - \\
\{ - \} \fi: \_\_cs\_generate\_variant\_loop\_long:WWNNn \} - \\
\{ \} \\
\s__cs\_stop \\
\exp_not:N \#1 \{\#2\} \{\#4\} \\
\} \\
\_\_cs\_generate\_variant:Nnww \#1 \{\#2\} \{\#3\}
```
End definition for \texttt{\_cs_generate_variant:Nnnw}.

\begin{itemize}
\item \#1: Last few consecutive letters common between the base and variant (more precisely, \texttt{\_cs_generate_variant_same:N} \langle letter\rangle for each letter).
\item \#2: Next variant letter.
\item \#3: Remainder of variant form.
\item \#4: Next base letter.
\end{itemize}

The first argument is populated by \texttt{\_cs_generate_variant_loop_same:w} when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is \texttt{N} and the variant is \texttt{c}, or when the base is \texttt{n} and the variant is \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, or \texttt{x}. Otherwise, call \texttt{\_cs_generate_variant_loop_invalid:NNwNNnn} to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of \texttt{\_cs_generate_variant:wwNN}. If the letters are distinct and the base letter is indeed \texttt{n} or \texttt{N}, leave in the input stream whatever argument \#1 was collected, and the next variant letter \#2, then loop by calling \texttt{\_cs_generate_variant_loop:nNwN}.

The loop can stop in three ways.

\begin{itemize}
\item If the end of the variant form is encountered first, \#2 is \texttt{\_cs_generate_variant_loop_end:nwwNNnn} (expanded by the conditional \texttt{\if:w}), which inserts some tokens to end the conditional; grabs the \langle base name\rangle as \#7, the \langle variant signature\rangle \#8, the \langle next base letter\rangle \#1 and the part \#3 of the base signature that wasn’t read yet; and combines those into the \langle new function\rangle to be defined.
\item If the end of the base form is encountered first, \#4 is \texttt{~\fi}; which ends the conditional (with an empty expansion), followed by \texttt{\_cs_generate_variant_loop_long:wNNn}, which places an error as the second argument of \texttt{\_cs_generate_variant:wwNN}.
\item The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (\texttt{n} or \texttt{N} to support the variant). In that case too an error is placed as the second argument of \texttt{\_cs_generate_variant:wwNN}.
\end{itemize}

Note that if the variant form has the same length as the base form, \#2 is as described in the first point, and \#4 as described in the second point above. The \texttt{\_cs_generate_variant_loop_end:nwwNNnn} breaking function takes the empty brace group in \#4 as its first argument: this empty brace group produces the correct signature for the full variant.

\begin{verbatim}
\cs_new:Npn \_cs_generate_variant_loop:nNwN #1#2#3 \s\_cs_mark #4
\{
  \if:w #2 #4
  \exp_after:wN \_cs_generate_variant_loop_same:w
  \else:
    \if:w #1 \_cs_generate_variant_loop_base:N #2 \else:
      \if:w 0
        \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
        \if:w \scan_stop: \_cs_generate_variant_loop_base:N #2 1 \fi:
      0
      \_cs_generate_variant_loop_special:NNwNNnn #4#2
  \fi:
\end{verbatim

369
\__cs_generate_variant_loop_invalid:NNwNNnn \#4\#2
\fi:
\fi:
\fi:
#1
\prg_do_nothing:
#2
\__cs_generate_variant_loop:nNwN { } \#3 \s__cs_mark
\cs_new:Npn \__cs_generate_variant_loop_base:N #1
{
\if:w c #1 N \else:
\if:w o #1 n \else:
\if:w V #1 n \else:
\if:w V #1 n \else:
\if:w V #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w e #1 n \else:
\if:w x #1 n \else:
\if:w x #1 n \else:
\if:w n #1 n \else:
\if:w N #1 N \else:
\if:w N #1 N \else:
\scan_stop:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
#5 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
\cs_new:Npn \__cs_generate_variant_loop_same:w #1 \prg_do_nothing: \#2\#3\#4
\cs_new:Npn \__cs_generate_variant_loop_same:N #2
\cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
#1#2 \s__cs_mark \#3 \s__cs_stop #5#6#7#8
\scan_stop: \scan_stop: \fi:
\a__cs_mark \s__cs_stop
\exp_not:N \#6
\exp_not:c \{ \#7 : \#8 \#1 \#3 \}
\cs_new:Npn \__cs_generate_variant_loop_long:wNNnn \#1 \s__cs_stop \#2\#3\#4\#5
\exp_not:n
{ \s__cs_mark
\__kernel_msg_error:nxxx \{ kernel \} \{ variant-too-long \}
\{ \#5 \} \{ \token_to_str:N \#3 \}
\use_none:nnn
\s__cs_stop
#3
#3
\cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
  \#1#2 \fi: \fi: \fi: \fi: \cs_stop \#4\#5\#6\#7
\{\fi: \fi: \fi: \exp_not:n\{
  \s__cs_mark\__kernel_msg_error:nnxxxx { kernel } { invalid-variant }\
  \#7} { \token_to_str:N \#5 } \#1 \#2\use_none:nnn\s__cs_stop\#5
\end{definition}

\cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
  \#1#2#3 \cs_stop \#4\#5\#6\#7
\{\if:w \#3 \s__cs_stop \#4 \#5 \{\#6\} \{\#7\}
\exp_not:n\{
  \__kernel_msg_error:nnxxxx
  \{ kernel \} \{ deprecated-variant \}
  \#7} { \token_to_str:N \#5 } \#1 \#2\}
\end{definition}

(End definition for \__cs_generate_variant_loop:NNWn and others.)

\__cs_generate_variant_same:N When the base and variant letters are identical, don’t do any expansion. For most
argument types, we can use the n-type no-expansion, but the N and p types require a
slightly different behaviour with respect to braces. For V-type this function could output
N to avoid adding useless braces but that is not a problem.
\cs_new:Npn \__cs_generate_variant_same:N  \#1
  \if:w \N \#1 \#1 \else:\if:w \p \#1 \#1 \else:
  \token_to_str:N \#1 \n\if:w \n \#1 \else:
  \__cs_generate_variant_loop_special:NNwNNnn \#1\#1
  \fi:
  \fi:
  \fi:
  \fi:
\end{definition}

(End definition for \__cs_generate_variant_same:N.)

\__cs_generate_variant:wwNN If the variant form has already been defined, log its existence (provided log-functions
is active). Otherwise, make sure that the \exp_args:N \#3 form is defined, and if it
contains x, change \__cs_tmp:w locally to \cs_new_protected:Npx. Then define the
variant by combining the \exp_args:N \#3 variant and the base function.
\cs_new_protected:Npn \__cs_generate_variant:wwNN
\end{document}
\begin{verbatim}
\__cs_generate_internal_variant:n #1 \s__cs_mark #2 \s__cs_stop #3#4
{
  \__cs_generate_internal_variant:n {\exp_not:c { \exp_args:N #1 } \exp_not:N #3 }
\group_end:
}
\end{verbatim}

(End definition for \__cs_generate_variant:wwNN.)

First test for the presence of x (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting \__cs_tmp:w). Then call \__cs_generate_internal_variant:NNN with arguments \cs_new_protected:cpn \use:x (for protected) or \cs_new:cpn \tex_expanded:D (expandable) and the signature. If p appears in the signature, or if the function to be defined is expandable and the primitive \exp_last_unbraced:NNN is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate \__cs commands. Otherwise, call \__cs_generate_internal_one_go:NNn to construct the \exp_args:N... function as a macro taking up to 9 arguments and expanding them using \use:x or \tex_expanded:D.

\begin{verbatim}
\cs_new_protected:Npx \__cs_generate_internal_variant:n #1 \s__cs_mark
{ \exp_not:N \__cs_generate_internal_variant:wwnNwn #1 \s__cs_mark { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx } \cs_new_protected:cpn \use:x \token_to_str:N x \s__cs_stop {#1} \cs_set_protected:Npn \__cs_tmp:w #1
{
\cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
{\if_catcode:w X \use_none:nnnnnnnn ##3 \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: \prg_do_nothing: X
}
\cs_if_free:cT { \exp_args:N #7 }
{ \__cs_generate_internal_variant:NNn #4 #5 {#7} }
\group_end:
{#1}
\exp_last_unbraced:NNN
\cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwn #1 \s__cs_mark
{ \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx } \cs_new_protected:cpn \use:x \token_to_str:N x \s__cs_mark { } \cs_new:cpn \exp_not:N \tex_expanded:D \s__cs_stop \s__cs_stop}
\end{verbatim}

372
This command grabs char by char outputting \\:##1 (not expanded further). We avoid tests by putting a trailing : \use_i:nn, which leaves \cs_end: and removes the looping macro. The colon is in fact also turned into \\: so that the required structure for \exp_args:N... commands is correctly terminated.

(End definition for \_cs_generate_internal_variant:n and \_cs_generate_internal_variant_\-loop:n.)
This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides \texttt{NnpcofVvx}, in particular that there are no spaces. Then we just call the internal function.

```latex
\exp_args_generate:n
```

This function is documented on page 106.
6.8 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

Here are the actual function definitions, using the helper functions above. The group is used because \_\_cs\_generate\_internal\_variant:n redefines \_\_cs\_tmp:w locally.

(End definition for \exp\_args\_generate:n. This function is documented on page 261.)

(End definition for \exp\_args\_inc and others. These functions are documented on page 31.)
7 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

7.1 Quarks

\quark_new:N \q_nil Allocate a new quark.
\cs_new_protected:Npn \quark_new:N \q_mark \q_no_value \q_stop

Some “public” quarks. \q_stop is an “end of argument” marker, \q_nil is a empty value and \q_no_value marks an empty argument.

(End definition for \quark_new:N. This function is documented on page 38.)

(End definition for \q_nil and others. These variables are documented on page 39.)
Quarks for ending recursions. Only ever used there! \texttt{\textbackslash q.recursion_tail} is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. \texttt{\textbackslash q.recursion_stop} is placed directly after the list.

\begin{verbatim}
\quark_new:N \q_recursion_tail
\quark_new:N \q_recursion_stop
\end{verbatim}

(End definition for \texttt{\textbackslash q.recursion_tail} and \texttt{\textbackslash q.recursion_stop}. These variables are documented on page 39.)

\texttt{s_.quark}

Private scan mark used in \texttt{l3quark}. We don’t have \texttt{l3scan} yet, so we declare the scan mark here and add it to the scan mark pool later.

\begin{verbatim}
\cs_new_eq:NN \s__quark \scan_stop:
\end{verbatim}

(End definition for \texttt{s_.quark}.)

\texttt{\textbackslash q_.quark_nil}

Private quark use for some tests.

\begin{verbatim}
\quark_new:N \q__quark_nil
\end{verbatim}

(End definition for \texttt{\textbackslash q_.quark_nil}.)

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

\begin{verbatim}
\cs_new:Npn \quark_if_recursion_tail_stop:N #1
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
\end{verbatim}

(End definition for \texttt{\quark_if_recursion_tail_stop:N} and \texttt{\quark_if_recursion_tail_stop_do:Nn}. These functions are documented on page 40.)

\texttt{\textbackslash quark_if_recursion_tail_stop:n}
\texttt{\textbackslash quark_if_recursion_tail_stop:o}
\texttt{\textbackslash quark_if_recursion_tail_stop:do:n}
\texttt{\textbackslash quark_if_recursion_tail_stop:do:on}
\texttt{\__quark_if_recursion_tail:w}

See \texttt{\quark_if_nil:nTF} for the details. Expanding \texttt{\__quark_if_recursion_tail:w} once in front of the tokens chosen here gives an empty result if and only if \texttt{#1} is exactly \texttt{\textbackslash q.recursion_tail}.

\begin{verbatim}
\cs_new:Npn \quark_if_recursion_tail_stop:n #1
\end{verbatim}

(End definition for \texttt{\quark_if_recursion_tail_stop:n} and \texttt{\quark_if_recursion_tail_stop:do:Nn}. These functions are documented on page 40.)

378
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
\{
\tl_if_empty:oTF
{ \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
{ \use_i_delimit_by_q_recursion_stop:nw }
{ \use_none:n }
\}
\cs_new:Npn \__quark_if_recursion_tail:w
#1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
(End definition for \quark_if_recursion_tail_stop:n, \quark_if_recursion_tail_stop_do:nn, and \__quark_if_recursion_tail:w. These functions are documented on page 40.)

Analogues of the \quark_if_recursion_tail_break... functions. Break the mapping using #2.
\cs_new:Npn \quark_if_recursion_tail_break:NN #1 #2
{\if_meaning:w \q_recursion_tail #1 \exp_after:wN #2 \fi:}
\cs_new:Npn \quark_if_recursion_tail_break:nN #1 #2
{\tl_if_empty:oT \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! } {#2}
}

(End definition for \quark_if_recursion_tail_break:NN and \quark_if_recursion_tail_break:nN. These functions are documented on page 40.)

\prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
{ \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
{ \if_meaning:w \q_no_value #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
{ \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
{ \if_meaning:w \q_no_value #1 \prg_return_true: \else: \prg_return_false: \fi: }

\textsuperscript{8}It may still loop in special circumstances however!
Let us explain \quark\_if\_nil:n(TF). Expanding \__quark\_if\_nil:w once is safe thanks to the trailing \q\_nil ??!. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens ??!. Thanks to the leading \{}, the argument #1 is empty if and only if the argument of \quark\_if\_nil:n starts with \q\_nil. The argument #2 is empty if and only if this \q\_nil is followed immediately by ? or by \{}?, coming either from the trailing tokens in the definition of \quark\_if\_nil:n, or from its argument. In the first case, \__quark\_if\_nil:w is followed by \{}\q\_nil? \q\_nil??!, hence #3 is delimited by the final ??!, and the test returns true as wanted. In the second case, the result is not empty since the first ??! in the definition of \quark\_if\_nil:n stop #3. The auxiliary here is the same as \__tl\_if\_empty\_if:o, with the same comments applying.

The function \__kernel\_quark\_new\_test:N defines #1 in a similar way as \quark\_\_if\_recursion\_tail... functions (as described below), using \q\__⟨namespace⟩\_recursion\_tail as the test quark and \q\__⟨namespace⟩\_recursion\_stop as the delimiter quark, where the ⟨namespace⟩ is determined as the first \_\_delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:
Some text extracted from the document.
\cs_new_protected:Npn \__quark_new_conditional:Nnnn \#1 \#2 \#3 \#4
\{
  \if_meaning:w \q_nil \#2 \q_nil
  \__kernel_msg_error:n { kernel } { invalid-quark-function }
  \{ \token_to_str:N \#1 \}
  \else:
  \__kernel_msg_error:n { kernel } { invalid-quark-function }
  \{ \token_to_str:N \#1 \}
  \else:
  \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
  \{ \cs_split_function:N \#1 \}
  \} \{ conditional \}
  \{#2} \{#3} \{#4\}
  \fi:
  \fi:
\}
\cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nxx }
\cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
\{
  \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
  \{ \__kernel_msg_error:n { kernel } { invalid-quark-function }
  \{ \token_to_str:N \#4 \} \{#2\}
  \use_none:nnn
  \}
\}
\cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
\{
  \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 \{ none \} \{ \} \{ \}
  \__quark_new_test_define_tl:nNnNNn #1 \{ \}
\}
\cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
\{
  \__quark_new_test_aux_do:nNNnnnnNNn #4 \#2 \#3 \{ i \} \{ n \} \{\#1\} \{\#2\}
  \__quark_new_test_define_tl:nnNNnnn \#1 \{ \use_none:n \}
\}
\cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
\{
  \__quark_new_testAux_do:nnNNnnnn \#4 \#2 \#3 \{ i \} \{ n \} \{\#1\} \{\#2\}
  \__quark_new_test_define_break_tl:nnNNnnn \#1 \{ \}
\}
\cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
\{
  \__quark_new_testAux_do:nnNNnnnn \#4 \#2 \#3 \{ none \} \{ \} \{ \}
  \__quark_new_test_define_ifx:nNnNNn \#1 \{ \}
\}(End definition for \__kernel_quark_new_test:N and others.)

These macros implement the six possibilities mentioned above, passing the right arguments to \__quark_new_test_aux_do:nnNNnnnnNNn, which defines some auxiliaries, and then to \__quark_new_test_define_tl:nnNNnn \( :n(n) \) variants or to \__quark_new_test_define_ifx:nNnNNn \( :N(n) \) which define the main conditionals.
The control sequence names which will be used by \__quark_test_define_aux:NNNNnnNNn, and then later by \__quark_new_test_define_tl:nNnnNn or \__quark_new_test_define_ifx:nNnNNn. The control sequences defined here are analogous to \__quark_if_recursion_tail:w and to \use_(none|i)_delimit_by_q_recursion_stop:(|n)w.

The name is composed by the name-space and the name of the quarks. Suppose \__kernel_quark_new_test:N was used with:

\__kernel_quark_new_test:N \__test_quark_tail:n

then the first auxiliary will be \__test_recursion_tail:w, and the second one will be \__test_use_none_delimit_by_q_recursion_stop:w.

Note that the actual quarks are not defined here. They should be defined separately using \quark_new:N.

\cs_new_protected:Npn \__quark_new_test_aux_do:nNNnnNNn \__quark_new_test_define_ifx:nNnNNn \__quark_new_test_define_tl:nNnnNn

(End definition for \__quark_new_test_n:Nnnn and others.)

Finally, these two macros define the main conditional function using what’s been set up before.
These macros implement the two possibilities for branching quark conditionals, passing the right arguments to \__quark_new_conditional\_aux\_do:NNnnn, which defines some auxiliaries and defines the main conditionals.

\__quark_module\_name:N takes a control sequence and returns its ⟨module⟩ name, determined as the first non-empty non-single-character word, separated by _ or :: These rules give the correct result for public functions ⟨module⟩\_\..., private functions \__⟨module⟩\_\..., and variables such as \l⟨module⟩\_.\... If no valid module is found the
result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as l3tl is not fully available when \_kernel_quark_new_\_test:N is first used. If no (module) is found (such as in \::n) we get the trailing marker \use_none:n {}, which expands to nothing.

\cs_set:Npn \__quark_tmp:w #1#2
\cs_new:Npn \__quark_module_name:N ##1
{ \exp_last_unbraced:Nf \__quark_module_name:w \cs_to_str:N ##1 } #1 \s__quark
\cs_new:Npn \__quark_module_name:w #1 #2 \use_none:n { } #2 \s__quark
\cs_new:Npn \__quark_module_name_loop:w #1 #2 \use_none:n { } #2 \s__quark
{ \__quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
\cs_new:Npn \__quark_module_name_end:w #1 \fi:
\cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
{ \cs_new:Npn \__quark_quark_conditional_name:N ##1
{ \exp_last_unbraced:Nf \__quark_quark_conditional_name:w \cs_to_str:N ##1 } #1 #2 \s__quark
{ \__quark_quark_conditional_name:w ##1 #2 ##2 #1 \s__quark {##2} }
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark
\cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
{ \cs_new:Npn \__quark_quark_conditional_name:N ##1
{ \exp_last_unbraced:Nf \__quark_quark_conditional_name:w \cs_to_str:N ##1 } #1 \s__quark
\cs_new:Npn \__quark_quark_conditional_name:w #1 #2 \s__quark
#1 \fi: \#2 \s__quark { \fi: \#1 }
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ }
\cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
{ \cs_new:Npn \__quark_quark_conditional_name:N ##1
{ \exp_last_unbraced:Nf \__quark_quark_conditional_name:w \cs_to_str:N ##1 } #1 \s__quark
\cs_new:Npn \__quark_quark_conditional_name:w #1 #2 \s__quark
#1 \#2 \#2 \#1 \s__quark \#2}
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

(End definition for \_quark_module_name:N and others.)

\_quark_quark_conditional_name:N \_quark_quark_conditional_name:w \_quark_quark_conditional_name:N determines the quark name that the quark conditional function #1 queries, as the part of the function name between _quark_if_ and the trailing :. Again we define it through \_quark_tmp:w, which receives : as #1 and _quark_if_ as #2. The auxiliary \_quark_quark_conditional_name:w returns the part between the first _quark_if_ and the next :, and we apply this auxiliary to the function name followed by : (in case the function name is lacking a signature), and _quark_if_: so that \_quark_quark_conditional_name:N returns an empty string if _quark_if_ is not present.

\cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
{ \cs_new:Npn \__quark_quark_conditional_name:N ##1
{ \exp_last_unbraced:Nf \__quark_quark_conditional_name:w \cs_to_str:N ##1 } #1 \s__quark
\cs_new:Npn \__quark_quark_conditional_name:w #1 #2 \s__quark
#1 \#2 \#2 \#1 \s__quark \#2}
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

(End definition for \_quark_quark_conditional_name:N and \_quark_quark_conditional_name:w.)
7.2 Scan marks

\g__scan_marks_tl The list of all scan marks currently declared. No l3tl yet, so define this by hand.

\cs_gset:Npn \g__scan_marks_tl { }

(End definition for \g__scan_marks_tl.)

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to \scan_stop: globally.

\cs_new_protected:Npn \scan_new:N #1
\tl_if_in:NnTF \g__scan_marks_tl { #1 }
{ \__kernel_msg_error:nnx { kernel } { scanmark-already-defined } \token_to_str:N #1 }
{ \tl_gput_right:Nn \g__scan_marks_tl {#1} \cs_new_eq:NN #1 \scan_stop: }
}

(End definition for \scan_new:N. This function is documented on page 41.)

\s_stop We only declare one scan mark here, more can be defined by specific modules. Can’t use \scan_new:N yet because l3tl isn’t loaded, so define \s_stop by hand and add it to \g__scan_marks_tl. We also add \s__quark (declared earlier) to the pool here. Since it lives in a different namespace, a little l3docstrip cheating is necessary.

\cs_new_eq:NN \s_stop \scan_stop: \cs_gset_nopar:Npx \g__scan_marks_tl \s_stop
\exp_not:o \g__scan_marks_tl \s_stop \@@=quark \s__quark \@@=scan }

(End definition for \s_stop. This variable is documented on page 42.)

\use_none_delimit_by_s_stop:w Similar to \use_none_delimit_by_q_stop:w.

\cs_new:Npn \use_none_delimit_by_s_stop:w \s_stop { }

(End definition for \use_none_delimit_by_s_stop:w. This function is documented on page 42.)
8 \texttt{tl} implementation

A token list variable is a \LaTeX macro that holds tokens. By using the \texttt{\sTeX} primitive \texttt{\unexpanded} inside a \LaTeX \texttt{\edef} it is possible to store any tokens, including \#, in this way.

8.1 Functions

\texttt{\tl_new:N} Creating new token list variables is a case of checking for an existing definition and doing the definition.

\texttt{\tl_new:c} Constants are also easy to generate.

\texttt{\tl_clear:N} Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

\texttt{\tl_gclear:N} Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.
For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit.

\begin{verbatim}
\cs_new_protected:Npn \tl_set_eq:NN #1#2 { \cs_set_eq:NN #1 #2 }
\cs_new_protected:Npn \tl_gset_eq:NN #1#2 { \cs_gset_eq:NN #1 #2 }
\cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
\cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
\end{verbatim}

Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to \texttt{\textbackslash scan\_stop:} would lead to problems later on.

\begin{verbatim}
\cs_new_protected:Npn \tl_concat:NNN #1#2#3 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_new_protected:Npn \tl_gconcat:NNN #1#2#3 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_generate_variant:Nn \tl_concat:NNN { ccc }
\cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
\end{verbatim}

Copies of the \texttt{cs} functions defined in \texttt{l3basics}.

\begin{verbatim}
\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N \{ TF , T , F , p \}
\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c \{ TF , T , F , p \}
\end{verbatim}

\subsection{Constant token lists}

\texttt{\c_empty_tl} Never full. We need to define that constant before using \texttt{\tl_new:N}.

\begin{verbatim}
\tl_const:Nn \c_empty_tl { }
\end{verbatim}

\texttt{\c_novalue_tl} A special marker: as we don’t have \texttt{\char\_generate:nn} yet, has to be created the old-fashioned way.

\begin{verbatim}
\group_begin:
\tex_lccode:D 'A = '¬
\tex_lccode:D 'N = 'N
\tex_lccode:D 'V = 'V
\tex_lowercase:D
{ \group_end:
\tl_const:Nn \c_novalue_tl { ANoValue- }
}
\end{verbatim}

\texttt{\c_space_tl} A space as a token list (as opposed to as a character).

\begin{verbatim}
\tl_const:Nn \c_space_tl { ~ }
\end{verbatim}
8.3 Adding to token list variables

\texttt{\tl_set:Nn} \hspace{1em} By using \texttt{\exp_not:n} token list variables can contain \# tokens, which makes the token list registers provided by Te\TeX{} more or less redundant. The \texttt{\tl_set:No} version is done "by hand" as it is used quite a lot.

\begin{verbatim}
\tl_set:Nn \tl_set:NV \tl_set:Nv \tl_set:No \tl_set:Nf \tl_set:Nx \\
\tl_gset:Nn \tl_gset:NV \tl_gset:Nv \tl_gset:No \tl_gset:Nf \tl_gset:Nx \\
\end{verbatim}

By using \texttt{\exp_not:n} token list variables can contain \# tokens, which makes the token list registers provided by Te\TeX{} more or less redundant. The \texttt{\tl_set:No} version is done "by hand" as it is used quite a lot.

\begin{verbatim}
\cs_new_protected:Npn \tl_set:Nn #1#2 \{
  \cs_set_nopar:Npx #1 { \exp_not:n {#2} } \\
\cs_new_protected:Npn \tl_set:No #1#2 \{
  \cs_set_nopar:Npx #1 { \exp_not:o {#2} } \\
\cs_new_protected:Npn \tl_set:Nx #1#2 \{
  \cs_set_nopar:Npx #1 {#2} \\
\cs_new_protected:Npn \tl_gset:Nn #1#2 \{
  \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } \\
\cs_new_protected:Npn \tl_gset:No #1#2 \{
  \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } \\
\cs_new_protected:Npn \tl_gset:Nx #1#2 \{
  \cs_gset_nopar:Npx #1 {#2} \\
\end{verbatim}

(End definition for \texttt{\tl_set:Nn} and \texttt{\tl_gset:Nn}. These functions are documented on page 44.)

\begin{verbatim}
\tl_put_left:Nn \tl_put_left:NV \tl_put_left:No \tl_put_left:Nx \\
\tl_gput_left:Nn \tl_gput_left:NV \tl_gput_left:No \tl_gput_left:Nx \\
\end{verbatim}

Adding to the left is done directly to gain a little performance.

\begin{verbatim}
\cs_new_protected:Npn \tl_put_left:Nn #1#2 \{
  \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } \\
\cs_new_protected:Npn \tl_put_left:NV #1#2 \{
  \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } \\
\cs_new_protected:Npn \tl_put_left:No #1#2 \{
  \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } \\
\cs_new_protected:Npn \tl_put_left:Nx #1#2 \{
  \cs_set_nopar:Npx #1 {#2 \exp_not:o {#1} } \\
\end{verbatim}

(End definition for \texttt{\tl_put_left:Nn} and \texttt{\tl_gput_left:Nn}. These functions are documented on page 44.)

389
\textit{\textbackslash tl\_put\_right:Nn} The same on the right.
\textit{\textbackslash tl\_put\_right:NV}
\textit{\textbackslash tl\_put\_right:No}
\textit{\textbackslash tl\_put\_right:Nx}
\textit{\textbackslash tl\_put\_right:cn}
\textit{\textbackslash tl\_put\_right:cV}
\textit{\textbackslash tl\_put\_right:co}
\textit{\textbackslash tl\_put\_right:cx}
\textit{\textbackslash tl\_gput\_right:Nn}
\textit{\textbackslash tl\_gput\_right:NV}
\textit{\textbackslash tl\_gput\_right:No}
\textit{\textbackslash tl\_gput\_right:Nx}
\textit{\textbackslash tl\_gput\_right:cn}
\textit{\textbackslash tl\_gput\_right:cV}
\textit{\textbackslash tl\_gput\_right:co}
\textit{\textbackslash tl\_gput\_right:cx}

(End definition for \textit{\textbackslash tl\_put\_right:Nn} and \textit{\textbackslash tl\_gput\_right:Nn}. These functions are documented on page 44.)

8.4 Internal quarks and quark-query functions

\texttt{\__tl\_nil}
\texttt{\__tl\_mark}
\texttt{\__tl\_stop}

(End definition for \texttt{\__tl\_nil}, \texttt{\__tl\_mark}, and \texttt{\__tl\_stop}.)

\texttt{\__tl\_recursion\_tail}
\texttt{\__tl\_recursion\_stop}

(End definition for \texttt{\__tl\_recursion\_tail} and \texttt{\__tl\_recursion\_stop}.)

\texttt{\_\_kernel\_quark\_new\_test:N} \texttt{\_\_kernel\_quark\_new\_conditional:N}

(End definition for \texttt{\_\_kernel\_quark\_new\_test:N} and \texttt{\_\_kernel\_quark\_new\_conditional:NTF}.)
8.5 Reassigning token list category codes

The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

\begin{verbatim}
\tl_const:Nx \c__tl_rescan_marker_tl \{ : \token_to_str:N : \}
\end{verbatim}

(End definition for \c__tl_rescan_marker_tl.)

In a group, after some initial setup explained below and the user setup #3 (followed by \scan_stop: to be safe), there is a call to __tl_set_rescan:nNN. This shared auxiliary defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-line files, it calls (with the same arguments) __tl_set_rescan_multi:nNN, whose code is included here to help understand the approach. This function rescans its argument #1, closes the group, and performs the assignment.

One difficulty when rescanning is that \scantokens treats the argument as a file, and without the correct settings a \TeX{} error occurs:

! File ended while scanning definition of ...

A related minor issue is a warning due to opening a group before the \scantokens and closing it inside that temporary file; we avoid that by setting \tracingnesting. The standard solution to the “File ended” error is to grab the rescanned tokens as a delimited argument of an auxiliary, here __tl_rescan:NNw, that performs the assignment, then let \TeX{} “execute” the end of file marker. As usual in delimited arguments we use \prg_do_-nothing: to avoid stripping an outer set braces: this is removed by using o-expanding assignments. The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

For \tl_rescan:nn we cannot simply call __tl_set_rescan:nNN \prg_do_-nothing: \use:n because that would leave the end-of-file marker after the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that \newlinechar should be equal to \endlinechar because \newlinechar characters become new lines and then become \endlinechar characters when writing to an abstract file and reading back. This equality is ensured by setting \newlinechar equal to \endlinechar. Prior to this, \endlinechar is set to −1 if it was 32 (in particular true after \ExplSyntaxOn) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two \if_false:: \fi: are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after \group_begin: so that if one accidentally f-expands \tl_set_rescan:Nn braces remain balanced. This is essential in e-type arguments when \expanded is not available.

\begin{verbatim}
\cs_new_protected:Npn \tl_rescan:nn #1#2
{ \tl_set_rescan:Nn \tl__tl_internal_a_tl \{#1\} \{#2\}
  \exp_after:wN \tl_clear:N \exp_after:wN \tl__tl_internal_a_tl
  \tl__tl_internal_a_tl
}
\cs_new_protected:Npn \tl_set_rescan:Nnn
\tl_set_rescan:Nno
\tl_set_rescan:Nnx
\tl_gset_rescan:Nnn
\tl_gset_rescan:Nno
\tl_gset_rescan:Nnx
\tl_rescan:nn
\__tl_set_rescan:NNnn
\__tl_set_rescan_multi:nNN
\__tl_rescan:NNw
\end{verbatim}

391
The function \texttt{\_\_tl_set_rescan:NNn} calls \texttt{\_\_tl_set_rescan_multi:NNn} or \texttt{\_\_tl_set_rescan_single:nnNN \{ \texttt{'} \}} depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a \texttt{\newlinechar} character. If \texttt{\newlinechar} is out of range, the argument is assumed to be a single line.

For a single line, no \texttt{\newlinechar} should be added, so it is set to \texttt{-1}, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as \TeX{} removes these at a very low level. The only way to preserve them is to rescans not the argument but the argument followed by a character with a reasonable category code. Here, 11 (letter) and 12 (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from \texttt{'} (ASCII 39) to \texttt{-} (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the
number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on \_\_tl_set_rescan_multi:nNN.

Otherwise, once a valid character is found (let us use ‘ in this explanation) run some code very similar to \_\_tl_set_rescan_multi:nNN but with ‘ added at both ends of the input. Of course, we need to define the auxiliary \_\_tl_set_rescan_single:NNww on the fly to remove the additional ‘ that is just before :: (by which we mean \c__tl_rescan_marker_tl). Note that the argument must be delimited by ‘ with the current catcode; this is done thanks to \char_generate:nn. Yet another issue is that the res-canned token list may contain a comment character, in which case the ‘ we expected is not there. We fix this as follows: rather than just :: we set \everyeof to ::{(code1)} ‘::{(code2)} \s__tl_stop. The auxiliary \_\_tl_set_rescan_single:NNww runs the o-expanding assignment, expanding either {code1} or {code2} before its the main argument #3. In the typical case without comment character, {code1} is expanded, removing the leading ‘. In the rarer case with comment character, {code2} is expanded, calling \_\_tl_set_rescan_single_aux:w, which removes the trailing ::{(code1)} and the leading ‘.

\cs_new_protected:Npn \_\_tl_set_rescan:nNN #1
\{\int_compare:nNnTF \tex_newlinechar:D < 0
\{ \use_ii:nn \}
\{ \exp_args:Nnf \tl_if_in:nnTF {#1} \char_generate:nn { \tex_newlinechar:D } { 12 } \}
\{ \_\_tl_set_rescan_multi:nNN \}
\{ \int_set:Nn \tex_endlinechar:D { -1 } \_\_tl_set_rescan_single:nnNN { ‘ ‘ \}
\}
\}
\cs_new_protected:Npn \_\_tl_set_rescan_single:nnNN #1
\{\int_compare:nNnTF \int_compare:nNnTF \\int_compare:nNnTF \int_compare:nNnTF
\{ \char_value_catcode:n {#1} / 2 \} = 6
\{ \\exp_args:Nof \_\_tl_set_rescan_single_aux:nnnNN \c__tl_rescan_marker_tl
\char_generate:nn {#1} \{ \char_value_catcode:n {#1} \}
\}
\{ \\exp_args:Nf \_\_tl_set_rescan_single:nnNN \\int_eval:n { #1 + 1 } \}
\}
\}
\cs_new_protected:Npn \_\_tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
\{ \\tex_everyeof:D
\}

393
8.6 Modifying token list variables

All of the replace functions call \_\_tl_replace:NnNNnN with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (\_\_tl_replace_next:w) or stops (\_\_tl_replace_wrap:w) after the first replacement. Next comes an x-type assignment function \tl_set:Nx or \tl_gset:Nx for local or global replacements. Finally, the three arguments ⟨tl var⟩ {⟨pattern⟩} {⟨replacement⟩} provided by the user. When describing the auxiliary functions below, we denote the contents of the ⟨tl var⟩ by ⟨token list⟩.

To implement the actual replacement auxiliary \_\_tl_replace_auxii:nNNnNnN we need a ⟨delimiter⟩ with the following properties:

- all occurrences of the ⟨pattern⟩ #6 in “⟨token list⟩ ⟨delimiter⟩” belong to the ⟨token list⟩ and have no overlap with the ⟨delimiter⟩,
- the first occurrence of the ⟨delimiter⟩ in “⟨token list⟩ ⟨delimiter⟩” is the trailing ⟨delimiter⟩.

(End definition for \_\_tl_set_rescan:n and others.)

(End definition for \_\_tl_set_rescan:nNn and others. These functions are documented on page 45.)
We first find the building blocks for the (delimiter), namely two tokens ⟨A⟩ and ⟨B⟩ such that ⟨A⟩ does not appear in #6 and #6 is not ⟨B⟩ (this condition is trivial if #6 has more than one token). Then we consider the delimiters “⟨A⟩” and “⟨A⟩⟨B⟩⟨A⟩n ⟨B⟩”, for n ≥ 1, where ⟨A⟩n denotes n copies of ⟨A⟩, and we choose as our (delimiter) the first one which is not in the (token list).

Every delimiter in the set obeys the first condition: #6 does not contain ⟨A⟩ hence cannot be overlapping with the (token list) and the (delimiter), and it cannot be within the (delimiter) since it would have to be in one of the two ⟨B⟩ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the (delimiter) we choose does not appear in the (token list). Additionally, the set of delimiters is such that a (token list) of n tokens can contain at most O(n/2) of them, hence we find a (delimiter) with at most O(n/2) tokens in a time at most O(n3/2). Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “⟨A⟩” in the list of delimiters to try, so that the (delimiter) is simply \q__tl_mark in the most common situation where neither the (token list) nor the (pattern) contains \q__tl_mark.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty (pattern) #6 is an error, and if #1 is absent from both the (token list) #5 and the (pattern) #6 then we can use it as the (delimiter) through \_tl_replace_auxii:nNNNnn {#1}. Otherwise, we end up calling \_tl_replace:NnnNNNnn repeated with the first two arguments \q__tl_mark (?), \? {??}, \?? {???}, and so on, until #6 does not contain the control sequence #1, which we take as our ⟨A⟩. The argument #2 only serves to collect 7 characters for #1. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose ⟨B⟩ to be \q__tl_nil or \q__tl_stop such that it is not equal to #6.

The \_tl_replace_auxii:NnnNNNnn auxiliary receives ⟨⟨A⟩⟩ and ⟨⟨A⟩n⟨B⟩⟩ as its arguments, initially with n = 1. If “⟨A⟩ ⟨A⟩n⟨B⟩ ⟨A⟩n⟨B⟩” is in the (token list) then increase n and try again. Once it is not anymore in the (token list) we take it as our (delimiter) and pass this to the auxii auxiliary.
The auxiliary \_\_tl_replace_auxii:nNNNnn receives the following arguments:

\{⟨delimiter⟩\} ⟨function⟩ ⟨assignment⟩
⟨tl var⟩ ⟨pattern⟩ \{⟨replacement⟩\}

All of its work is done between \group_align_safe_begin: and \group_align_safe_end: to avoid issues in alignments. It does the actual replacement within \#3 \#4 \{\ldots\}, an x-expanding ⟨assignment⟩ \#3 to the ⟨tl var⟩ \#4. The auxiliary \_\_tl_replace_next:w is called, followed by the ⟨token list⟩, some tokens including the ⟨delimiter⟩ \#1, followed by the ⟨pattern⟩ \#5. This auxiliary finds an argument delimited by \#5 (the presence of a trailing \#5 avoids runaway arguments) and calls \_\_tl_replace_wrap:w to test whether this \#5 is found within the ⟨token list⟩ or is the trailing one.

If on the one hand it is found within the ⟨token list⟩, then \#1 cannot contain the ⟨delimiter⟩ \#1 that we worked so hard to obtain, thus \_\_tl_replace_wrap:w gets \#1 as its own argument \#1, and protects it against the x-expanding assignment. It also finds \exp_not:n as \#2 and does nothing to it, thus letting through \exp_not:n \{⟨replacement⟩\} into the assignment. Note that \_\_tl_replace_next:w and \_\_tl_replace_wrap:w are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of \exp_not:o and \use_none:nn, rather than simply \exp_not:n. Afterwards, \_\_tl_replace_next:w is called to repeat the replacement, or \_\_tl_replace_wrap:w if we only want a single replacement. In this second case, \#1 is the ⟨remaining tokens⟩ in the ⟨token list⟩ and \#2 is some ⟨ending code⟩ which ends the assignment and removes the trailing tokens \#5 using some \if_false: \{ \fi: \} trickery because \#5 may contain any delimiter.

If on the other hand the argument \#1 of \_\_tl_replace_next:w is delimited by the trailing ⟨pattern⟩ \#5, then \#1 is “{ } { } ⟨token list⟩ ⟨delimiter⟩ ⟨ending code⟩”, hence \_\_tl_replace_wrap:w finds “{ } { } ⟨token list⟩” as \#1 and the ⟨ending code⟩ as \#2. It leaves the ⟨token list⟩ into the assignment and unbraces the ⟨ending code⟩ which removes what remains (essentially the ⟨delimiter⟩ and ⟨replacement⟩).
\\exp_after:wN \__tl_replace_next:w
4538\exp_after:wN { \exp_after:wN }
4539\exp_after:wN { \exp_after:wN }
4540#4
4541#1
4542{
4543\if_false: { \fi:
4544\exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4545}
4546#5
4547\group_align_safe_end:
4548}
4549\cs_new_eq:NN \__tl_replace_wrap:w ?
4550\cs_new_eq:NN \__tl_replace_next:w ?
8.7 Token list conditionals
\tl_if_blank_p:n
4551\tl_if_blank_p:V
4552\tl_if_blank:o
4553\__tl_if_blank_p:NNw
\__tl_if_empty_if:o is a fast emptiness test, converting its argument to a string (after one expansion) and using the test \if_meaning:w \q__tl_nil \ldots \q__tl_nil.
4556\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
4557\prg_return_true:
4558\else:
4559\prg_return_false:
4563\tl_if_blank_all:cn
4564\tl_if_blank_all:cn
4565\cs_new_protected:Npp \tl_remove_all:NN #1#2
4566\cs_new_protected:Npp \tl_gremove_all:NN #1#2
4567\cs_generate_variant:Nn \tl_remove_all:NN { c }
4568\cs_generate_variant:Nn \tl_gremove_all:NN { c }
8.7 Token list conditionals
\tl_if_blank_p:n
\tl_if_blank_p:V
\tl_if_blank:o
\__tl_if_blank_p:NNw
\__tl_if_empty_if:o is a fast emptiness test, converting its argument to a string (after one expansion) and using the test \if_meaning:w \q__tl_nil \ldots \q__tl_nil.
\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
\prg_return_true:
\else:
\prg_return_false:
\fi:
\}
These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
{ \if_meaning:w #1 \c_empty_tl \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \tl_if_empty:N { c } { p , T , F , TF }

(End definition for \tl_if_empty:NTF and \__tl_if_empty_p:NNw. This function is documented on page 46.)

\tl_if_empty_p:N Convert the argument to a string: this is empty if and only if the argument is. Then \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
{ \exp_after:wN \if_meaning:w \exp_after:wN \q__tl_nil \__kernel_tl_to_str:w \exp_after:wN {#1} \q__tl_nil \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \tl_if_empty:n { V } { p , TF , T , F }

(End definition for \tl_if_empty:nTF. This function is documented on page 47.)

The auxiliary function \__tl_if_empty_if:o is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl_if_empty:nTF, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don’t put \prg_return_true: and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code.
\cs_new:Npn \__tl_if_empty_if:o #1
{ \exp_after:wN \if_meaning:w \exp_after:wN \q__tl_nil \__kernel_tl_to_str:w \exp_after:wN (#1) \q__tl_nil \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
\tl_if_eq_p:NN  \tl_if_eq_p:Nc  \tl_if_eq_p:cN  \tl_if_eq_p:cc
\tl_if_eq:NN  \tl_if_eq:Nc  \tl_if_eq:cN  \tl_if_eq:cc
\tl_if_eq:nn  \tl_if_eq:nnTF
\tl_if_in:Nn  \tl_if_in:cn
\tl_if_in:nnTF  \tl_if_in:nnFN
\tl_if_in:nnTF  \tl_if_in:nnFN
\tl_if_in:nnTF  \tl_if_in:nnFN
\tl_if_in:nnTF  \tl_if_in:nnFN
\tl_if_in:nnTF

Returns \texttt{c_true_bool} if and only if the two token list variables are equal.

\begin{verbatim}
\newcommand{\tl_if_eq_p}{\newcommand{\tl_if_eq_p}{\prg_new_conditional:NpNN\tl_if_eq_p{\prg_return_true}{\if_meaning:w#1#2\prg_return_true}{\prg_return_false}}\prg_generate_conditional_variant:Nnn\tl_if_eq_p{\tl_if_eq_p}{\tl_if_eq_p}{p,T,F,TF}}
\end{verbatim}

\begin{verbatim}
\newcommand{\tl_if_eq}{\newcommand{\tl_if_eq}{\exp_after:wN\tl_set:Nn\l__tl_internal_a_tl{#1}\tl_set:Nn\l__tl_internal_b_tl{#2}\if_meaning:w\l__tl_internal_a_tl\l__tl_internal_b_tl\prg_return_true}{\prg_return_false}}\tl_new:N\l__tl_internal_a_tl\tl_new:N\l__tl_internal_b_tl
\end{verbatim}

\begin{verbatim}
\newcommand{\tl_if_in}{\newcommand{\tl_if_in}{\exp_args:No\tl_if_in:nnTF}}\newcommand{\tl_if_in}{\exp_args:No\tl_if_in:nnFN}\newcommand{\tl_if_in}{\exp_args:No\tl_if_in:nnTF}\prg_generate_conditional_variant:Nnn\tl_if_in\tl_if_in{\tl_if_in}{\tl_if_in}{c}{p,T,F,TF}}
\end{verbatim}

(End definition for \tl_if_eq:nnTF and \tl_if_eq:nnFN. This function is documented on page 47.)

(End definition for \tl_if_eq:NNTF. This function is documented on page 47.)

(End definition for \tl_if_in:nnTF, \tl_if_in:nnFN, \tl_if_in:nnTF, and \tl_if_in:nnFN. This function is documented on page 47.)

(End definition for \tl_if_in:nnTF. This function is documented on page 47.)
Once more, the test relies on the emptiness test for robustness. The function \_\_tl_-_tmp:w removes tokens until the first occurrence of #2. If this does not appear in #1, then the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See \tl_if_empty:nTF for details on the emptiness test.

Treating correctly cases like \tl_if_in:nnTF \{a state\}{states\}, where #1#2 contains #2 before the end, requires special care. To cater for this case, we insert { } between the two token lists. This marker may not appear in #2 because of \TeX{} limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The \if_false: constructions are a faster way to do \group_align_safe_begin: and \group_align_safe_end:. The \scan_stop: ensures that f-expanding \tl_if_in:nn does not lead to unbalanced braces.

\begin{verbatim}
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
\prg_generate_conditional_variant:Nnn \tl_if_in:nn
\prg_generate_conditional_variant:Nnn \tl_if_in:nn
\end{verbatim}

(End definition for \tl_if_in:nnTF. This function is documented on page 47.)

\begin{verbatim}
\tl_if_novalue_p:n
\tl_if_novalue:n
\_\_tl_if_novalue:w
\prg_new_protected_conditional:Npnn \tl_if_novalue:p:n #1
\prg_new_protected_conditional:Npnn \tl_if_novalue:n #1
\prg_new_protected_conditional:Npnn \tl_if_novalue:n  
\exp_args:No \tl_if_novalue:nT 
\exp_args:No \tl_if_novalue:nF 
\exp_args:No \tl_if_novalue:nTF 
\end{verbatim}

(End definition for \tl_if_novalue:nTF and \_\_tl_if_novalue:w. This function is documented on page 47.)

\begin{verbatim}
\_\_tl_if_single:p:n
\_\_tl_if_single:n
\cs_set_protected:Npn \tl_if_single:p:n { \exp_args:No \tl_if_single:p:n } 
\cs_set_protected:Npn \tl_if_single:n { \exp_args:No \tl_if_single:n } 
\cs_set_protected:Npn \tl_if_single:nT { \exp_args:No \tl_if_single:nT } 
\cs_set_protected:Npn \tl_if_single:nF { \exp_args:No \tl_if_single:nF } 
\cs_set_protected:Npn \tl_if_single:nTF { \exp_args:No \tl_if_single:nTF } 
\end{verbatim}

(End definition for \tl_if_single:nTF. This function is documented on page 48.)
This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if #1 is blank, a single `?` if #1 has a single item, and otherwise yields some tokens ending with `??`. Then, `\tl_to_str:n` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTEX. Instead, `\__tl_if_single:nw` picks the second token in front of it. If #1 is empty, this token is the trailing `?` and the catcode test yields false. If #1 has a single item, the token is `^` and the catcode test yields true. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the catcode test yields false. Note that `\if_catcode:w` and `\__kernel_tl_to_str:w` are primitives that take care of expansion.

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.
To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then \texttt{#1} is the code to insert, \texttt{#2} is the next case to check on and \texttt{#3} is all of the rest of the cases code. That means that \texttt{#4} is the true branch code, and \texttt{#5} tidies up the spare \texttt{s__tl_mark} and the false branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that \texttt{#1} is empty, \texttt{#2} is the first \texttt{s__tl_mark} and so \texttt{#4} is the false code (the true code is mopped up by \texttt{#3}).

(End definition for \texttt{tl_case:NnTF} and others. This function is documented on page 48.)

8.8 Mapping to token lists

Expandable loop macro fortoken lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the loop terminated.

(End definition for \texttt{tl_case:NnTF} and others. This function is documented on page 48.)

8.8 Mapping to token lists

Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the loop terminated.
The inline functions are straightforward by now. We use a little trick with the counter \g__kernel_prg_map_int to make them nestable. We can also make use of \__tl_map_function:Nn from before.

\cs_new_protected:Npn \tl_map_inline:nn #1#2
{ \int_gincr:N \g__kernel_prg_map_int
  \cs_gset_protected:cpn
  { \__tl_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
  \exp_args:Nc \__tl_map_function:Nn
  { \__tl_map_ \int_use:N \g__kernel_prg_map_int :w }
  #1 \q__tl_recursion_tail
  \prg_break_point:Nn \tl_map_break:
  { \int_gdecr:N \g__kernel_prg_map_int }
}
\cs_new_protected:Npn \tl_map_inline:Nn
{ \exp_args:No \tl_map_inline:nn }
\cs_generate_variant:Nn \tl_map_inline:Nn { c }

(End definition for \tl_map_function:nN, \tl_map_function:Nn, and \__tl_map_function:Nn. These functions are documented on page 49.)

\tl_map_tokens:nn
\tl_map_tokens:Nn
\__tl_map_tokens:nn

Much like the function mapping.

\cs_new:Npn \tl_map_tokens:nn #1#2
{ \__tl_map_tokens:nn {#2} #1
  \q__tl_recursion_tail
  \prg_break_point:Nn \tl_map_break:
  { }
}
\cs_new:Npn \tl_map_tokens:Nn
{ \exp_args:No \tl_map_tokens:nn }
\cs_generate_variant:Nn \tl_map_tokens:Nn { c }
\cs_new:Npn \__tl_map_tokens:nn #1#2
{ \__tl_if_recursion_tail_break:nN {#2} \tl_map_break:
  \use:n {#1} {#2}
  \__tl_map_tokens:nn {#1}
}

(End definition for \tl_map_tokens:nn, \tl_map_tokens:Nn, and \__tl_map_tokens:nn. These functions are documented on page 49.)
\tl_map_variable:nNn  \tl_map_variable:nNn (token list) \langle tl var \rangle \langle action \rangle assigns \langle tl var \rangle to each element and executes \langle action \rangle. The assignment to \langle tl var \rangle is done after the quark test so that this variable does not get set to a quark.

\cs_new_protected:Npn \tl_map_variable:nNn #1 #2 #3
\__tl_map_variable:Nnn #2 { #3 } #1 \q__tl_recursion_tail \prg_break_point:Nn \tl_map_break: { }
\cs_new_protected:Npn \tl_map_variable:NNn
{ \exp_args:No \tl_map_variable:nNn }
\cs_new_protected:Npn \__tl_map_variable:Nnn #1 #2 #3
{ \__tl_if_recursion_tail_break:nN { #3 } \tl_map_break:
\tl_set:Nn #1 { #3 }
\use:n { #2 }
\__tl_map_variable:Nnn #1 { #2 } }
\cs_generate_variant:Nn \tl_map_variable:NNn { c }

(End definition for \tl_map_variable:nNn, \tl_map_variable:NNn, and \__tl_map_variable:Nnn. These functions are documented on page 49.)

\tl_map_break:  The break statements use the general \prg_map_break:Nn.
\cs_new:Npn \tl_map_break:
{ \prg_map_break:Nn \tl_map_break: { } }
\cs_new:Npn \tl_map_break:n
{ \prg_map_break:Nn \tl_map_break: }

(End definition for \tl_map_break: and \tl_map_break:n. These functions are documented on page 50.)

8.9 Using token lists

\tl_to_str:n  Another name for a primitive: defined in l3basics.
\tl_to_str:V  \cs_generate_variant:Nn \tl_to_str:n { V }

(End definition for \tl_to_str:n. This function is documented on page 51.)

\tl_to_str:N  These functions return the replacement text of a token list as a string.
\tl_to_str:c  \cs_new:Npn \tl_to_str:N #1 \tl_to_str:w \exp_after:wN {#1} }
\cs_generate_variant:Nn \tl_to_str:N { c }

(End definition for \tl_to_str:N. This function is documented on page 51.)

\tl_use:N  Token lists which are simply not defined give a clear \TeX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.
\cs_new:Npn \tl_use:N #1
{ \tl_if_exist:NTF #1 { #1 }
{ \tl_if_exist:NNTF #1 { #1 }
{ \__kernel_msg_expandable_error:nnn { kernel } { bad-variable } { #1 } }
} }

404
8.10 Working with the contents of token lists

\textbf{\texttt{\textbackslash tl\_count:n}}\quad Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. \texttt{\_\_tl\_count:n} grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

\begin{verbatim}
\cs_new:Npn \tl_count:n #1 {
  \int_eval:n { 0 \tl_map_function:nN {#1} \_\_tl\_count:n }
}
\cs_new:Npn \_\_tl\_count:n #1 { + 1 }
\cs_generate_variant:Nn \tl_count:n { V , o }
\cs_generate_variant:Nn \_\_tl\_count:n { c }
\end{verbatim}

(End definition for \texttt{\_\_tl\_count:n} and others. This function is documented on page 51.)

\textbf{\texttt{\_\_tl\_count:n}}\quad

\textbf{\texttt{\textbackslash tl\_count\_tokens:n}}\quad

\textbf{\texttt{\_\_tl\_act\_count\_normal:nN}}\quad

\textbf{\texttt{\_\_tl\_act\_count\_group:nn}}\quad

\textbf{\texttt{\_\_tl\_act\_count\_space:n}}\quad

The token count is computed through an \texttt{\_\_tl\_act\_count\_normal:nN} construction. Each 1+ is output to the left, into the integer expression, and the sum is ended by the \texttt{\_\_tl\_act\_end:wn} (which is technically implemented as \texttt{\textbackslash c\_zero\_int}). Somewhat a hack!

\begin{verbatim}
\cs_new:Npn \_\_tl\_act\_end:wn { 1 + }
\cs_new:Npn \_\_tl\_act\_count\_normal:nN { 2 + \tl\_count\_tokens:n {#2} + }
\end{verbatim}

(End definition for \texttt{\_\_tl\_act\_end:wn} and others. This function is documented on page 52.)

\textbf{\texttt{\textbackslash tl\_reverse\_items:n}}\quad

\textbf{\texttt{\_\_tl\_reverse\_items:nWn}}\quad

\textbf{\texttt{\_\_tl\_reverse\_items:nWn}}\quad

Reversal of a token list is done by taking one item at a time and putting it after \texttt{\textbackslash s\_\_tl\_stop}.
\end{definition}

Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial \texttt{s__tl_mark}, and whose second argument is a \texttt{⟨continuation⟩}, which receives as a braced argument \texttt{\use_none:n s__tl_mark} \texttt{(trimmed token list)}. In the case at hand, we take \texttt{\exp_not:o} as our continuation, so that space trimming behaves correctly within an \texttt{x}-type expansion.

\begin{verbatim}
\cs_set:Npn \__tl_tmp:w #1
\{ \cs_new:Npn \__tl_trim_spaces:nn #1 \exp_not:o \}
\cs_generate_variant:Nn \__tl_trim_spaces:nn { o }
\cs_new_protected:Npn \__tl_trim_spaces_auxi:w
\{ \__tl_trim_spaces_auxii:w \}
\cs_new_protected:Npn \__tl_trim_spaces_auxii:w\{ \__tl_trim_spaces_auxiii:w \}
\cs_new_protected:Npn \__tl_trim_spaces_auxiii:w\{ \__tl_trim_spaces_auxiv:w \}
\end{verbatim}

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in \texttt{\__tl_tmp:w}, which then receives a single space as its argument: \texttt{#1} is \texttt{\_\_tl_trim_spaces:nn}. Removing leading spaces is done with \texttt{\_\_tl_trim_spaces_auxi:w}, which loops until \texttt{s__tl_mark} matches the end of the token list: then \texttt{#1} is the token list and \texttt{#3} is \texttt{\_\_tl_trim_spaces_auxii:w}. This hands the relevant tokens to the loop \texttt{\_\_tl_trim_spaces_auxiii:w}, responsible for trimming trailing spaces. The end is reached when \texttt{s__tl_nil} matches the one present in the definition of \texttt{\_\_tl_trim_spaces:nn}. Then \texttt{\_\_tl_trim_spaces_auxiv:w} puts the token list into a group, with \texttt{\use_none:n} placed there to gobble a lingering \texttt{s__tl_mark}, and feeds this to the \texttt{⟨continuation⟩}. 

\begin{verbatim}
\cs_set:Npn \__tl_tmp:w #1
\{ \cs_new:Npn \__tl_trim_spaces:nn #1 \exp_not:o \}
\cs_generate_variant:Nn \__tl_trim_spaces:nn { o }
\cs_new_protected:Npn \__tl_trim_spaces_auxi:w
\{ \__tl_trim_spaces_auxii:w \}
\cs_new_protected:Npn \__tl_trim_spaces_auxii:w\{ \__tl_trim_spaces_auxiii:w \}
\cs_new_protected:Npn \__tl_trim_spaces_auxiii:w\{ \__tl_trim_spaces_auxiv:w \}
\end{verbatim}
\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn
\tl_sort:nN

Implemented in l3sort.

(End definition for \tl_sort:Nn, \tl_gsort:Nn, and \tl_sort:nN. These functions are documented on page 53.)

8.11 Token by token changes

The \__tl_act... functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only \q__tl_act_mark and \q__tl_act_stop may not appear in the token lists manipulated by \__tl_act:NNNNnn functions.

\__tl_act:NNNNnn
\__tl_act_output:n
\__tl_act_reverse_output:n
\__tl_act_loop:w
\__tl_act_normal:NwnNNN
\__tl_act_group:nwnNNN
\__tl_act_space:wwnNNN
\__tl_act_end:w

To help control the expansion, \__tl_act:NNNNnn should always be proceeded by \exp:w and ends by producing \exp_end: once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker \q__tl_act_mark is used both to avoid
losing outer braces and to detect the end of the token list more easily. The result is stored
as an argument for the dummy function \_\_tl\_act\_result:n.

\cs_new:Npn \_\_tl\_act:NNNnn \#1\#2\#3\#4\#5
\group_align_safe_begin:
\_\_tl\_act\_loop:w \#5 \q\_\_tl\_act\_mark \q\_\_tl\_act\_stop
\{\#4\} \#1 \#2 \#3
\_\_tl\_act\_result:n \{ \}

In the loop, we check how the token list begins and act accordingly. In the “normal” case,
we may have reached \q\_\_tl\_act\_mark, the end of the list. Then leave \exp_end: and
the result in the input stream, to terminate the expansion of \exp:w. Otherwise, apply
the relevant function to the “arguments”, \#3 and to the head of the token list. Then
repeat the loop. The scheme is the same if the token list starts with a group or with a
space. Some extra work is needed to make \_\_tl\_act\_space:wwnNNN gobble the space.

\cs_new:Npn \_\_tl\_act\_loop:w \#1 \q\_\_tl\_act\_stop
\{ \tl_if_head_is_N_type:nTF {\#1} \{ \_\_tl\_act\_normal:NwnNNN \}
\{ \tl_if_head_is_group:nTF {\#1} \{ \_\_tl\_act\_group:nwnNNN \}
\{ \_\_tl\_act\_space:wwnNNN \}
\}
\_\_tl\_act\_stop
\}
\cs_new:Npn \_\_tl\_act\_normal:NwnNNN \#1 \#2 \q\_\_tl\_act\_stop \#3\#4
\{ \if_meaning:w \q\_\_tl\_act\_mark \#1 \exp_after:wN \_\_tl\_act\_end:wn \fi:
\#4 \{\#3\} \#1
\_\_tl\_act\_loop:w \#2 \q\_\_tl\_act\_stop
\{\#3\} \#4
\}
\cs_new:Npn \_\_tl\_act\_end:wn \#1 \_\_tl\_act\_result:n \#2
\{ \group_align_safe_end: \exp_end: \#2 \}
\cs_new:Npn \_\_tl\_act\_group:nwnNNN \#1 \#2 \q\_\_tl\_act\_stop \#3\#4\#5
\{ \#5 \{\#3\} \{\#1\}
\_\_tl\_act\_loop:w \#2 \q\_\_tl\_act\_stop
\{\#3\} \#4 \#5
\}
\exp_last_unbraced:NNo
\cs_new:Npn \_\_tl\_act\_space:wwnNNN \c_space_tl \#1 \q\_\_tl\_act\_stop \#2\#3\#4\#5
\{ \#5 \{\#2\}
\_\_tl\_act\_loop:w \#1 \q\_\_tl\_act\_stop
\{\#2\} \#3 \#4 \#5
\}

Typically, the output is done to the right of what was already output, using \_\_tl\_act\_output:n, but for the \_\_tl\_act\_reverse functions, it should be done to the left.

408
The goal here is to reverse without losing spaces nor braces. This is done using the general internal function \__tl_act:NNn. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by \__tl_act:NNn when changing case (to record which direction the change is in), but not when reversing the tokens.

This reverses the list, leaving \exp_stop_f: in front, which stops the f-expansion.

Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping to a list. The empty brace groups in \tl_-_head:h ensure that a blank argument gives an empty result. The result is returned within the \unexpanded primitive. The approach here is to use \if_false: to allow us to use } as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it’s own code. Using a marker, we can see if what we are
grabbing is exactly the marker, or there is anything else to deal with. Is there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in http://tex.stackexchange.com/a/70168.

To correctly leave the tail of a token list, it's important not to absorb any of the tail part as an argument. For example, the simple definition

\begin{verbatim}
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\end{verbatim}

would give the wrong result for \tl_tail:n { a { bc } } (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (i.e. that the argument is not blank) and assuming there is to dispose of the first item. As with \tl_head:n, the result is protected from further expansion by \unexpanded. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

(End definition for \tl_head:n and others. These functions are documented on page 54.)
Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in \(\texttt{tl_if_head_eq_charcode:nN}\). Here, \(\texttt{tl_head:w}\) yields the first token of the token list, then passed to \(\texttt{exp_not:N}\).

\[
\texttt{\textbackslash if_charcode:w} \\
\texttt{\exp_after:wN \exp_not:N \tl_head:w \#1 \textbackslash q_nil \textbackslash q_stop} \\
\texttt{\exp_not:N \#2}
\]

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the true branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use \(\texttt{str_head:n}\) to access it (this works even if it is a space character). An empty argument results in \(\texttt{tl_head:w}\) leaving two tokens: ? which is taken in the \(\texttt{if_charcode:w}\) test, and \(\texttt{use_none:nn}\), which ensures that \(\texttt{prg_return_false}\) is returned regardless of whether the charcode test was true or false.

\[
\texttt{prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN \#1\#2 \{ p , T , F , TF \}} \\
\texttt{\{ \}} \\
\texttt{\texttt{\if_charcode:w}} \\
\texttt{\exp_not:N \#2} \\
\texttt{\tl_if_head_is_N_type:nTF \{ \#1 ? \}} \\
\texttt{\{ \exp_after:wN \exp_not:N} \\
\texttt{\_\_\tl_tl_head:w \#1 \{ \textbackslash use_none:nn \} \_\_\tl_stop}\texttt{\}} \\
\texttt{\{ \str_head:n \{\#1\} \}} \\
\texttt{\prg_return_true:} \\
\texttt{\else:} \\
\texttt{\prg_return_false:} \\
\texttt{\fi:} \\
\texttt{\}} \\
\texttt{\prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN \#1 \textbackslash use_none:nn \{ f \} \{ p , TF , T , F \}}
\]

For \(\texttt{tl_if_head_eq_catcode:nN}\), again we detect special cases with a \(\texttt{tl_if_head_is_N_type:n}\). Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either \(\texttt{c_group_begin_token}\) or \(\texttt{c_space_token}\). Again, for an empty argument, a hack is used, removing \(\texttt{prg_return_true:}\) and \(\texttt{else:}\) with \(\texttt{use_none:nn}\) in case the catcode test with the (arbitrarily chosen) ? is true.

\[
\texttt{prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN \#1 \#2 \{ p , T , F , TF \}} \\
\texttt{\{ \}} \\
\texttt{\texttt{\if_catcode:w}} \\
\texttt{\exp_not:N \#2} \\
\texttt{\tl_if_head_is_N_type:nTF \{ \#1 ? \}} \\
\texttt{\{ \exp_after:wN \exp_not:N} \\
\texttt{\_\_\tl_tl_head:w \#1 \{ \textbackslash use_none:nn \} \_\_\tl_stop}\texttt{\}} \\
\texttt{\{ \}} \\
\texttt{\prg_return_true:} \\
\texttt{\else:}
For `{\tl_if_head_eq_meaning:nN}`, again, detect special cases. In the normal case, use `{\tl_head:w}`, with no `{\exp_not:N}` this time, since `{\if_meaning:w}` causes no expansion. With an empty argument, the test is `{true}`, and `{\use_none:nnn}` removes #2 and the usual `{\prg_return_true:}` and `{\else}`. In the special cases, we know that the first token is a character, hence `{\if_charcode:w}` and `{\if_catcode:w}` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

(End definition for `{\tl_if_head_eq_meaning:nNTF}` and others. These functions are documented on page 412.)
A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving \_\_tl_if_head_is_N_type:w produces \(^\) (and otherwise nothing). In the third case (begin-group token), the lines involving \exp_after:wN produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the \* to the beginning: if \#1 contains primitive conditionals, all of its occurrences must be dealt with before the \if_catcode:w tries to skip the true branch of the conditional.

\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
\_\_tl_if_head_is_N_type:w

The auxiliary’s argument is all that is before the first explicit space in \(?#1\)?~. If that is a single \? the test yields true. Otherwise, that is more than one token, and the test yields false. The work is done within braces (with an \if_false: { \fi: ... } construction) both to hide potential alignment tab characters from \TeX in a table, and to allow for removing what remains of the token list after its first space. The \exp:w and

\tl_if_head_is_group_p:n
\tl_if_head_is_group:nTF
\tl_if_head_is_space_p:n
\tl_if_head_is_space:nTF
\_\_tl_if_head_is_space:w

\cs_new:Npn \__tl_if_head_is_space:w #1 ~
\tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
\exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
}

(End definition for \tl_if_head_is_N_type:nTF and \_\_tl_if_head_is_N_type:w. This function is documented on page 55.)
ensure that the result of a single step of expansion directly yields a balanced
token list (no trailing closing brace).

\begin{verbatim}
\begin{prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
\end{prg_new_conditional:Npnn}
\begin{prg_new_conditional:Npnn \tl_if_head_is_space:w #1 ? #1 ? ~ }
\end{prg_new_conditional:Npnn}
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\begin{cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
\end{verbatim}

(End definition for \tl_if_head_is_space:nTF and \__tl_if_head_is_space:w. This function is documented on page 55.)

### 8.13 Using a single item

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then \__tl_if_recursion_tail\break:nN terminates the loop, and returns nothing at all.

\begin{verbatim}
\begin{cs_new:Npn \tl_item:nn { \exp_args:No \tl_item:nn }
\begin{cs_generate_variant:Nn \tl_item:nn { c }
\end{verbatim}

(End definition for \tl_item:nn and others. These functions are documented on page 56.)

Importantly \tl_item:nn only evaluates its argument once.

\begin{verbatim}
\begin{cs_new:Npn \tl_rand_item:n #1
\begin{cs_new:Npn \tl_rand_item:N
\begin{cs_new:Npn \tl_rand_item:c
\end{verbatim}

414
To avoid checking for the end of the token list at every step, start by counting the number \( l \) of items and “normalizing” the bounds, namely clamping them to the interval \([0, l]\) and dealing with negative indices. More precisely, \( __\text{tl_range_items}:nnNn \) receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either \( __\text{tl_range}:w \) or the token list itself. If nothing should be kept, leave \{\}: this stops the f-expansion of \( \text{tl_head}:f \) and that function produces an empty result. Otherwise, repeatedly call \( __\text{tl_range_skip}:w \) to delete \#1 items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version \( __\text{tl_range_braced}:w \) sets up \( __\text{tl_range_collect_braced}:w \) which stores items one by one in an argument after the semicolon. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and \( \text{tl_head}:f \) cleans up and gives the result in \exp_not:n.
\if_int_compare:w #1 > 0 \exp_stop_f:
  \exp_after:wN \__tl_range_skip:w
\int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
\else:
  \exp_after:wN \exp_end:
\fi:

{ \cs_new:Npn \__tl_range:w #1 ; #2 }
{ \exp_after:wN \__tl_range_collect:nn }
{ \__tl_range_skip_spaces:n #1 }
{ \cs_new:Npn \__tl_range_collect:nn #1#2 }
{ \__tl_range_collect_space:nw #1 ~ }
{ \__tl_range_collect_N:nN }
{ \__tl_range_collect_group:nn }
\cs_generate_variant:Nn \__tl_range_collect:nn { ff }

(End definition for \tl_range:Nnn and others. These functions are documented on page 57.)

\__tl_range_normalize:nn
This function converts an \langle index \rangle argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the \langle index \rangle #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

\cs_new:Npn \__tl_range_normalize:nn #1#2
(End definition for \_\_tl_range_normalize:nn.)

\_\_tl_show:NN
\_\_tl_show:n
\_\_tl_show:w

8.14 Viewing token lists

Showing token list variables is done after checking that the variable is defined (see \_\_kernel_register_show:N).

\cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
\cs_generate_variant:Nn \tl_show:N { c }
\cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
\cs_generate_variant:Nn \tl_log:N { c }
\cs_new_protected:Npn \__tl_show:NN #1#2
\tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s__tl_stop }

(End definition for \_\_tl_show:N, \_\_tl_log:N, and \_\_tl_show:NN. These functions are documented on page 58.)

\__tl_show:n

Many show functions are based on \_\_tl_show:n. The argument of \_\_tl_show:n is line-wrapped using \iow_wrap:nnnN but with a leading >- and trailing period, both removed before passing the wrapped text to the \showtokens primitive. This primitive shows the result with a leading >- and trailing period.

The token list \_\_tl_internal_a_tl containing the result of all these manipulations is displayed to the terminal using \tex_showtokens:D and an odd \exp_after:wN which expand the closing brace to improve the output slightly. The calls to \_\_kernel_iow_with:Nnn ensure that the \newlinechar is set to 10 so that the \iownewline: inserted by the line-wrapping code are correctly recognized by \TeX, and that \errorcontextlines is -1 to avoid printing irrelevant context.
5258 \_\_kernel_iow_with:Nnn \tex_newlinechar:D \{ 10 \}
5259 \{
5260 \_\_kernel_iow_with:Nnn \tex_errorcontextlines:D \{ -1 \}
5261 \{
5262 \tex_showntokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5263 \{ \exp_after:wN \l__tl_internal_a_tl \}
5264 \}
5265 \}
5266 \cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}

(End definition for \_tl_show:n, __tl_show:n, and __tl_show:w. This function is documented on page 58.)\tl_log:n
Logging is much easier, simply line-wrap. The >> and trailing period is there to match the output of \_tl_show:n.
5268 \cs_new_protected:Npn \tl_log:n \#1
5269 \{ \iow_wrap:nnnN { > ~ \tl_to_str:n \{#1\} . } \} \l__tl_stop \{#2\}

(End definition for \tl_log:n. This function is documented on page 58.)

8.15 Internal scan marks
\s__tl_nil \s__tl_mark \s__tl_stop
Internal scan marks. These are defined here at the end because the code for \scan_new:N depends on some \l3tl functions.
5270 \scan_new:N \s__tl_nil
5271 \scan_new:N \s__tl_mark
5272 \scan_new:N \s__tl_stop

(End definition for \s__tl_nil, \s__tl_mark, and \s__tl_stop.)

8.16 Scratch token lists
\g_tmpa_tl \g_tmpb_tl
Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can’t count on other macros not to redefine them from under you.
5273 \tl_new:N \g_tmpa_tl
5274 \tl_new:N \g_tmpb_tl

(End definition for \g_tmpa_tl and \g_tmpb_tl. These variables are documented on page 59.)
\l_tmpa_tl \l_tmpb_tl
These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.
5275 \tl_new:N \l_tmpa_tl
5276 \tl_new:N \l_tmpb_tl

(End definition for \l_tmpa_tl and \l_tmpb_tl. These variables are documented on page 59.)
9 l3str implementation

\section*{9.1 Internal auxiliaries}

\begin{verbatim}
\s__str_mark \s__str_stop
\end{verbatim}

Internal scan marks.

\begin{verbatim}
\scan_new:N \s__str_mark
\scan_new:N \s__str_stop
\end{verbatim}

(End definition for \s__str_mark and \s__str_stop.)

\begin{verbatim}
\str_use_none_delimit_by_s_stop:w
\str_use_i_delimit_by_s_stop:nw
\end{verbatim}

Functions to gobble up to a scan mark.

\begin{verbatim}
\cs_new:Npn \__str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
\cs_new:Npn \__str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
\end{verbatim}

(End definition for \__str_use_none_delimit_by_s_stop:w and \__str_use_i_delimit_by_s_stop:nw.)

\begin{verbatim}
\q__str_recursion_tail \q__str_recursion_stop
\end{verbatim}

Internal recursion quarks.

\begin{verbatim}
\quark_new:N \q__str_recursion_tail
\quark_new:N \q__str_recursion_stop
\end{verbatim}

(End definition for \q__str_recursion_tail and \q__str_recursion_stop.)

\begin{verbatim}
\str_new:N \str_new:c \str_use:N \str_use:c \str_clear:N \str_clear:c \str_gclear:N \str_gclear:c \str_clear_new:N \str_clear_new:c \str_gclear_new:N \str_gclear_new:c \str_set_eq:NN \str_set_eq:cN \str_set_eq:Nc \str_set_eq:cc \str_gset_eq:NN \str_gset_eq:cN \str_gset_eq:Nc \str_gset_eq:cc \str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc
\end{verbatim}

Functions to query recursion quarks.

\begin{verbatim}
\__kernel_quark_new_test:N \__str_if_recursion_tail_break:NN \__str_if_recursion_tail_stop_do:Nn
\end{verbatim}

(End definition for \__str_if_recursion_tail_break:NN and \__str_if_recursion_tail_stop_do:Nn.)

\subsection*{9.2 Creating and setting string variables}

A string is simply a token list. The full mapping system isn’t set up yet so do things by hand.

\begin{verbatim}
\cs_set_protected:Npn \__str_tmp:n #1
\tl_if_blank:nF {#1}
\cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
\exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
\__str_tmp:n
{ new }
{ use }
{ clear }
{ gclear }
{ clear_new }
{ gclear_new }
{ }
\group_end:
\cs_new_eq:NN \tl_set_eq:NN \str_set_eq:NN \str_set_eq:cN \str_set_eq:Nc \str_set_eq:cc \str_gset_eq:NN \str_gset_eq:cN \str_gset_eq:Nc \str_gset_eq:cc \str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc
\end{verbatim}

419
Simply convert the token list inputs to strings.

9.3 Modifying string variables

Start by applying \texttt{\textbackslash tl\_to\_str:n} to convert the old and new token lists to strings, and also apply \texttt{\textbackslash tl\_to\_str:N} to avoid any issues if we are fed a token list variable. Then the code is a much simplified version of the token list code because neither the delimiter nor the replacement can contain macro parameters or braces. The delimiter \texttt{\textbackslash s\_\textbackslash str\_mark} cannot appear in the string to edit so it is used in all cases. Some x-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. The ending code is much simplified and does not need to hide in braces.
\cs_new_protected:Npn \__str_replace:NNNnn \tl_gset:Nx }
\cs_generate_variant:Nn \str_replace_once:Nnn { c }
\cs_generate_variant:Nn \str_greplace_once:Nnn { c }
\cs_generate_variant:Nn \str_replace_all:Nnn { c }
\cs_generate_variant:Nn \str_greplace_all:Nnn { c }
\cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
{
\tl_if_empty:nTF {#4}
{
\__kernel_msg_error:nnx { kernel } { empty-search-pattern } {#5}
}
{
\use:x
{
\exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
{ \tl_to_str:N #3 }
{ \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
}
}
}
\cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
{
\cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
#2 #3
{
\__str_replace_next:w
#4
\__str_use_none_delimit_by_s_stop:w
#5
\s__str_stop
}
}
\cs_new_eq:NN \__str_replace_next:w ?
(End definition for \str_replace_all:Nnn and others. These functions are documented on page 62.)

Removal is just a special case of replacement.
\cs_new_protected:Npn \str_remove_once:Nn #1#2
{ \str_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \str_gremove_once:Nn #1#2
{ \str_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \str_remove_once:Nn { c }
\cs_generate_variant:Nn \str_gremove_once:Nn { c }

(End definition for \str_remove_once:Nn and \str_gremove_once:Nn. These functions are documented on page 62.)

Removal is just a special case of replacement.
\cs_new_protected:Npn \str_remove_all:Nn #1#2
{ \str_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \str_gremove_all:Nn #1#2
{ \str_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \str_remove_all:Nn { c }
\cs_generate_variant:Nn \str_gremove_all:Nn { c }

421
End definition for \str_remove_all:Nn and \str_gremove_all:Nn. These functions are documented on page 62.

9.4 String comparisons

More copy-paste!

\_\_str_if_eq:nn
\_\_str_escape:n

String comparisons rely on the primitive \pdfstrcmp if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not always use the primitive, the correct approach is to wrap up in a function with defined behaviour. That’s done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is loaded in \l3bootstrap. The need to detokenize and force expansion of input arises from the case where a \# token is used in the input, e.g. \_\_str_if_eq:nn \# \{ \tl_to_str:n \# \}, which otherwise would fail as \text_luaescapestring:D does not double such tokens.

\cs_new:Npn \_\_str_if_eq:nn #1#2 { \tex_strcmp:D {#1} {#2} }
\cs_if_exist:NT \tex_luatexversion:D { \cs_set_eq:NN \lua_escape:e \tex_luaescapestring:D \cs_set_eq:NN \lua_now:e \tex_directlua:D \cs_set:Npn \_\_str_if_eq:nn #1#2 { \lua_now:e { l3kernel.strcmp ( " \_\_str_escape:n \{#1\} " , " \_\_str_escape:n \{#2\} " ) } } \cs_new:Npn \_\_str_escape:n #1 { \lua_escape:e { \use:e { {#1} } } } (End definition for \_\_str_if_eq:nn and \_\_str_escape:n.)

\str_if_p:nn
\str_if_p:Vn
\str_if_p:on
\str_if_p:nV
\str_if_p:no
\str_if_p:ee
\str_if_p:mm
\str_if_eq:nTF
\str_if_eq:VnTF
\str_if_eq:onTF
\str_if_eq:nVTF
\str_if_eq:noVTF
\str_if_eq:VTF

Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The nn and xx versions are created directly as this is most efficient.
\prg_new_conditional:Npnn \str_if_eq:nn #1#2 \{ p , T , F , TF \}  
\prg_if_int_compare:w \__str_if_eq:nn \exp_not:n {#1} \exp_not:n {#2} \}= 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi: 
\prg_generate_conditional_variant:Nnn \str_if_eq:nn \{ V , v , o , nV , no , VV , nv \} \{ p , T , F , TF \} 
\prg_new_conditional:Npnn \str_if_eq:ee #1#2 \{ p , T , F , TF \}  
\prg_if_int_compare:w \__str_if_eq:nn \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi: 
\prg_generate_conditional_variant:Nnn \str_if_eq:ee \{ c , Nc , cc \} \{ T , F , TF , p \} 
\str_if_eq_p:NN \str_if_eq_p:Nc \str_if_eq_p:cN \str_if_eq_p:cc \str_if_eq:NN TF \str_if_eq:Nc TF \str_if_eq:cN TF \str_if_eq:cc TF 
\prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 \{ T , F , TF \}  
\use:x \{ \tl_if_in:nnTF \tl_to_str:N #1 \tl_to_str:n {#2} \} \{ \prg_return_true: \} \{ \prg_return_false: \} 
\prg_generate_conditional_variant:Nnn \str_if_in:Nn \{ c \} \{ T , F , TF \} 
\prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 \{ T , F , TF \}  
\use:x \{ \tl_if_in:nnTF \tl_to_str:n #1 \tl_to_str:n #2 \} \{ \prg_return_true: \} \{ \prg_return_false: \} 
\str_case:nn \str_case:Vn \str_case:on \str_case:nV \str_case:nv \str_case:nn TF \str_case:Vn TF \str_case:on TF \str_case:nV TF \str_case:nv TF \str_case_e:nn \str_case:e:nn TF 
\str_case_e:nw \__str_case:nnTF \__str_case_e:nnTF \__str_case_end:nw 
\cs_new:Npn \str_case:nn #1#2 
\exp:w

(End definition for \str_if_eq:nnTF. This function is documented on page 63.)
9.5 Mapping to strings

The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require \__kernel_str_to_other:n, quadratic in the string length. To deal with spaces in that case, \__str_map_function:w replaces the following space by a braced space and a further call to itself. These are received by \__str_map_function:inN, which passes the space to #1 and calls \__str_map_function:w to deal with the next space. The space before the braced space allows to optimize the \q__str_recursion_tail test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when \TeX{} tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is \q__str_recursion_tail (also by converting 9 spaces at a time in the \str_map_function:nN case).

For the map_variable functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.
9.6 Accessing specific characters in a string

First apply \texttt{\tl_to_str:n}, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the \texttt{\s__str_mark} and \texttt{\s__str_stop} markers. The end is detected when \texttt{\_str_to_other_loop:w} finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then \texttt{\_str_to_other_end:w} is called, and finds the result between \texttt{\s__str_mark} and the first A (well, there is also the need to remove a space).

\begin{verbatim}
\cs_new:Npn \__kernel_str_to_other:n #1
{ \exp_after:wN \__str_to_other_loop:w \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop }
\end{verbatim}

(End definition for \texttt{\str_map_function:NN} and others. These functions are documented on page 64.)
The difference with \texttt{\_kernel\_str\_to\_other:nn} is that the converted part is left in the input stream, making these commands only restricted-expandable.

The \texttt{\str\_item:nn} hands its argument with spaces escaped to \texttt{\_str\_item:nn}, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The \texttt{\str\_item\_ignore\_spaces:nn} function does not escape spaces, which are thus ignored by \texttt{\_str\_item:nn} since everything else is done with undelimited arguments. Evaluate the \langle index \rangle argument #2 and count characters in the string, passing
those two numbers to \_\_str_item:w for further analysis. If the \langle index \rangle is negative, shift it by the \langle count \rangle to know the how many character to discard, and if that is still negative give an empty result. If the \langle index \rangle is larger than the \langle count \rangle, give an empty result, and otherwise discard \langle index \rangle \text{–} 1 characters before returning the following one. The shift by \text{–}1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the \langle index \rangle is zero.

\begin{verbatim}
\cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
\cs_generate_variant:Nn \str_item:Nn { c }
\cs_new:Npn \str_item:nn #1#2
{ \exp_after:wN \__str_item:w \int_value:w \int_eval:n {#2} \exp_after:wN ;
  \int_value:w \__str_count:n {#1} ;
  #1 \s__str_stop }
\cs_new:Npn \__str_item:w #1; #2;
{ \int_compare:nNnTF {#1} < 0
  { \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w #1 ; { }
    \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w #2 ; { }
  }
  { \int_compare:nNnTF {#1} > {#2}
    { \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w #2 #1 ; { }
      \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w #1 ; { }
    }
    { \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w { } ; { }
      \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w { } ; { }
    }
  }
}
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \str_item_ignore_spaces:nn #1
{ \exp_after:wN \__str_item:nn { \tl_to_str:n {#1} } }
\cs_new:Npn \__str_item:nn #1#2
{ \exp_after:wN \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w #1 ; { }
  \__str_use_i_delimit_by_s_stop:nw \exp:w \exp_after:wN \__str_skip_exp_end:w { } ; { }
}
\end{verbatim}

(End definition for \str_item:Nn and others. These functions are documented on page 67.)

\begin{verbatim}
\__str_skip_exp_end:w \__str_skip_loop:wnnnnnnnnn \__str_skip_end:w \__str_skip_end:wnnnnnnnnn
\end{verbatim}

Removes \text{max}(\#1,0) characters from the input stream, and then leaves \exp_end:. This should be expanded using \exp:w. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the \texttt{\if_case:w} construction leaves between 0 and 8 times the \texttt{\or:} control sequence, and those \texttt{\or:} become arguments of \_\_str_skip_end:wnnnnnnnnnn. If the number of characters to remove is 6, say, then there
are two \or: left, and the 8 arguments of \__str_skip_end:NNNNNNNN are the two \or:, and 6 characters from the input stream, exactly what we wanted to remove. Then close the \if_case:w conditional with \fi:, and stop the initial expansion with \exp_end: (see places where \__exp_end:w is called).

\cs_new:Npn \__str_skip_exp_end:w #1;
\begin{verbatim}
  \if_int_compare:w #1 > 8 \exp_stop_f:
    \exp_after:wN \__str_skip_loop:wNNNNNNNN
  \else:
    \exp_after:wN \__str_skip_end:w
    \int_value:w \int_eval:w
    \fi:
    #1 ;
\end{verbatim}
\cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
\begin{verbatim}
  \exp_after:wN \__str_skip_exp_end:w
  \int_value:w \int_eval:n { #1 - 8 } ;
\end{verbatim}
\cs_new:Npn \__str_skip_end:w #1 ;
\begin{verbatim}
  \exp_after:wN \__str_skip_end:NNNNNNNN
  \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or: 
\end{verbatim}
\cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the \langle start index \rangle, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

\cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
\cs_generate_variant:Nn \str_range:Nnn { c }
\cs_new:Npn \str_range:nnn #1#2#3
\begin{verbatim}
  \exp_after:wN \__str_range:w
  \int_value:w \__str_count:n {#1} \exp_after:wN ;
  \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
  \int_value:w \int_eval:n {#3} ;
  #1 \s__str_stop
\end{verbatim}
\cs_new:Npn \str_range_ignore_spaces:nnn #1
\begin{verbatim}
  \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} }
\end{verbatim}
\cs_new:Npn \__str_range:nnn #1#2#3
\begin{verbatim}
  \exp_after:wN \__str_range:w
  \int_value:w \__str_count:n {#1} \exp_after:wN ;
  \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
  \int_value:w \int_eval:n {#3} ;
  #1 \s__str_stop
\end{verbatim}
\cs_new:Npn \__str_range:w #1; #2; #3; 
{ \exp_args:Nf \__str_range:nnw { \__str_range_normalize:nn {#2} {#1} } 
\{ \__str_range_normalize:nn {#3} {#1} \} }
\cs_new:Npn \__str_range:nnw #1#2
{ \exp_after:wN \__str_collect_delimit_by_q_stop:w 
\int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ; 
\exp:w \__str_skip_exp_end:w #1 ; }
\__str_collect_delimit_by_q_stop:w
\__str_collect_loop:wn
\__str_collect_end:wn
\__str_collect_end:nnnnnnnnw
\__str_range_normalize:nn
This function converts an \emph{(index)} argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the \emph{(index)} \#1 and the string count \#2. If \#1 is negative, replace it by \#1 + \#2 + 1, then limit to the range [0, \#2].
\cs_new:Npn \__str_range_normalize:nn \#1\#2
{ \int_eval:n 
{ \if_int_compare:w \#1 < 0 \exp_stop_f: 
\if_int_compare:w \#1 < -\#2 \exp_stop_f: 
0 
\else: 
\#1 + \#2 + 1 
\fi: 
\else: 
\if_int_compare:w \#1 < \#2 \exp_stop_f: 
\#1 
\else: 
\#2 
\fi: 
\fi: }
}
\__str_collect_delimit_by_q_stop:w
\__str_collect_loop:wn
\__str_collect_end:wn
\__str_collect_end:nnnnnnnnw
Collects max(\#1,0) characters, and removes everything else until \s__str_stop. This is somewhat similar to \__str_skip_exp_end:w, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an \if_case:w trick again, so that the 8 first arguments of \__str_collect_end:nnnnnnnnw are some \or:, followed by an \fi:, followed by \#1 characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the \or: disappear.
9.7 Counting characters

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing \texttt{X\{number\}}, and that \texttt{\{number\}} is added to the sum of 9 that precedes, to adjust the result.

\begin{verbatim}
\cs_new:Npn \str_count_spaces:N
{ \exp_args:No \str_count_spaces:n }
\cs_generate_variant:Nn \str_count_spaces:N { c }
\cs_new:Npn \str_count_spaces:n #1
{ \int_eval:n
{ \exp_after:wN \__str_count_spaces_loop:w
\tl_to_str:n {#1} ~ X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~ \s__str_stop }
}
\cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
{ \if_meaning:w X #9
\__str_use_i_delimit_by_s_stop:nw
\fi:
9 + \__str_count_spaces_loop:w
}
\end{verbatim}

(End definition for \texttt{\__str_count_delimit_by_q_stop:w} and others.)

\section*{\texttt{\str_count:N}}

To count characters in a string we could first escape all spaces using \texttt{\__kernel_str-to_other:n}, then pass the result to \texttt{\tl_count:n}. However, the escaping step would
be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces \( \text{str\_count\_spaces:n} \) and the result of \( \text{tl\_count:n} \), which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function \( \_\_\text{str\_count:n} \), used in \( \text{str\_item:nn} \) and \( \text{str\_range:nnn} \), is similar to \( \text{str\_count\_ignore\_spaces:n} \) but expects its argument to already be a string or a string with spaces escaped.

```
\cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
\cs_generate_variant:Nn \str_count:N { c }
\cs_new:Npn \str_count:n #1
    { \__str_count_aux:n
      \str_count_spaces:n {#1}
      + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
    }
\cs_new:Npn \__str_count:n #1
    { \__str_count_aux:n \__str_count_loop:NNNNNNNNN #1 }
\cs_new:Npn \str_count_ignore_spaces:n #1
    { \__str_count_aux:n \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
\cs_new:Npn \__str_count_aux:n #1
    { \int_eval:n { #1 } }
\cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
    { \if_meaning:w X #9
      \exp_after:wN \__str_use_none_delimit_by_s_stop:w
      \fi:
      9 + \__str_count_loop:NNNNNNNNN
    }
(End definition for \( \text{str\_count:n} \) and others. These functions are documented on page 66.)
```

### 9.8 The first character in a string

\textit{str\_head:N} The \texttt{\_ignore\_spaces} variant applies \( \text{tl\_to\_str:n} \) then grabs the first item, thus skipping spaces. As usual, \textit{str\_head:N} expands its argument and hands it to \textit{str\_head:n}.
To circumvent the fact that \TeX skips spaces when grabbing undelimited macro parameters, \__str_head:w takes an argument delimited by a space. If \#1 starts with a non-space character, \__str_use_i_delimit_by_s_stop:nw leaves that in the input stream. On the other hand, if \#1 starts with a space, the \__str_head:w takes an empty argument, and the single (initially braced) space in the definition of \__str_head:w makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of \str_head:n gives an empty result after passing through \__str_use_i_delimit_by_s_stop:nw.

5827 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
5828 \cs_generate_variant:Nn \str_head:N { c }
5829 \cs_new:Npn \str_head:n #1
5830 { \exp_after:wN \__str_head:w \
5831 \tl_to_str:n {#1} { } \s__str_stop 
5832 }
5833 \cs_new:Npn \__str_head:w #1 ~ %
5834 { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
5835 \cs_new:Npn \str_head_ignore_spaces:n #1
5836 { \exp_after:wN \__str_use_i_delimit_by_s_stop:nw \
5837 \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop 
5838 }
5839 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
5840 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

(End definition for \str_head:N and others. These functions are documented on page 67.)

Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with \reverse_if:N \if_charcode:w \scan_stop:. This removes the first character, and necessarily makes the test true, since the character cannot match \scan_stop:. The auxiliary function then inserts the required \fi: to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker X be unexpandable and not a control sequence). The _ignore_spaces is rather simpler: after converting the input to a string, \__str_tail_auxii:w removes one undelimited argument and leaves everything else until an end-marker \s__str_mark. One can check that an empty (or blank) string yields an empty tail.

5842 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
5843 \cs_generate_variant:Nn \str_tail:N { c }
5844 \cs_new:Npn \str_tail:n #1
5845 { \exp_after:wN \__str_tail_auxi:w \
5846 \reverse_if:N \if_charcode:w \scan_stop: \tl_to_str:n {#1} X X \s__str_stop 
5847 }
5848 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
5849 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark \s__str_mark \s__str_stop { #2 }

(End definition for \str_tail:N and others. These functions are documented on page 67.)
9.9 String manipulation

Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.

\begin{verbatim}
\def\str_foldcase:n #1 { \__str_change_case:nn {#1} { fold } }
\def\str_lower:n #1 { \__str_change_case:nn {#1} { lower } }
\def\str_uppercase:n #1 { \__str_change_case:nn {#1} { upper } }
\cs_generate_variant:Nn \str_foldcase:n { V }
\cs_generate_variant:Nn \str_lower:n { f }
\cs_generate_variant:Nn \str_uppercase:n { f }
\def\str_new:Npn \__str_change_case:nn #1 #2 { \exp_after:wN \__str_change_case_aux:nn \exp_after:wN { \tl_to_str:n {#1} } }
\def\__str_change_case辅助:nw #1#2 \__str_change_case_result:n #3 { #2 \__str_change_case_result:n { #3 #1 } }
\cs_generate_variant:Nn \__str_change_case辅助:nw { f }
\def\__str_change_case结束:nw #1 \__str_change_case_result:n #2 { \tl_to_str:n {#2} }
\def\__str_change_case循环:nw #1#2 \q__str_recursion_tail \q__str_recursion_stop { \tl_if_head_is_space:nTF {#2} { \__str_change_case空白:n } { \__str_change_case字符:nN } {#1} #2 \q__str_recursion_stop }
\def\\tl_if_head_is_space:nTF #2 { \__str_change_case空白:n } { \__str_change_case字符:nN } {#1} #2 \q__str_recursion_stop }
\def\\exp_last_unbraced:NNNN { }
\def\\str_new:Npn \__str_change_case空白:n #1 \c_space_tl { \__str_change_case辅助:nw { ~ } \__str_change_case循环:nw {#1} }
\def\\str_new:Npn \__str_change_case字符:nN #1#2 { \__str_if_recursion_tail_stop_do:Nn #2 { \__str_change_case结束:nw } { \use:c { char_str_ #1 case:N } #2 } \__str_change_case循环:nw {#1} }
\end{verbatim}

(End definition for \str_foldcase:n and others. These functions are documented on page 70.)

For all of those strings, use \cs_to_str:N to get characters with the correct category code without worries

\begin{verbatim}
\def\str_const:Nx \c_\textunderscore_str { \cs_to_str:N \& }
\\cs_to_str:N \c_\textunderscore_str \&
\def\str_const:Nx \c_\textunderscore_str \&
\end{verbatim}

434
Scratch strings.
\l_tmpa_str
\l_tmpb_str
\g_tmpa_str
\g_tmpb_str

(End definition for \l_tmpa_str and others. These variables are documented on page 71.)

9.10 Viewing strings
\str_show:n Displays a string on the terminal.
\str_show:N
\str_show:c
\str_log:n
\str_log:N
\str_log:c

(End definition for \str_show:n and others. These functions are documented on page 70.)

10 l3str-convert implementation

10.1 Helpers

10.1.1 Variables and constants

Internal scratch space for some functions.
\__str_tmp:w
\l__str_internal_int
\l__str_internal_tl
\g__str_result_tl

(End definition for \__str_tmp:w, \l__str_internal_int, and \l__str_internal_tl.)

The \g__str_result_tl variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.
When converting, invalid bytes are replaced by the Unicode replacement character "\uFFFD."

The maximal byte number.

Internal scan marks.

Internal quarks.

To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

Conversions from one ⟨encoding⟩/⟨escaping⟩ pair to another are done within x-expanding assignments. Errors are signalled by raising the relevant flag.
10.2 String conditionals

\__str_if_contains_char:NNT \langle \text{token list} \rangle \langle \text{char} \rangle

Expects the \langle \text{token list} \rangle to be an \langle \text{other string} \rangle: the caller is responsible for ensuring that no (too)-special catcodes remain. Spaces with catcode 10 are ignored. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return “false”.

\prg_new_conditional:Npnn \__str_if_contains_char:NN #1#2 { T , TF }
\prg_new_conditional:Npnn \__str_if_contains_char:nN #1#2 { TF }
\cs_new:Npn \__str_if_contains_char_aux:NN #1#2 { \prg_break:n { ? \fi: } }
\prg_break_point:
\prg_return_false:

\prg_new_conditional:Npnn \__str_if_contains_char_true: { \prg_break:n { \prg_return_true: \use_none:n } }

\__str_octal_use:NTF \langle \text{token} \rangle \langle \text{true code} \rangle \langle \text{false code} \rangle

If the \langle \text{token} \rangle is an octal digit, it is left in the input stream, followed by the \langle \text{true code} \rangle. Otherwise, the \langle \text{false code} \rangle is left in the input stream.

\textbf{\TeX{}hackers note:} This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. \TeX{} dutifully detects octal digits for us: if \#1 is an octal digit, then the right-hand side of the comparison is \texttt{'1}\#1, greater than 1. Otherwise, the right-hand side stops as \texttt{'1}, and the conditional takes the false branch.

\prg_new_conditional:Npnn \__str_octal_use:N #1 { TF }
\prg_new_conditional:Npnn \__str_octal_use:NTF \langle \text{token} \rangle \langle \text{true code} \rangle \langle \text{false code} \rangle

(End definition for \__str_if_contains_char:NNT and others.)

(End definition for \__str_octal_use:NTF.)
\_\_str\_hexadecimal\_use:NTF  \TeX{} detects uppercase hexadecimal digits for us (see \_\_str\_octal\_use:NTF), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

\begin{verbatim}
\prg_new_conditional:Npnn \__str\_hexadecimal\_use:N #1 { TF }
{ \if_int_compare:w 1 < \texttt{1} \token_to_str:N #1 \exp_stop_f:
\texttt{1} \prg_return_true:
\else:
\if_case:w \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - \texttt{a}}
A \or: B \or: C \or: D \or: E \or: F \else:
\prg_return_false:
\exp_after:wN \use_none:n
\fi:
\prg_return_true:
\fi: }
\end{verbatim}

(End definition for \_\_str\_hexadecimal\_use:NTF.)

10.3 Conversions

10.3.1 Producing one byte or character

\begin{verbatim}
\c\_str\_byte\_0_tl
\c\_str\_byte\_1_tl
\c\_str\_byte\_255_tl
\c\_str\_byte\_\texttt{-1}\_tl
\end{verbatim}

For each integer \( N \) in the range \([0, 255]\), we create a constant token list which holds three character tokens with category code other: the character with character code \( N \), followed by the representation of \( N \) as two hexadecimal digits. The value \(-1\) is given a default token list which ensures that later functions give an empty result for the input \(-1\).

\begin{verbatim}
\group_begin:
\tl_set:Nx \l__str\_internal\_tl { \tl_to_str:n { 0123456789ABCDEF } }
\tl_map_inline:Nn \l\_\_str\_internal\_tl
{ \tl_map_inline:Nn \l\_\_str\_internal\_tl
{ \tl_const:cx { \c\_str\_byte\_\int\_eval:n{"\texttt{#1}\texttt{#1}}\_tl }
{ \char\_generate:nn { "\texttt{#1}\texttt{#1} } \{ 12 \} \#1 \#1 } }
}
\group_end:
\tl_const:cn { \c\_str\_byte\_\texttt{-1}\_tl } { { } \use\_none:n { } }
\end{verbatim}

(End definition for \c\_str\_byte\_0_tl and others.)

\begin{verbatim}
\c\_str\_output\_byte:n
\c\_str\_output\_byte:w
\c\_str\_output\_hexadecimal:n
\c\_str\_output\_end:
\end{verbatim}

Those functions must be used carefully: feeding them a value outside the range \([-1, 255]\) will attempt to use the undefined token list variable \c\_str\_byte\_(number)\_tl. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value \(-1\) produces an empty result in both cases.

\begin{verbatim}
\cs\_new:Npn \c\_str\_output\_byte:n #1
\end{verbatim}

438
{ \__str_output_byte:w #1 \__str_output_end: }
\cs_new:Npn \__str_output_byte:w
{ \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nnn \cs:w c__str_byte_ \int_eval:w }
\cs_new:Npn \__str_output_hexadecimal:n #1
{ \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end: }
\cs_new:Npn \__str_output_end:
{ \scan_stop: _tl \cs_end: }
\__str_output_byte_pair_be:n
\__str_output_byte_pair_le:n
\__str_output_byte_pair:nnN
\__str_convert_gmap:N
\__str_convert_gmap_loop:NN

\__str_output_byte_pair_be:n
\__str_output_byte_pair_be:n
\__str_output_hexadecimal:n
\__str_output_pair:nnN
\__str_convert_gmap:N
\__str_convert_gmap_loop:NN

Conversion a number in the range [0, 65535] to a pair of bytes, either big-endian or little-endian.
\cs_new:Npn \__str_output_byte_pair_be:n #1
{ \exp_args:Nf \__str_output_byte_pair:nnN { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn }
\cs_new:Npn \__str_output_byte_pair_le:n #1
{ \exp_args:Nf \__str_output_byte_pair:nnN { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn }
\cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
{ #3 \{ \__str_output_byte:n { #1 } \} \{ #2 - #1 * "100 } \} }
\cs_new:Npn \__str_convert_gmap:N #1
{ \tl_gset:Nx \g__str_result_tl { \exp_after:wN \__str_convert_gmap_loop:NN \exp_after:wN #1 \g__str_result_tl { ? \prg_break: } \prg_break_point: }
}\cs_new_protected:Npn \__str_convert_gmap:N #1
{ \tl_gset:Nx \g__str_result_tl { \exp_after:wN \__str_convert_gmap_loop:NN \exp_after:wN #1 \g__str_result_tl { ? \prg_break: } \prg_break_point: }
}

10.3.2 Mapping functions for conversions

This maps the function #1 over all characters in \g__str_result_tl, which should be a byte string in most cases, sometimes a native string.
\cs_new:Npn \__str_convert_gmap_loop:NN #1#2
{
    \use_none:n #2
    #1#2
    \__str_convert_gmap_loop:NN #1
}

(End definition for \__str_convert_gmap:N and \__str_convert_gmap_loop:NN.)

This maps the function \#1 over all character codes in \g__str_result_tl, which must be in the internal representation.

\cs_new_protected:Npn \__str_convert_gmap_internal:N #1
{
    \tl_gset:Nx \g__str_result_tl
    {
        \exp_after:wN \__str_convert_gmap_internal_loop:Nww
        \exp_after:wN #1
        \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
        \prg_break_point:
    }
}

\cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
{
\__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
#1 {#3}
\__str_convert_gmap_internal_loop:Nww #1
}

(End definition for \__str_convert_gmap_internal:N and \__str_convert_gmap_internal_loop:Nw.)

10.3.3 Error-reporting during conversion

When converting using the function \str_set_convert:Nnnn, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically \_\_\_error), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions \str_set_convert:NnnnTF, errors should be suppressed. This is done by changing \__str_if_flag_error:nnx into \__str_if_flag_no_error:nnx locally.

\cs_new_protected:Npn \__str_if_flag_error:nnx #1#2#3
{\flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }

(End definition for \__str_if_flag_error:nnx and \__str_if_flag_no_error:nnx.)

\__str_if_flag_times:nT

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints \#2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

\cs_new:Npn \__str_if_flag_times:nT #1#2
{\flag_if_raised:nT (#1) { #2-(x \flag_height:n (#1) ) } }
10.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of \TeX\ tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

\[
\langle \text{bytes} \rangle \s__str \langle \text{Unicode code point} \rangle \s__str
\]

where we have collected the \langle \text{bytes} \rangle which combined to form this particular Unicode character, and the \langle \text{Unicode code point} \rangle is in the range \[0, "10FFFF\].
- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

The input string is stored in \g__str_result_tl, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on \g__str_result_tl. Errors are silenced for the conditional functions by redefining \_str_if_flag_error:nxx locally.

\begin{verbatim}
\cs_new_protected:Npn \str_set_convert:Nnnn { \__str_convert:nNNnnn { } \tl_set_eq:NN }
\cs_new_protected:Npn \str_gset_convert:Nnnn { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
\prg_new_protected_conditional:Npnn \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
\prg_set_in_g爨os:Nn \__str_convert:nNNnnn
\prg_new_protected_conditional:Npnn \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
\prg_set_in_g爨os:Nn \__str_convert:nNNnnn
\end{verbatim}

441
\_str_convert:wwwnn \_str_convert:NNnNN

The task of \_str_convert:wwwnn is to split \langle encoding\rangle/\langle escaping\rangle pairs into their components, \#1 and \#2. Calls to \_str_convert:nnn ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- \#1 is the encoding conversion function;
- \#2 is the escaping function;
- \#3 is the escaping name for use in an error message;
- \#4 is \texttt{prg\_do\_nothing:} for unescaping/decoding, and \texttt{use\_ii\_i:nn} for encoding/escaping;
- \#5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do \#2\#1 in this order, and in the encoding/escaping phase, the order should be reversed: \#4\#2\#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, \#1, should be performed.
The arguments of \_\_str_convert:nnn are: **enc** or **esc**, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to \_\_str_convert:nnnn. The task is then to make sure that the conversion function \#3\_\#1 corresponding to the type \#3 and filtered name \#1 is defined, then set our initial conversion function \#3\_\#4 equal to that.

How do we get the \#3\_\#1 conversion to be defined if it isn’t? Two main cases.

First, if \#1 is a key in \_g__str_alias_prop, then the value \_l__str_internal_tl tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on \_l__str_internal_tl already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, \#1 may be absent from the property list. The \cs_if_exist:cF test is automatically false, and we search for a file defining the encoding or escaping \#1 (this should allow third-party .def files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on \_l__str_internal_tl is defined, so we can set the \#3\_\#1 function equal to that. In some cases (*e.g.*, utf16be), the \#3\_\#1 function is actually defined within the file we just loaded, and it is different from the \_\_str-internal_tl-based function: we mustn’t clobber that different definition.

(End definition for \_\_str_convert:wwmn and \_\_str_convert:NNnN.)
\__str_convert_lowercase_alphanum:n
\__str_convert_lowercase_alphanum_loop:N

This function keeps only letters and digits, with upper case letters converted to lower case.

\cs_new:Npn \__str_convert_lowercase_alphanum:n #1
\exp_after:wN \__str_convert_lowercase_alphanum_loop:N
\tl_to_str:n {#1} { ? \prg_break: }
\prg_break_point:
\cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
\use_none:n #1
\if_int_compare:w '#1 > 'Z \exp_stop_f:
  \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
    \if_int_compare:w '#1 < 'a \exp_stop_f:
      \else:
        \__str_output_byte:n { '#1 + 'a - 'A }
        \fi:
        \fi:
        \fi:
    \else:
      \__str_output_byte:n { '#1 + 'a - 'A }
    \fi:
  \else:
    \__str_output_byte:n { '#1 + 'a - 'A }
  \fi:
\else:
  \__str_output_byte:n { '#1 + 'a - 'A }
\fi:
\__str_load_catcodes:
Since encoding files may be loaded at arbitrary places in a \TeX document, including within verbatim mode, we set the catcodes of all characters appearing in any encoding definition file.
10.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag \texttt{str\_byte}. Spaces have already been given the correct category code when this function is called.

\begin{verbatim}
\_str_filter_bytes:n        \_str_filter_bytes_aux:N
\end{verbatim}

445
\_\str\_convert\_unescape: The simplest unescaping method removes non-bytes from \_g\_str\_result\_tl.

\_\str\_convert\_unescape\_bytes: \_\str\_convert\_unescape_: The simplest unescaping method removes non-bytes from \_g\_str\_result\_tl.

\_\str\_convert\_escape: The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

\_\str\_convert\_escape\_bytes: \_\str\_convert\_escape_: The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

10.3.6 Native strings

\_\str\_convert\_decode: \_\str\_decode\_native\_char: Convert each character to its character code, one at a time.

\_\str\_convert\_encode: \_\str\_encode\_native\_char: The conversion from an internal string to native character tokens basically maps \char\_generate:nn through the code-points, but in non-Unicde-aware engines we use a fallback character rather than nothing when given a character code outside [0, 255]. We detect the presence of bad characters using a flag and only produce a single error after the x-expanding assignment.
\__str_convert_encode_:  
\flag_clear:n { str_error }  
\__str_convert_gmap_internal:N \__str_encode_native_char:n  
\__str_if_flag_error:nnx { str_error }  
{ native-overflow } { }  
}  
\cs_new:Npn \__str_encode_native_char:n #1  
{  
\if_int_compare:w #1 > \c__str_max_byte_int  
\flag_raise:n { str_error }  
?  
\else:  
\char_generate:nn {#1} {12}  
\fi:  
}  
\__kernel_msg_new:nnnn { str } { native-overflow }  
{ Character-code-too-large-for-this-engine. }  
{ This-engine-only-support-8-bit-characters:-valid-character-codes-are-in-the-range-[0,255].-To-manipulate-arbitrary-Unicode,-use-LuaTeX-or-XeTeX. }  
}  
\cs_new:Npn \__str_convert_decode_clist:  
\__str_decode_clist_char:n  
Convert each integer to the internal form. We first turn \g__str_result_tl into a clist variable, as this avoids problems with leading or trailing commas.  
\cs_new_protected:Npn \__str_convert_decode_clist:  
{  
\clist_gset:No \g__str_result_tl \g__str_result_tl  
\tl_gset:Nx \g__str_result_tl { \exp_args:No \clist_map_function:nN \g__str_result_tl \__str_decode_clist_char:n  }  
}  
\cs_new:Npn \__str_decode_clist_char:n #1  
{ #1 \s__str \int_eval:n {#1} \s__str }  
(\End definition for \__str_convert_decode_clist: and \__str_decode_clist_char:n.)  
\__str_convert_encode_clist:  
\__str_encode_clist_char:n  
Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since \tl_tail:N does not trigger an error in this case).  
\cs_new_protected:Npn \__str_convert_encode_clist:  
{  
\__str_convert_gmap_internal:N \__str_encode_clist_char:n  
\tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }  
}
10.3.8 8-bit encodings

This section will be entirely rewritten: it is not yet clear in what situations 8-bit encodings are used, hence I don’t know what exactly should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings. An approach based on csnames would have a smaller constant load time for each individual conversion, but has a large hash table cost. Using a range of \count registers works for decoding, but not for encoding: one possibility there would be to use a binary tree for the mapping of Unicode characters to bytes, stored as a box, one per encoding.

Since the section is going to be rewritten, documentation lacks.

All the 8-bit encodings which l3str supports rely on the same internal functions.

All the 8-bit encoding definition file start with \str_declare_eight_bit_encoding:nnn \{\{encoding name\}\}\{\{mapping\}\}\{\{missing bytes\}\}. The \{mapping\} argument is a token list of pairs \{(byte)\} \{(Unicode)\} expressed in uppercase hexadecimal notation. The \{missing\} argument is a token list of \{(byte)\}. Every \{byte\} which does not appear in the \{mapping\} nor the \{missing\} lists maps to the same code point in Unicode.
\cs_new_protected:Npn \__str_decode_eight_bit_load:nn #1#2
\{\__str_use_none_delimit_by_s_stop:w #1 \s__str_stop
\text_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
\text_skip:D \l__str_internal_int =="#1 sp \scan_stop:
\text_toks:D \l__str_internal_int \exp_after:wN \{ \int_value:w "#2 \}
\int_incr:N \l__str_internal_int
\__str_decode_eight_bit_load:nn\}
\cs_new_protected:Npn \__str_decode_eight_bit_load_missing:n #1
\{\__str_use_none_delimit_by_s_stop:w #1 \s__str_stop
\text_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
\text_skip:D \l__str_internal_int =="#1 sp \scan_stop:
\text_toks:D \l__str_internal_int \exp_after:wN \{ \int_use:N \c__str_replacement_char_int \}
\int_incr:N \l__str_internal_int
\__str_decode_eight_bit_load_missing:n\}
\cs_new:Npn \__str_decode_eight_bit_char:N #1
\{\if_int_compare:w \text_dimen:D '#1 < \l__str_internal_int
\if_int_compare:w \text_skip:D \text_dimen:D '#1 ="#1 \exp_stop_f:
\text_the:D \text_toks:D \text_dimen:D \fi:
\fi:
\int_value:w '#1 \s__str
\}

(End definition for \__str_convert_decode_eight_bit:n and others.)

[End of definition for \__str_convert_decode_eight_bit:n and others.]
10.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```
\__kernel_msg_new:nnn { str } { unknown-esc }
\{ Escaping-scheme-‘#1’-(filtered: ‘#2’)-unknown. \}
\__kernel_msg_new:nnn { str } { unknown-enc }
\{ Encoding-scheme-‘#1’-(filtered: ‘#2’)-unknown. \}
\__kernel_msg_new:nnnn { str } { native-escaping }
\{ The-‘native’-encoding-scheme-does-not-support-any-escaping. \}
\{ Since-native-strings-do-not-consist-in-bytes,-
\none-of-the-escaping-methods-make-sense.-
\The-specified-escaping,-‘#1’,-will be ignored. \}
\__kernel_msg_new:nnn { str } { file-not-found }
\{ File-‘l3str-#1.def’-not-found. \}
```

Message used when the “bytes” unescaping fails because the string given to \texttt{\str_set_convert:Nnnn} contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.
6443 \__kernel_msg_new:nnnn \{ \str \} \{ \text{non-byte} \}
6444 \{ \text{String-invalid-in-escaping-"#1":it-may-only-contain-bytes.} \}
6445 \{
6446 \text{Some-characters-in-the-string-you-asked-to-convert-are-not-}
6447 \text{8-bit-characters.-Perhaps-the-string-is-a-'native'-Unicode-string?-}
6448 \text{If-it-is,-try-using\}\}
6449 \\}
6450 \text{iow-indent:n}
6451 \{
6452 \text{\textbackslash iow_char:N\str_set_convert:Nnnn \}
6453 \text{\ \ <str-var>-\{-<string>-\}-\{-native\}-\{-<target-encoding>-\}}
6454 \}
6455 \}
6456 \}
6457 \}

Those messages are used when converting to and from 8-bit encodings.

6458 \__kernel_msg_new:nnnn \{ \str \} \{ \text{decode-8-bit} \}
6459 \{ \text{Invalid-string-in-encoding-"#1".} \}
6460 \{
6461 \text{LaTeX-came-across-a-byte-which-is-not-defined-to-represent-}
6462 \text{any-character-in-the-encoding-"#1".}
6463 \}
6464 \__kernel_msg_new:nnnn \{ \str \} \{ \text{encode-8-bit} \}
6465 \{ \text{Unicode-string-cannot-be-converted-to-encoding-"#1".} \}
6466 \{
6467 \text{The-encoding-"#1"-only-contains-a-subset-of-all-Unicode-characters.-}
6468 \text{LaTeX-was-asked-to-convert-a-string-to-that-encoding,-but-that-}
6469 \text{string-contains-a-character-that-"#1"-does-not-support.}
6470 \}

10.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage
methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is
defined by default);
- **hex** or **hexadecimal**, as per the pdf\TeX primitive `\pdfescapehex`
- **name**, as per the pdf\TeX primitive `\pdf escapename`
- **string**, as per the pdf\TeX primitive `\pdfescapestring`
- **url**, as per the percent encoding of urls.

10.5.1 Unescape methods

Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending
a 0 in all cases, and dropping it if it is alone.

\cs_new_protected:Npn \__str_convert_unescape_hex:
\texttt{\{\group_begin:\n  \flag_clear:n \str_error \n  \int_set:Nn \tex_escapechar:D \{ \texttt{92} \n  \tl_gset:Nx \g__str_result_tl \n  \{ \n    \_\str_output_byte:w " \n    \exp_last_unbraced:Nf \_\str_unescape_hex_auxi:N \n    \{ \tl_to_str:N \g__str_result_tl \} \n    0 \{ ? 0 - 1 \prg_break: \} \prg_break_point: \n    \_\str_output_end: \n  \} \n  \_\str_if_flag_error:nxn \str_error \} \} \} \group_end:\n\cs_new:Npn \_\str_unescape_hex_auxi:N \#1 \n  \{ \use_none:n \#1 \n  \_\str_hexadecimal_use:NTF \#1 \n  \{ \_\str_unescape_hex_auxii:N \} \n  \{ \n    \flag_raise:n \str_error \n    \_\str_unescape_hex_auxi:N \n  \} \n\} \cs_new:Npn \_\str_unescape_hex_auxii:N \#1 \n  \{ \use_none:n \#1 \n  \_\str_hexadecimal_use:NTF \#1 \n  \{ \_\str_output_end: \n    \_\str_output_byte:w " \_\str_unescape_hex_auxi:N \n  \} \n  \{ \n    \flag_raise:n \str_error \n    \_\str_unescape_hex_auxii:N \n  \} \n\} \_\kernel_msg_new:nnnn \str \{ \texttt{unescape-hex} \} \{ \texttt{String-invalid-in-escaping-`hex':-only-hexadecimal-digits-allowed.} \} \{ \texttt{Some-characters-in-the-string-you-asked-to-convert-are-not-hexadecimal-digits-(0-9,-A-F,-a-f)-nor-spaces.} \} \}

(End definition for \_\str_convert_unescape_hex:, \_\str_unescape_hex_auxi:N, and \_\str_unescape_hex_auxii:N.)

The \_\str_convert_unescape_name: function replaces each occurrence of \# followed by two hexadecimal digits in \g__str_result_tl by the corresponding byte. The \url function is identical, with escape character \% instead of \#. Thus we define the two together. The arguments of \_\str_tmp:w are the character code of \# or \% in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

452
The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test \_\_str_hexadecimal_use:NTF leaves the upper-case digit in the input stream, hence we surround the test with \_\_str_output_byte:w and \_\_str_output_end:. If both characters are hexadecimal digits, they should be removed before looping: this is done by \use_i:nnn. If one of the characters is not a hexadecimal digit, then feed "#1 to \_\_str_output_byte:w to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove \use_i:nnn).

\cs_set_protected:Npn \_\_str_tmp:w { #1#2#3 }
\cs_new_protected:cpn { __str_convert_unescape_#2: } { }{ }
\group_begin:{ }
\flag_clear:n { str_byte }
\flag_clear:n { str_error }
\int_set:Nn \tex_escapechar:D { 92 }
\tl_gset:Nx \g__str_result_tl { \exp_after:wN #3 \g__str_result_tl }
#1 ? { ? \prg_break: } \prg_break_point:
\__str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
\__str_if_flag_error:nnx { str_error } { unescape-#2 } { }
\group_end: }
\cs_new:Npn #3 { ##1##2##3 }
\__str_filter_bytes:n {##1} \use_none:n ##3 \__str_output_byte:w 
###\__str_hexadecimal_use:NTF \_\_str_hexadecimal_use:NTF #2 \{ \}
\__kernel_msg_new:nnnn { str } { unescape-#2 } { String invalid in escaping-#2. } { \LaTeX came across the escape character-'#1'-not followed by-two hexadecimal digits. This is invalid in the escaping-'#2'. }
The string escaping is somewhat similar to the name and url escapings, with escape character \. The first step is to convert all three line endings, \^J, \^M, and \^M\^J to the common \^J, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

```
\n Line feed (10)
\r Carriage return (13)
\t Horizontal tab (9)
\b Backspace (8)
\f Form feed (12)
( Left parenthesis
\) Right parenthesis
\ Backslash
\ddd (backslash followed by 1 to 3 octal digits) Byte ddd (octal), subtracting 256 in case of overflow.
```

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.
\_\_str_if_flag_error:nxx { str_byte } { non-byte } { string }
\_\_str_if_flag_error:nxx { str_error } { unescape-string } { }
group_end:
\exp_args:No \_\_str_tmp:w { \c_backslash_str }
\exp_last_unbraced:NNNNo
\cs_new:Npn \_\_str_unescape_string_loop:wNNN \c_backslash_str #2\#3\#4
{ \_\_str_filter_bytes:n {#1} \use_none:n #4 \_\_str_output_byte:w , \_\_str_octal_use:NTF #2
{ \_\_str_octal_use:NTF #3
{ \_\_str_octal_use:NTF #4
{ \_\_str_unescape_string_repeat:NNNNNN ? }
{ \_\_str_unescape_string_repeat:NNNNNN ?? } \str_case_e:nnF {#2}
{ \c_backslash_str } { 134 }
{ ( ) } { 50 }
{ ) } { 51 }
{ r } { 15 }
{ f } { 14 }
{ n } { 12 }
{ t } { 11 }
{ b } { 10 }
{ "\J } { 0 - 1 }
}
{ \flag_raise:n { str_error } 0 - 1 \use_i:nn }
\_\_str_output_end:
\use_i:nn \_\_str_unescape_string_loop:wNNN #2\#3\#4
\cs_new:Npn \_\_str_unescape_string_repeat:NNNNNN \_\_\_str_unescape_string_loop:wNNN #1\_\_\_str_unescape_string_repeat:NNNNNN #2\#3\#4
\cs_new:Npn \_\_str_unescape_string_newlines:wN \_\_\_str_unescape_string_repeat:NNNNNN \_\_\_str_unescape_string_repeat:NNNNNN #1
\_\_\_str_unescape_string_repeat:NNNNNN #2\#3\#4\#5\#6
\cs_new:Npn \_\_str_unescape_string_newlines:wN \_\_\_str_unescape_string_repeat:NNNNNN #1 "\n #2
{ #1
\if_charcode:w ^^J #2 \else: ^^J \fi:
\_str_unescape_string_newlines:wN #2
}
\_kernel_msg_new:nnnn { str } { unescape-string }
{ String-\texttt{invalid-in-escaping-'string'}. }
{ \LaTeX\text{-}came-across-an-escape-character-'\texttt{c\_backslash_str}'-
\texttt{not-followed-by-any-of:-'n','r','t','b','f','t'(-(t,-))','-
'\texttt{c\_backslash_str}','-one-to-three-octal-digits,}-or-the-end-
of-a-line. }
\group_end:
(\textit{End definition for \texttt{\_str_convert_unescape_string:} and others.})

\textbf{10.5.2 Escape methods}

Currently, none of the escape methods can lead to errors, assuming that their input is
made out of bytes.

\texttt{\_str_convert_escape_hex:}
Loop and convert each byte to hexadecimal.
\texttt{\_str_escape_hex_char:N}
\texttt{\_str_convert_gmap:N \_str_escape_hex_char:N}
\texttt{\cs_new_protected:Npn \_str_convert_escape_hex:}
\texttt{\{ \_str_convert_gmap:N \_str_escape_hex_char:N \}}
\texttt{\cs_new:Npn \_str_escape_hex_char:N #1}
\texttt{\{ \_str_output_hexadecimal:n { '#1 } \}}
\texttt{(End definition for \texttt{\_str_convert_escape_hex:} and \texttt{\_str_escape_hex_char:N}.)}

\texttt{\_str_convert_escape_name:}
\texttt{\_str_escape_name_char:N}
\texttt{\_str_if_escape_name:NTF}
\texttt{\_str_convert_escape_name:}
\texttt{\cs_new_protected:Npn \_str_convert_escape_name:}
\texttt{\{ \_str_convert_gmap:N \_str_escape_name_char:N \}}
\texttt{\cs_new:Npn \_str_escape_name_char:N #1}
\texttt{\{ \_str_if_escape_name:NTF #1 {#1} \}}
\texttt{\{ \_str_if_escape_name:NTF \c__str_escape_name_not_str \}}
\texttt{\c__str_escape_name_str}
\texttt{\c_hash_str \_str_output_hexadecimal:n \{ '#1 \}}
\texttt{\prg_new_conditional:Npnn \_str_if_escape_name:N #1 { TF } \}}
\texttt{\prg_new_conditional:Npnn \_str_if_escape_name:N #1 { TF } \}}
\texttt{\if_int_compare:w \#1 < "2A \exp_stop_f:}
\texttt{\_str_if_contains_char:NNTF \c__str_escape_name_not_str \#1}
\texttt{\prg_return_true: \prg_return_false:}
\texttt{\else: \if_int_compare:w \#1 > "7E \exp_stop_f:}
\texttt{\prg_return_false: \else:}
\texttt{\_str_if_contains_char:NNTF \c__str_escape_name_str \#1}
\texttt{\prg_return_false: \prg_return_true: \fi: \fi: \fi:}
\texttt{456}
Any character below (and including) space, and any character above (and including) del, are converted to octal. One backslash is added before each parenthesis and backslash.

This function is similar to \_\_str_convert_escape_name:, escaping different characters.
10.6 Encoding definitions

The native encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- **UTF-8**;
- **UTF-16**, big-, little-endian, or with byte order mark;
- **UTF-32**, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

10.6.1 UTF-8 support

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? \texttt{\_\_str\_convert\_utf\_viii\_loop::w\_\_n\_n\_w} takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient \texttt{#1} is less than the limit \texttt{#3} for that range, output the leading byte (\texttt{#1} shifted by \texttt{#4}) and stop. Otherwise, we need one more step: use the quotient of \texttt{#1} by 64, and \texttt{#1} as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder \texttt{#2} − 64\texttt{#1} + 128. The bizarre construction \texttt{- 1 + 0 *} removes the spurious initial continuation byte (better methods welcome).
\cs_new_protected:cpn { \__str_convert_encode_utf8: }
{ \\__str_convert_gmap_internal:N \\__str_encode_utf_viii_char:n }
\cs_new:Npn \\__str_encode_utf_viii_char:n #1
{ \\__str_encode_utf_viii_loop:wwnnw #1 ; - 1 + 0 * ;
 { 128 } \{ 0 \}
 { 32 } \{ 192 \}
 { 16 } \{ 224 \}
 { 8 } \{ 240 \}
 \s__str_stop
}
\cs_new:Npn \\__str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \s__str_stop
{ \if_int_compare:w #1 < #3 \exp_stop_f:
 \\__str_output_byte:n { #1 + #4 }
 \exp_after:wN \\__str_use_none_delimit_by_s_stop:w
 \fi:
 \exp_after:wN \\__str_encode_utf_viii_loop:wwnnw
 \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ; #5 \s__str_stop
 \\__str_output_byte:n \{ #2 - 64 * ( #1 - 2 ) \}
}
(End definition for \\__str_convert_encode_utf8:, \\__str_encode_utf_viii_char:n, and \\__str_encode_utf_viii_loop:wwnnw.)

When decoding a string that is purportedly in the utf-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using \flag_clear_new:n rather than \flag_new:n, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, i.e., not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, “C080 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one \LaTeX error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct utf-8 string should look like, then add error-specific information.

\flag_clear_new:n { str_missing }
\flag_clear_new:n { str_extra }
\flag_clear_new:n { str_overlong }
\flag_clear_new:n { str_overflow }
\__kernel_msg_new:nnnn { str } { utf8-decode }
{
 Invalid-UTF-8-string:
 \exp_last_unbraced:Nf \use_none:n

459
In the UTF-8 encoding, each Unicode character consists in 1 to 4 bytes, with the following bit pattern: \[
\begin{align*}
\text{Code-point-} & \quad \langle 0 \langle 128 \rangle \langle 2048 \rangle < 65536 \rangle \langle 1114112 \rangle \\
\text{A leading byte (in the range }[192, 255] & \text{)} was not followed by the appropriate number of continuation bytes.
\end{align*}
\]

Bytes of the form 10xxxxxx are called continuation bytes.

\[
\begin{align*}
\text{The start auxiliary tests the first byte:}
\end{align*}
\]

- [0, 7F]: the byte stands alone, and is converted to its own character code;
- [80, BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character `\texttt{\char123FFFD};`
• ["C0", "FF"]: this byte should be followed by some continuation byte(s).

In the first two cases, \use_none_delimit_by_q_stop: removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the _continuation auxiliary. We expect \#3 to be in the range ["80", "BF"]. The test for this goes as follows: if the character code is less than "80", we compare it to −"C0", yielding false; otherwise to "C0", yielding true in the range ["80", "BF"] and false otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the _start auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the _aux function.

The _aux function tests whether we should look for more continuation bytes or not. If the number it receives as \#1 is less than the maximum \#4 for the current range, then we are done: check for an overlong representation by comparing \#1 with the maximum \#3 for the previous range. Otherwise, we call the _continuation auxiliary again, after shifting the “current code point” by \#4 (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD" for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the _start auxiliary leaves its first argument in the input stream: the end-marker begins with \prg_break:, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the \use_none:n \#3 construction removes the first token from the end-marker, and leaves the _end auxiliary, which raises the appropriate error flag before ending the mapping.

\cs_new_protected:cpn { __str_convert_decode_utf8: }  \{  \flag_clear:n { str_error }  \flag_clear:n { str_missing }  \flag_clear:n { str_extra }  \flag_clear:n { str_overlong }  \flag_clear:n { str_overflow }  \tl_gset:Nx \g__str_result_tl { \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl { \prg_break: \__str_decode_utf_viii_end: } }  \prg_break_point:  \__str_if_flag_error:nnx { str_error } { \utf8-decode } { }  \}  \cs_new:Npn \__str_decode_utf_viii_start:N #1  \{  #1  \if_int_compare:w '#1 < "C0 \exp_stop_f:  \s__str  \if_int_compare:w '#1 < "80 \exp_stop_f:  \int_value:w '#1  \prg_break: \__str_decode_utf_viii_end: }  \prg_break_point:  \}  \__str_if_flag_error:nnx { str_error } { \utf8-decode } { }  \}  \cs_new:Npn \__str_decode_utf_viii_start:N #1  \{  #1  \if_int_compare:w ' #1 < "$C0 \exp_stop_f:  \s__str  \if_int_compare:w ' #1 < "$80 \exp_stop_f:  \int_value:w ' #1  \else:  \flag_raise:n { str_extra }  \}  

461
10.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- [0, "D7FE]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 − "10000/400.

For the duration of this operation, __str__tmp:w is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to __str__encode_utf_xvi_be:n or its le analog: those compute the remainder, and output two bytes for the quotient and remainder.
When encoding a Unicode string to UTF-16, only one error can occur: code points in the range \([U+D800, U+DFFF]\), corresponding to surrogates, cannot be encoded. We use the all-purpose flag \@_error to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

When encoding a Unicode string to UTF-16, only one error can occur: code points in the range \([U+D800, U+DFFF]\), corresponding to surrogates, cannot be encoded. We use the all-purpose flag \@_error to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.
In the UTF-16 encoding, each Unicode character is encoded as 2-or-4-bytes:\n
\begin{itemize}
\item Code-point-in-\[U+0000, U+D7FF\]: two bytes
\item Code-point-in-\[U+D800, U+DFFF\]: illegal
\item Code-point-in-\[U+E000, U+FFFF\]: two bytes
\item Code-point-in-\[U+10000, U+10FFFF\]:
\begin{itemize}
\item a lead surrogate and a trail surrogate
\end{itemize}
\end{itemize}

Lead surrogates are pairs of bytes in the range \[0xD800, 0xDBFF\], and trail surrogates are in the range \[0xDC00, 0xDFFF\].

If a lead surrogate was not followed by a trail surrogate,

\begin{itemize}
\item \flag_if_raised:nT { str_missing }
\item A lead surrogate was not followed by a trail surrogate.
\end{itemize}

If LaTeX came across a trail surrogate when it was not expected,

\begin{itemize}
\item \flag_if_raised:nT { str_extra }
\item LaTeX came across a trail surrogate when it was not expected.
\end{itemize}

The string contained an odd number of bytes. This is invalid:

the basic code unit for UTF-16 is 16 bits (2 bytes).

As for utf-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are *FE and *FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianess. The three endianness cases are based on a common auxiliary whose first argument is \lstr_stop for big-endian and \lstr_stop for little-endian, and whose second argument, delimited by the scan mark \sstr_stop, is expanded once (the string may be long; passing \gstr_result_tl as an argument before expansion is cheaper).

The \_str_decode_utf_xvi:w function defines \_str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using \_str_decode_utf_xvi_pair:NN described below.

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are *FE and *FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianess. The three endianness cases are based on a common auxiliary whose first argument is \lstr_stop for big-endian and \lstr_stop for little-endian, and whose second argument, delimited by the scan mark \sstr_stop, is expanded once (the string may be long; passing \gstr_result_tl as an argument before expansion is cheaper).

The \_str_decode_utf_xvi:w function defines \_str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using \_str_decode_utf_xvi_pair:NN described below.

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are *FE and *FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianess. The three endianness cases are based on a common auxiliary whose first argument is \lstr_stop for big-endian and \lstr_stop for little-endian, and whose second argument, delimited by the scan mark \sstr_stop, is expanded once (the string may be long; passing \gstr_result_tl as an argument before expansion is cheaper).

The \_str_decode_utf_xvi:w function defines \_str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using \_str_decode_utf_xvi_pair:NN described below.

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are *FE and *FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianess. The three endianness cases are based on a common auxiliary whose first argument is \lstr_stop for big-endian and \lstr_stop for little-endian, and whose second argument, delimited by the scan mark \sstr_stop, is expanded once (the string may be long; passing \gstr_result_tl as an argument before expansion is cheaper).

The \_str_decode_utf_xvi:w function defines \_str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using \_str_decode_utf_xvi_pair:NN described below.

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are *FE and *FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianess. The three endianness cases are based on a common auxiliary whose first argument is \lstr_stop for big-endian and \lstr_stop for little-endian, and whose second argument, delimited by the scan mark \sstr_stop, is expanded once (the string may be long; passing \gstr_result_tl as an argument before expansion is cheaper).

The \_str_decode_utf_xvi:w function defines \_str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using \_str_decode_utf_xvi_pair:NN described below.
Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- `["D8","DB"]` signals a lead surrogate, and the integer expression yields 1 (←TEX rounds ties away from zero);

- `["DC","DF"]` signals a trail surrogate, unexpected here, and the integer expression yields 2;

- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the _pair auxiliary.

The case of a lead surrogate is treated by the _quad auxiliary, whose arguments #1, #2, #4 and #5 are the four bytes. We expect the most significant byte of #4#5 to be in the range `["DC","DF"]` (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where #4#5 is indeed a trail surrogate, leave #1#2#4#5 \s__str ⟨code point⟩ \s__str, and remove the pair #4#5 before looping with \_str_decode_utf_xvi_pair:NN. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7×"400 = "D800×"400+"DC00−"10000.
Every time we read a pair of bytes, we test for the end-marker `\__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```latex
\cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
\begin{verbatim}
\if_meaning:w \q__str_nil #2
\__str_decode_utf_xvi_pair_end:Nw #1
\fi:
\if_case:w
\int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
\or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
\or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
\fi:
#1#2 \s__str
\int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
\__str_decode_utf_xvi_pair:NN
\end{verbatim}
\cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
\begin{verbatim}
\if_meaning:w \q__str_nil #5
\__str_decode_utf_xvi_error:nNN { missing } #1#2
\__str_decode_utf_xvi_pair_end:Nw #4
\fi:
\if_int_compare:w
\if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
0 = 1
\else:
\__str_tmp:w #4#5 < "E0
\fi:
\exp_stop_f:
#1 #2 #4 #5 \s__str
\int_eval:n
\begin{verbatim}
{ ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
+ "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
}
\end{verbatim}
\s__str
\exp_after:wN \use_i:nnn
\else:
\__str_decode_utf_xvi_error:nNN { missing } #1#2
\fi:
\__str_decode_utf_xvi_pair:NN #4#5
\end{verbatim}
\cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1
\begin{verbatim}
\fi:
\if_meaning:w \q__str_nil #1
\else:
\__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
\fi:
\prg_break:
\end{verbatim}
\cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
\end{verbatim}
```
\begin{verbatim}
{ \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
\cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3 {
    \flag_raise:n { str_error }
    \flag_raise:n { str_##1 }
    \int_use:N \c__str_replacement_char_int \s__str
}
\end{verbatim}

(End definition for \__str_decode_utf_xvi_pair:NN and others.)

Restore the original catcodes of bytes 254 and 255.

\subsection*{10.6.3 utf-32 support}

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

\begin{verbatim}
\__str_convert_encode_utf32: \__str_convert_encode_utf32be:
\__str_convert_encode_utf32le:
\__str_encode_utf_xxxii_be:n \__str_encode_utf_xxxii_be_aux:nn
\__str_encode_utf_xxxii_le:n \__str_encode_utf_xxxii_le_aux:nn
\cs_new_protected:cpn { \__str_convert_encode_utf32: }
\cs_new_protected:cpn { \__str_convert_encode_utf32be: }
\cs_new_protected:cpn { \__str_convert_encode_utf32le: }
\cs_new_protected:cpn { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
\tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
\cs_new_protected:cpn { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
\cs_new_protected:cpn { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
\cs_new:Npn \__str_convert_encode_utf32be:n #1 {
    \exp_args:Nf \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
    \tl_gput_left:Nx \g__str_result_tl { ^^00 "\#1 * "100 ^"fe ^"ff }
}
\cs_new:Npn \__str_convert_encode_utf32le:n #1 {
    \exp_args:Nf \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n
    \tl_gput_left:Nx \g__str_result_tl { "\#1 * "100 ^"fe ^"ff }
}
\cs_new:Npn \__str_convert_encode_utf32le:n #1 {
    \exp_args:Nf \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n
    \tl_gput_left:Nx \g__str_result_tl { "\#1 * "100 ^"fe ^"ff }
}
\end{verbatim}

Convert each integer in the comma-list \g__str_result_tl to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the \__str_output_byte:n instructions are reversed.
There can be no error when encoding in utf-32. When decoding, the string may not have length $4n$, or it may contain code points larger than $\times 10FFFF$. The latter case often happens if the encoding was in fact not utf-32, because most arbitrary strings are not valid in utf-32.

```
\__str_output_byte_pair_le:n {#1}
^~00
}

(End definition for \_\_str_convert_encode_utf32: and others.)

**str_overflow** There can be no error when encoding in utf-32. When decoding, the string may not have length $4n$, or it may contain code points larger than $\times 10FFFF$. The latter case often happens if the encoding was in fact not utf-32, because most arbitrary strings are not valid in utf-32.

```
\flag_clear_new:n { str_overflow }
\flag_clear_new:n { str_end }
\__kernel_msg_new:nnnn { str } { utf32-decode }
{
    Invalid-UTF-32-string:
    \exp_last_unbraced:Nf \use_none:n
    {
        \__str_if_flag_times:nT { str_overflow } { ,code-point-too-large }
        \__str_if_flag_times:nT { str_end } { ,truncated-string }
    }
}
```

(End definition for str_overflow and str_end. These variables are documented on page ???.)

The structure is similar to utf-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly \_\_str_stop if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The \_\_str_decode_utf_xxxii:Nw auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets \_\_str_tmp:w to expand to the character code of either of its two arguments depending on endianness, then triggers the _loop auxiliary inside an x-expanding assignment to \g__str_result_tl.

The _loop auxiliary first checks for the end-of-string marker \_\_str_stop, calling the _end auxiliary if appropriate. Otherwise, leave the ⟨4 bytes⟩ \_\_str_stop behind, then
check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `s__str_stop`. Break the map.

```latex
\cs_new_protected:cpn { __str_convert_decode_utf32be: }
\cs_new_protected:cpn { __str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
\cs_new_protected:cpn { __str_convert_decode_utf32le: }
\cs_new_protected:cpn { __str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
\cs_new_protected:cpn { __str_convert_decode_utf32: }
{
    \exp_after:wN __str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
    \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
}
\cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
{
    \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
    { __str_decode_utf_xxxii:Nw 2 }
    {
        \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
        { __str_decode_utf_xxxii:Nw 1 }
        { __str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
    }
}
\cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
{
    \flag_clear:n { str_overflow }
    \flag_clear:n { str_end }
    \flag_clear:n { str_error }
\exp_after:wN \__str_decode_utf_xxxii_loop:NNNN #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
\__str_if_flag_error:nnx { str_error } { utf32-decode } { }
\cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
{
    \flag_clear:n { str_overflow }
    \flag_clear:n { str_end }
    \flag_clear:n { str_error }
    \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN \g__str_result_tl
    \s__str_stop \s__str_stop \s__str_stop \s__str_stop
\prg_break_point:
}
\__str_if_flag_error:nxn { str_error } { utf32-decode } { }
\cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
{
    \if_meaning:w \s__str_stop #4
    \exp_after:wN \__str_decode_utf_xxxii_end:w
    \fi:
\exp_after:wN \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4 \s__str
\\if_int_compare:w \__str_tmp:w #1#4 > 0 \exp_stop_f:
    \flag_raise:n { str_overflow }
    \flag_raise:n { str_error }
    \int_use:N \c__str_replacement_char_int
\else:
    \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
        \flag_raise:n { str_overflow }
        \flag_raise:n { str_error }
        \int_use:N \c__str_replacement_char_int
    \else:
        \int_eval:n
```

470
\[ \text{__str_tmp:w #2#3**10000 + \__str_tmp:w #3#2**100 + \__str_tmp:w #4#1 } \]
\fi:
\fi:
\s__str
\__str_decode_utf_xxxii_loop:NNNN
\cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
{
\tl_if_empty:nF {#1}
{
 \flag_raise:n { str_end }
 \flag_raise:n { str_error }
 #1 \s__str
 \int_use:N \c__str_replacement_char_int \s__str
}\prg_break:
}
(End definition for \__str_convert_decode_utf32: and others.)

Restore the original catcodes of bytes 0, 254 and 255.
\group_end:
\⟨/initex\ | \package⟩

10.6.4 iso 8859 support

The iso-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the iso-8859 family, we keep track only of differences, and of unassigned bytes.
\⟨iso88591⟩
\str_declare_eight_bit_encoding:nnn \{ iso88591 \}
\{ 
\}
\{ 
\} 
\⟨/iso88591⟩
\⟨iso88592⟩
\str_declare_eight_bit_encoding:nnn \{ iso88592 \}
\{ 
\{ A1 \} \{ 0104 \}
\{ A2 \} \{ 02D8 \}
\{ A3 \} \{ 0141 \}
\{ A5 \} \{ 013D \}
\{ A6 \} \{ 015A \}
\{ A9 \} \{ 0160 \}
\{ AA \} \{ 015E \}
\{ AB \} \{ 0164 \}
\{ AC \} \{ 0179 \}
\{ AE \} \{ 017D \}
\{ AF \} \{ 017B \}
\{ B1 \} \{ 0105 \}
\{ B2 \} \{ 02DB \}
\{ B3 \} \{ 0142 \}
\{ B5 \} \{ 013E \}

471
\str_declare_eight_bit_encoding:nnn { iso88593 }

{ A1 } { 0126 }
{ A2 } { 02D8 }
{ A6 } { 0124 }
{ A9 } { 0130 }
{ AA } { 015E }
\begin{lstlisting}
{ AB }\{ 011E \}
{ AC }\{ 0134 \}
{ AF }\{ 0178 \}
{ B1 }\{ 0127 \}
{ B6 }\{ 0125 \}
{ B9 }\{ 0131 \}
{ BA }\{ 015F \}
{ BB }\{ 011F \}
{ BC }\{ 0135 \}
{ BF }\{ 017C \}
{ C5 }\{ 010A \}
{ C6 }\{ 0108 \}
{ D5 }\{ 0120 \}
{ D8 }\{ 011C \}
{ DD }\{ 016C \}
{ DE }\{ 015C \}
{ E5 }\{ 010B \}
{ E6 }\{ 0109 \}
{ F5 }\{ 0121 \}
{ F6 }\{ 011D \}
{ FD }\{ 016D \}
{ FE }\{ 015D \}
{ FF }\{ 02D9 \}

{ A5 }
{ AE }
{ BE }
{ C3 }
{ D0 }
{ E3 }
{ F0 }

\end{lstlisting}
\str_declare_eight_bit_encoding:nnn { iso88595 }

{ A1 } { 0401 }
{ A2 } { 0402 }
{ A3 } { 0403 }
{ A4 } { 0404 }
{ A5 } { 0405 }
{ A6 } { 0406 }
{ A7 } { 0407 }
{ A8 } { 0408 }
{ A9 } { 0409 }
{ AA } { 040A }
{ AB } { 040B }
{ AC } { 040C }
{ AE } { 040E }
{ AF } { 040F }
\str_declare_eight_bit_encoding:nnn { \texttt{iso88596} }

\langle iso88595 \rangle

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>0633</td>
</tr>
<tr>
<td>D4</td>
<td>0634</td>
</tr>
<tr>
<td>D5</td>
<td>0635</td>
</tr>
<tr>
<td>D6</td>
<td>0636</td>
</tr>
<tr>
<td>D7</td>
<td>0637</td>
</tr>
<tr>
<td>D8</td>
<td>0638</td>
</tr>
<tr>
<td>D9</td>
<td>0639</td>
</tr>
<tr>
<td>DA</td>
<td>063A</td>
</tr>
<tr>
<td>E0</td>
<td>0640</td>
</tr>
<tr>
<td>E1</td>
<td>0641</td>
</tr>
<tr>
<td>E2</td>
<td>0642</td>
</tr>
<tr>
<td>E3</td>
<td>0643</td>
</tr>
<tr>
<td>E4</td>
<td>0644</td>
</tr>
<tr>
<td>E5</td>
<td>0645</td>
</tr>
<tr>
<td>E6</td>
<td>0646</td>
</tr>
<tr>
<td>E7</td>
<td>0647</td>
</tr>
<tr>
<td>E8</td>
<td>0648</td>
</tr>
<tr>
<td>E9</td>
<td>0649</td>
</tr>
<tr>
<td>EA</td>
<td>064A</td>
</tr>
<tr>
<td>EB</td>
<td>064B</td>
</tr>
<tr>
<td>EC</td>
<td>064C</td>
</tr>
<tr>
<td>ED</td>
<td>064D</td>
</tr>
<tr>
<td>EE</td>
<td>064E</td>
</tr>
<tr>
<td>EF</td>
<td>064F</td>
</tr>
<tr>
<td>F0</td>
<td>0650</td>
</tr>
<tr>
<td>F1</td>
<td>0651</td>
</tr>
<tr>
<td>F2</td>
<td>0652</td>
</tr>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td></td>
</tr>
<tr>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>AE</td>
<td></td>
</tr>
<tr>
<td>AF</td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td></td>
</tr>
<tr>
<td>B9</td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td></td>
</tr>
</tbody>
</table>
\str_declare_eight_bit_encoding:nnn \{ iso88597 \}

{ BH } { 2017 } { F6 } { 0396 } { F7 } { 0397 } { F8 } { 0398 } { F9 } { 0399 } { FA } { 039A } { FB } { 039B } { CC } { 039C } { CD } { 039D } { CE } { 039E } { CF } { 039F } { D0 } { 03A0 } { D1 } { 03A1 } { D3 } { 03A3 } { D4 } { 03A4 } { D5 } { 03A5 } { D6 } { 03A6 } { D7 } { 03A7 } { D8 } { 03A8 } { D9 } { 03A9 } { DA } { 03AA } { DB } { 03AB }
\str_declare_eight_bit_encoding:nnn { iso88598 }
{ E7 } { 05D7 }
{ E8 } { 05D8 }
{ E9 } { 05D9 }
{ EA } { 05DA }
{ EB } { 05DB }
{ EC } { 05DC }
{ ED } { 05DD }
{ EE } { 05DE }
{ EF } { 05DF }
{ F0 } { 05E0 }
{ F1 } { 05E1 }
{ F2 } { 05E2 }
{ F3 } { 05E3 }
{ F4 } { 05E4 }
{ F5 } { 05E5 }
{ F6 } { 05E6 }
{ F7 } { 05E7 }
{ F8 } { 05E8 }
{ F9 } { 05E9 }
{ FA } { 05EA }
{ FD } { 200E }
{ FE } { 200F }

{ A1 }
{ BF }
{ C0 }
{ C1 }
{ C2 }
{ C3 }
{ C4 }
{ C5 }
{ C6 }
{ C7 }
{ C8 }
{ C9 }
{ CA }
{ CB }
{ CC }
{ CD }
{ CE }
{ CF }
{ D0 }
{ D1 }
{ D2 }
{ D3 }
{ D4 }
{ D5 }
{ D6 }
{ D7 }
{ D8 }
{ D9 }
{ DA }
{ DB }
\str_declare_eight_bit_encoding:nnn \texttt{iso88599} \\
\{ D0 \} \{ 011E \}
\{ DD \} \{ 0130 \}
\{ DE \} \{ 015E \}
\{ F0 \} \{ 011F \}
\{ FD \} \{ 0131 \}
\{ FE \} \{ 015F \}
\}
\}

\str_declare_eight_bit_encoding:nnn \texttt{iso885910} \\
\{ A1 \} \{ 0104 \}
\{ A2 \} \{ 0112 \}
\{ A3 \} \{ 0122 \}
\{ A4 \} \{ 012A \}
\{ A5 \} \{ 0128 \}
\{ A6 \} \{ 0136 \}
\{ A8 \} \{ 013B \}
\{ A9 \} \{ 0110 \}
\{ AA \} \{ 0160 \}
\{ AB \} \{ 0166 \}
\{ AC \} \{ 017D \}
\{ AE \} \{ 016A \}
\{ AF \} \{ 014A \}
\{ B1 \} \{ 0105 \}
\{ B2 \} \{ 0113 \}
\{ B3 \} \{ 0123 \}
\{ B4 \} \{ 012B \}
\{ B5 \} \{ 0129 \}
\{ B6 \} \{ 0137 \}
\{ B8 \} \{ 013C \}
\{ B9 \} \{ 0111 \}
\{ BA \} \{ 0161 \}
\{ BB \} \{ 0167 \}
\{ BC \} \{ 017E \}
\{ BD \} \{ 2015 \}
\{ BE \} \{ 016B \}
\{ BF \} \{ 014B \}
\{ C0 \} \{ 0100 \}
\{ C7 \} \{ 012E \}
\{ C8 \} \{ 010C \}

481
\strDeclareEightBitEncoding:nnn { iso885910 }

\strDeclareEightBitEncoding:nnn { iso885911 }

{ A1 } { 0E01 }
{ A2 } { 0E02 }
{ A3 } { 0E03 }
{ A4 } { 0E04 }
{ A5 } { 0E05 }
{ A6 } { 0E06 }
{ A7 } { 0E07 }
{ A8 } { 0E08 }
{ A9 } { 0E09 }
{ AA } { 0E0A }
{ AB } { 0E0B }
{ AC } { 0E0C }
{ AD } { 0E0D }
{ AE } { 0E0E }
{ AF } { 0E0F }
{ B0 } { 0E10 }
{ B1 } { 0E11 }
{ B2 } { 0E12 }
{ B3 } { 0E13 }
{ B4 } { 0E14 }
{ B5 } { 0E15 }
{ B6 } { 0E16 }
{ B7 } { 0E17 }
{ B8 } { 0E18 }
{ B9 } { 0E19 }
{ BA } { 0E1A }
{ BB } { 0E1B }
{ BC } { 0E1C }
{ BD } { 0E1D }
{ BE } { 0E1E }
{ BF } { 0E1F }
<table>
<thead>
<tr>
<th>Code</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>♂</td>
</tr>
<tr>
<td>C1</td>
<td>♀</td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td></td>
</tr>
<tr>
<td>C9</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>CB</td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td></td>
</tr>
<tr>
<td>CE</td>
<td></td>
</tr>
<tr>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td></td>
</tr>
<tr>
<td>D7</td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td></td>
</tr>
<tr>
<td>D9</td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td></td>
</tr>
<tr>
<td>DF</td>
<td></td>
</tr>
<tr>
<td>E0</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td></td>
</tr>
<tr>
<td>E7</td>
<td></td>
</tr>
<tr>
<td>E8</td>
<td></td>
</tr>
<tr>
<td>E9</td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td></td>
</tr>
<tr>
<td>EB</td>
<td></td>
</tr>
<tr>
<td>EC</td>
<td></td>
</tr>
<tr>
<td>ED</td>
<td></td>
</tr>
<tr>
<td>EE</td>
<td></td>
</tr>
<tr>
<td>EF</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td></td>
</tr>
<tr>
<td>F9</td>
<td></td>
</tr>
</tbody>
</table>

483
{ FA } { OE5A }
{ FB } { OE5B }
{
{ DB }
{ DC }
{ DD }
{ DE }
}
{/iso885911}
{%iso885913%
\str_declare_eight_bit_encoding:nnn { iso885913 }
{
{ A1 } { 201D }
{ A5 } { 201E }
{ A8 } { 00D8 }
{ AA } { 0156 }
{ AF } { 00C6 }
{ B4 } { 201C }
{ B8 } { 00F8 }
{ BA } { 0157 }
{ BF } { 00E6 }
{ CO } { 0104 }
{ C1 } { 012E }
{ C2 } { 0100 }
{ C3 } { 0106 }
{ C6 } { 0118 }
{ C7 } { 0112 }
{ C8 } { 010C }
{ CA } { 0179 }
{ CB } { 0116 }
{ CC } { 0122 }
{ CD } { 0136 }
{ CE } { 012A }
{ CF } { 013B }
{ D0 } { 0160 }
{ D1 } { 0143 }
{ D2 } { 0145 }
{ D4 } { 014C }
{ D8 } { 0172 }
{ D9 } { 0141 }
{ DA } { 015A }
{ DB } { 016A }
{ DD } { 017B }
{ DE } { 017D }
{ E0 } { 0105 }
{ E1 } { 012F }
{ E2 } { 0101 }
{ E3 } { 0107 }
{ E6 } { 0119 }
{ E7 } { 0113 }
{ EB } { 010D }
{ EA } { 017A }
{ EB } { 0117 }
}
\begin{verbatim}
\str_declare_eight_bit_encoding:nnn { iso885914 }
{
  { A1 } { 1E02 }
  { A2 } { 1E03 }
  { A4 } { 010A }
  { A5 } { 010B }
  { A6 } { 1E0A }
  { A8 } { 1E80 }
  { AA } { 1E82 }
  { AB } { 1E8B }
  { AC } { 1EF2 }
  { AF } { 017B }
  { B0 } { 1E1E }
  { B1 } { 1E1F }
  { B2 } { 0120 }
  { B3 } { 0121 }
  { B4 } { 1E40 }
  { B5 } { 1E41 }
  { B7 } { 1E56 }
  { B8 } { 1E81 }
  { B9 } { 1E57 }
  { BA } { 1E83 }
  { BB } { 1E60 }
  { BC } { 1EF3 }
  { BD } { 1E84 }
  { BE } { 1E85 }
  { BF } { 1E61 }
  { D0 } { 0174 }
  { D7 } { 1E6A }
  { DE } { 0176 }
  { F0 } { 0175 }
  { F7 } { 1E6B }
  { FE } { 0177 }
}
\end{verbatim}
\str_declare_eight_bit_encoding:nnn { iso885915 }
{ 
{ A4 } { 20AC }
{ A6 } { 0160 }
{ A8 } { 0161 }
{ B4 } { 017D }
{ B8 } { 017E }
{ BC } { 0152 }
{ BD } { 0153 }
{ BE } { 0178 }
}

\str_declare_eight_bit_encoding:nnn { iso885916 }
{ 
{ A1 } { 0104 }
{ A2 } { 0105 }
{ A3 } { 0141 }
{ A4 } { 20AC }
{ A5 } { 201E }
{ A6 } { 0160 }
{ A8 } { 0161 }
{ AA } { 0218 }
{ AC } { 0179 }
{ AE } { 017A }
{ AF } { 017B }
{ B2 } { 010C }
{ B3 } { 0142 }
{ B4 } { 017D }
{ B5 } { 201D }
{ B8 } { 017E }
{ B9 } { 010D }
{ BA } { 0219 }
{ BC } { 0152 }
{ BD } { 0153 }
{ BE } { 0178 }
{ BF } { 017C }
{ C3 } { 0102 }
{ C5 } { 0106 }
{ D0 } { 0110 }
{ D1 } { 0143 }
{ D5 } { 0150 }
{ D7 } { 015A }
{ D8 } { 0170 }
{ DD } { 0118 }
{ DE } { 021A }
{ E3 } { 0103 }
}
A sequence is a control sequence whose top-level expansion is of the form \texttt{\_\_seq_item:n \{item\} ... \_\_seq_item:n \{item\}}", with a leading scan mark followed by \(n\) items of the same form. An earlier implementation used the structure \texttt{\seq_elt:w \{item\} \seq_elt_end: \ldots \seq_elt:w \{item\} \seq_elt_end:}. This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items \_\_seq_item:n

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

\_\_seq_push_item_def:n
\_\_seq_push_item_def:x

Saves the definition of \_\_seq_item:n and redefines it to accept one parameter and expand to \{code\}. This function should always be balanced by use of \_\_seq_pop_item_def:. 

\_\_seq_pop_item_def:

Restores the definition of \_\_seq_item:n most recently saved by \_\_seq_push_item_def:n. This function should always be used in a balanced pair with \_\_seq_push_item_def:n.

\s__seq
This private scan mark.

(End definition for \s__seq.)

\s__seq_mark
Private scan marks.

(End definition for \s__seq_mark and \s__seq_stop.)
The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

\__seq_item:n
\_\_kernel_msg\_expandable\_error:nn \{ kernel \} \{ misused\_sequence \}
\use\_none:n

(End definition for \__seq_item:n.)

\l__seq_internal\_a\_tl
Scratch space for various internal uses.
\l__seq_internal\_b\_tl

(End definition for \l__seq_internal\_a\_tl and \l__seq_internal\_b\_tl.)

\__seq\_tmp:w
Scratch function for internal use.

\c_empty_seq
A sequence with no item, following the structure mentioned above.

(End definition for \c_empty_seq. This variable is documented on page 85.)

11.1 Allocation and initialisation

\seq\_new:N
\seq\_new:c
Sequences are initialized to \c_empty_seq.

\seq\_clear:N
\seq\_clear:c
Clearing a sequence is similar to setting it equal to the empty one.

\seq\_clear\_new:N
\seq\_clear\_new:c
\seq\_gclear\_new:N
\seq\_gclear\_new:c
Once again we copy code from the token list functions.

488
Copying a sequence is the same as copying the underlying token list.

\begin{itemize}
\item \texttt{\seq_set_eq:NN}
\item \texttt{\seq_set_eq:cN}
\item \texttt{\seq_set_eq:Nc}
\item \texttt{\seq_set_eq:cc}
\item \texttt{\seq_gset_eq:NN}
\item \texttt{\seq_gset_eq:cN}
\item \texttt{\seq_gset_eq:Nc}
\item \texttt{\seq_gset_eq:cc}
\end{itemize}

Setting a sequence from a comma-separated list is done using a simple mapping.

\begin{itemize}
\item \texttt{\seq_set_from_clist:NN}
\item \texttt{\seq_set_from_clist:cN}
\item \texttt{\seq_set_from_clist:Nc}
\item \texttt{\seq_set_from_clist:cc}
\item \texttt{\seq_gset_from_clist:NN}
\item \texttt{\seq_gset_from_clist:cN}
\item \texttt{\seq_gset_from_clist:Nc}
\item \texttt{\seq_gset_from_clist:cc}
\end{itemize}

Almost identical to \texttt{\seq_set_from_clist:Nn}.

\begin{itemize}
\item \texttt{\seq_const_from_clist:Nn}
\item \texttt{\seq_const_from_clist:cN}
\end{itemize}
When the separator is empty, everything is very simple, just map \$_{\text{seq\_wrap\_item:n}}$ through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After \texttt{\textbackslash tl\_replace\_all:Nnn}, the token list \texttt{\textbackslash l\_seq\_internal\_a\_tl} is a repetition of the pattern \texttt{\textbackslash _seq\_set\_split\_auxi:w \textbackslash prg\_do\_nothing: (item with spaces) \textbackslash _seq\_set\_split\_end:}. Then, x-expansion causes \texttt{\textbackslash _seq\_set\_split\_auxi:w} to trim spaces, and leaves its result as \texttt{\textbackslash _seq\_set\_split\_auxi:w (trimmed item) \textbackslash _seq\_set\_split\_end:}. This is then converted to the \texttt{l3seq} internal structure by another x-expansion. In the first step, we insert \texttt{\textbackslash prg\_do\_nothing:} to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

(End definition for \texttt{\textbackslash seq\_set\_split:Nnn} and others. These functions are documented on page 76.)

When concatenating sequences, one must remove the leading \texttt{\textbackslash s\_\_seq} of the second
sequence. The result starts with \s__seq (of the first sequence), which stops f-expansion.

\cs_new_protected:Npn \seq_concat:NNN #1#2#3
\{ \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } \}
\cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
\{ \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } \}
\cs_generate_variant:Nn \seq_concat:NNN { ccc }
\cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

(End definition for \seq_concat:NNN and \seq_gconcat:NNN. These functions are documented on page 76.)

\seq_if_exist_p:N \seq_if_exist_p:c \seq_if_exist:NTF \seq_if_exist:c
\prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
\prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c

(End definition for \seq_if_exist:NTF. This function is documented on page 76.)

11.2 Appending data to either end

When adding to the left of a sequence, remove \s__seq. This is done by \__seq_put_left_aux:w, which also stops f-expansion.

\cs_new_protected:Npn \seq_put_left:Nn #1#2
\{ \tl_set:Nx #1 { \exp_not:n { \s__seq \__seq_item:n {#2} } \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 } } \}
\cs_new_protected:Npn \seq_gput_left:Nn #1#2
\{ \tl_gset:Nx #1 { \exp_not:n { \s__seq \__seq_item:n {#2} } \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 } } \}
\cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }

\seq_put_right:Nn \seq_put_right:NV \seq_put_right:Nv \seq_put_right:No \seq_put_right:Nx
\{ \tl_set:Nx \seq_put_left:Nn \seq_put_left:NV \seq_put_left:Nv \seq_put_left:No \seq_put_left:Nx \}
\seq_put_right:cn \seq_put_right:cV \seq_put_right:cv \seq_put_right:co \seq_put_right:cx
\seq_gput_right:Nn \seq_gput_right:NV \seq_gput_right:Nv \seq_gput_right:No \seq_gput_right:Nx
\{ \tl_gset:Nx \seq_gput_left:Nn \seq_gput_left:NV \seq_gput_left:Nv \seq_gput_left:No \seq_gput_left:Nx \}
\seq_gput_right:cn \seq_gput_right:cV \seq_gput_right:cv \seq_gput_right:co \seq_gput_right:cx

\__seq_put_left_aux:w \s__seq

(End definition for \seq_put_left:Nn, \seq_put_left:NN, and \__seq_put_left_aux:w. These functions are documented on page 76.)

\seq_put_right:Nn \seq_put_right:NV \seq_put_right:Nv \seq_put_right:No \seq_put_right:Nx
\{ \tl_set:Nx \seq_put_right:Nn \seq_put_right:NV \seq_put_right:Nv \seq_put_right:No \seq_put_right:Nx \}
\seq_gput_right:Nn \seq_gput_right:NV \seq_gput_right:Nv \seq_gput_right:No \seq_gput_right:Nx
\seq_gput_right:cn \seq_gput_right:cV \seq_gput_right:cv \seq_gput_right:co \seq_gput_right:cx

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in \__seq_item:n.
11.3 Modifying sequences

\__seq_wrap_item:n
\cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

\l__seq_remove_seq
An internal sequence for the removal routines.
\seq_new:N \l__seq_remove_seq

\seq_remove_duplicates:N \seq_remove_duplicates:c
\seq_gremove_duplicates:N \seq_gremove_duplicates:c
\__seq_remove_duplicates:NN
Removing duplicates means making a new list then copying it.
\cs_new_protected:Npn \seq_remove_duplicates:N
\cs_new_protected:Npn \seq_gremove_duplicates:N
\cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
{ \seq_clear:N \l__seq_remove_seq
\seq_map_inline:Nn #2
{ \seq_if_in:NnF \l__seq_remove_seq {##1}
{ \seq_put_right:Nn \l__seq_remove_seq {##1} }
}
#1 #2 \l__seq_remove_seq }
\cs_generate_variant:Nn \seq_remove_duplicates:N { c }
\cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

\seq_remove_all:Nn \seq_remove_all:cn
\seq_gremove_all:Nn \seq_gremove_all:cn
\__seq_remove_all_aux:NNn
The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in \__seq_pop_right:NNN, using a “flexible” x-type expansion to do most of the work. As \tl_\_if_eq:nNT is not expandable, a two-part strategy is needed. First, the x-type expansion uses \str_\_if_eq:nNT to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the \tl_\_if_eq:nNT test. The x-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) ensures that nothing is lost.
\cs_new_protected:Npn \seq_remove_all:Nn
\cs_new_protected:Npn \seq_gremove_all:Nn
\cs_new_protected:Npn \seq_remove_all:cn
\cs_new_protected:Npn \seq_gremove_all:cn
\cs_new_protected:Npn \__seq_remove_all_aux:NNn
\texttt{\cs_new_protected:Npn \_\_seq_remove_all Aux:NNn \#1\#3} \\
\{ \\
\_\_seq_push_item_def:n \\
\{ \\
\str_if_eq:nnT {\#1} {\#3} \\
\{ \\
\if_false: { \fi: } \\
\tl_set:Nn \_\_seq_internal_b_tl {\#1} \\
#1 #2 \\
\{ \if_false: \} \fi: \\
\exp_not:o {#2} \\
\tl_if_eq:NNT \_\_seq_internal_a_tl \_\_seq_internal_b_tl \\
\{ \use_none:nn \} \\
\_\_seq_wrap_item:n {\#1} \\
\} \\
\_\_seq_pop_item_def: \\
\} \\
\_\_seq_reverse:NN \_\_seq_remove_all>All:NN \_\_seq_reverse:NN \_\_seq_remove_all Aux:NNn \\
\_\_seq_reverse_item:nwn

(End definition for \texttt{\_\_seq_remove_all Aux:NNn}, \texttt{\_\_seq_remove_all Aux:NN}, and \texttt{\_\_seq_remove_all Aux:NNn}.)

Previously, \texttt{\seq_reverse:N} was coded by collecting the items in reverse order after an \texttt{\exp_stop_f}: marker.

\texttt{\cs_new_protected:Npn \seq_reverse:N \#1} \\
\{ \\
\cs_set_eq:NN \@@_item:n \@@_reverse_item:nw \\
\tl_set:Nf \#2 { \#2 \exp_stop_f: } \\
\} \\
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f: \\
\{ \\
#2 \exp_stop_f: \\
\@@_item:n {\#1} \\
\}

At first, this seems optimal, since we can forget about each item as soon as it is placed after \texttt{\exp_stop_f}:. Unfortunately, \texttt{\TeX}’s usual tail recursion does not take place in this case: since the following \texttt{\_\_seq_reverse_item:nw} only reads tokens until \texttt{\exp_stop_f}:, and never reads the \texttt{\_\_item:n} \{\#1\} left by the previous call, \texttt{\TeX} cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the \texttt{\_\_seq_reverse_item:nw} are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. \texttt{\TeX} can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of \texttt{\exp_not:n}. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

\texttt{\cs_new_protected:Npn \seq_reverse:N} \\
\{ \_\_seq_reverse:NN \_\_seq_set:Nx \}
\cs_new_protected:NN \seq_greverse:N \__seq_reverse:NN \tl_gset:Nx  
\cs_new_protected:NN \__seq_reverse:NN \_\_seq_reverse_item:nwn  
\cs_set_eq:NN \__seq_tmp:w \__seq_item:n  
\cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn  
\_\_seq_reverse_item:nwn \_\_seq_item:n \exp_not:n { }  
\cs_set_eq:NN \__seq_item:n \__seq_tmp:w  
\cs_generate_variant:Nn \seq_reverse:N { c }  
\cs_generate_variant:Nn \seq_greverse:N { c }  
(End definition for \seq_reverse:N and others. These functions are documented on page 79.)

\seq_sort:Nn \seq_sort:cn \seq_gsort:Nn \seq_gsort:cn  
Implemented in l3sort.  
(End definition for \seq_sort:Nn and \seq_gsort:Nn. These functions are documented on page 79.)

11.4 Sequence conditionals

\seq_if_empty_p:N \seq_if_empty_p:c \seq_if_empty:NTF \seq_if_empty:cTF  
Similar to token lists, we compare with the empty sequence.

\seq_shuffle:N \seq_shuffle:c \seq_gshuffle:N \seq_gshuffle:c \__seq_shuffle:NN \_\_seq_shuffle_item:n \g__seq_internal_seq  
We apply the Fisher–Yates shuffle, storing items in \toks registers. We use the primitive \tex_uniformdeviate:D for speed reasons. Its non-uniformity is of order its argument divided by \(2^{28}\), not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds (13! \(> \) \(2^{28}\)) so the question of uniformity is somewhat moot. The integer variables are declared in l3int: load-order issues.

\cs_if_exist:NTF \tex_uniformdeviate:D  
\seq_new:N \g__seq_internal_seq  
\cs_new_protected:NN \seq_shuffle:N \_\_seq_shuffle:NN \_\_seq_shuffle_item:n  
\__seq_shuffle:NN \seq_set_eq:NN \_\_seq_shuffle_item:n  
\g__seq_internal_seq \_\_seq_shuffle_item:n \exp_not:n \#2  
\cs_set_eq:NN \__seq_tmp:w \__seq_item:n  
\cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn  
\_\_seq_reverse_item:nwn \_\_seq_item:n \exp_not:n \#3  
\cs_set_eq:NN \__seq_item:n \__seq_tmp:w  
\cs_generate_variant:Nn \seq_reverse:N { c }  
\cs_generate_variant:Nn \seq_greverse:N { c }  
(End definition for \seq_shuffle:NTF. This function is documented on page 80.)
\__kernel_msg_error:nnx { kernel } { shuffle-too-large }
{ \token_to_str:N #2 }
}

\group_begin:
\int_zero:N \l__seq_internal_a_int
\__seq_push_item_def:
\cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n #2
\__seq_pop_item_def:
\seq_gset_from_inline_x:Nnn \g__seq_internal_seq
{ \int_step_function:nN \l__seq_internal_a_int }
{ \tex_the:D \tex_toks:D ##1 }
\group_end:
#1 #2 \g__seq_internal_seq
\seq_gclear:N \g__seq_internal_seq
}

\cs_new_protected:Npn \__seq_shuffle_item:n
{ \int_incr:N \l__seq_internal_a_int
\int_set:Nn \l__seq_internal_b_int
{ 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
\tex_toks:D \l__seq_internal_a_int
= \tex_toks:D \l__seq_internal_b_int
\tex_toks:D \l__seq_internal_b_int
}

\cs_new_protected:Npn \seq_shuffle:N #1
{ \__kernel_msg_error:nnn { kernel } { fp-no-random }
\seq_shuffle:N #1 }
\cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
\cs_generate_variant:Nn \seq_shuffle:N { c }
\cs_generate_variant:Nn \seq_gshuffle:N { c }

The approach here is to define \__seq_item:n to compare its argument with the test
sequence. If the two items are equal, the mapping is terminated and \group_end: \prg-
return_true: is inserted after skipping over the rest of the recursion. On the other hand,
if there is no match then the loop breaks, returning \prg_return_false:. Everything
is inside a group so that \__seq_item:n is preserved in nested situations.

(End definition for \seq_shuffle:N and others. These functions are documented on page 80.)
11.5 Recovering data from sequences

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

\begin{verbatim}
\cs_new:Npn \__seq_pop:NNNN #1#2#3#4
{ \if_meaning:w #3 \c_empty_seq \tl_set:Nn #4 \q_no_value \else: #1#2#3#4 \fi: }
\cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
{ \if_meaning:w #3 \c_empty_seq \prg_return_false: \else: #1#2#3#4 \prg_return_true: \fi: }
\end{verbatim}

(End definition for \__seq_pop:NNNN and \__seq_pop_TF:NNNN.)

\begin{verbatim}
\cs_new_protected:Npn \__seq_get_left:NN #1#2
\tl_set:Nx #2 { \exp_after:wN \__seq_get_left:wnw \__seq_item:n { \q_no_value } \s__seq_stop }
\end{verbatim}

(End definition for \__seq_get_left:NN and \__seq_get_left:cN.)

Getting an item from the left of a sequence is pretty easy: just trim off the first item after \__seq_item:n at the start. We append a \q_no_value item to cover the case of an empty sequence.

\begin{verbatim}
\cs_new_protected:Npn \__seq_get_left:NN\seq_get_left:\seq_get_left:\seq_get_left:wnw
\tl_set:Nx #1 #2
{ \exp_after:wN \__seq_get_left:wnw \__seq_item:n { \q_no_value } \s__seq_stop }
\end{verbatim}
The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

\begin{verbatim}
\cs_new_protected:Npn \seq_pop_left:NN #1\__seq_pop_left:NN \tl_set:Nn
\cs_new_protected:Npn \seq_gpop_left:NN \__seq_pop_left:NN \tl_gset:Nn
\cs_new_protected:Npn \__seq_pop_left:NNN \__seq_pop_left:wnwNNN \tl_set:Nn
\cs_new_protected:Npn \__seq_pop_left:wnwNNN \__seq_item:n \s__seq_stop \tl_set:Nn
\cs_generate_variant:Nn \seq_pop_left:NN { c }
\cs_generate_variant:Nn \seq_gpop_left:NN { c }
\end{verbatim}

(End definition for \seq_pop_left:NN and others. These functions are documented on page 77.)

First remove \s__seq and prepend \q_no_value. The first argument of \__seq_get_right_loop:nw is the last item found, and the second argument is empty until the end of the loop, where it is code that applies \exp_not:n to the last item and ends the loop.

\begin{verbatim}
\cs_new_protected:Npn \seq_get_right:NN #1\__seq_get_right_loop:nw \__seq_get_right_end:NnN
\cs_new_protected:Npn \__seq_get_right_loop:nw #1\__seq_item:n
\cs_new_protected:Npn \__seq_get_right_end:NnN #1#2#3 \tl_set:Nn
\cs_generate_variant:Nn \seq_get_right:NN { c }
\end{verbatim}

(End definition for \seq_get_right:NN, \__seq_get_right_loop:nw, and \__seq_get_right_end:NnN. This function is documented on page 77.)

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the { \if_false: } \fi: ...\if_false: \fi: \exp_not:n \tl_set:Nn \tl_gset:Nn construct. Using an x-type expansion and a “non-expanding” definition for \__seq_item:n, the left-most $n - 1$ entries in a sequence of $n$ items are stored back in the sequence. That needs a loop of unknown length, hence using the
strange `if_false`: way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and \tl_set:Nn \#3 is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and \use_none:nn, which finally stops the loop.

\cs_new_protected:Npn \seq_pop_right:NN { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
\cs_new_protected:Npn \seq_gpop_right:NN { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3 { \cs_set_eq:NN \__seq_tmp:w \__seq_item:n \cs_set_eq:NN \__seq_item:n \scan_stop: #1 #2 { \if_false: } \fi: \s__seq \exp_after:wN \use_i:nnn \exp_after:wN \__seq_pop_right_loop:nn #2 { \if_false: { \fi: } \tl_set:Nx #3 } \use_none:nn \cs_set_eq:NN \__seq_item:n \__seq_tmp:w }
\prg_generate_variant:Nnn \seq_pop_right:NN { c }
\prg_generate_variant:Nnn \seq_pop_right:NN { c }
(End definition for \seq_pop_right:NN and others. These functions are documented on page 77.)

\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF } { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF } { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nx #2 } \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c } { T , F , TF } \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c } { T , F , TF }
(End definition for \seq_left:NNTF and \seq_right:NNTF. These functions are documented on page 78.)

\prg_new_protected_conditional:Npnn \seq_pop_left:NNn \seq_pop_left:NNn \prg_do_nothing: \seq_pop_left:NN #1#2 \prg_new_protected_conditional:Npnn \seq_gpop_left:NNn \seq_gpop_left:NNn \prg_do_nothing: \seq_pop_left:NN #1#2 \prg_generate_conditional_variant:Nnn \seq_pop_left:NNn \seq_pop_left:NNn { c } { T , F , TF } \prg_generate_conditional_variant:Nnn \seq_gpop_left:NNn \seq_gpop_left:NNn { c } { T , F , TF }

(End definition for \seq_left:NNTF and \seq_right:NNTF. These functions are documented on page 78.)

\prg_new_protected_conditional:Npnn \seq_pop_right:NNn \seq_pop_right:NNn \prg_do_nothing: \seq_pop_right:NN #1#2 \prg_new_protected_conditional:Npnn \seq_gpop_right:NNn \seq_gpop_right:NNn \prg_do_nothing: \seq_pop_right:NN #1#2 \prg_generate_conditional_variant:Nnn \seq_pop_right:NNn \seq_pop_right:NNn { c } { T , F , TF } \prg_generate_conditional_variant:Nnn \seq_gpop_right:NNn \seq_gpop_right:NNn { c } { T , F , TF }

(End definition for \seq_left:NNTF and \seq_right:NNTF. These functions are documented on page 78.)
\seq_item:Nn

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by \__seq_item:n is \prg_break: instead of being empty, terminating the loop and returning nothing at all.

\seq_rand_item:N
Importantly, \seq_item:Nn only evaluates its argument once.
11.6 Mapping to sequences

To break a function, the special token \texttt{\textbackslash prg\_break\_point:Nn} is used to find the end of the code. Any ending code is then inserted before the return value of \texttt{\textbackslash seq\_map\_break:n} is inserted.

\begin{verbatim}
\cs_new:Npn \seq_map_break:
{ \prg_map_break:Nn \seq_map_break: { } }
\cs_new:Npn \seq_map_break:n
{ \prg_map_break:Nn \seq_map_break: }
\end{verbatim}

(End definition for \texttt{\seq_map_break:n} and \texttt{\seq_map_break:n}. These functions are documented on page 81.)

The idea here is to apply the code of \texttt{#2} to each item in the sequence without altering the definition of \texttt{\_\_seq\_item:n}. The argument delimited by \texttt{\_\_seq\_item:n} is almost always empty, except at the end of the loop where it is \texttt{\textbackslash prg\_break:}. This allows to break the loop without needing to do a (relatively-expensive) quark test.

\begin{verbatim}
\cs_new:Npn \seq_map_function:NN #1#2
{ \exp_after:wN \use_i_ii:nnn
  \exp_after:wN \__seq_map_function:Nw
  \exp_after:wN #2
#1
\prg_break: \_\_seq\_item:n { } \prg_break_point:
\prg_break_point:Nn \seq_map_break: { }
}
\cs_new:Npn \__seq_map_function:Nw #1#2 \_\_seq\_item:n #3
{ #2
  #1 {#3}
  \_\_seq_map_function:Nw #1
}
\cs_generate_variant:Nn \seq_map_function:NN { c }
\end{verbatim}

(End definition for \texttt{\seq_map_function:NN} and \texttt{\_\_seq_map_function:NNn}. This function is documented on page 80.)

The definition of \texttt{\_\_seq\_item:n} needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

\begin{verbatim}
\cs_new_protected:Npn \_\_seq\_push\_item\_def:n
{ \_\_seq\_push\_item\_def:
  \cs_gset:Npn \_\_seq\_item:n #1
}
\cs_new_protected:Npn \_\_seq\_push\_item\_def:x
{ \_\_seq\_push\_item\_def:
  \cs_gset:Npx \_\_seq\_item:n #1
}
\end{verbatim}
\cs_new_protected:Npn \__seq_push_item_def:
\int_gincr:N \g__kernel_prg_map_int
\cs_gset_eq:cN { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
\__seq_item:n
\cs_new_protected:Npn \__seq_pop_item_def:
\cs_gset_eq:Nc \__seq_item:n { \__seq_map_ \int_use:N \g__kernel_prg_map_int }
\int_gdecr:N \g__kernel_prg_map_int

(End definition for \__seq_push_item_def:n, \__seq_push_item_def:, and \__seq_pop_item_def:.)

\seq_map_inline:Nn
\seq_map_tokens:Nn
\__seq_map_tokens:nw
\seq_map_variable:NNn
\seq_map_variable:Ncn
\seq_map_variable:ccn
\seq_map_variable:ccn

The idea here is that \__seq_item:n is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining \__seq_item:n.

\cs_new_protected:Npn \seq_map_inline:Nn #1#2
\__seq_push_item_def:n {#2}
\prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
\prg_break_point:Nn \seq_map_break: { }
\cs_generate_variant:Nn \seq_map_inline:Nn { c }

(End definition for \seq_map_inline:Nn. This function is documented on page 80.)

\seq_map_tokens:Nn
\__seq_map_tokens:nw
\seq_map_variable:NNn
\seq_map_variable:Ncn
\seq_map_variable:ccn
\seq_map_variable:ccn

This is based on the function mapping but using the same tricks as described for \prop_map_tokens:Nn. The idea is to remove the leading \s__seq and apply the tokens such that they are safe with the break points, hence the \use:n.

\cs_new:Npn \seq_map_tokens:Nn #1#2
\exp_last_unbraced:Nno \use_i:nn { \__seq_map_tokens:nw {#2} } #1
\prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
\prg_break_point:Nn \seq_map_break: { }
\cs_generate_variant:Nn \seq_map_tokens:Nn { c }

(End definition for \seq_map_tokens:Nn and \__seq_map_tokens:nw. This function is documented on page 81.)

\seq_map_variable:NNn
\seq_map_variable:Ncn
\seq_map_variable:ccn
\seq_map_variable:ccn

This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
\__seq_push_item_def:x
Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, \#9 is \seq_count:w instead of being empty. It removes \+ and instead places the number of \seq_item:n that \seq_count:w grabbed before reaching the end of the sequence.

\seq_count:N
\seq_count:c
\seq_count:w
\seq_count_end:w

\seq_use:Nnnn
\seq_use:cnnn
\seq_use:NNnn
\seq_use:nwww
\seq_use:NN
\seq_use:cn

\seq_count:N
\seq_count:c
\seq_count:w
\seq_count_end:w

(End definition for \seq_map_variable:NNn. This function is documented on page 81.)
11.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:N
\seq_push:NV
\seq_push:Nv
\seq_push:No
\seq_push:cn
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gp pushed:N
\seq_gp pushed:NV
\seq_gp pushed:Nv
\seq_gp pushed:No
\seq_gp pushed:cn
\seq_gp pushed:cV
\seq_gp pushed:cv
\seq_gp pushed:co
\seq_gp pushed:cx

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 82.)
\cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

(End definition for \seq_push:Nn and \seq_gpush:Nn. These functions are documented on page 84.)

\seq_get:NN \seq_get:cN \seq_pop:NN \seq_pop:cN \seq_gpop:NN \seq_gpop:cN

In most cases, getting items from the stack does not need to specify that this is from the left. So aliases are provided.

\cs_new_eq:NN \seq_get:NN \seq_get_left:NN \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

(End definition for \seq_get:NN, \seq_pop:NN, and \seq_gpop:NN. These functions are documented on page 83.)

\seq_get:NTF \seq_get:cNTF \seq_pop:NTF \seq_pop:cNTF \seq_gpop:NTF \seq_gpop:cNTF

(End definition for \seq_get:NNTF, \seq_pop:NNTF, and \seq_gpop:NNTF. These functions are documented on page 83.)

11.9 Viewing sequences

\seq_show:N \seq_show:cN \seq_log:N \seq_log:c \__seq_show:NN

Apply the general \msg_show:nnnnnn.

\cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nnnnnn }
\cs_generate_variant:Nn \seq_show:N { c }
\cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nnnnnn }
\__seq_show:NN

{ \__kernel_chk_defined:NT #2
  \#1 \{ LaTeX/kernel \} \{ show-seq \}
  \{ \token_to_str:N #2 \#2 \}
  \{ \seq_map_function:NN \#2 \msg_show_item:n \}
  \}
}

(End definition for \seq_show:N, \seq_log:N, and \__seq_show:NN. These functions are documented on page 86.)
11.10 Scratch sequences

Temporary comma list variables.
\begin{verbatim}
\l_tmpa_seq \l_tmpb_seq \g_tmpa_seq \g_tmpb_seq
\end{verbatim}

(End definition for \l_tmpa_seq and others. These variables are documented on page 86.)

12 l3int implementation

The following test files are used for this code: m3int001, m3int002, m3int03.

Done in l3basics.

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

(End definition for \c_max_register_int. This variable is documented on page 99.)

\begin{verbatim}
\__int_to_roman:w \if_int_compare:w
\end{verbatim}

Done in l3basics.

(End definition for \__int_to_roman:w and \if_int_compare:w. This function is documented on page 100.)

\begin{verbatim}
\or:
\end{verbatim}

Done in l3basics.

(End definition for \or:. This function is documented on page 100.)

Here are the remaining primitives for number comparisons and expressions.

\begin{verbatim}
\int_value:w \__int_eval:w \__int_eval_end: \if_int_odd:w \if_case:w
\end{verbatim}

(End definition for \int_value:w and others. These functions are documented on page 100.)

\begin{verbatim}
\s__int_mark \s__int_stop
\end{verbatim}

Scan marks used throughout the module.

(End definition for \s__int_mark and \s__int_stop.)

\begin{verbatim}
\__int_use_none_delimit_by_s_stop:w
\end{verbatim}

Function to gobble until a scan mark.

(End definition for \__int_use_none_delimit_by_s_stop:w.)

\begin{verbatim}
\q__int_recursion_tail \q__int_recursion_stop
\end{verbatim}

Quarks for recursion.

(End definition for \q__int_recursion_tail and \q__int_recursion_stop.)
Functions to query quarks.

(End definition for \_\_int_if_recursion_tail_stop_do:NN and \_\_int_if_recursion_tail_stop:NN.)

12.1 Integer expressions

\int_eval:n \int_eval:w

Wrapper for \_\_int_eval:w: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

(End definition for \_\_int_eval:n and \_\_int_eval:w. These functions are documented on page 88.)

\int_sign:n \_\_int_sign:Nw

See \int_abs:n. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in \_\_int_eval:w...\exp_stop_f: to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

(End definition for \_\_int_sign:Nw. This function is documented on page 89.)

\int_abs:n \_\_int_abs:N \int_max:nn \int_min:nn \_\_int_maxmin:wwN

Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

(End definition for \_\_int_abs:N and \_\_int_abs:N. This function is documented on page 89.)
As \__int_eval:w rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., \tl_count:n). If the numerator \#1\#2 is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator \#1\#2 towards 0 by (\#3\#4−1)/2, which we round away from zero. It turns out that this quantity exactly compensates the difference between \texttt{c-\TeX}’s rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

For the sake of completeness:

\begin{verbatim}
\cs_new:Npn \int_div_truncate:nn #1#2
\begin{verbatim}
\int_eval:w \__int_div_truncate:NwNw #1#2; #3#4;
\end{verbatim}
\end{verbatim}
\end{verbatim}
\begin{verbatim}
\int_eval:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: 
\end{verbatim}

For the sake of completeness:
Finally there’s the modulus operation.

```
\cs_new:Npn \int_mod:nn #1#2
\__int_eval:w \__int_eval:w \exp_after:wN \__int_mod:ww
\__int_value:w \__int_value:w #1 \exp_after:wN \__int_value:w #2 ;
\__int_eval:w \__int_eval:w \__int_eval_end:
```

```
\cs_new:Npn \__int_mod:ww #1; #2;
{ \#1 - ( \__int_div_truncate:NwNw \#1 ; \#2 ; ) * \#2 }
```

(End definition for \int_div_truncate:nn and others. These functions are documented on page 89.)

\_kernel\_int\_add:nn

Equivalent to \int_eval:n {#1+#2+#3} except that overflow only occurs if the final result overflows \([-2^{31} + 1, 2^{31} - 1]\). The idea is to choose the order in which the three numbers are added together. If \#1 and \#2 have opposite signs (one is in \([-2^{31} + 1, -1]\) and the other in \([0, 2^{31} - 1]\)) then \#1+\#2 cannot overflow so we compute the result as \#1+\#2+\#3. If they have the same sign, then either \#3 has the same sign and the order does not matter, or \#3 has the opposite sign and any order in which \#3 is not last will work. We use \#1+\#3+\#2.

```
\cs_new:Npn \__kernel_int_add:nnn #1#2#3
\__int_value:w \__int_eval:w #1
\if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
\if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
\__int_eval:w \__int_eval:w \__int_eval_end:
```

(End definition for \_kernel\_int\_add:nnn.)

12.2 Creating and initialising integers

\int_new:N \int_new:c

Two ways to do this: one for the format and one for the \LaTeX2ε package. In plain \TeX, \newcount (and other allocators) are \outer: to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to \newbox, \newdimen and so on.)

```
\cs_new_protected:Npn \int_new:N #1
{ \__kernel_chk_if_free_cs:N #1 \cs:w newcount \cs_end: #1 }
```

(End definition for \int_new:N. This function is documented on page 89.)

\int_const:Nn \int_const:cn
\__int_constdef:Nw \c__int_max_constdef_int

As stated, most constants can be defined as \chardef or \mathchardef but that’s engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use \int_gset:Nn because (when check-declarations is enabled) this runs some checks that constants would fail.

```
\cs_new_protected:Npn \int_const:Nn #1#2
```

508
\begin{verbatim}
\int_zero:N \int_zero:c \int_gzero:N \int_gzero:c
\end{verbatim}

Functions that reset an \texttt{(integer)} register to zero.

\begin{verbatim}
\int_zero_new:N \int_zero_new:c \int_gzero_new:N \int_gzero_new:c
\end{verbatim}

Create a register if needed, otherwise clear it.

(End definition for \texttt{\int_zero:N}, \texttt{\int_zero_new:N}, and \texttt{\int_gzero:N}. These functions are documented on page 90.)

(End definition for \texttt{\int_zero:N} and \texttt{\int_gzero:N}. These functions are documented on page 90.)
\int_set_eq:NN \int_set_eq:cN \int_set_eq:cc \int_gset_eq:NN \int_gset_eq:cN \int_gset_eq:cc

Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as TeX does it for us.

\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

(End definition for \int_set_eq:NN and \int_gset_eq:NN. These functions are documented on page 90.)

\int_if_exist_p:N \int_if_exist_p:c \int_if_exist:N \int_if_exist:c \int_if_exist:TF \int_if_exist:cTF

Copies of the cs functions defined in l3basics.

\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
\prg_new_eq_conditional:NNn { TF , T , F , p } \prg_new_eq_conditional:NNn { TF , T , F , p }

(End definition for \int_if_exist:NTF. This function is documented on page 90.)

12.3 Setting and incrementing integers

\int_add:Nn \int_add:cn \int_gadd:Nn \int_gadd:cn \int_sub:Nn \int_sub:cn \int_gsub:Nn \int_gsub:cn

Adding and subtracting to and from a counter.

\cs_new_protected:Npn \int_add:Nn #1#2 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_sub:Nn #1#2 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gadd:Nn #1#2 { \tex_global:D \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gsub:Nn #1#2 { \tex_global:D \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\cs_generate_variant:Nn \int_add:Nn { c }
\cs_generate_variant:Nn \int_sub:Nn { c }
\cs_generate_variant:Nn \int_gadd:Nn { c }
\cs_generate_variant:Nn \int_gsub:Nn { c }

(End definition for \int_add:Nn and others. These functions are documented on page 90.)

\int_incr:N \int_incr:c \int_gincr:N \int_gincr:c \int_decr:N \int_decr:c \int_gdecr:N \int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

\cs_new_protected:Npn \int_incr:N #1 { \tex_advance:D #1 \c_one_int }
\cs_new_protected:Npn \int_gincr:N #1 { \tex_global:D \tex_advance:D #1 \c_one_int }
\cs_new_protected:Npn \int_decr:N #1 { \tex_advance:D #1 - \c_one_int }
\cs_new_protected:Npn \int_gdecr:N #1 { \tex_global:D \tex_advance:D #1 - \c_one_int }
\cs_generate_variant:Nn \int_incr:N { c }
\cs_generate_variant:Nn \int_gincr:N { c }
\cs_generate_variant:Nn \int_decr:N { c }
\cs_generate_variant:Nn \int_gdecr:N { c }

(End definition for \int_incr:N and others. These functions are documented on page 90.)
12.4 Using integers

Here is how counters are accessed:

\cs_new_eq:NN \int_use:N \tex_the:D

We hand-code this for some speed gain:

\cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

(End definition for \int_use:N. This function is documented on page 91.)

12.5 Integer expression conditionals

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as != and >= are supported. The tests first evaluate their left-hand side, with a trailing \__int_compare_error:. This marker is normally not expanded, but if the relation symbol is missing from the test’s argument, then the marker inserts = (and itself) after triggering the relevant \TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed \__int_compare_error:Nw gets expanded, cleaning up the end of the test and telling the user what the problem was.

\cs_new_protected:Npn \__int_compare_error: \__int_compare_error:Nw

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as != and >= are supported, and multiple comparisons can be performed at once, for instance 0 < 5 <= 1. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary \__- int_compare:Nw reads one \langle operand \rangle and one \langle comparison \rangle symbol, and leaves roughly
in the input stream. Each call to this auxiliary provides the second operand of the last call's \if_int_compare:w. If one of the \langle \textit{comparisons} \rangle is false, the true branch of the \TeX conditional is taken (because of \reverse_if:N), immediately returning false as the result of the test. There is no \TeX conditional waiting the first operand, so we add an \if_false: and expand by hand with \int_value:w, thus skipping \prg_return_false: on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let \TeX evaluate this left hand side of the (in)equality using \__int_eval:w. Since the relation symbols \langle <, >, = \rangle are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, \__int_compare_error: is expanded, inserting \texttt{=} and itself after an error. In all cases, \__int_compare:w receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items \texttt{e} and \langle \texttt{nd,} \rangle, with a trailing \s__int_stop used to grab the entire argument when necessary.

The goal here is to find an \langle \textit{operand} \rangle and a \langle \textit{comparison} \rangle. The \langle \textit{operand} \rangle is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying \__int_to_roman:w, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with \__int_compare:Nw to determine the relation symbol, building a control sequence from it (\token_to_str:N gives better errors if \texttt{#1} is not a character). All the extended forms have an extra \texttt{=} hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by \TeX into \scan_stop:, ignored thanks to \unexpanded, and \__int_compare_error:Nw raises an error.
When the last \textit{operand} is seen, \texttt{__int\_compare\_end\_=:NNw} receives \texttt{e} and \texttt{nd\_} as arguments, hence calling \texttt{__int\_compare\_end\_=:NNw} to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls \texttt{__int\_compare\_nnN} where \texttt{#1} is \texttt{\if\_int\_compare\_w} or \texttt{\reverse\_if\_N} \texttt{\if\_int\_compare\_w}, \texttt{#2} is the \textit{operand}, and \texttt{#3} is one of \texttt{<, =, or >}. As announced earlier, we leave the \textit{operand} for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return \texttt{false}. Otherwise, we apply the conditional \texttt{#1} to the \textit{operand} \texttt{#2} and the comparison \texttt{#3}, and call \texttt{__int\_compare\_Nw} to look for additional operands, after evaluating the following expression.

\begin{verbatim}
\cs_new:cpn { __int\_compare\_end\_=:NNw } #1#2#3 e #4 \s__int\_stop
{\#3} \exp\_stop\_f:
\prg\_return\_false: \else: \prg\_return\_true: \fi:
\cs\_new\_Npn \__int\_compare\_nnN \#1\#2\#3
{\#2} \exp\_stop\_f:
\prg\_return\_false: \exp\_after\_wN \__int\_use\_none\_delimit\_by\_s\_stop:w
\fi:
\#1 \#2 \#3 \exp\_after\_wN \__int\_compare\_Nw \int\_value\_w \__int\_eval\_w
\end{verbatim}

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding \texttt{__int\_compare\_error\_Nw \langle token\rangle} responsible for error detection.

\begin{verbatim}
\cs\_new\_Ncpn { __int\_compare\_=:NNw } \#1\#2\#3 =
{ \__int\_compare\_nnN { \reverse\_if\_N \if\_int\_compare\_w } \#3 = }
\cs\_new\_Ncpn { __int\_compare\_<:NNw } \#1\#2\#3 <
{ \__int\_compare\_nnN { \reverse\_if\_N \if\_int\_compare\_w } \#3 < }
\cs\_new\_Ncpn { __int\_compare\_>:NNw } \#1\#2\#3 >
{ \__int\_compare\_nnN { \reverse\_if\_N \if\_int\_compare\_w } \#3 > }
\cs\_new\_Ncpn { __int\_compare\_==:NNw } \#1\#2\#3 ==
{ \__int\_compare\_nnN { \if\_int\_compare\_w } \#3 == }
\cs\_new\_Ncpn { __int\_compare\_!=:NNw } \#1\#2\#3 !=
{ \__int\_compare\_nnN { \if\_int\_compare\_w } \#3 != }
\cs\_new\_Ncpn { __int\_compare\_<:NNw } \#1\#2\#3 <=
{ \__int\_compare\_nnN { \if\_int\_compare\_w } \#3 <= }
\cs\_new\_Ncpn { __int\_compare\_:NNw } \#1\#2\#3 >=
{ \__int\_compare\_nnN { \if\_int\_compare\_w } \#3 >= }
\end{verbatim}

(End definition for \texttt{\_int\_compare\_nTF} and others. This function is documented on page \pageref{int_compare:nTF}.)

\texttt{\_int\_compare\_p\_Nn}\texttt{\_int\_compare\_nTNF} More efficient but less natural in typing.

\begin{verbatim}
\prg\_new\_conditional\_Npnn \int\_compare\_nN \#1\#2\#3 \{ p, T, F, TF \}
{ \if\_int\_compare\_w \__int\_eval\_w \#1 \#2 \__int\_eval\_w \#3 \__int\_eval\_end:
\prg\_return\_true:
\end{verbatim}
For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for \texttt{t}l_{case:nn}(TF) as described in l3tl.

\begin{verbatim}
\cs_new:Npn \int_case:nnTF #1 #2 #3 #4
\{ \exp:w \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { } \}
\cs_new:Npn \int_case:nnT #1 #2 #3
\{ \exp:w \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { } \}
\cs_new:Npn \int_case:nnF #1 #2 #3
\{ \exp:w \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { } \}
\cs_new:Npn \int_case:nn #1 #2 #3
\{ \exp:w \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { } \}
\cs_new:Npn \__int_case:nnTF #1 #2 #3 #4
\{ \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop \}
\cs_new:Npn \__int_case:nw #1 #2 #3 #4 #5 #6 #7
\{ \exp_end: #1 #4 \}
\end{verbatim}

(End definition for \texttt{int_compare:nNnTF}. This function is documented on page 91.)

\texttt{\int_if_odd_p:n}
\texttt{\int_if_odd:n TF}
\texttt{\int_if_even_p:n}
\texttt{\int_if_even:n TF}

A predicate function.

\begin{verbatim}
\prg_new_conditional:Nppnn \int_if_odd:n #1 \ p , \ T , \ F , \ TF \}
\{ \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
\prg_return_true:
\else:
\prg_return_false:
\fi:
\}
\prg_new_conditional:Nppnn \int_if_even:n #1 \ p , \ T , \ F , \ TF \}
\{ \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
\end{verbatim}
\begin{verbatim}
\prg_return_true:
\else:
\prg_return_false:
\fi:
\end{verbatim}

(End definition for \texttt{int\_if\_odd:nTF} and \texttt{int\_if\_even:nTF}. These functions are documented on page 93.)

\section{Integer expression loops}
\begin{itemize}
\item \texttt{int\_while\_do:nn}
\item \texttt{int\_until\_do:nn}
\item \texttt{int\_do\_while:nn}
\item \texttt{int\_do\_until:nn}
\end{itemize}
These are quite easy given the above functions. The \texttt{while} versions test first and then execute the body. The \texttt{do\_while} does it the other way round.

\begin{verbatim}
\cs_new:Npn \int\_while\_do:nn #1#2
\{ \int_compare:nT {#1} \{ \int\_while\_do:nn {#1} {#2} \} \}
\cs_new:Npn \int\_until\_do:nn #1#2
\{ \int_compare:nF {#1} \{ \int\_until\_do:nn {#1} {#2} \} \}
\cs_new:Npn \int\_do\_while:nn #1#2
\{ \int\_do\_while:nn {#1} {#2} \}
\cs_new:Npn \int\_do\_until:nn #1#2
\{ \int\_do\_until:nn {#1} {#2} \}
\end{verbatim}

(End definition for \texttt{int\_while\_do:nn} and others. These functions are documented on page 94.)

\begin{verbatim}
\int\_while\_do:nNn \int\_until\_do:nNn \int\_do\_while:nNn \int\_do\_until:nNn
\cs_new:Npn \int\_while\_do:nNn #1#2#3#4
\{ \int\_compare:nNnT {#1} #2 \{ \int\_while\_do:nNn {#1} {#2} \} \}
\cs_new:Npn \int\_until\_do:nNn #1#2#3#4
\{ \int\_compare:nNnF {#1} #2 \{ \int\_until\_do:nNn {#1} {#2} \} \}
\cs_new:Npn \int\_do\_while:nNn #1#2#3#4
\{ \int\_do\_while:nNn {#1} {#2} \}
\cs_new:Npn \int\_do\_until:nNn #1#2#3#4
\{ \int\_do\_until:nNn {#1} {#2} \}
\end{verbatim}

As above but not using the more natural syntax.
Before all else, evaluate the initial value, step, and final value. Repeating a function by
steps first needs a check on the direction of the steps. After that, do the function for the
start value then step and loop around. It would be more symmetrical to test for a step
size of zero before checking the sign, but we optimize for the most frequent case (positive
step).

\begin{verbatim}
\cs_new:Npn \int_step_function:nnnN #1#2#3
\{ \exp_after:wN \__int_step:wwwN \int_value:w \__int_eval:w #1 \exp_after:wN ;
\int_value:w \__int_eval:w #2 \exp_after:wN ; \int_value:w \__int_eval:w #3 ; \}
\cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
\{ \int_compare:nNnTF {#2} > \c_zero_int
{ \__int_step:NwnnN > }
\int_compare:nNnTF {#2} = \c_zero_int
{ \__int_step:NwnnN > }
\int_compare:nNnTF {#2} < \c_zero_int
{ \__int_step:NwnnN < }
\prg_break_point:
\}
\cs_new:Npn \__int_step:NwnnN #1; #2; #3; #4
\{ \exp_after:wN \__int_step:NwnnN \int_value:w \__int_eval:w #1 \exp_after:wN ;
\int_value:w \__int_eval:w #2 \exp_after:wN ; \int_value:w \__int_eval:w #3 ; \}
\end{verbatim}

(End definition for \int_while_do:nNnn and others. These functions are documented on page 94.)

\section{Integer step functions}

\subsection*{\int_step_function:nnnN}

Before all else, evaluate the initial value, step, and final value. Repeating a function by
steps first needs a check on the direction of the steps. After that, do the function for the
start value then step and loop around. It would be more symmetrical to test for a step
size of zero before checking the sign, but we optimize for the most frequent case (positive
step).
The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using \int_step_function:nnnN We put a \prg_break_point:Nn so that map_break functions from other modules correctly decrement \g__kernel_prg_map_int before looking for their own break point. The first argument is \scan_stop:, so that no breaking function recognizes this break point as its own.
12.8 Formatting integers

Nothing exciting here.

\begin{Verbatim}
\cs_new_eq:NN \int_to_arabic:n \int_eval:n
\end{Verbatim}

(End definition for |\int_to_arabic:n| and others. These functions are documented on page 95.)

\int_to_symbols:nnn\int_to_symbols:nnn

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don’t have to number from zero. Using an \texttt{f}-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a ‘nice’ number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

\begin{Verbatim}
\cs_new:Npn \int_to_symbols:nnn \#1\#2\#3
{\int_compare:nNnTF {#1} > {#2} {
\exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
\int_case:nn
\begin{Verbatim}
\{ 1 + \int_mod:nn { #1 - 1 } {#2} \} {#2} \} {#3}
\{#1\} {#2} {#3}
\} \int_case:nn {#1} {#3} }
\int_case:nn {#1} {#3}
\end{Verbatim}
\exp_args:Nf \int_to_symbols:nnn
\int_div_truncate:nn \#2 - 1 {#3} {#4}
\{#3\} {#4}
\{#1\}
\end{Verbatim}

(End definition for |\int_to_symbols:nnn| and \__int_to_symbols:nnnn. This function is documented on page 96.)

\int_to_alph:n\int_to_Alph:n

These both use the above function with input functions that make sense for the alphabet in English.

\begin{Verbatim}
\cs_new:Npn \int_to_alph:n \#1
{\int_to_symbols:nnn \#1 \{ 26 \}
\begin{Verbatim}
\{ 1 \} \{ a \}
\{ 2 \} \{ b \}
\{ 3 \} \{ c \}
\{ 4 \} \{ d \}
\{ 5 \} \{ e \}
\{ 6 \} \{ f \}
\{ 7 \} \{ g \}
\end{Verbatim}
\end{Verbatim}
\cs_new:Npn \int_to_Alph:n #1
\{ int_to_symbols:nnn {#1} { 26 } \}

(End definition for \int_to_alph:n and \int_to_Alph:n. These functions are documented on page 96.)
Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn’t perform it twice. Then check the sign, store it, either – or \c_empty_tl, and feed the absolute value to the next auxiliary function.

\begin{verbatim}
\cs_new:Npn \int_to_base:nn #1
\{ \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } \}
\cs_new:Npn \int_to_Base:nn #1
\{ \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } \}
\cs_new:Npn \__int_to_base:nn #1#2
\{\int_compare:nNnTF {#1} < 0
\{ \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - \}
\{ \__int_to_base:nnN {#1} {#2} \c_empty_tl \}
\}
\cs_new:Npn \__int_to_Base:nn #1#2
\{\int_compare:nNnTF {#1} < 0
\{ \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - \}
\{ \__int_to_Base:nnN {#1} {#2} \c_empty_tl \}
\}
\cs_new:Npn \__int_to_base:nnN #1#2#3
\{\int_compare:nNnTF {#1} < {#2}
\{ \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } \}
\{ \exp_args:Nf \__int_to_base:nnnN { \__int_to_letter:n { \int_mod:nn {#1} {#2} } } {#1} \}
\{#1\}
\{#2\}
\}
\cs_new:Npn \__int_to_Base:nnN #1#2#3
\{\int_compare:nNnTF {#1} < {#2}
\{ \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } \}
\{ \exp_args:Nf \__int_to_Base:nnnN { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } } \}
\{#1\}
\}
\cs_new:Npn \__int_to_base:nnnN #1#2#3#4
\{\int_compare:nNnTF {#1} < {#2}
\{ \exp_last_unbraced:Nf #3 #4 { \__int_to_letter:n {#1} } \}
\{ \exp_args:Nf \__int_to_base:nnnN { \__int_to_letter:n { \int_mod:nn {#1} {#2} } } \}
\{#1\}
\{#2\}
\{#3\}
\}
\cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
\{\int_compare:nNnTF {#1} < {#2}
\{ \exp_last_unbraced:Nf #3 #4 { \__int_to_Letter:n {#1} } \}
\{ \exp_args:Nf \__int_to_Base:nnnN { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } } \}
\{#1\}
\}
\end{verbatim}

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.
\cs_new:Npn \__int_to_Base:nnnN #1 #2 #3  
{ \exp_args:Nf \__int_to_Base:nnN \{ \int_div_truncate:nn {#2} {#3} \} \#3 \#4 \#1 \}

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use \int_case:nn, but in our case, the cases are contiguous, so it is forty times faster to use the \if_case:w primitive. The first \exp_after:wN expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since \#1 might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing \fi::

\cs_new:Npn \__int_to_letter:n #1  
{ \exp_after:wN \exp_after:wN \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:\a \or: b \or: c \or: d \or: e \or: f \or: g \or: h \or: i \or: j \or: k \or: l \or: m \or: n \or: o \or: p \or: q \or: r \or: s \or: t \or: u \or: v \or: w \or: x \or: y \or: z \else: \int_value:w \__int_eval:w \#1 \exp_after:wN \__int_eval_end: \fi: \}

\cs_new:Npn \__int_to_Letter:n #1  
{
\exp_after:wN \exp_after:wN
\if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
  \or: A
  \or: B
  \or: C
  \or: D
  \or: E
  \or: F
  \or: G
  \or: H
  \or: I
  \or: J
  \or: K
  \or: L
  \or: M
  \or: N
  \or: O
  \or: P
  \or: Q
  \or: R
  \or: S
  \or: T
  \or: U
  \or: V
  \or: W
  \or: X
  \or: Y
  \or: Z
\else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
\fi:
}

(End definition for \int_to_base:nn and others. These functions are documented on page 97.)

\int_to_bin:n Wrappers around the generic function.
\int_to_hex:n \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n \{ \int_to_base:nn {#1} { 2 } \}
\int_to_oct:n \cs_new:Npn \int_to_hex:n #1
\int_to_Base:nn \{ \int_to_base:nn {#1} { 16 } \}
\int_to_roman:n \cs_new:Npn \int_to_hex:n #1
\int_to_Roman:nn \{ \int_to_base:nn {#1} { 16 } \}
\int_to_oct:n \{ \int_to_base:nn {#1} { 8 } \}

(End definition for \int_to_bin:n and others. These functions are documented on page 97.)

\__int_to_roman:w The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the Q.
\__int_to_roman_i:w \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w \{ \exp_after:wN \__int_to_roman:N \__int_to_roman:w \int_eval:n {#1} \}
\__int_to_roman_x:w \__int_to_roman_d:w \__int_to_roman_c:w
\__int_to_roman_l:w \__int_to_roman_m:w \__int_to_roman_q:w
\__int_to_Roman_i:w \__int_to_Roman_v:w \__int_to_Roman_x:w
\__int_to_Roman_l:w \__int_to_Roman_c:w \__int_to_Roman_d:w

522
\cs_new:Npn \__int_to_roman:N #1
\use:c { \__int_to_roman_ #1 :w }
\__int_to_roman:N
\cs_new:Npn \int_to_Roman:n #1
\exp_after:wN \__int_to_Roman_aux:N
\__int_to_roman:w \int_eval:n {#1} Q
\cs_new:Npn \__int_to_Roman_aux:N #1
\use:c { \__int_to_Roman_ #1 :w }
\__int_to_Roman_aux:N
\cs_new:Npn \__int_to_Roman_i:w { i }
\cs_new:Npn \__int_to_Roman_v:w { v }
\cs_new:Npn \__int_to_Roman_x:w { x }
\cs_new:Npn \__int_to_Roman_l:w { l }
\cs_new:Npn \__int_to_Roman_c:w { c }
\cs_new:Npn \__int_to_Roman_d:w { d }
\cs_new:Npn \__int_to_Roman_m:w { m }
\cs_new:Npn \__int_to_Roman_Q:w #1 { }
\cs_new:Npn \__int_to_Roman_i:w { I }
\cs_new:Npn \__int_to_Roman_v:w { V }
\cs_new:Npn \__int_to_Roman_x:w { X }
\cs_new:Npn \__int_to_Roman_l:w { L }
\cs_new:Npn \__int_to_Roman_c:w { C }
\cs_new:Npn \__int_to_Roman_d:w { D }
\cs_new:Npn \__int_to_Roman_m:w { M }
\cs_new:Npn \__int_to_Roman_Q:w #1 { }

\__int_pass_signs:wn \__int_pass_signs_end:wn
\__int_pass_signs:wn
\if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
\exp_after:wN \__int_pass_signs:wn
\else:
\exp_after:wN \__int_pass_signs_end:wn
\exp_after:wN \#1
\fi:
\cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 \#1 }

(End definition for \int_to_Roman:n and others. These functions are documented on page 97.)

12.9 Converting from other formats to integers

Called as \__int_pass_signs:wn \{signs and digits\} \s__int_stop \{\langle code\rangle\}, this function leaves in the input stream any sign it finds, then inserts the \langle code\rangle before the first non-sign token (and removes \s__int_stop). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

\cs_new:Npn \__int_pass_signs:wn \#1
\if:w + \if:w - \exp_not:N \#1 + \fi: \exp_not:N \#1
\exp_after:wN \__int_pass_signs:wn
\else:
\exp_after:wN \__int_pass_signs_end:wn
\exp_after:wN \#1
\fi:
\cs_new:Npn \__int_pass_signs_end:wn \#1 \s__int_stop \#2 \{ \#2 \#1 }

(End definition for \__int_pass_signs:wn and \__int_pass_signs_end:wn.)
First take care of signs then loop through the input using the recursion quarks. The \__int_from_alph:nN auxiliary collects in its first argument the value obtained so far, and the auxiliary \__int_from_alph:N converts one letter to an expression which evaluates to the correct number.

\cs_new:Npn \int_from_alph:n #1
\int_eval:n
\exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
\s__int_stop { \__int_from_alph:nN { 0 } }
\q__int_recursion_tail \q__int_recursion_stop

\cs_new:Npn \__int_from_alph:nN #1#2
\__int_if_recursion_tail_stop_do:Nn #1 {#2}
\exp_args:Nf \__int_from_alph:nN
\int_eval:n { #1 * 26 + \__int_from_alph:N #2 }

\cs_new:Npn \__int_from_alph:N #1
{ '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

(End definition for \int_from_alph:n, \__int_from_alph:nN, and \__int_from_alph:N. This function is documented on page 97.)

Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of \__int_from_base:nnN. To convert a single character, \__int_from_base:N checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use \int_eval:n, hence is not safe for general use.

\cs_new:Npn \int_from_base:nn #1#2
\int_eval:n
\exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
\s__int_stop { \__int_from_base:nnN { 0 } {#2} }
\q__int_recursion_tail \q__int_recursion_stop

\cs_new:Npn \__int_from_base:nnN #1#2#3
\__int_if_recursion_tail_stop_do:Nn #3 {#1}
\exp_args:Nf \__int_from_base:nnN
\int_eval:n { #1 * #2 + \__int_from_base:N #3 }

\cs_new:Npn \__int_from_base:N #1
{ \int_compare:nNnTF { '#1 } < { 58 } { '#1 } }
{ '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }

(End definition for \int_from_base:nn, \__int_from_base:nnN, and \__int_from_base:N. This function is documented on page 98.)
Wrappers around the generic function.

\int_from_bin:n \int_from_hex:n \int_from_oct:n

\cs_new:Npn \int_from_bin:n #1
\cs_new:Npn \int_from_hex:n #1
\cs_new:Npn \int_from_oct:n #1

\int_const:cn { c__int_from_roman_i_int } { 1 }
\int_const:cn { c__int_from_roman_v_int } { 5 }
\int_const:cn { c__int_from_roman_x_int } { 10 }
\int_const:cn { c__int_from_roman_l_int } { 50 }
\int_const:cn { c__int_from_roman_c_int } { 100 }
\int_const:cn { c__int_from_roman_d_int } { 500 }
\int_const:cn { c__int_from_roman_m_int } { 1000 }
\int_const:cn { c__int_from_roman_I_int } { 1 }
\int_const:cn { c__int_from_roman_V_int } { 5 }
\int_const:cn { c__int_from_roman_X_int } { 10 }
\int_const:cn { c__int_from_roman_L_int } { 50 }
\int_const:cn { c__int_from_roman_C_int } { 100 }
\int_const:cn { c__int_from_roman_D_int } { 500 }
\int_const:cn { c__int_from_roman_M_int } { 1000 }

The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by \TeX. If any unknown letter is found, skip to the closing parenthesis and insert \texttt{*0-1} afterwards, to replace the value by \texttt{-1}.

\cs_new:Npn \int_from_roman:n #1
\int_if_exist:cF { c__int_from_roman_ #1 _int } { \__int_from_roman_error:w }
\int_if_exist:cF { c__int_from_roman_ #2 _int } { \__int_from_roman_error:w }
\int_compare:nNnTF
\begin{verbatim}
\use:c { c__int_from_roman_ #1 _int } }
<
\use:c { c__int_from_roman_ #2 _int } }
{
 + \use:c { c__int_from_roman_ #2 _int }
 - \use:c { c__int_from_roman_ #1 _int }
 \__int_from_roman:NN
}
{
 + \use:c { c__int_from_roman_ #1 _int }
 \__int_from_roman:NN #2
}
\cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
{ #2 * 0 - 1 }
\end{verbatim}

(End definition for \texttt{\int_from_roman:n}, \texttt{\__int_from_roman:NN}, and \texttt{\__int_from_roman_error:w}. This function is documented on page 98.)

12.10 Viewing integer

\begin{verbatim}
\int_show:N
\int_show:c
\__int_show:nN
\cs_new_eq:NN \int_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \int_show:N { c }
\end{verbatim}

(End definition for \texttt{\int_show:N} and \texttt{\__int_show:nN}. This function is documented on page 99.)

\int_show:n
We don’t use the \TeX{} primitive \texttt{\showthe} to show integer expressions: this gives a more unified output.

\begin{verbatim}
\cs_new_protected:Npn \int_show:n
{ \msg_show_eval:Nn \int_eval:n }
\end{verbatim}

(End definition for \texttt{\int_show:n}. This function is documented on page 99.)

\int_log:N
\int_log:c
\__int_log:nN
\cs_new_eq:NN \int_log:N \__kernel_register_log:N
\cs_generate_variant:Nn \int_log:N { c }

(End definition for \texttt{\int_log:N}. This function is documented on page 99.)

\int_log:n
Similar to \texttt{\int_show:n}.

\begin{verbatim}
\cs_new_protected:Npn \int_log:n
{ \msg_log_eval:Nn \int_eval:n }
\end{verbatim}

(End definition for \texttt{\int_log:n}. This function is documented on page 99.)

12.11 Random integers

\int_rand:nn
Defined in \texttt{l3fp-random}.

(End definition for \texttt{\int_rand:nn}. This function is documented on page 98.)
12.12 Constant integers

\c_zero_int The zero is defined in l3basics.
\c_one_int \int_const:Nn \c_one_int { 1 }  

(End definition for \c_zero_int and \c_one_int. These variables are documented on page 99.)

\c_max_int The largest number allowed is \(2^{31} - 1\)
\int_const:Nn \c_max_int { 2 147 483 647 }  

(End definition for \c_max_int. This variable is documented on page 99.)

\c_max_char_int The largest character code is 1114111 (hexadecimal 10FFFF) in Xe\TeX{} and Lua\TeX{} and 255 in other engines. In many places \TeX{} and up\TeX{} support larger character codes but for instance the values of \lccode{} are restricted to \([0, 255]\).
\int_const:Nn \c_max_char_int  
\{  
if_int_odd:w 0  
\cs_if_exist:NT \tex_luatexversion:D { 1 }  
\cs_if_exist:NT \tex_XeTeXversion:D { 1 } -  
"10FFFF  
\else:  
"FF  
\fi:  
\}  

(End definition for \c_max_char_int. This variable is documented on page 99.)

12.13 Scratch integers

\l_tmpa_int \l_tmpb_int \g_tmpa_int \g_tmpb_int  
\l_tmpa_int \int_new:N \l_tmpa_int  
\l_tmpb_int \int_new:N \l_tmpb_int  
\g_tmpa_int \int_new:N \g_tmpa_int  
\g_tmpb_int \int_new:N \g_tmpb_int  

(End definition for \l_tmpa_int and others. These variables are documented on page 99.)

12.14 Integers for earlier modules

<@@=seq>
\l__int_internal_a_int \l__int_internal_b_int  
\l__int_internal_a_int \int_new:N \l__int_internal_a_int  
\l__int_internal_b_int \int_new:N \l__int_internal_b_int  

(End definition for \l__int_internal_a_int and \l__int_internal_b_int.)  
\langle\textbackslash in@tex | package\rangle
13 \texttt{l3flag} implementation

The following test files are used for this code: \texttt{m3flag001}.

13.1 Non-expandable flag commands

The height $h$ of a flag (initially zero) is stored by setting control sequences of the form \texttt{\flag{\textlangle name\textrangle \textlangle integer\textrangle}} to \texttt{\relax} for $0 \leq \textlangle integer\textrangle < h$. When a flag is raised, a “trap” function \texttt{\flag{\textlangle name\textrangle}} is called. The existence of this function is also used to test for the existence of a flag.

\texttt{\flag_new:n} For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to \texttt{\relax}. This can be done expandably!

\begin{verbatim}
\cs_new_protected:Npn \flag_new:n #1
\{ \cs_new:cpn { flag~#1 } ##1 \; \{ \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: \} \}
\end{verbatim}

(End definition for \texttt{\flag_new:n}. This function is documented on page 102.)

\texttt{\flag_clear:n} Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don’t use \texttt{\cs_undefine:c} because that would act globally. When the option \texttt{\check-declarations} is used, check for the function defined by \texttt{\flag_new:n}.

\begin{verbatim}
\cs_new_protected:Npn \flag_clear:n #1 \{ \__flag_clear:wn 0 ; {#1} \}
\cs_new_protected:Npn \__flag_clear:wn #1 ; #2 \{ \if_cs_exist:w flag~#2~#1 \cs_end: \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D \exp_after:wN \__flag_clear:wn \int_value:w \int_eval:w 1 + #1 \else:\use_i:nnn \fi: ; {#2} \}
\end{verbatim}

(End definition for \texttt{\flag_clear:n} and \texttt{\__flag_clear:wn}. This function is documented on page 102.)

\texttt{\flag_clear_new:n} As for other datatypes, clear the (flag) or create a new one, as appropriate.

\begin{verbatim}
\cs_new_protected:Npn \flag_clear_new:n #1 \{ \if_cs_exist:w flag~#1 \cs_end: \cs_set_eq:cN { flag~#1 } \tex_undefined:D \exp_after:wN \__flag_clear:wn \int_value:w \int_eval:w 1 + #1 \else:\use_i:nnn \fi: ; {#1} \}
\end{verbatim}

(End definition for \texttt{\flag_clear_new:n}. This function is documented on page 102.)

\texttt{\flag_show:n} Show the height (terminal or log file) using appropriate \texttt{l3msg} auxiliaries.

\begin{verbatim}
\cs_new_protected:Npn \flag_show:n \{ \__flag_show:Nn \tl_show:n \}
\cs_new_protected:Npn \flag_log:n \{ \__flag_show:Nn \tl_log:n \}
\cs_new_protected:Npn \__flag_show:Nn #1#2 \{ \exp_args:Nc \__kernel_chk_defined:NT \{ flag~#2 \}
\}
\end{verbatim}

528
13.2 Expandable flag commands

\flag_if_exist_p:n \flag_if_exist:nTF
A flag exist if the corresponding trap \flag \{flag name\}:n is defined.

\flag_if_raised_p:n \flag_if_raised:nTF
Test if the flag has a non-zero height, by checking the 0 control sequence.

\flag_height:n \__flag_height_loop:wn \__flag_height_end:wn
Extract the value of the flag by going through all of the control sequences starting from 0.

\flag_raise:n
Simply apply the trap to the height, after expanding the latter.
14 \texttt{l3prg} implementation

The following test files are used for this code: \texttt{m3prg001.lvt,m3prg002.lvt,m3prg003.lvt}.

\*initex | package

14.1 Primitive conditionals

Those two primitive \TeX{} conditionals are synonyms. \texttt{\if_bool:N} is defined in \texttt{l3basics}, as it’s needed earlier to define quark test functions.

\begin{verbatim}
\cs_new_eq:NN \if_predicate:w \tex_ifodd:D
\end{verbatim}

(End definition for \texttt{\if_bool:N} and \texttt{\if_predicate:w}. These functions are documented on page 112.)

14.2 Defining a set of conditional functions

These are all defined in \texttt{l3basics}, as they are needed “early”. This is just a reminder!

(End definition for \texttt{\prg_set_conditional:Npnn} and others. These functions are documented on page 104.)

14.3 The boolean data type

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

\begin{verbatim}
\bool_new:N \bool_new:c
\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_new:N { c }
\end{verbatim}

(End definition for \texttt{\bool_new:N}. This function is documented on page 106.)

A merger between \texttt{\tl_const:Nn} and \texttt{\bool_set:Nn}.

\begin{verbatim}
\bool_const:Nn \bool_const:cn
\cs_new_protected:Npm \bool_set_true:N \bool_set_true:c
\cs_new_protected:Npm \bool_set_false:N \bool_set_false:c
\cs_new_protected:Npm \bool_gset_true:N \bool_gset_true:c
\cs_new_protected:Npm \bool_gset_false:N \bool_gset_false:c
\end{verbatim}

(End definition for \texttt{\bool_const:Nn}. This function is documented on page 107.)

Setting is already pretty easy. When \texttt{check-declarations} is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on \TeX{} registers.

\begin{verbatim}
\cs_new_protected:Npm \bool_set_true:N \bool_set_true:N \bool_set_true:c
\cs_new_protected:Npm \bool_set_false:N \bool_set_false:c
\cs_new_protected:Npm \bool_gset_true:N \bool_gset_true:c
\cs_new_protected:Npm \bool_gset_false:N \bool_gset_false:c
\end{verbatim}

530
(End definition for \texttt{\bool_set_true:N} and others. These functions are documented on page 107.)

\begin{verbatim}
\bool_set_eq:NN \bool_set_eq:cn \bool_set_eq:Nc \bool_set_eq:cc
\bool_gset_eq:NN \bool_gset_eq:cn \bool_gset_eq:Nc \bool_gset_eq:cc
\end{verbatim}

The usual copy code. While it would be cleaner semantically to copy the \texttt{\cs_set_eq:NN} family of functions, we copy \texttt{\tl_set_eq:NN} because that has the correct checking code.

\begin{verbatim}
\cs_set_eq:NN \cs_set_eq:cn \cs_set_eq:Nc \cs_set_eq:cc
\end{verbatim}

(End definition for \texttt{\bool_set_eq:NN} and \texttt{\bool_gset_eq:NN}. These functions are documented on page 107.)

\begin{verbatim}
\bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn
\end{verbatim}

This function evaluates a boolean expression and assigns the first argument the meaning \texttt{\c_true_bool} or \texttt{\c_false_bool}. Again, we include some checking code. It is important to evaluate the expression before applying the \texttt{\chardef} primitive, because that primitive sets the left-hand side to \texttt{\scan_stop} before looking for the right-hand side.

\begin{verbatim}
\cs_new_protected:Npn \bool_set:Nn \bool_set:cn \cs_generate_variant:Nn { c }
\end{verbatim}

(End definition for \texttt{\bool_set:Nn} and \texttt{\bool_gset:Nn}. These functions are documented on page 107.)

\section{14.4 Internal auxiliaries}

Internal recursion quarks.

\begin{verbatim}
\q__bool_recursion_tail \q__bool_recursion_stop
\end{verbatim}

(End definition for \texttt{\q__bool_recursion_tail} and \texttt{\q__bool_recursion_stop}.)

\begin{verbatim}
\bool_use_i_delimit_by_q_recursion_stop:nw
\end{verbatim}

Functions to gobble up to a quark.

\begin{verbatim}
\cs_new:Np \__bool_use_i_delimit_by_q_recursion_stop:nw #1 \q__bool_recursion_stop \{#1\}
\end{verbatim}

(End definition for \texttt{\__bool_use_i_delimit_by_q_recursion_stop:nw}.)

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

Functions to query recursion quarks.

\begin{verbatim}
\__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn
\end{verbatim}

(End definition for \texttt{\__bool_if_recursion_tail_stop_do:nn}.)
\bool_if_p:N
\bool_if_p:c
\bool_if:N
\bool_if:c
\bool_if:NTF
\bool_if:cTF

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\prg_return_true:
\prg_return_false:
\fi:
\prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }

(End definition for \bool_if:NTF. This function is documented on page 107.)

\bool_show:n
\bool_log:n
\__bool_to_str:n

Show the truth value of the boolean, as true or false.

\cs_new_protected:Npm \bool_show:n
\cs_new_protected:Npm \bool_log:n
\cs_new:Npm \__bool_to_str:n

(End definition for \bool_show:n, \bool_log:n, and \__bool_to_str:n. These functions are documented on page 107.)

\bool_show:N
\bool_show:c
\__bool_to_str:n

Show the truth value of the boolean, as true or false.

\cs_new_protected:Npm \bool_show:N { \__bool_to_str:n \tl_show:n }
\cs_new_protected:Npm \bool_log:N { \tl_log:n }
\cs_new_protected:Npm \__bool_to_str:n \tl_show:n \#1
\exp_args:Nx \token_to_str:N \#2 = \__bool_to_str:n \#2

(End definition for \bool_show:N, \bool_log:N, and \__bool_to_str:n. These functions are documented on page 107.)

\l_tmpa_bool
\l_tmpb_bool
\g_tmpa_bool
\g_tmpb_bool

A few booleans just if you need them.

\bool_new:N \l_tmpa_bool
\bool_new:N \l_tmpb_bool
\bool_new:N \g_tmpa_bool
\bool_new:N \g_tmpb_bool

(End definition for \l_tmpa_bool and others. These variables are documented on page 108.)

\bool_if_exist:p:N
\bool_if_exist:p:c
\bool_if_exist:NTF
\bool_if_exist:cTF

Copies of the cs functions defined in \l3basics.

\prg_new_eq_conditional:Nnn \bool_if_exist:N \cs_if_exist:N
\prg_new_eq_conditional:Nnn \bool_if_exist:c \cs_if_exist:c

(End definition for \bool_if_exist:NTF. This function is documented on page 108.)
14.5 Boolean expressions

\bool_if_p:n Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with ( and ) for grouping, ! for logical “Not”, && for logical “And” and || for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value (true) or (false).

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

\langle true \rangle And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

\langle false \rangle And Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return (false).

\langle true \rangle Or Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return (true).

\langle false \rangle Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

\langle true \rangle Close Current truth value is true, Close seen, return (true).

\langle false \rangle Close Current truth value is false, Close seen, return (false).

To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be \c_true_bool or \c_false_bool. We use a version of \tl_if_single:nTF optimized for speed since we know that an empty #1 is an error. The auxiliary \_\bool_if_p_aux:w removes the trailing parenthesis and gets rid of any space. For the general case, first issue a \group_align_safe_begin: as we are using && as syntax shorthand for the And operation and we need to hide it for
\TeX. This group is closed after \texttt{\_bool_get_next:NN} returns \texttt{\c_true_bool} or \texttt{\c_false_bool}. That function requires the trailing parenthesis to know where the expression ends.

\begin{verbatim}
\cs_new:Npn \bool_if_p:n { \exp_args:Nf \_bool_if_p:n }
\cs_new:Npn \_bool_if_p:n #1
\{\tl_if_empty:oT { \use_none:nn #1 . } { \_bool_if_p_aux:w }
\group_align_safe_begin:
\exp_after:wN
\group_align_safe_end:
\exp:w \exp_end_continue_f:w % ( \__bool_get_next:NN \use_i:nnnn #1 )
}\}
\cs_new:Npn \_bool_if_p_aux:w #1 \use_i:nnnn #2\#3 {#2}
\end{verbatim}

(End definition for \texttt{\bool_if_p:n}, \texttt{\_bool_if_p:n}, and \texttt{\_bool_if_p_aux:w}. This function is documented on page 109.)

\texttt{\_bool_get_next:NN} The GetNext operation. Its first argument is \texttt{\use_i:nnnn}, \texttt{\use_ii:nnnn}, \texttt{\use_iii:nnnn}, or \texttt{\use_iv:nnnn} (we call these “states”). In the first state, this function eventually expand to the truth value \texttt{\c_true_bool} or \texttt{\c_false_bool} of the expression which follows until the next unmatched closing parenthesis. For instance “\texttt{\_bool-get_next:NN \use_i:nnnn \c_true_bool || \c_true_bool )}” (including the closing parenthesis) expands to \texttt{\c_true_bool}. In the second state (after a !) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after \texttt{\c_true_bool||}) it always returns \texttt{\c_true_bool}. In the fourth state (after \texttt{\c_false_bool&&}) it always returns \texttt{\c_false_bool} and also stops when encountering ||, not only parentheses. This code itself is a switch: if what follows is neither ! nor (, we assume it is a predicate.

\begin{verbatim}
\cs_new:Npn \_bool_get_next:NN #1#2
{\use:c
 {\_bool-
 \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
 :Nw
 #1 \#2
 )}
\end{verbatim}

(End definition for \texttt{\_bool_get_next:NN}.)

\texttt{\_bool_!:Nw} The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after \texttt{\c_true_bool||} or \texttt{\c_false_bool&&} the ! is ignored.

\begin{verbatim}
\cs_new:cpn { \_bool_!:Nw } #1#2
{\exp_after:wN \_bool_get_next:NN
 #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
}
\end{verbatim}

(End definition for \texttt{\_bool_!:Nw}.)
The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

The arguments are \#1: a function such as \texttt{\texttt{use_i:nnn}}, \#2: 0 or 1 encoding the current truth value, \#3: the next operation, And, Or or Close. We distinguish three cases according to a combination of \#1 and \#2. Case 2 is when \#1 is \texttt{\texttt{use_iii:nnnn}} (state 3), namely after \texttt{\texttt{c_true_bool ||}}. Case 1 is when \#1 is \texttt{\texttt{use_i:nnnn}} and \#2 is true or when \#1 is \texttt{\texttt{use_ii:nnnn}} and \#2 is false, for instance for \texttt{\texttt{!c_false_bool}}. Case 0 includes the same with true/false interchanged and the case where \#1 is \texttt{\texttt{use_iv:nnnn}} namely after \texttt{\texttt{c_false_bool &&}}.

When seeing ) the current subexpression is done, leave the appropriate boolean. In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \texttt{\texttt{c_false_bool &&}} because that binds more tightly than ||. In the other two cases start skipping for \texttt{\texttt{c_true_bool ||}}.
\bool_lazy_all_p:n  Go through the list of expressions, stopping whenever an expression is \texttt{false}. If the end is reached without finding any \texttt{false} expression, then the result is \texttt{true}.

\bool_lazy_all:nTF

\_bool_lazy_all:n

\cs_new:Npn \bool_lazy_all_p:n #1
\prg_new_conditional:Nnn \bool_lazy_all:n #1 { \texttt{T} , \texttt{F} , \texttt{TF} }
\begin{verbatim}
\{ \_bool_lazy_all:n #1 \q__bool_recursion_tail \q__bool_recursion_stop \}
\end{verbatim}

\begin{verbatim}
\if_predicate:w \bool_lazy_all_p:n {#1}
\prg_return_true:
\else:
\prg_return_false:
\fi:
\end{verbatim}

\{ }
\cs_new:Npn \__bool_lazy_all:n #1
\begin{verbatim}
\__bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
\bool_if:nF {#1}
\{ \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } \}
\__bool_lazy_all:n
\end{verbatim}

(End definition for \texttt{\bool_lazy_all:nTF} and \texttt{\_bool_lazy_all:n}. This function is documented on page 109.)

\bool_lazy_and_p:nn
\bool_lazy_and:nnTF

\_bool_lazy_and:n

\prg_new_conditional:Nnn \bool_lazy_and:nn #1#2 { \texttt{p} , \texttt{T} , \texttt{F} , \texttt{TF} }
\begin{verbatim}
\if_predicate:w \bool_lazy_and_p:nn {#1}{#2} { \c_true_bool }
\bool_if:nF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
\prg_return_true:
\else:
\prg_return_false:
\fi:
\end{verbatim}

(End definition for \texttt{\bool_lazy_and:nnTF}. This function is documented on page 109.)

\bool_lazy_any_p:n
\bool_lazy_any:nTF

\_bool_lazy_any:n

\cs_new:Npn \bool_lazy_any_p:n #1
\prg_new_conditional:Nnn \bool_lazy_any:n #1 { \texttt{T} , \texttt{F} , \texttt{TF} }
\begin{verbatim}
\{ \_bool_lazy_any:n #1 \q__bool_recursion_tail \q__bool_recursion_stop \}
\end{verbatim}

\begin{verbatim}
\if_predicate:w \bool_lazy_any_p:n {#1}
\prg_return_true:
\else:
\prg_return_false:
\fi:
\end{verbatim}

\{ }
\cs_new:Npn \__bool_lazy_any:n #1
\begin{verbatim}
\__bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
\bool_if:nT {#1}
\{ \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } \}
\end{verbatim}

(End definition for \texttt{\bool_lazy_any:nTF} and \texttt{\_bool_lazy_any:n}. This function is documented on page 109.)
(End definition for \bool_lazy_any:nTF and \bool_lazy_any:n. This function is documented on page 110.)

\bool_lazy_or_p:nn \bool_lazy_or:nn

Only evaluate the second expression if the first is false.

\bool_not_p:n

The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

\bool_xor_p:nn \bool_xor:nn

Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

14.6 Logical loops

\bool_while_do:Nn \bool_while_do:cn
\bool_until_do:Nn \bool_until_do:cn

A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

(End definition for \bool_while_do:Nn and \bool_until_do:Nn. These functions are documented on page 110.)
A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

Loop functions with the test either before or after the first body expansion.

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed
down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of \texttt{\exp:w}, which ensures that \texttt{\prg_replicate:nn} only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don’t think this is a real limitation for any ordinary use, and if necessary, it is possible to write \texttt{\prg_replicate:nn \{1000\} \{\prg_replicate:nn \{100\} \{\texttt{code}\}\}}. An alternative approach is to create a string of \texttt{m}'s with \texttt{\exp:w} which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

\begin{verbatim}
9803 \cs_new:Npn \prg_replicate:nn #1 { \exp:w \exp_after:wN \__prg_replicate_first:N \int_value:w \int_eval:n {#1} \cs_end: }
9804 \cs_new:Npn \__prg_replicate_first:N #1 { \cs:w __prg_replicate_first_ #1 :n \__prg_replicate:N }
9805 \cs_new:Npn \__prg_replicate:N #1 { \cs_end: }
9806 \cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
9807 \cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
9808 \cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
9809 \cs_new:cpn { __prg_replicate_3:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
9810 \cs_new:cpn { __prg_replicate_4:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
9811 \cs_new:cpn { __prg_replicate_5:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
9812 \cs_new:cpn { __prg_replicate_6:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
9813 \cs_new:cpn { __prg_replicate_7:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
9814 \cs_new:cpn { __prg_replicate_8:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
9815 \cs_new:cpn { __prg_replicate_9:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
9816 \cs_new:cpn { __prg_replicate_first_-:n } #1 { \exp_end: \__kernel_msg_expandable_error:nn { kernel } { negative-replication } }
\end{verbatim}

Then comes all the functions that do the hard work of inserting all the copies. The first function takes \texttt{:n} as a parameter.

\begin{verbatim}
9817 \cs_new:Npn \__prg_replicate_:n #1 { \cs_end: }
9818 \cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
9819 \cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
9820 \cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
9821 \cs_new:cpn { __prg_replicate_3:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
9822 \cs_new:cpn { __prg_replicate_4:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
9823 \cs_new:cpn { __prg_replicate_5:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
9824 \cs_new:cpn { __prg_replicate_6:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
9825 \cs_new:cpn { __prg_replicate_7:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
9826 \cs_new:cpn { __prg_replicate_8:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
9827 \cs_new:cpn { __prg_replicate_9:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1#1 }
9828 \cs_new:cpn { __prg_replicate_first_-:n } #1 { \exp_end: \__kernel_msg_expandable_error:nn { kernel } { negative-replication } }
\end{verbatim}

Users shouldn’t ask for something to be replicated once or even not at all but...

\begin{verbatim}
9831 \cs_new:cpn { __prg_replicate_first_-:n } #1 { \exp_end: \__kernel_msg_expandable_error:nn { kernel } { negative-replication } }
\end{verbatim}

539
\cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
\cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
\cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
\cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
\cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_9:n } #1 { \exp_end: #1#1#1#1#1#1#1#1#1 }

(End definition for \prg_replicate:nn and others. This function is documented on page 111.)

14.8 Detecting \TeX’s mode

\mode_if_vertical_p:
\mode_if_vertical:TF
For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the \exp_end: in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

\mode_if_horizontal_p:
\mode_if_horizontal:TF
For testing horizontal mode.

\mode_if_inner_p:
\mode_if_inner:TF
For testing inner mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

\mode_if_math_p:
\mode_if_math:TF
(End definition for \mode_if_math:TF. This function is documented on page 111.)
14.9 Internal programming functions

\TeX's alignment structures present many problems. As Knuth says himself in *\TeX: The Program*: “It's sort of a miracle whenever \halign or \valign work, [...]” One problem relates to commands that internally issues a \textbackslash cr but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a & with category code 4 we get some sort of weird error message because the underlying \texttt{futurelet} stores the token at the end of the alignment template. This could be a & giving a message like \texttt{! Misplaced \textbackslash cr.} or even worse: it could be the \texttt{\endtemplate} token causing even more trouble! To solve this we have to open a special group so that \TeX still thinks it's on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The \TeXbook*...We place the \texttt{\if_false: { \fi:} part at that place so that the successive expansions of \texttt{\group_align_safe_begin/end:} are always brace balanced.

\begin{verbatim}
\cs_new:Npn \group_align_safe_begin: 
  { \if_int_compare:w \if_false: { \fi: } = \c_zero_int \fi: }
\cs_new:Npn \group_align_safe_end: 
  { \if_int_compare:w '{ = \c_zero_int } \fi: }
\end{verbatim}

(End definition for \texttt{\group_align_safe_begin:} and \texttt{\group_align_safe_end:}. These functions are documented on page 113.)

\texttt{\g__kernel_prg_map_int} A nesting counter for mapping.

\begin{verbatim}
\int_new:N \g__kernel_prg_map_int
\end{verbatim}

(End definition for \texttt{\g__kernel_prg_map_int}.)

\texttt{\prg_break_point:Nn} \texttt{\prg_map_break:Nn} These are defined in \texttt{l3basics}, as they are needed “early”. This is just a reminder that is the case!

(End definition for \texttt{\prg_break_point:Nn} and \texttt{\prg_map_break:Nn}. These functions are documented on page 112.)

\texttt{\prg_break_point:} \texttt{\prg_break:} \texttt{\prg_break:n} Also done in \texttt{l3basics} as in format mode these are needed within \texttt{l3alloc}.

(End definition for \texttt{\prg_break_point:}, \texttt{\prg_break:}, and \texttt{\prg_break:n}. These functions are documented on page 115.)

\begin{verbatim}
\langle @@=sys \rangle
\end{verbatim}

15 \texttt{l3sys} implementation

\begin{verbatim}
\langle @@=sys \rangle
\end{verbatim}

15.1 Kernel code

\begin{verbatim}
\langle *initex | package \rangle
\end{verbatim}

15.1.1 Detecting the engine

\texttt{\sys_const:nn} Set the T, F, TF, p forms of \texttt{#1} to be constants equal to the result of evaluating the boolean expression \texttt{#2}.

\begin{verbatim}
\cs_new_protected:Npn \sys_const:nn #1#2
  { \bool_if:nTF {#2} \cs_new:Npn \sys_const:nn #1\#2}
\end{verbatim}

541
\__sys_const:nn { sys_if_engine_ #1 } \str_if_eq_p:Vn \c_sys_engine_str {#1} \c_true_bool
}{
\c_if_exist_p:N \tex_uniformdeviate:D
}{ \__sys_const:nn { sys_if_engine_ #1 } \str_if_eq_p:Vn \c_sys_engine_str {#1} }
\tl_map_inline:nn { { luatex } { pdftex } { pteX } { uptex } { xetex } }
{ \__sys_const:nn { sys_if_engine_ #1 } \str_if_eq_p:Vn \c_sys_engine_str {#1} }

(End definition for \sys_if_engine_luatex:TF and others. These functions are documented on page 114.)

15.1.2 Randomness

This candidate function is placed there because \sys_if_rand_exist:TF is used in l3fp-rand.
\__sys_const:nn { sys_if_rand_exist } \cs_if_exist_p:N \tex_uniformdeviate:D

(End definition for \sys_if_rand_exist:TF. This function is documented on page 267.)

15.1.3 Platform

Setting these up requires the file module (file lookup), so is actually implemented there.

(End definition for \sys_if_platform_unix:TF, \sys_if_platform_windows:TF, and \c_sys_platform_str. These functions are documented on page 115.)
15.1.4 Configurations

Loading the backend code is pretty simply: check that the backend is valid, then load it up.

\cs_new_protected:Npn \sys_load_backend:n #1
\{\sys_finalise:\\c_sys_backend_str
\str_if_exist:NTF \c_sys_backend_str
\{ \str_if_eq:VnF \c_sys_backend_str {#1}
\{ \__kernel_msg_error:nn { sys } { backend-set } \}
\}
\}
\str_if_exist:TF \c_sys_backend_str
\{ \__sys_load_backend_check:N \c_sys_backend_str
\str_const:Nx \c_sys_backend_str { \c_sys_backend_str }
\__kernel_sys_configuration_load:n
\{ \__sys_load_backend_check:N \c_sys_backend_str \}
\}
\}
\cs_new_protected:Npn \__sys_load_backend_check:N #1
\{ \str_case:VnF #1
\{ dvisvgm \} \{}\}
\{ xdvipdfmx \} \{}
\}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}\{ \__kernel_msg_error:nnxx { sys } { wrong-backend }
\str_if_eq:VnF \c_sys_backend_str
\{ pdfmode \} \{}
\}
\_kernel\_msg\_error:nnxx \{ sys \} \{ wrong-backend \}
\ #1 \{ dvips \}
\tl\_get:Nn \#1 \{ dvips \}
\}
\}
\}
\}

\__(End definition for \sys\_load\_backend:n, \_sys\_load\_backend\_check:N, and \c\_sys\_backend\_str. These functions are documented on page 117.\)
\sys_if_shell:TF
{ \exp_args:No \__sys_get:nnN { \tl_to_str:n (#1) } (#2) #3 }
{ \prg_return_false: }
\cs_new_protected:Npn \__sys_get:nnN #1#2#3
{ \tl_if_in:nnTF {#1} { " } 
\__kernel_msg_error:nnx
{ kernel } { quote-in-shell } {#1}
\prg_return_false: }
\group_begin:
\if_false: { \fi:
\int_set_eq:NN \tex_tracingnesting:D \c_zero_int
\exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
#2 \scan_stop:
\exp_after:wN \__sys_get_do:Nw
\exp_after:wN #3
\exp_after:wN \prg_do_nothing:
\tex_input:D \ "#1" \scan_stop:
\if_false: } \fi:
\prg_return_true:
\group_end:
\tl_set:No #1 {#2}
(End definition for \sys_get_shell:nnNTF and others. These functions are documented on page 116.)

\__sys_shell_stream_int
This is not needed for Lua\TeX: shell escape there isn’t done using a \TeX interface.
\sys_if_engine_luatex:F
\{ \int_const:Nn \c__sys_shell_stream_int { 18 } \}
(End definition for \c__sys_shell_stream_int.)

\sys_shell_now:n
Execute commands through shell escape immediately.
\sys_if_engine_luatex:TF
{ \cs_new_protected:Npn \sys_shell_now:n #1
{ \lua_now:e
\l3kernel.shellescape(" \lua_escape:e \{ \tl_to_str:n (#1) \} ")
}
}
{ \iow_now:Nn \c__sys_shell_stream_int (#1) }

\sys_shell_shipout:n  \sys_shell_shipout:n Execute commands through shell escape at shipout.

\sys_shell_shipout:n\cs_new_protected:Npn \sys_shell_shipout:n #1\lua_shipout_e:n\l3kernel.shellescape(" \lua_escape:e { \tl_to_str:n {#1} } ") \}
\cs_new_protected:Npn \sys_shell_shipout:n #1\iow_shipout:Nn \c__sys_shell_stream_int {#1} \}
\cs_generate_variant:Nn \sys_shell_shipout:n { x }\end definition for \sys_shell_shipout:n. This function is documented on page 116.

15.2 Dynamic (every job) code
\sys_everyjob:\sys_everyjob:n\__sys_everyjob:n\tl_use:N \g__sys_everyjob_tl\tl_gclear:N \g__sys_everyjob_tl\cs_new_protected:Npn \__sys_everyjob:n #1\tl_gput_right:Nn \g__sys_everyjob_tl {#1} \}
\tl_new:N \g__sys_everyjob_tl\cs_generate_variant:Nn \sys_everyjob:n { x }\end definition for \sys_everyjob:n, \__sys_everyjob:n, and \g__sys_everyjob_tl. This function is documented on page ??.

15.2.1 The name of the job\c_sys_jobname_str Inherited from the \LaTeX3 name for the primitive. This has to be the primitive as it’s set in \everyjob. If the user does

\pdflatex \input some-file-name
then \everyjob is inserted before \jobname is changed form texput, and thus we would have the wrong result.
\__sys_everyjob:n\cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D \end definition for \c_sys_jobname_str. This variable is documented on page 114.)
15.2.2 Time and date

Copies of the information provided by \TeX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have \texttt{primitive} and it is working correctly. For \texttt{Init\TeX} of course that is all redundant but does no harm.

\begin{verbatim}
\__sys_everyjob:n
    \group_begin:
    \cs_set:Npn \__sys_tmp:w #1
    { \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 } { #1 }
    \cs_if_exist:NTF \tex_primitive:D
    { \bool_lazy_and:nnTF
        \sys_if_engine_xetex_p:
        { \int_compare_p:nNn { \exp_after:wN \use_none:n \tex_XeTeXrevision:D } { 99999 } { 0 } }
    { 0 }
    \} \{ \tex_primitive:D #1 \}
    \{ 0 \}
    \group_end:
\end{verbatim}

(End definition for \texttt{c/sys_minute_int} and others. These variables are documented on page 114.)

15.2.3 Random numbers

\texttt{sys_rand_seed}:

Unpack the primitive. When random numbers are not available, we return zero after an error (and incidentally make sure the number of expansions needed is the same as with random numbers available).

\begin{verbatim}
\__sys_everyjob:n
    \sys_if_rand_exist:TF
    \cs_new:Npn \sys_rand_seed:
        { \tex_the:D \tex_randomseed:D }
    \cs_new:Npn \sys_rand_seed:
        { \int_value:w 547
        }
\end{verbatim}
\_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
{ \sys_rand_seed: }
c_zero_int
)

(End definition for \sys_rand_seed:. This function is documented on page 115.)

\sys_gset_rand_seed:n
The primitive always assigns the seed globally.

\sys_gset_rand_seed:n
The primitive always assigns the seed globally.

\sys_shell_escape_int
Expose the engine’s shell escape status to the user.

\sys_gset_rand_seed:n
The primitive always assigns the seed globally.

15.2.4 Access to the shell

\sys_shell_escape_int
Expose the engine’s shell escape status to the user.

\sys_if_shell_p:
\sys_if_shell:TF
\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted:TF
Performs a check for whether shell escape is enabled. The first set of functions returns
true if either of restricted or unrestricted shell escape is enabled, while the other two sets
of functions return true in only one of these two cases.

\sys_shell_escape_int
Expose the engine’s shell escape status to the user.

\sys_gset_rand_seed:n
The primitive always assigns the seed globally.

(End definition for \sys_gset_rand_seed:n. This function is documented on page 115.)

15.2.4 Access to the shell

\sys_shell_escape_int
Expose the engine’s shell escape status to the user.

\sys_gset_rand_seed:n
The primitive always assigns the seed globally.

(End definition for \sys_gset_rand_seed:n. This function is documented on page 115.)

548
(End definition for \sys_if_shell:TF, \sys_if_shell_unrestricted:TF, and \sys_if_shell_restricted:TF. These functions are documented on page 116.)

15.2.5 Held over from l3file

\g_file_curr_name_str

See comments about \c_sys_jobname_str: here, as soon as there is file input/output, things get “tided up”.

(End definition for \g_file_curr_name_str. This variable is documented on page 163.)

15.3 Last-minute code

A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

(End definition for \sys_finalise:, \__sys_finalise:n, and \g_sys_finalise_tl. This function is documented on page 117.)

15.3.1 Detecting the output

This is a simple enough concept: the two views here are complementary.

(End definition for \sys_if_output_dvi:TF, \sys_if_output_pdf:TF, and \c_sys_output_str. These functions are documented on page 115.)

549
15.3.2 Configurations

\g__sys_backend_tl

As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```latex
\tl_new:N \g__sys_backend_tl
\__sys_finalise:n
{
  \tl_gset:Nx \g__sys_backend_tl
  {
    \sys_if_engine_xetex:TF
    {
      \sys_if_output_pdf:TF
      {
        \sys_if_engine_pdf:TF
        {
          \tl_gset:Nn \g__sys_backend_tl { pdfmode }
        }
        {
          \tl_gset:Nn \g__sys_backend_tl { dvips }
        }
      }
      {
        \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx }
      }
    }
    \sys_if_engine_dvisvgm:TF
    {
      \tl_gset:Nn \g__sys_backend_tl { dvisvgm }
    }
    \sys_if_engine_pdftex:TF
    {
      \tl_gset:Nn \g__sys_backend_tl { pdftex }
    }
    \sys_if_engine_xetex:TF
    {
      \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx }
    }
  }
}
\__sys_finalise:n
{
  \cs_if_exist:NT \@classoptionslist
  {
    \cs_if_eq:NNF \@classoptionslist \scan_stop:
    {
      \clist_map_inline:Nn \@classoptionslist
      {
        \str_case:nnT {#1}
        {
          { dvipdfmx }
          \tl_gset:Nn \g__sys_backend_tl { dvipdfmx }
        }
        { dvips }
        \tl_gset:Nn \g__sys_backend_tl { dvips }
      }
      \sys_if_engine_dvisvgm:TF
      {
        { dvisvgm }
        \tl_gset:Nn \g__sys_backend_tl { dvisvgm }
      }
      \sys_if_engine_pdf:TF
      {
        { pdfmode }
        \tl_gset:Nn \g__sys_backend_tl { pdfmode }
      }
      \sys_if_engine_xetex:TF
      {
        { xetex }
        \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx }
      }
      \sys_if_engine_pdftex:TF
      {
        { xdvipdfmx }
        \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx }
      }
    }
    \clist_remove_all:Nn \@unusedoptionlist {#1}
  }
}
(End definition for \g__sys_backend_tl)
```

(End definition for \g__sys_backend_tl)

```latex
\end{verbatim}

(End definition for \g__sys_backend_tl)
16  \texttt{l3clist} implementation

The following test files are used for this code: \texttt{m3clist002}.

\begin{verbatim}
\l__clist_internal_clist
Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before \texttt{clist_new:N}
\end{verbatim}

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

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

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

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

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

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

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

\begin{verbatim}
\l__clist_internal_clist
\end{verbatim}
16.1 Removing spaces around items

\__clist_trim_next:w Called as \exp:w \__clist_trim_next:w \prg_do_nothing: ⟨comma list⟩ ... it expands to {⟨trimmed item⟩} where the ⟨trimmed item⟩ is the first non-empty result from removing spaces from both ends of comma-delimited items in the ⟨comma list⟩. The \prg_do_nothing: marker avoids losing braces. The test for blank items is a somewhat optimized \tl_if_empty:oTF construction; if blank, another item is sought, otherwise trim spaces.

\cs_new:Npn \__clist_trim_next:w #1 ,
\begin{Verbatim}
\tl_if_empty:oTF { \use_none:nn #1 ? }
\{ \__clist_trim_next:w \prg_do_nothing: \}
\{ \tl_trim_spaces_apply:oN {#1} \exp_end: \}
\end{Verbatim}

(End definition for \__clist_trim_next:w.)

\__clist_sanitize:n \__clist_sanitize:Nn The auxiliary \__clist_sanitize:Nn receives a delimiter (\c_empty_tl the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls \__clist_wrap_item:w to unbrace the item (using a comma delimiter is safe since #2 came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

\cs_new:Npn \__clist_sanitize:n #1
\begin{Verbatim}
\exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
\exp:w \__clist_trim_next:w \prg_do_nothing:
\{ \__clist_if_recursion_tail_stop:n {#2}
\\#1 , \q__clist_recursion_tail , \q__clist_recursion_stop
\}
\end{Verbatim}
\cs_new:Npn \__clist_sanitize:Nn #1#2
\begin{Verbatim}
\__clist_if_recursion_tail_stop:n {#2}
\\#1 \__clist_wrap_item:w #2 ,
\\exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
\\exp:w \__clist_trim_next:w \prg_do_nothing:
\end{Verbatim}

(End definition for \__clist_sanitize:n and \__clist_sanitize:Nn.)

\__clist_if_wrap:nTF \__clist_if_wrap:w True if the argument must be wrapped to avoid getting altered by some clist operations. That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

All \l3clist functions go through the same test when they need to determine whether to brace an item, so it is not a problem that this test has false positives such as “\s__clist_mark ?”. If the argument starts or end with a space or contains a comma then one of the three arguments of \__clist_if_wrap:w will have its end delimiter (partly) in one of the three copies of #1 in \__clist_if_wrap:nTF; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise,
the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```
\prg_new_conditional:Nnnn \__clist_if_wrap:n #1 { TF }
{ }
\tl_if_empty:oTF
{ \__clist_if_wrap:w
  \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
  \s__clist_mark , ~ \s__clist_mark #1 ,
}
{ \tl_if_head_is_group:nTF { #1 { } }
  \tl_if_empty:nTF {#1}
  { \prg_return_true: }
  \tl_if_empty:oTF { \use_none:n #1}
  { \prg_return_true: }
  { \prg_return_false: }
}
{ \prg_return_false: }
{ \prg_return_true: }
\cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }
```

(End definition for \__clist_if_wrap:nTF and \__clist_if_wrap:w.)

```
\__clist_wrap_item:w
Safe items are put in \exp_not:n, otherwise we put an extra set of braces.
```

```
\cs_new:Npn \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:c \tl_new:c
```

(End definition for \clist_new:N. This function is documented on page 118.)

```
\__clist_wrap_item:w
Safe items are put in \exp_not:n, otherwise we put an extra set of braces.
```

```
\clist_const:Nn
\clist_const:cn
\clist_const:Nx
\clist_const:cx
```

Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```
\cs_new_protected:Npn \clist_const:Nn #1 \s__clist_mark ? - #2 - \s__clist_mark #3 , { }
```

(End definition for \clist_const:Nn. This function is documented on page 119.)
Clearing comma lists is just the same as clearing token lists.

Once again a copy from the token list functions.

Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.
Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

\begin{verbatim}
\cs_new_protected:Npn \clist_concat:NNN { \__clist_concat:NNNN \tl_set:Nx }
\cs_new_protected:Npn \clist_gconcat:NNN { \__clist_concat:NNNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4 { #1 #2 \\
\exp_not:o #3 \\
\clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } } \\
\exp_not:o #4 \\
}
\cs_generate_variant:Nn \clist_concat:NNN { ccc }
\cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
\end{verbatim}

(End definition for \clist_concat:NNN, \clist_gconcat:NNN, and \__clist_concat:NNNN. These functions are documented on page 119.)

Copies of the cs functions defined in \texttt{l3basics}.

\begin{verbatim}
\prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N \\
\prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
\end{verbatim}

(End definition for \clist_if_exist:NTF. This function is documented on page 119.)

### 16.3 Adding data to comma lists

\begin{verbatim}
\clist_set:Nn \clist_set:NV \clist_set:No \clist_set:Nx \\
\clist_gset:Nn \clist_gset:NV \clist_gset:No \clist_gset:Nx \\
\clist_put_left:Nn \clist_put_left:NV \clist_put_left:No \clist_put_left:Nx \\
\clist_gput_left:Nn \clist_gput_left:NV \clist_gput_left:No \clist_gput_left:Nx \\
\__clist_put_left:NNNn
\end{verbatim}

Everything is based on concatenation after storing in \texttt{\_\_clist_internal_clist}. This avoids having to worry here about space-trimming and so on.

\begin{verbatim}
\cs_new_protected:Npm \clist_put_left:Nn \clist_gput_left:Nn
\end{verbatim}

(End definition for \clist_set:Nn and \clist_gset:Nn. These functions are documented on page 120.)
10335 \{ \_clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn \}
10336 \cs_new_protected:Npn \_clist_put_left:NNNn #1#2#3#4
10337 { \_clist_internal_clist \#4 \#1 \#3 \_clist_internal_clist \#3 }
10338 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
10339 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
10340 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
10341 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

(End definition for \clist_put_left:Nn, \clist_gput_left:Nn, and \_clist_put_left:NNNn. These functions are documented on page 126.)

16.4 Comma lists as stacks

\clist_put_right:Nn \clist_put_right:NN \clist_put_right:NV \clist_put_right:Nx \clist_put_right:cn \clist_put_right:cV \clist_put_right:co \clist_put_right:cx \_clist_put_right:NNNn

\clist_get:NN \clist_get:cN \__clist_get:wN (End definition for \clist_put_right:Nn, \clist_gput_right:Nn, and \_clist_put_right:NNNn. These functions are documented on page 126.)

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list variables are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

\clist_get:NN \clist_get:cN \__clist_get:wN

An empty clist leads to \q_no_value, otherwise grab until the first comma and assign to the variable. The second argument of \_clist_pop:wwNNN is a comma list ending in a
comma and \texttt{s__clist_mark}, unless the original clist contained exactly one item: then
the argument is just \texttt{s__clist_mark}. The next auxiliary picks either \texttt{\exp_not:n} or
\texttt{\use_none:n} as \#2, ensuring that the result can safely be an empty comma list.

\begin{verbatim}
\cs_new_protected:Npn \clist_pop:NN { \__clist_pop:NNN \tl_set:Nx }
\cs_new_protected:Npn \clist_gpop:NN { \__clist_pop:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_pop:NNN #1 #2 #3 {
  \if_meaning:w #2 \c_empty_clist
    \tl_set:Nn #3 { \q_no_value }
  \else:
    \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1 #2 #3
  \fi:
}
\cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3 #4 #5 {
  \tl_set:Nn #5 {#1}
  #3 #4
  \prg_do_nothing:
    #2 \exp_not:o , \s__clist_mark \use_none:n \s__clist_stop
}
\cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
\cs_generate_variant:Nn \clist_pop:NN { c }
\cs_generate_variant:Nn \clist_gpop:NN { c }
\end{verbatim}

The same, as branching code: very similar to the above.

\begin{verbatim}
\prg_new_protected_conditional:Npnn \clist_get:NN #1 #2 { T , F , TF }
\prg_new_protected_conditional:Npnn \clist_pop:NN { c } { T , F , TF }
\prg_new_protected_conditional:Npnn \clist_gpop:NN { c } { T , F , TF }
\prg_new_protected_conditional:Npnn \__clist_pop:TF:NNN { \tl_set:Nx \tl_gset:Nx \tl_set:Nx } { T , F , TF }
\prg_new_protected_conditional:Npnn \__clist_get:NNN { \prg_do_nothing: } { T , F , TF }
\end{verbatim}

(End definition for \texttt{\clist_pop:NN} and others. These functions are documented on page 125.)
16.5 Modifying comma lists

Removing duplicates means making a new list then copying it.

Removing duplicates means making a new list then copying it.

```latex
\clist_new:N \l__clist_internal_remove_clist
\seq_new:N \l__clist_internal_remove_seq
```

(End definition for \l__clist_internal_remove_clist and \l__clist_internal_remove_seq.)

Pushing to a comma list is the same as adding on the left.

Pushing to a comma list is the same as adding on the left.

```latex
\clist_push:Nn \clist_push:NV \clist_push:No \clist_push:Nx \clist_push:cn \clist_push:cV \clist_push:co \clist_push:cx

\clist_gpush:Nn \clist_gpush:NV \clist_gpush:No \clist_gpush:Nx \clist_gpush:cn \clist_gpush:cV \clist_gpush:co \clist_gpush:cx
```

(End definition for \clist_push:Nn and \clist_gpush:Nn. These functions are documented on page 126.)

An internal comma list and a sequence for the removal routines.

An internal comma list and a sequence for the removal routines.

```latex
\clist_remove_duplicates:N \clist_remove_duplicates:c \clist_gremove_duplicates:N \clist_gremove_duplicates:c
```

(End definition for \clist_remove_duplicates:N and \clist_gremove_duplicates:N.)
The method used here for safe items is very similar to \tl_replace_all:Nnn. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with \l3seq code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For “safe” items, build a function delimited by the ⟨item⟩ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the ⟨item⟩. The loop is controlled by the argument grabbed by \__clist_remove_all:w when the item was found, the \s__clist_mark delimiter used is the one inserted by \__clist_tmp:w, and \__clist_use_none_delimit_by_s_stop:w is deleted. At the end, the final ⟨item⟩ is grabbed, and the argument of \__clist_tmp:w contains \s__clist_mark: in that case, \__clist_remove_all:w removes the second \s__clist_mark (inserted by \__clist_tmp:w), and lets \__clist_use_none_delimit_by_s_stop:w act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.
\begin{verbatim}
{
\exp_after:wN \use_none:n #3
}
}
\cs_new:Npn \__clist_remove_all:w \__clist_tmp:w ,
{\exp_not:n {#1}}
\cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , \#2 , { \exp_not:n {#1}}
\cs_generate_variant:Nn \clist_remove_all:Nn { c }
\cs_generate_variant:Nn \clist_gremove_all:Nn { c }
\end{verbatim}

(End definition for \clist_remove_all:Nn and others. These functions are documented on page 121.)

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

Use \clist_reverse:n in an x-expanding assignment. The extra work that \clist_reverse:n does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

\begin{verbatim}
\cs_new_protected:Npn \clist_reverse:N #1
{\tl_set:Nx #1 {\exp_args:No \clist_reverse:n {#1}}} \cs_new_protected:Npn \clist_greverse:N #1
{\tl_gset:Nx #1 {\exp_args:No \clist_reverse:n {#1}}}
\cs_generate_variant:Nn \clist_reverse:N { c }
\cs_generate_variant:Nn \clist_greverse:N { c }
\end{verbatim}

(End definition for \clist_reverse:N and \clist_greverse:N. These functions are documented on page 121.)

\clist_reverse:n
\__clist_reverse:wwNww
\__clist_reverse_end:ww

The reversed token list is built one item at a time, and stored between \s__clist_stop and \s__clist_mark, in the form of \"item\". We start from a comma list \"item\textsubscript{1},...,item\textsubscript{n}\". During the loop, the auxiliary \__clist_reverse:wwNww receives \"?item\textsubscript{i}\" as #1, \"item\textsubscript{i+1},...,item\textsubscript{n}\" as #2, \__clist_reverse:wwNww as #3, what remains until \s__clist_stop as #4, and \"item\textsubscript{i-1},...,item\textsubscript{1}\" as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, \__clist_reverse:wwNww receives \"s__clist_mark \__clist_reverse:wwNww \!\" as its argument #1, thus \__clist_reverse_end:ww as its argument #3. This second auxiliary cleans up until the marker \!, removes the trailing comma (introduced when the first item was moved after \s__clist_stop), and leaves its argument #1 within \exp_not:n. There is also a need to remove a leading comma, hence \exp_not:o and \use_none:n.

\begin{verbatim}
\cs_new:Npn \__clist_reverse:wwNww
#1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
{ #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark}
\cs_new:Npn \__clist_reverse_end:ww
#1 \! #2 , \s__clist_mark
{ \exp_not:o { \use_none:n #2}}
\end{verbatim}

(End definition for \clist_reverse:n, \__clist_reverse:wwNww, and \__clist_reverse_end:ww. This function is documented on page 121.)
Implemented in l3sort.

(End definition for \clist_sort:Nn and \clist_gsort:Nn. These functions are documented on page 121.)

16.6 Comma list conditionals

Simple copies from the token list variable material.

As usual, we insert a token (here ?) before grabbing any argument: this avoids losing braces. The argument of \tl_if_empty:oTF is empty if #1 is ? followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs \prg_return_false: as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing \s__clist_mark \prg_return_false: item.

For “safe” items, we simply surround the comma list, and the item, with commas, then use the same code as for \tl_if_in:Nn. For “unsafe” items we follow the same route as \seq_if_in:Nn, mapping through the list a comparison function. If found, return true and remove \prg_return_false:.
\cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
\{\__clist_if_wrap:nTF {#2}\}
\cs_set:Npx \__clist_tmp:w ##1 \{\exp_not:N \tl_if_eq:nnT {##1} \exp_not:n \{#2\} \clist_map_break:n \{\prg_return_true: \use_none:n \}\}
\clist_map_function:NN #3 \__clist_tmp:w \prg_return_false:\}
\prg_generate_conditional_variant:Nnn \clist_if_in:Nn \{N,V,No,c,cV,co\} \{T,F,TF\}
\prg_generate_conditional_variant:Nnn \clist_if_in:nn \{N,V,\} \{T,F,TF\}

(End definition for \clist_if_in:NnTF, \clist_if_in:nnTF, and \__clist_if_in_return:nnN. These functions are documented on page 192.)

16.7 Mapping to comma lists

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by \q__clist_recursion_tail. The auxiliary function \__clist_map_function:Nw is also used in \clist_map_inline:Nn.

\cs_new:Npn \clist_map_function:NN #1#2
\{\clist_if_empty:NF #1 \exp_last_unbraced:NNo \__clist_map_function:Nw #2 \ exp_last_unbraced:NNo \__clist_map_function:Nw \q__clist_recursion_tail \ \prg_break_point:Nn \clist_map_break: {}\}
\cs_new:Npn \__clist_map_function:Nw #1#2 ,
\{\__clist_if_recursion_tail_break:nN \{#2\} \clist_map_break: #1 \{#2\} \__clist_map_function:Nw \q__clist_recursion_tail \ \prg_break_point:Nn \clist_map_break: {}\}
\cs_generate_variant:Nn \clist_map_function:NN \{c\}
The \texttt{\_\_clist_map_function_n:NN} function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on \texttt{\_\_clist_trim_next:w}. The auxiliary \texttt{\_\_clist_map_function_n:NN} receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by \texttt{\_\_clist_map_unbrace:Nw}.

```
\cs_new:Npn \clist_map_function:nN #1#2
\exp_after:wN \_\_clist_map_function_n:Nn \exp_after:wN #2
\exp:w \_\_clist_trim_next:w \prg_do_nothing: #1 , \q__clist_recursion_tail ,
\prg_break_point:Nn \clist_map_break: { }
\prg_break_point:Nn \clist_map_break: { }
\cs_new:Npn \_\_clist_map_function_n:Nn #1 #2
\__clist_if_recursion_tail_break:nN {#2} \clist_map_break:
\_\_clist_map_unbrace:Nw #1 #2,
\exp_after:wN \_\_clist_map_function_n:NN \exp_after:wN #1
\exp:w \_\_clist_trim_next:w \prg_do_nothing:
\_\_clist_map_unbrace:Nw #1 #2, { #1 (#2) }
\cs_new:Npn \_\_clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with \TeX’s groups. We use a different function for each level of nesting.

```
\cs_new_protected:Npn \clist_map_inline:Nn \clist_map_inline:cn \clist_map_inline:nn
{\clist_if_empty:NF #1
\int_gincr:N \g__kernel_prg_map_int
\cs_gset_protected:cpn
\__clist_map:_int_use:N \g__kernel_prg_map_int :w } #1 {#2}
\exp_last_unbraced:Nco \_\_clist_map_function:Nw
\__clist_map:_int_use:N \g__kernel_prg_map_int :w }
\q__clist_recursion_tail ,
\prg_break_point:Nn \clist_map_break:
\{ \int_gdec:N \g__kernel_prg_map_int }
\prg_break_point:Nn \clist_map_break: { }
\cs_new_protected:Npn \clist_map_inline:nn #1
\clist_set:Nn \l__clist_internal_clist {#1}
\clist_map_inline:Nn \l__clist_internal_clist
\cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(End definition for \texttt{\_\_clist_map_function:nN} and \texttt{\_\_clist_map_function:NN}. This function is documented on page 122.)

(End definition for \texttt{\_\_clist_map_function:nN}, \texttt{\_\_clist_map_function:NN}, and \texttt{\_\_clist_map_unbrace:Nw}. These functions are documented on page 123.)
As for other comma-list mappings, filter out the case of an empty list. Same approach as \clist_map_function:Nn, additionally we store each item in the given variable. As for inline mappings, space trimming for the n variant is done by storing the comma list in a variable. The quark test is done before assigning the item to the variable: this avoids storing a quark which the user wouldn’t expect. The strange \use:n avoids unlikely problems when #2 would contain \q__clist_recursion_stop.

\begin{verbatim}
\cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
{ \clist_if_empty:NF #1
  { \exp_args:Nno \use:nn
    { \__clist_map_variable:Nnw #2 {#3} }
    #1
    \q__clist_recursion_tail \q__clist_recursion_stop
    \prg_break_point:Nn \clist_map_break: { } }
}
\cs_new_protected:Npn \clist_map_variable:nNn #1
{ \clist_set:Nn \l__clist_internal_clist {#1} \clist_map_variable:NNn \l__clist_internal_clist }
\cs_new:Npn \__clist_map_variable:Nnw #1#2#3, 
{ \__clist_if_recursion_tail_stop:n {#3} \tl_set:Nn #1 {#3} \use:n {#2} \__clist_map_variable:Nnw #1 {#2} }
\cs_generate_variant:Nn \clist_map_variable:NNn { c }
\end{verbatim}

\begin{itemize}
\item \clist_map_break: The break statements use the general \prg_map_break:Nn mechanism.
\item \clist_map_break:n
\end{itemize}

\begin{verbatim}
\cs_new:Npn \clist_map_break:
{ \prg_map_break:Nn \clist_map_break: { } }
\cs_new:Npn \clist_map_break:n
{ \prg_map_break:Nn \clist_map_break: }
\end{verbatim}

\begin{itemize}
\item \clist_count:N
\item \clist_count:c
\item \__clist_count:n
\item \__clist_count:w
\end{itemize}

Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics. In the case of an n-type comma-list, we could of course use \clist_map_function:nN, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not {}), hence the extra spaces.

\begin{verbatim}
\cs_new:Npn \clist_count:N #1
{ \int_eval:n
  { \clist_map_function:NN #1 \__clist_count:n
    #1
  }
}
\end{verbatim}

564
16.8 Using comma lists

First check that the variable exists. Then count the items in the comma list. If it has
none, output nothing. If it has one item, output that item, brace stripped (note that
space-trimming has already been done when the comma list was assigned). If it has two,
place the ⟨separator between two⟩ in the middle.

Otherwise, \clist_use:Nnnn takes the following arguments; 1: a ⟨separator⟩,
2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6:
a ⟨continuation⟩ function (use_ii or use_iii with its ⟨separator⟩ argument), 7: junk,
and 8: the temporary result, which is built in a brace group following \q__clist_stop.
The ⟨separator⟩ and the first of the three items are placed in the result, then we use
the ⟨continuation⟩, placing the remaining two items after it. When we begin this loop,
the three items really belong to the comma list, the first \q__clist_mark is taken as a
delimiter to the use_ii function, and the continuation is use_ii itself. When we reach
the last two items of the original token list, \q__clist_mark is taken as a third item,
and now the second \q__clist_mark serves as a delimiter to use_ii, switching to the other
(continuation), use_iii, which uses the ⟨separator between final two⟩.
\s__clist_mark , \{ \s__clist_use:nwwn \{#4\} \}
\s__clist_stop \{ \}
}

\cs_generate_variant:Nn \clist_use:Nnnn { c }
\cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
\cs_new:Npn \__clist_use:nwwwwnwn #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8 \{ #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop \{ #8 #1 #2 \} \}
\cs_new:Npn \__clist_use:nwwn #1#2 , #3 \s__clist_stop #4 { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \clist_use:Nn #1#2 \{ \clist_use:Nnnn #1 {#2} {#2} {#2} \}
\cs_generate_variant:Nn \clist_use:Nn { c }

(End definition for \clist_use:nnnn and others. These functions are documented on page 124.)

16.9 Using a single item

To avoid needing to test the end of the list at each step, we first compute the \langle length \rangle of the list. If the item number is 0, less than \langle length \rangle, or more than \langle length \rangle, the result is empty. If it is negative, but not less than \langle length \rangle, add \langle length \rangle + 1 to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

\begin{verbatim}
\cs_new:Npn \clist_use:Nn #1#2 \{ \clist_count:N #1 \{ \int_eval:n {#2} \} #1 \}
\cs_new:Npn \__clist_item:ffon \{ \int_compare:nNnTF {#2} < 0 \{ \__clist_use:none_delimit_by_s_stop:w \} \{ \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } \} \}
\cs_new:Npn \__clist_item:N_loop:nw #1 #2, \}
\cs_new:Npn \__clist_item:fnN \{ \int_compare:nNnTF {#2} > {#1} \{ \__clist_use:none_delimit_by_s_stop:w \} \{ \exp_args:Nf #4 { \int_eval:n { #2 - 1 - #1 } } \} \}
\cs_new:Npn \__clist_item:N_loop:nw #1 #2, \}
\cs_generate_variant:Nn \__clist_item:ffon { ffo, ff }
\end{verbatim}

566
\begin{quote}
\texttt{\textbackslash int\_compare:NNTF \{#1\} = 0}
\texttt{\{ \_\_clist\_use\_i\_delimit\_by\_s\_stop:nw \{ \exp\_not:n \{#2\} \} \}}
\texttt{\{ \exp\_args:Nf \_\_clist\_item\_N\_loop:nw \{ \int\_eval:n \{#1 - 1\}\} \}}
\texttt{\}}
\texttt{\cs\_generate\_variant:Nn \_\_clist\_item:Nn \{ c \}}
\end{quote}

(End definition for \texttt{\_\_clist\_item:Nn}, \texttt{\_\_clist\_item:nnN}, and \texttt{\_\_clist\_item\_N\_loop:nw}. This function is documented on page 126.)

This starts in the same way as \texttt{\_\_clist\_item:NN} by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking \texttt{\prg\_do\_nothing:} to avoid losing braces. Blank items are ignored.

\begin{quote}
\texttt{\cs\_new:Npn \_\_clist\_item:nn \#1\#2}
\texttt{\texttt{\{ \_\_clist\_item:ffnN}}
\texttt{\{ \clist\_count:n \{#1\}\}}
\texttt{\{ \int\_eval:n \{#2\}\}}
\texttt{\}}
\texttt{\_\_clist\_item:n:nw}
\texttt{\_\_clist\_item\_N\_loop:nw}
\texttt{\_\_clist\_item\_N\_end:n}
\texttt{\_\_clist\_item\_strip:n}
\texttt{\_\_clist\_item\_strip:w}
\end{quote}

The \texttt{N}-type function is not implemented through the \texttt{n}-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptyness of an \texttt{n}-type comma-list is slow, so we count items first and use that both for the emptyness test and the pseudo-random integer. Importantly, \texttt{\_\_clist\_item:NN} and \texttt{\_\_clist\_item:nn} only evaluate their argument once.

\begin{quote}
\texttt{\cs\_new:Npn \_\_clist\_rand\_item:n \#1}
\texttt{\texttt{\{ \exp\_args:Nf \_\_clist\_rand\_item:nn \{ \clist\_count:n \{#1\}\} \{#1\}\}}}
\texttt{\cs\_new:Npn \_\_clist\_rand\_item:N \#1 #2 \_\_clist\_stop}
\texttt{\{ \_\_clist\_trim\_spaces\_apply:nw \{#1\} \_\_clist\_item\_N\_strip:n \}}
\texttt{\cs\_new:Npn \_\_clist\_rand\_item:c \#1 \_\_clist\_item\_N\_strip:w #1 , \{ \exp\_not:n \{#1\}\}}
\texttt{\cs\_new:Npn \_\_clist\_rand\_item:nn \#1 #2 \_\_clist\_item\_N\_strip:w #1 , \{ \exp\_not:n \{#1\}\}}
\end{quote}

(End definition for \texttt{\_\_clist\_item:nn} and others. This function is documented on page 126.)
16.10 Viewing comma lists

\clist_show:N  \clist_log:N  \__clist_show:NN
Apply the general \__kernel_chk_defined:NT and \msg_show:nnnnnn.
\clist_show:c  \clist_log:c  \__clist_show:NN
\clist_new:NN \clist_new:cN \clist_new:N \clist_new:N
Temporary comma list variables.
\l_tmpa_clist  \l_tmpb_clist  \g_tmpa_clist  \g_tmpb_clist

(End definition for \clist_show:N, \clist_log:N, and \__clist_show:NN. These functions are documented on page 126.)

(End definition for \clist_show:N, \clist_log:N, and \__clist_show:NN. These functions are documented on page 127.)

16.11 Scratch comma lists

\l_tmpa_clist  \l_tmpb_clist  \g_tmpa_clist  \g_tmpb_clist

(End definition for \l_tmpa_clist and others. These variables are documented on page 127.)
17 13token implementation

17.1 Internal auxiliaries

\s__char_stop

Internal scan mark.

\scan_new:N \s__char_stop

(End definition for \s__char_stop.)

\q__char_no_value

Internal recursion quarks.

\quark_new:N \q__char_no_value

(End definition for \q__char_no_value.)

\__char_quark_if_no_value_p:N \__char_quark_if_no_value:N

Functions to query recursion quarks.

\__kernel_quark_new_conditional:Nn \__char_quark_if_no_value:N { TF }

(End definition for \__char_quark_if_no_value:NTF.)

17.2 Manipulating and interrogating character tokens

Simple wrappers around the primitives.

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n

Simple wrappers around the primitives.

\cs_new_protected:Npn \char_set_catcode:nn #1#2
{ \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }

\cs_new:Npn \char_value_catcode:n #1
{ \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }

\cs_new_protected:Npn \char_show_value_catcode:n #1
{ \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }

(End definition for \char_set_catcode:nn, \char_value_catcode:n, and \char_show_value_catcode:n. These functions are documented on page 131.)

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_if_no_value_p:N \char_set_catcode_if_no_value:N
\char_set_catcode_comment:N
\char_set_catcode_active:N
\char_set_catcode_invalid:N

(End definition for \char_set_catcode:nn, \char_value_catcode:n, and \char_show_value_catcode:n. These functions are documented on page 131.)
Pretty repetitive, but necessary!
\section*{17.3 Creating character tokens}

Four simple functions with very similar definitions, so set up using an auxiliary. These functions are similar to LuaTeX's \texttt{\let\char\code} primitive.

\begin{verbatim}
\char_set_active_eq:NN \char_gset_active_eq:Nc
\char_set_active_eq:Nc \char_gset_active_eq:Nc
\char_set_active_eq:nn \char_gset_active_eq:nn
\end{verbatim}

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example \texttt{\ExplSyntaxOn}).

\begin{verbatim}
\l_char_active_seq \l_char_special_seq
\end{verbatim}

(End definition for \texttt{\char_set_mathcode:nn} and others. These functions are documented on page 132.)

\begin{verbatim}
\l_char_active_seq \l_char_special_seq
\end{verbatim}

(End definition for \texttt{\char_set_active_eq:nn} and \texttt{\l_char_special_seq}. These variables are documented on page 132.)
\__char_int_to_roman:w

For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

\char_generate:nn
\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
\l__char_tmp_tl
\__char_generate_invalid_catcode:

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (Xe\TeX, Lua\TeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

\cs_new:Npn \char_generate:nn #1#2
{ \exp:w \exp_after:wN \__char_generate_aux:w
\int_value:w \int_eval:n {#1} \exp_after:wN ;
\int_value:w \int_eval:n {#2} ;
}

Before doing any actual conversion, first some special case filtering. Spaces are out here as Lua\TeX emulation only makes normal (charcode 32 spaces). However, \^^@ is filtered out separately as that can’t be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

\cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
{ \if_int_compare:w #2 = 10 \exp_stop_f:
\if_int_compare:w #1 = 0 \exp_stop_f:
\__kernel_msg_expandable_error:nn { kernel } { char-null-space }
\else:
\__kernel_msg_expandable_error:nn { kernel } { char-space }
\fi:
\else:
\__kernel_msg_expandable_error:nn { kernel } { char-invalid-catcode }
\fi:
\else:
\if_int_odd:w 0
\if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
\if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
\if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
\if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
\__kernel_msg_expandable_error:nn { kernel }
{ char-invalid-catcode }
\else:
\if_int_odd:w 0
\if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
\fi:
Engine-dependent definitions are now needed for the implementation. For \texttt{LuaTEX} and \texttt{XeTEX} there is engine-level support. They can do cases that macro emulation can’t. All of those are filtered out here using a primitive-based boolean expression to avoid fixing the category code of the null character used in the false branch (for 8-bit engines). The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of \texttt{XeTEX} cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.
For engines where \Ucharcat isn’t available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a \tl containing \^)^\@ with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a \if_case:w low-level conditional. There are a few things to notice here. As \^L is \outer we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using \if_false: pairing and are then x-type expanded together into the desired form.

\tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
\char_set_catcode_group_begin:n { 0 } % { 
\tl_put_right:Nn \l__char_tmp_tl { \^)^\@ \if_false: } }
\char_set_catcode_group_end:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \^)^\@ { \fi: \exp_not:N \or: } % }
\tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
\char_set_catcode_math_toggle:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_alignment:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_parameter:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_math_superscript:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_math_subscript:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\tl_put_right:Nn \l__char_tmp_tl { \or: }

For making spaces, there needs to be an o-type expansion of a \use:n (or some other tokenization) to avoid dropping the space.

\char_set_catcode_space:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \use:n { \or: } \^)^\@ }
\char_set_catcode_letter:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_other:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }
\char_set_catcode_active:n { 0 }
\tl_put_right:Nn \l__char_tmp_tl { \or: \^)^\@ }

Convert the above temporary list into a series of constant token lists, one for each character code, using \tex_lowercase:D to convert \^)^\@ in each case. The x-type expansion ensures that \tex_lowercase:D receives the contents of the token list. In package mode, \^L is awkward hence this is done in three parts. Notice that at this stage \^)^\@ is active.

\cs_set_protected:Npn \__char_tmp:n #1
\char_set_lccode:nn { 0 } {#1}
\char_set_lccode:nn { 32 } {#1}
\exp_args:Nx \tex_lowercase:D
As \TeX{} is very unhappy if it finds an alignment character inside a primitive \texttt{\halign} even when skipping false branches, some precautions are required. \TeX{} is happy if the token is hidden between braces within \texttt{\if_false: ... \fi:}.

\begin{verbatim}
\cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
{#3}
\if_false: { \fi:
\exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
\exp_after:wN \exp_after:wN \if_case:w #2
\exp_last_unbraced:Nv \exp_stop_f:
{ \exp_not:c { \c__char_ \__char_int_to_roman:w #1 _tl }
\or: }
\fi:
\fi:
\group_end:
\end{verbatim}

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see \url{https://en.wikipedia.org/wiki/UTF-8} for the octet pattern.
```latex
\else:
  \if_int_compare:w #1 < "10000 \exp_stop_f:
    \__char_to_utfviii_bytes_outputi:nw
    \{ \__char_to_utfviii_bytes_auxii:Nnn E {#1} { 64 * 64 } \}
    \__char_to_utfviii_bytes_outputii:nw
    \{ \int_div_truncate:nn {#1} { 64 } \}
    \__char_to_utfviii_bytes_outputiii:nw
    \{ \__char_to_utfviii_bytes_auxiii:n {#1} \}
  \else:
    \__char_to_utfviii_bytes_outputi:nw
    \{ \__char_to_utfviii_bytes_auxii:Nnn F {#1} { 64 * 64 * 64 } \}
    \__char_to_utfviii_bytes_outputii:nw
    \{ \int_div_truncate:nn {#1} { 64 * 64 } \}
    \__char_to_utfviii_bytes_outputiii:nw
    \{ \int_div_truncate:nn {#1} { 64 } \}
    \__char_to_utfviii_bytes_outputiv:nw
    \{ \__char_to_utfviii_bytes_auxiii:n {#1} \}
  \fi:
\fi:
\__char_to_utfviii_bytes_end: { } { } { } { }
\cs_new:Npn \__char_to_utfviii_bytes_auxii:Nnn #1#2#3
  { "#10 + \int_div_truncate:nn {#2} {#3} }
\cs_new:Npn \__char_to_utfviii_bytes_auxiiii:n { \int_div_truncate:nn {#1} { 64 } + 128 }
\cs_new:Npn \__char_to_utfviii_bytes_auxiii:n #1
  { \int_mod:nn {#1} { 64 } + 128 }
\cs_new:Npn \__char_to_utfviii_bytes_outputi:nw #1
  \{ \__char_to_utfviii_bytes_outputi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputii:nw #1
  \{ \__char_to_utfviii_bytes_outputii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiii:nw #1
  \{ \__char_to_utfviii_bytes_outputiii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputiv:nw #1
  \{ \__char_to_utfviii_bytes_outputiv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputv:nw #1
  \{ \__char_to_utfviii_bytes_outputv:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvi:nw #1
  \{ \__char_to_utfviii_bytes_outputvi:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputvii:nw #1
  \{ \__char_to_utfviii_bytes_outputvii:nw \}
\cs_new:Npn \__char_to_utfviii_bytes_outputviii:nw #1
  \{ \__char_to_utfviii_bytes_outputviii:nw \}
```

576
Look up any \texttt{nfd} and recursively produce the result.

\begin{verbatim}
\char_to_nfd:N \__char_to_nfd:n \__char_to_nfd:Nw
\end{verbatim}

To ensure that the category codes produced are predictable, every character is re-generated even if it is otherwise unchanged. This makes life a little interesting when we might have multiple output characters: we have to grab each of them and case change them in reverse order to maintain \texttt{f}-type expandability.

\begin{verbatim}
\end{verbatim}
\cs_new:Npn \__char_change_case:nNN #1#2#3
{ \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl } }
{ \exp_args:Nf \__char_change_case:nN { #2 { '#3 } } #3 }
}

\cs_new:Npn \__char_change_case:NNN #1#2#3
{ \exp_args:Nnf \use:nn
{ \__char_change_case:NN #1 #2 }
{ \__char_change_case:NN #1 #3 }
}

\cs_new:Npn \__char_change_case:NNNN #1#2#3#4
{ \exp_args:Nnff \use:nnn
{ \__char_change_case:NN #1 #2 }
{ \__char_change_case:NN #1 #3 }
{ \__char_change_case:NN #1 #4 }
}

\cs_new:Npn \__char_quark_if_no_value:NTF #1
{ \__char_quark_if_no_value:NTF #1 }

\cs_new:Npn \__char_quark_if_no_value:NTF #2
{ \__char_quark_if_no_value:NTF #2 }

\cs_new:Npn \__char_generate:nn { \__char_change_case:NNN #1 #2 #3 #4 }
Same story for the string version, except category code is easier to follow. This of course makes this version significantly faster.
\bool_lazy_or:nnF \cs_if_exist_p:N \tex_luatexversion:D \cs_if_exist_p:N \tex_XeTeXversion:D \cs_set:Npn \_char_str_change_case:nN #1#2 \tl_to_str:n {#2} \}

(End definition for \char_lowercase:N and others. These functions are documented on page 129.)

\c_catcode_other_space_tl
Create a space with category code 12: an “other” space.
\tl_const:Nx \c_catcode_other_space_tl \char_generate:nn { ‘ ‘ } { 12 } \)
(End definition for \c_catcode_other_space_tl. This function is documented on page 129.)

17.4 Generic tokens
\s__token_stop
Internal scan marks.
\scan_new:N \s__token_stop \)
(End definition for \s__token_stop.)

\token_to_meaning:N \token_to_meaning:c \token_to_str:N \token_to_str:c \c_group_begin_token \c_group_end_token \c_math_toggle_token \c_alignment_token \c_parameter_token \c_math_superscript_token \c_math_subscript_token \c_space_token \c_catcode_letter_token \c_catcode_other_token
These are all defined in l3basics, as they are needed “early”. This is just a reminder!
\token_to_meaning:N \token_to_str:N \c_group_begin_token \c_group_end_token \c_math_toggle_token \c_alignment_token \c_parameter_token \c_math_superscript_token \c_math_subscript_token \c_space_token \c_catcode_letter_token \c_catcode_other_token
We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for \cs_new_eq:NN does not cover them so we do things by hand. (As currently coded it would work with \cs_new_eq:NN but that’s not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the \__kernel_chk_if_free_cs:N check.
\group_begin:
\__kernel_chk_if_free_cs:N \c_group_begin_token \__kernel_chk_if_free_cs:N \c_group_end_token { \__kernel_chk_if_free_cs:N \c_group_end_token } \__kernel_chk_if_free_cs:N \c_math_toggle_token * \__kernel_chk_if_free_cs:N \c_alignment_token * \__kernel_chk_if_free_cs:N \c_parameter_token # \__kernel_chk_if_free_cs:N \c_math_superscript_token \__kernel_chk_if_free_cs:N \c_math_subscript_token \__kernel_chk_if_free_cs:N \c_space_token \__kernel_chk_if_free_cs:N \c_catcode_letter_token \__kernel_chk_if_free_cs:N \c_catcode_other_token \group_end:
(End definition for \c_group_begin_token and others. These functions are documented on page 133.)
\c_catcode_active_tl  Not an implicit token!
\group_begin:
\char_set_catcode_active:N \*
\tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
\group_end:
(End definition for \c_catcode_active_tl. This variable is documented on page 133.)

17.5 Token conditionals

\token_if_group_begin_p:N  \token_if_group_begin:NTF
Check if token is a begin group token. We use the constant \c_group_begin_token for this.
\prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF } { \if_catcode:w \exp_not:N #1 \c_group_begin_token \prg_return_true: \else: \prg_return_false: \fi: }
(End definition for \token_if_group_begin:NTF. This function is documented on page 134.)

\token_if_group_end_p:N  \token_if_group_end:NTF
Check if token is a end group token. We use the constant \c_group_end_token for this.
\prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF } { \if_catcode:w \exp_not:N #1 \c_group_end_token \prg_return_true: \else: \prg_return_false: \fi: }
(End definition for \token_if_group_end:NTF. This function is documented on page 134.)

\token_if_math_toggle_p:N  \token_if_math_toggle:NTF
Check if token is a math shift token. We use the constant \c_math_toggle_token for this.
\prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF } { \if_catcode:w \exp_not:N #1 \c_math_toggle_token \prg_return_true: \else: \prg_return_false: \fi: }
(End definition for \token_if_math_toggle:NTF. This function is documented on page 134.)

\token_if_alignment_p:N  \token_if Alignment:NTF
Check if token is an alignment tab token. We use the constant \c_alignment_token for this.
\prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF } { \if_catcode:w \exp_not:N #1 \c_alignment_token \prg_return_true: \else: \prg_return_false: \fi: }
(End definition for \token_if_alignment:NTF. This function is documented on page 134.)
Check if token is a parameter token. We use the constant \c_parameter_token for this. We have to trick \TeX{} a bit to avoid an error message: within a group we prevent \c_parameter_token from behaving like a macro parameter character. The definitions of \prg_new_conditional:Npn are global, so they remain after the group.

\begin{verbatim}
\token_if_parameter_p:N \token_if_parameter:N \token_if_parameter:NTF
Check if token is a parameter token. We use the constant \c_parameter_token for this.
\begin{verbatim}
\group_begin:
  \cs_set_eq:NN \c_parameter_token \scan_stop:
  \prg_new_conditional:Npn \token_if_parameter:N #1 \{ p , T , F , TF \}
  \{
    \if_catcode:w \exp_not:N #1 \c_parameter_token 
    \prg_return_true: \else: \prg_return_false: \fi:
  \}
\group_end:
(End definition for \token_if_parameter:NTF. This function is documented on page 134.)
\end{verbatim}
\end{verbatim}

Check if token is a math superscript token. We use the constant \c_math_superscript_token for this.

\begin{verbatim}
\token_if_math_superscript_p:N \token_if_math_superscript:N \token_if_math_superscript:NTF
Check if token is a math superscript token. We use the constant \c_math_superscript_token for this.
\begin{verbatim}
\prg_new_conditional:Npn \token_if_math_superscript:N #1 \{ p , T , F , TF \}
  \{
    \if_catcode:w \exp_not:N #1 \c_math_superscript_token
    \prg_return_true: \else: \prg_return_false: \fi:
  \}
\end{verbatim}
(End definition for \token_if_math_superscript:NTF. This function is documented on page 134.)
\end{verbatim}

Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.

\begin{verbatim}
\token_if_math_subscript_p:N \token_if_math_subscript:N \token_if_math_subscript:NTF
Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.
\begin{verbatim}
\prg_new_conditional:Npn \token_if_math_subscript:N #1 \{ p , T , F , TF \}
  \{
    \if_catcode:w \exp_not:N #1 \c_math_subscript_token
    \prg_return_true: \else: \prg_return_false: \fi:
  \}
\end{verbatim}
(End definition for \token_if_math_subscript:NTF. This function is documented on page 134.)
\end{verbatim}

Check if token is a space token. We use the constant \c_space_token for this.

\begin{verbatim}
\token_if_space_p:N \token_if_space:N \token_if_space:NTF
Check if token is a space token. We use the constant \c_space_token for this.
\begin{verbatim}
\prg_new_conditional:Npn \token_if_space:N #1 \{ p , T , F , TF \}
  \{
    \if_catcode:w \exp_not:N #1 \c_space_token
    \prg_return_true: \else: \prg_return_false: \fi:
  \}
\end{verbatim}
(End definition for \token_if_space:NTF. This function is documented on page 134.)
\end{verbatim}

Check if token is a letter token. We use the constant \c_catcode_letter_token for this.

\begin{verbatim}
\token_if_letter_p:N \token_if_letter:N \token_if_letter:NTF
Check if token is a letter token. We use the constant \c_catcode_letter_token for this.
\begin{verbatim}
\prg_new_conditional:Npn \token_if_letter:N #1 \{ p , T , F , TF \}
  \{
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
    \prg_return_true: \else: \prg_return_false: \fi:
  \}
\end{verbatim}
(End definition for \token_if_letter:NTF. This function is documented on page 135.)
\end{verbatim}

582
Check if token is an another char token. We use the constant \c_catcode_other_token for this.
\begin{verbatim}
\prg_new_conditional:Npnn \token_if_other:NTF #1 { p , T , F , TF }
\{ 
  \if_catcode:w \exp_not:N #1 \c_catcode_other_token 
  \prg_return_true: \else: \prg_return_false: \fi:
\}
\end{verbatim}
(End definition for \token_if_other:NTF. This function is documented on page 135.)

Check if token is an active char token. We use the constant \c_catcode_active_tl for this. A technical point is that \c_catcode_active_tl is in fact a macro expanding to \exp_not:N *, where * is active.
\begin{verbatim}
\prg_new_conditional:Npnn \token_if_active:NTF #1 { p , T , F , TF }
\{ 
  \if_catcode:w \exp_not:N #1 \c_catcode_active_tl 
  \prg_return_true: \else: \prg_return_false: \fi:
\}
\end{verbatim}
(End definition for \token_if_active:NTF. This function is documented on page 135.)

Check if the tokens #1 and #2 have same meaning.
\begin{verbatim}
\prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
\{ 
  \if_meaning:w #1 #2 
  \prg_return_true: \else: \prg_return_false: \fi:
\}
\end{verbatim}
(End definition for \token_if_eq_meaning:NNTF. This function is documented on page 135.)

Check if the tokens #1 and #2 have same category code.
\begin{verbatim}
\prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
\{ 
  \if_catcode:w \exp_not:N #1 \exp_not:N #2 
  \prg_return_true: \else: \prg_return_false: \fi:
\}
\end{verbatim}
(End definition for \token_if_eq_catcode:NNTF. This function is documented on page 135.)

Check if the tokens #1 and #2 have same character code.
\begin{verbatim}
\prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
\{ 
  \if_charcode:w \exp_not:N #1 \exp_not:N #2 
  \prg_return_true: \else: \prg_return_false: \fi:
\}
\end{verbatim}
(End definition for \token_if_eq_charcode:NNTF. This function is documented on page 135.)

When a token is a macro, \token_to_meaning:N always outputs something like \long macro:#1->#1 so we could naively check to see if the meaning contains \->. However, this can fail the five \...mark primitives, whose meaning has the form \...mark:{⟨user material⟩}. The problem is that the ⟨user material⟩ can contain \-. However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first :, and analyse what is left. However, macros can have
any combination of \long, \protected or \outer (not used in \LaTeX3) before the string \texttt{macro}: We thus only select the part of the meaning between the first \texttt{ma} and the first following \texttt{:}. If this string is \texttt{cro}, then we have a macro. If the string is \texttt{rk}, then we have a mark. The string can also be \texttt{cro parameter character} for a colon with a weird category code (namely the usual category code of \texttt{#}). Otherwise, it is empty.

This relies on the fact that \texttt{long}, \texttt{protected}, \texttt{outer} cannot contain \texttt{ma}, regardless of the escape character, even if the escape character is \texttt{m}...

Both \texttt{ma} and \texttt{:} must be of category code 12 (other), so are detokenized.

\begin{verbatim}
\use:x
\{ 
  \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
  { p , T , F , TF }
  { 
    \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
    \exp_not:N \token_to_meaning:N #1 \tl_to_str:n \{ ma : \}
    \s__token_stop
  }
  \cs_new:Npn \exp_not:N \__token_if_macro_p:w
    ##1 \tl_to_str:n \{ ma \} \c_colon_str \#2 \c_colon_str \#3 \s__token_stop
  { 
    \str_if_eq:nnTF { #2 } { cro } { \prg_return_true: } { \prg_return_false: }
  }
\}
\end{verbatim}

(End definition for \texttt{token_if_macro:NTF} and \texttt{__token_if_macro_p:w}. This function is documented on page 135.)

\begin{verbatim}
\token_if_cs_p:N
\token_if_cs:NTF
Check if token has same catcode as a control sequence. This follows the same pattern as
for \texttt{token_if_letter:N} etc. We use \texttt{scan_stop:} for this.
\begin{verbatim}
\prg_new_conditional:Npnn \exp_not:N \token_if_cs:N #1
  { p , T , F , TF }
  { 
    \if_catcode:w \exp_not:N #1 \scan_stop:
    \prg_return_false: \else: \prg_return_true: \fi:
  }
\end{verbatim}
(End definition for \texttt{token_if_cs:NTF}. This function is documented on page 135.)

\begin{verbatim}
\token_if_expandable_p:N
\token_if_expandable:NTF
Check if token is expandable. We use the fact that \TeX temporarily converts \texttt{exp_-}
\texttt{not:N} \texttt{⟨token⟩} into \texttt{scan_stop:} if \texttt{⟨token⟩} is expandable. An undefined token is not
considered as expandable. No problem nesting the conditionals, since the third \texttt{#1} is only
skipped if it is non-expandable (hence not part of \TeX’s conditional apparatus).
\begin{verbatim}
\prg_new_conditional:Npnn \exp_not:N \token_if_expandable:N #1
  { p , T , F , TF }
  { 
    \exp_after:wN \if_meaning:w \exp_not:N #1 \#1
    \prg_return_false: \else:
    \if_cs_exist:N #1
    \prg_return_true: \else:
    \prg_return_false: \fi:
  }
\end{verbatim}
\end{verbatim}

584
These auxiliary functions are used below to define some conditionals which detect whether the \meaning of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by \texttt{__token_stop}, and returns the first one and its delimiter. This result is eventually compared to another string.

Each of these conditionals tests whether its argument’s \meaning starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the \meaning to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using \texttt{\str_if_eq:eeTF} to the result of applying \texttt{\token_to_str:N} to a control sequence. Second, the \meaning of primitives such as \texttt{\dimen} or \texttt{\dimendef} starts in the same way as registers such as \texttt{\dimen123}, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through \texttt{\tl_to_str:n}. This requires doing all definitions within \texttt{x-expansion}. The temporary function \texttt{\_token_tmp:w} used to define each conditional receives three arguments: the name of the conditional, the auxiliary’s delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary’s result. Note that the \meaning of a protected long macro starts with \texttt{\protected\long macro}, with no space after \texttt{\protected} but a space after \texttt{\long}, hence the mixture of \texttt{\token_to_str:N} and \texttt{\tl_to_str:n}.

For the first five conditionals, \texttt{\cs_if_exist:cT} turns out to be \texttt{false}, and the code boils down to a string comparison between the result of the auxiliary on the \meaning of the conditional’s argument \texttt{####1}, and \texttt{#3}. Both are evaluated at run-time, as this is important to get the correct escape character.
The other five conditionals have additional code that compares the argument to two TeX primitives which would wrongly be recognized as registers otherwise. Despite using TeX's primitive conditional construction, this does not break when is itself a conditional, because branches of the conditionals are only skipped if is one of the two primitives that are tested for (which are not TeX conditionals).

\begin{verbatim}
\group_begin:
\cs_set_protected:Npn \_token_tmp:w #1#2#3
 { \use:x
 { \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } 11438
 { p , T , F , TF }
 { \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \if_meaning:w 11439
 { \exp_not:c { tex_ #2 :D }
 \exp_not:N \prg_return_false:
 \exp_not:N \else:
 \exp_not:N \if_meaning:w 11440
 { \exp_not:c { tex_ #2 def:D }
 \exp_not:N \prg_return_false:
 \exp_not:N \else:

 } 11441
 \exp_not:N \str_if_eq:eeTF
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 } 11442
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }
 { \exp_not:N \exp_after:wN
 \exp_not:c { __token_delimit_by_ #2 :w }
 \exp_not:N \token_to_meaning:N ####1
 ? \tl_to_str:n {#2} \s__token_stop
 }
 { \exp_not:n {#3} }
 { \exp_not:N \prg_return_true: }
 { \exp_not:N \prg_return_false: }
 \cs_if_exist:cT { tex_ #2 :D }
 { \exp_not:N \fi:
 \exp_not:N \fi:
 }

\__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
\__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
\__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
\__token_tmp:w { protected_macro } { macro }
\__token_tmp:w { protected } macro
\__token_tmp:w { protected_long_macro } { macro }
\__token_tmp:w { protected } \long macro
\__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
\__token_tmp:w { int_register } { count } { \token_to_str:N \count }
\__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
\__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
\__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
\group_end:
\end{verbatim}
We filter out macros first, because they cause endless trouble later otherwise. Primitives are almost distinguished by the fact that the result of \token_to_-meaning:N is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., \count123) or a double quote (e.g., char"A).

Ten exceptions: on the one hand, \tex_undefined:D is not a primitive, but its meaning is undefined, only letters; on the other hand, \space, \italiccorr, \hyphen, \firstmark, \topmark, \botmark, \splitfirstmark, \splitbotmark, and \nullfont are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on \endlinechar), and takes care of three of the exceptions: \space, \italiccorr and \hyphen, whose meaning is at most two characters. This leaves a string terminated by some : , and \s__token_stop.

The meaning of each one of the five \...mark primitives has the form (letters):(user material). In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either " , or a space, or a digit. Two exceptions remain: \tex_undefined:D, which is not a primitive, and \nullfont, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for \nullfont. Otherwise, we go through characters one by one, and stop at the first character less than 'A (this is not quite a test for “only letters”, but is close enough to work in this context). If this first character is : then we have a primitive, or \tex_undefined:D, and if it is " or a digit, then the token is not a primitive.
(End definition for \token_ifPrimitive:NTF and others. This function is documented on page 137.)

17.6 Peeking ahead at the next token

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.
\l_peek_token Storage tokens which are publicly documented: the token peeked.
\g_peek_token
\cs_new_eq:NN \l_peek_token ?
\cs_new_eq:NN \g_peek_token ?

(End definition for \l_peek_token and \g_peek_token. These variables are documented on page 137.)
\l__peek_search_token The token to search for as an implicit token: cf. \l__peek_search_tl.
\cs_new_eq:NN \l__peek_search_token ?

(End definition for \l__peek_search_token.)
\l__peek_search_tl The token to search for as an explicit token: cf. \l__peek_search_token.
\tl_new:N \l__peek_search_tl

(End definition for \l__peek_search_tl.)
\__peek_true:w Functions used by the branching and space-stripping code.
\__peek_true_aux:w
\__peek_false:w
\__peek_tmp:w
\cs_new:Npn \__peek_true:w { }
\cs_new:Npn \__peek_true_aux:w { }
\cs_new:Npn \__peek_false:w { }
\cs_new:Npn \__peek_tmp:w { }

(End definition for \__peek_true:w and others.)
\s__peek_mark Internal scan marks.
\s__peek_stop
\scan_new:N \s__peek_mark
\scan_new:N \s__peek_stop

(End definition for \s__peek_mark and \s__peek_stop.)
\__peek_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
\cs_new:Npn \__peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }

(End definition for \__peek_use_none_delimit_by_s_stop:w.)
\peek_after:Nw Simple wrappers for \futurelet: no arguments absorbed here.
\peek_gafter:Nw
\cs_new_protected:Npn \peek_after:Nw { \tex_futurelet:D \l_peek_token }
\cs_new_protected:Npn \peek_gafter:Nw { \tex_global:D \tex_futurelet:D \g_peek_token }

(End definition for \peek_after:Nw and \peek_gafter:Nw. These functions are documented on page 137.)
\__peek_true_remove:w A function to remove the next token and then regain control.
\cs_new_protected:Npn \__peek_true_remove:w
\tex_afterassignment:D \__peek_true_aux:w
\cs_set_eq:NN \__peek_tmp:w

(End definition for \__peek_true_remove:w.)
Repeatedly use \_\_peek_true_remove:w to remove a space and call \_\_peek_true_aux:w.

\cs_new_protected:Npn \peek_remove_spaces:n #1
\cs_set:Npx \__peek_false:w { \exp_not:n {#1} }
\group_align_safe_begin:
\cs_set:Npn \__peek_true_aux:w { \peek_after:Nw \_\_peek_remove_spaces: }
\__peek_true_aux:w
\cs_new_protected:Npn \_\_peek_remove_spaces:
{\if_meaning:w \l_peek_token \c_space_token
\exp_after:wN \_\_peek_true_remove:w
\else:
\group_align_safe_end:
\exp_after:wN \_\_peek_false:w
\fi:}
\group_align_safe_end:
\peek_after:Nw #2

(End definition for \peek_remove_spaces:n and \_\_peek_remove_spaces:. This function is documented on page 270.)

\__peek_token_remove_generic:NN For token removal there needs to be a call to the auxiliary function which does the work.
\cs_new_protected:Npn \_\_peek_token_remove_generic:NNTF
{ \_\_peek_token_remove_generic:NNTF \_\_peek_true_aux:w }
\cs_new_protected:Npn \_\_peek_token_remove_generic:NNTF #1#2#3#4#5
{ \group_align_safe_begin:
\cs_set_eq:NN \l_peek_search_token #3
\tl_set:Nn \l_peek_search_tl {#3}
\cs_set:Npx \_\_peek_true_aux:w
{ \exp_not:N \group_align_safe_end:
\exp_not:n {#4}
}
\cs_set_eq:NNF \_\_peek_true:w #1
\cs_set:Npx \_\_peek_false:w
{ \exp_not:N \group_align_safe_end:
\exp_not:n {#5}
}
\peek_after:Nw #2
}

(End definition for \_\_peek_token_remove_generic:NNTF.)
\cs_new_protected:Npn \__peek_token_remove_generic:NNTF
\{ \__peek_token_remove_generic_aux:NNTF \__peek_true_remove:w \}
\cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
\{ \__peek_token_remove_generic:NNTF #1 #2 {#3} { } \}
\cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
\{ \__peek_token_remove_generic:NNTF #1 #2 { } {#3} \}

(End definition for \__peek_token_generic:NNTF and \__peek_token_remove_generic:NNTF.)

\__peek_execute_branches_meaning:
The meaning test is straightforward.
\cs_new:Npn \__peek_execute_branches_meaning:
\{ \if_meaning:w \l_peek_token \l__peek_search_token
\exp_after:wN \__peek_true:w
\else:
\exp_after:wN \__peek_false:w
\fi:
\}

(End definition for \__peek_execute_branches_meaning:.)

\__peek_execute_branches_catcode: The catcode and charcode tests are very similar, and in order to use the same auxiliaries
\__peek_execute_branches_charcode: we do something a little bit odd, firing \if_catcode:w and \if_charcode:w before
\__peek_execute_branches_catcode_aux: finding the operands for those tests, which are only given in the auxii:N and auxiii:
\__peek_execute_branches_catcode_auxii:N auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (\emph{e.g.}, macro, primitive);
- active characters which are not equal to a non-active character token (\emph{e.g.}, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

\__peek_execute_branches_catcode_auxii:N The first two cases are not distinguishable simply using \TeX{}’s \texttt{\futurelet}, because we
can only access the \texttt{\meaning} of tokens in that way. In those cases, detected thanks to
\__peek_execute_branches_catcode_auxiii: a comparison with \texttt{\scan_stop:}, we grab the following token, and compare it explicitly
\__peek_execute_branches_catcode_auxii:N with the explicit search token stored in \texttt{\l__peek_search_tl}. The \texttt{\exp_not:N} prevents
\__peek_execute_branches_catcode_auxiii: outer macros (coming from non-L\LaTeX{}3 code) from blowing up. In the third case, \texttt{\l_-
\__peek_token} is good enough for the test, and we compare it again with the explicit search
token. Just like the peek token, the search token may be of any of the three types above,
hence the need to use the explicit token that was given to the peek function.
\cs_new:Npn \__peek_execute_branches_catcode:
\{ \if_catcode:w \__peek_execute_branches_catcode_aux: \}
\cs_new:Npn \__peek_execute_branches_charcode:
\{ \if_charcode:w \__peek_execute_branches_catcode_aux: \}
\cs_new:Npn \__peek_execute_branches_catcode_aux:
\{ \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
\exp_after:wN \exp_after:wN
\exp_after:wN \__peek_execute_branches_catcode_auxii:N
\exp_after:wN \exp_not:N
\else:
\endlinecharTest:  591
The public functions themselves cannot be defined using \prg_new_conditional:Nnnn. Instead, the TF, T, F variants are defined in terms of corresponding variants of \__peek_token_generic:NNTF or \__peek_token_remove_generic:NNTF, with first argument one of \__peek_execute_branches_catcode:, \__peek_execute_branches_charcode:, or \__peek_execute_branches_meaning:.

To ignore spaces, remove them using \peek_remove_spaces:n before running the tests.
All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since \l_peek_token might be outer, we cannot use the convenient \bool_if:NTF function, and must resort to the old trick of using \ifodd to expand a set of tests. The false branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call \__peek_false:w. In the true branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for outer in the \meaning of \l_peek_token. If that is absent, \__peek_use_none_delimit_by_s_stop:w cleans up, and we call \__peek_true:w. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains outer, it can be the primitive \outer, or it can be an outer token. Macros and marks would have ma in the part before the first occurrence of outer; the meaning of \outer has nothing after outer, contrarily to outer macros; and that covers all cases, calling \__peek_true:w or \__peek_false:w as appropriate. Here, there is no ⟨search token⟩, so we feed a dummy \scan_stop: to the \__peek_token_generic:NNTF function.

\group_begin:
\cs_set_protected:Npn \__peek_tmp:w \#1 \s__peek_stop
{\cs_new_protected:Npn \__peek_execute_branches_N_type:
\cs_set_protected:Npn \__peek_token_generic:NNTF
\cs_new_protected:cpx \peek_#1_ignore_spaces:NTF \##1\##2\##3
{ \exp_not:c \peek_#1:NTF \##1 \{\##2\} \{\##3\} }
\cs_new_protected:cpx \peek_#1_ignore_spaces:NT \##1\##2
{ \peek_remove_spaces:n \exp_not:c \peek_#1:NT \##1 \{\##2\} }
\cs_new_protected:cpx \peek_#1_ignore_spaces:NF \##1\##2
{ \peek_remove_spaces:n \exp_not:c \peek_#1:NF \##1 \{\##2\} }
}

(End definition for \peek_catcode_ignore_spaces:NTF and others. These functions are documented on page 158.)

\peek_N_type:TF
\__peek_execute_branches_N_type: w
\__peek_N_type_aux: nnw
\__peek_use_none_delimit_by_s_stop: w
\__peek_false: w
\__peek_true: w
The following test files are used for this code: m3prop001, m3prop002, m3prop003, m3prop004, m3show001.

A property list is a macro whose top-level expansion is of the form
\s__prop \__prop_pair:wn ⟨key1⟩ \s__prop {⟨value1⟩}
...
\__prop_pair:wn ⟨keyn⟩ \s__prop {⟨valuen⟩}

where \s__prop is a scan mark (equal to \scan_stop:), and \__prop_pair:wn can be used to map through the property list.

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

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.
\l__prop_internal_tl  Token list used to store new key–value pairs to be inserted by functions of the \prop_put:Nnn family.

(End definition for \l__prop_internal_tl.)

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

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

\s__prop  A private scan mark is used as a marker after each key, and at the very beginning of the property list.

\l__prop_internal_tl  Token list used to store the new key–value pair inserted by \prop_put:Nnn and friends.

(End definition for \l__prop_internal_tl.)

\c_empty_prop  An empty prop.

(End definition for \c_empty_prop. This variable is documented on page 119.)
18.1 Internal auxiliaries

\s__prop_mark
\s__prop_stop

Internal scan marks.

\scan_new:N \s__prop_mark
\scan_new:N \s__prop_stop

(End definition for \s__prop_mark and \s__prop_stop.)

\q__prop_recursion_tail
\q__prop_recursion_stop

Internal recursion quarks.

\quark_new:N \q__prop_recursion_tail
\quark_new:N \q__prop_recursion_stop

(End definition for \q__prop_recursion_tail and \q__prop_recursion_stop.)

\prop_new:N
\prop_new:c

Property lists are initialized with the value \c_empty_prop.

\cs_new_protected:Npn \prop_new:N #1
{ \__kernel_chk_if_free_cs:N #1 \cs_gset_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_new:N { c }

(End definition for \prop_new:N. This function is documented on page 143.)

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

The same idea for clearing.

\cs_new_protected:Npm \prop_clear:N #1
{ \prop_set_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_clear:N { c }
\cs_new_protected:Npm \prop_gclear:N #1
{ \prop_gset_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_gclear:N { c }

(End definition for \prop_clear:N and \prop_gclear:N. These functions are documented on page 143.)

\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c

Once again a simple variation of the token list functions.

\cs_new_protected:Npm \prop_clear_new:N #1
{ \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_clear_new:N { c }
\cs_new_protected:Npm \prop_gclear_new:N #1
{ \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new:N { c }

(End definition for \prop_clear_new:N and \prop_gclear_new:N. These functions are documented on page 143.)

596
These are simply copies from the token list functions.

(End definition for \prop_set_eq:NN and \prop_gset_eq:NN. These functions are documented on page 143.)

We can now initialize the scratch variables.

(End definition for \l_tmpa_prop and others. These variables are documented on page 148.)

Property list used by \prop_set_from_keyval:Nn and others.

To avoid tracking throughout the loop the variable name and whether the assignment is local/global, do everything in a scratch variable and empty it afterwards to avoid wasting memory. Loop through items separated by commas, with \prg_do_nothing: to avoid losing braces. After checking for termination, split the item at the first and then at the second = (which ought to be the first of the trailing = that we added). For both splits trim spaces and call a function (first \__prop_from_keyval_key:w, followed by the trimmed material, \s__prop_mark, the subsequent part of the item, and the trailing =s and \s__prop_stop. After finding the \{key\} just store it after \s__prop_stop. After finding the \{value\} ignore completely empty items (both trailing = were used as delimiters and all parts are empty); if the remaining part #2 consists exactly of the second trailing = (namely there was exactly one = in the item) then output one key–value pair for the property list; otherwise complain about a missing or extra =.
This function is used by most of the module, and hence must be fast. It receives a ⟨property list⟩, a ⟨key⟩, a ⟨true code⟩ and a ⟨false code⟩. The aim is to split the ⟨property list⟩ at the given ⟨key⟩ into the ⟨extract1⟩ before the key–value pair, the ⟨value⟩ associated
with the ⟨key⟩ and the ⟨extract2⟩ after the key–value pair. This is done using a delimited function, whose definition is as follows, where the ⟨key⟩ is turned into a string.

\cs_set:Npn \_prop_split_aux:w #1
\_prop_pair:wn ⟨key⟩ \s__prop #2
#3 \s__prop_mark #4 \s__prop_stop
{ #4 {⟨true code⟩} {⟨false code⟩} }

If the ⟨key⟩ is present in the property list, \_prop_split_aux:w’s #1 is the part before the ⟨key⟩, #2 is the ⟨value⟩, #3 is the part after the ⟨key⟩, #4 is \use_i:nn, and #5 is additional tokens that we do not care about. The ⟨true code⟩ is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 \_prop_pair:wn ⟨key⟩ \s__prop (#2) \s__prop (#3).

If the ⟨key⟩ is not there, then the ⟨function⟩ is \use_ii:nn, which keeps the ⟨false code⟩.

\cs_new_protected:Npn \_prop_split:NnTF #1#2
{ \exp_args:NNo \_prop_split_aux:NnTF #1 { \tl_to_str:n ⟨#2⟩ } }
\cs_new_protected:Npn \_prop_split:NnTF #1#2#3#4
{ \cs_set:Npn \_prop_split_aux:w ##1
\_prop_pair:wn #2 \s__prop #3 \s__prop_mark #4 \s__prop_stop
{ ##4 #2 #3 } }
\exp_after:wN \_prop_split_aux:w #1 \s__prop_mark \use_i:nn
\_prop_pair:wn #2 \s__prop \s__prop_stop
\_prop_split:NnTF #1
\cs_new:Npn \_prop_split_aux:w { }

(End definition for \_prop_split:NnTF, \_prop_split_aux:NnTF, and \_prop_split_aux:w.)

\prop_remove:Nn \prop_gremove:Nn
\prop_remove:NV \prop_gremove:NV
\prop_remove:cn \prop_gremove:cn
\prop_remove:cV \prop_gremove:cV

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

\cs_new_protected:Npn \prop_remove:Nn #1#2
{ \_prop_split:NnTF #1 {#2} { \tl_set:Nn #1 { ##1 ##3 } } { } }
\cs_new_protected:Npn \prop_gremove:Nn #1#2
{ \_prop_split:NnTF #1 {#2} { \tl_gset:Nn #1 { ##1 ##3 } } { } }
\cs_generate_variant:Nn \prop_remove:Nn { NV }
\cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

(End definition for \prop_remove:Nn and \prop_gremove:Nn. These functions are documented on page 145.)
Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to \texttt{q_no_value}.

\begin{verbatim}
\cs_new_protected:Npn \prop_get:NnN #1#2#3
\__prop_split:NnTF #1 {#2}
\{ \tl_set:Nn #3 {##2} \}
{ \tl_set:Nn #3 { \q_no_value } }
\end{verbatim}

(End definition for \texttt{prop_get:Nn}. This function is documented on page 144.)

Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save \texttt{q_no_value} in the token list.

\begin{verbatim}
\cs_new_protected:Npn \prop_pop:NnN #1#2#3
\__prop_split:NnTF #1 {#2}
\{ \tl_set:Nn #3 {##2}
\tl_set:Nn #1 { ##1 ##3 }
\}
{ \tl_set:Nn #3 { \q_no_value } }
\end{verbatim}

(End definition for \texttt{prop_pop:Nn} and \texttt{prop_gpop:Nn}. These functions are documented on page 144.)

Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one \texttt{⟨key⟩–⟨value⟩} pair at a time: the arguments of \texttt{__prop_item_Nn:nwn} are the \texttt{⟨key⟩} we are looking for, a \texttt{⟨key⟩} of the property list, and its associated value. The \texttt{⟨keys⟩} are compared (as strings). If they match, the \texttt{⟨value⟩} is returned, within \texttt{exp_not:n}. The loop terminates even if the \texttt{⟨key⟩} is missing, and yields an empty value, because we have appended the appropriate \texttt{⟨key⟩–⟨empty value⟩} pair to the property list.

\begin{verbatim}
\cs_new:Npn \prop_item:Nn #1#2
\exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
\end{verbatim}

\begin{verbatim}
\__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
\end{verbatim}

\prg_break_point:

600
\prop_count:N
\prop_count:c
\_prop_count:nn

Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

\prop_pop:NnN
\prop_pop:cnN
\prop_gpop:NnN
\prop_gpop:cnN

Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, \prg_return_true: is used after the assignments.

(End definition for \prop_item:Nn and \_prop_item:Nn:nn. This function is documented on page 145.)
Since the branches of \_\_prop_split:NnTF are used as the replacement text of an internal macro, and since the \langle key\rangle and new \langle value\rangle may contain arbitrary tokens, it is not safe to include them in the argument of \_\_prop_split:NnTF. We thus start by storing in l\_prop_internal_tl tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in \_\_prop_split:NnTF. If the \langle key\rangle was absent, append the new key–value to the list. Otherwise concatenate the extracts \#1 and \#3 with the new key–value pair l\_prop_internal_tl. The updated entry is placed at the same spot as the original \langle key\rangle in the property list, preserving the order of entries.

Adding conditionally also splits. If the key is already present, the three brace groups given by \_\_prop_split:NnTF are removed. If the key is new, then the value is added, being careful to convert the key to a string using \tl_to_str:n.

\cs_new_protected:Npn \prop_put_if_new:Nnn { \_\_prop_put_if_new:NNnn \tl_set:Nx }\cs_new_protected:Npn \prop_gput_if_new:Nnn { \_\_prop_put_if_new:NNnn \tl_gset:Nx }\cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n {#4} } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put_if_new:Nnn { NnV , Nno , Nnx , NV , NVV , No , Noo }\cs_generate_variant:Nn \prop_gput_if_new:Nnn { NnV , Nno , Nnx , NV , NVV , No , Noo }\cs_generate_variant:Nn \__prop_put_if_new:NNnn { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \prop_put_if_new:cnn { \_\_prop_put_if_new:NNnn \tl_set:Nx }\cs_new_protected:Npn \prop_gput_if_new:cnn { \_\_prop_put_if_new:NNnn \tl_gset:Nx }\cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put_if_new:cnn { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \cs_new_protected:Nn \prop_put:NNn { \__prop_put:NNnn \tl_set:Nx }\cs_new_protected:Npn \prop_gput:NNn { \__prop_put:NNnn \tl_gset:Nx }\cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n {#4} } \__prop_split:NnTF #2 {#3} { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put:NNn { NnV , Nno , Nnx , NV , NVV , No , Noo }\cs_generate_variant:Nn \prop_gput:NNn { NnV , Nno , Nnx , NV , NVV , No , Noo }\cs_generate_variant:Nn \__prop_put:NNnn { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \cs_new_protected:Nn \prop_put:cnn { \_\_prop_put:cnn \tl_set:Nx }\cs_new_protected:Npn \prop_gput:cnn { \_\_prop_put:cnn \tl_gset:Nx }\cs_new_protected:Npn \__prop_put:cnn #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put:cnn { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \cs_new_protected:Nn \prop_put:cnV { \__prop_put:cnV \tl_set:Nx }\cs_new_protected:Npn \prop_gput:cnV { \_\_prop_put:cnV \tl_gset:Nx }\cs_new_protected:Npn \__prop_put:cnV #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put:cnV { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \cs_new_protected:Nn \prop_put:cno { \__prop_put:cno \tl_set:Nx }\cs_new_protected:Npn \prop_gput:cno { \_\_prop_put:cno \tl_gset:Nx }\cs_new_protected:Npn \__prop_put:cno #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put:cno { c , cnV , cno , cnx , cV , cVV , co , coo }\cs_new_protected:Npn \cs_new_protected:Nn \prop_put:cnx { \__prop_put:cnx \tl_set:Nx }\cs_new_protected:Npn \prop_gput:cnx { \_\_prop_put:cnx \tl_gset:Nx }\cs_new_protected:Npn \__prop_put:cnx #1#2#3#4 { \tl_set:Nn \l__prop_internal_tl { \exp_not:N \__prop_pair:wn \tl_to_str:n {#3} \s__prop \exp_not:n { {#4} } } \__prop_split:NnTF #2 {#3} { } { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } } }\cs_generate_variant:Nn \prop_put:cnx { c , cnV , cno , cnx , cV , cVV , co , coo }

(End definition for \prop_put:NNn, \prop_gput:NNn, and \__prop_put:NNnn. These functions are documented on page 144.)

(End definition for \prop_put:Nnn, \prop_gput:Nnn, and \__prop_split:NnTF. These functions are documented on page 146.)
18.4 Property list conditionals

Copies of the cs functions defined in l3basics.

Same test as for token lists.

Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```
\prg_new_protected_conditional:Nnn \prop_if_in:Nn #1 #2

\exp_last_unbraced:Noo \__prop_if_in:nwwn \tl_to_str:n {#2} #1
\__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
\q__prop_recursion_tail
\prg_break_point:
```

but \_\_\_prop_split:NnTF is non-expandable.

Instead, the key is compared to each key in turn using \str_if_eq:ee, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second \tl_to_str:n is not expanded at the start, but only when included in the \str_if_eq:ee. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of \_\_\_prop_if_in:nwn is most often empty. When the ⟨key⟩ is found in the list, \_\_\_prop_if_in:N receives \_\_\_prop_pair:wn, and if it is found as the extra item, the function receives \q__prop_recursion_tail, easily recognizable.

Here, \prop_map_function:NN is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```
\prg_new_conditional:Nnn \prop_if_in:Nn \prop_map_function:NN #1 { p , T , F , TF } #2
```
18.5 Recovering values from property lists with branching

Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

\prop_get:NnNTF
\prop_get:NWNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF

(End definition for \prop_get:NnNTF. This function is documented on page 146.)

18.6 Mapping to property lists

The argument delimited by \__prop_pair:wn is empty except at the end of the loop where it is \prg_break:. No need for any quark test.

\prop_map_function:NN
\prop_map_function:Nc
\prop_map_function:cc
\__prop_map_function:Nwwn

(End definition for \prop_get:NnNTF. This function is documented on page 146.)
Mapping in line requires a nesting level counter. Store the current definition of \_prop_pair:wn, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form \_prop_pair:wn (key) \s__prop {value}, there are a leading and a trailing token, but both are equal to \scan_stop:, hence have no effect in such inline mapping. Such \scan_stop: could have affected ligatures if they appeared during the mapping.

The mapping is very similar to \prop_map_function:NN. The \use_i:nn removes the leading \s__prop. The odd construction \use:n (#1) allows #1 to contain any token without interfering with \prop_map_break:. The loop stops when the argument delimited by \_prop_pair:wn is \prg_break: instead of being empty.
The break statements are based on the general \prg_map_break:Nn.

\prop_map_break:
\prop_map_break:n

Apply the general \_kernel_chk_defined:NT and \msg_show:nnnnnn. Contrarily to sequences and comma lists, we use \msg_show_item:nn to format both the key and the value for each pair.

\prop_show:N
\prop_show:c
\prop_log:N
\prop_log:c

A general scratch for the module.

\l__msg_internal_tl

Used to save module info when creating messages.

(End definition for \prop_map_tokens:Nn and \_prop_map_tokens:nwn. This function is documented on page 147.)

(End definition for \prop_map_tokens:Nn and \_prop_map_tokens:nwn. This function is documented on page 147.)

18.7 Viewing property lists

(End definition for \prop_map_break: and \prop_map_break:n. These functions are documented on page 147.)

(End definition for \prop_map_break: and \prop_map_break:n. These functions are documented on page 147.)

(End definition for \prop_map_break: and \prop_map_break:n. These functions are documented on page 147.)

(End definition for \prop_map_break: and \prop_map_break:n. These functions are documented on page 147.)

19 \l3msg implementation

(End definition for \l__msg_internal_tl.)

(End definition for \l__msg_internal_tl.)

(End definition for \l__msg_internal_tl.)

(End definition for \l__msg_name_str and \l__msg_text_str.)

(End definition for \l__msg_name_str and \l__msg_text_str.)
19.1 Internal auxiliaries

\s__msg_mark
\s__msg_stop

Internal scan marks.

(End definition for \s__msg_mark and \s__msg_stop.)

\__msg_use_none_delimit_by_s_stop:w

Functions to gobble up to a scan mark.

(End definition for \__msg_use_none_delimit_by_s_stop:w)

19.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

\c__msg_text_prefix_tl
\c__msg_more_text_prefix_tl

Locations for the text of messages.

(End definition for \c__msg_text_prefix_tl and \c__msg_more_text_prefix_tl)

\msg_if_exist_p:nn
\msg_if_exist:nnTF

Test whether the control sequence containing the message text exists or not.

(End definition for \msg_if_exist:nnTF. This function is documented on page 151.)

\__msg_chk_if_free:nn

This auxiliary is similar to \__kernel_chk_if_free.cs:N, and is used when defining messages with \msg_new:nnn.

(End definition for \__msg_chk_if_free:nn.)

\msg_new:nnnn
\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

607
19.3 Messages: support functions and text

Simple pieces of text for messages.

\begin{verbatim}
\tl_const:Nn \c__msg_coding_error_text_tl
\tl_const:Nn \c__msg_continue_text_tl
\tl_const:Nn \c__msg_critical_text_tl
\tl_const:Nn \c__msg_fatal_text_tl
\tl_const:Nn \c__msg_help_text_tl
\tl_const:Nn \c__msg_no_info_text_tl
\tl_const:Nn \c__msg_on_line_text_tl
\tl_const:Nn \c__msg_return_text_tl
\tl_const:Nn \c__msg_trouble_text_tl
\end{verbatim}

(End definition for \msg_new:nnnn and others. These functions are documented on page 150.)
More-errors-will-almost-certainly-follow: \hfill
the-LaTeX-run-should-be-aborted.
\}

(End definition for \c__msg_coding_error_text_tl and others.)

For writing the line number nicely. \msg_line_context: was set up earlier, so this is not new.

\cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
\cs_gset:Npn \msg_line_context: { \c__msg_on_line_text_tl \c_space_tl \msg_line_number: }

(End definition for \msg_line_number: and \msg_line_context: These functions are documented on page 151.)

19.4 Showing messages: low level mechanism

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup \TeX ’s \texttt{\errhelp} register before issuing an \texttt{\errmessage}. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.
First setup \TeX’s \texttt{\textbackslash errhelp} register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary \texttt{\_msg\_interrupt\_more\_text:n} receives its argument as a line-wrapped string, which is thus unaffected by expansion. We ave to split the main text into two parts as only the “message” itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by \texttt{\textbackslash errmessage} itself.

\begin{verbatim}
\cs_new_protected:Npn \_msg\_interrupt\_wrap:nnn \#1\#2\#3 { \iow_wrap:nnnN { \#3 } { } { } \_msg\_interrupt\_more\_text:n \\
\group_begin:
\int_sub:Nn \l_iow_line_count_int { 2 }
\iow_wrap:nxnN \{ \l__msg_text_str : - \#1 \}
{ \str_count:N \l__msg_text_str - \str_count:N \l__msg_name_str + 2 }
{ - }
\}
\_msg\_interrupt\_text:n

\_msg\_interrupt\_more\_text:n { \exp_args:Nx \tex_errhelp:D { \#1 \iow_newline: } }
\end{verbatim}

The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: \TeX’s own information involves the macro in which \texttt{\textbackslash errmessage} is called, and the end of the argument of the \texttt{\textbackslash errmessage}, including the closing brace. We use an active ! to call the \texttt{\textbackslash errmessage} primitive, and end its argument with \texttt{\use:none:n \{ (spaces) \}} which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active ! is closed before producing the message: this ensures that tokens inserted by typing I in the command-line are inserted after the message is entirely cleaned up.

The \texttt{\_kernel_iow\_with:Nnn} auxiliary, defined in l3file, expects an \{integer variable\}, an integer \{value\}, and some \{code\}. It runs the \{code\} after ensuring that the
\(\text{integer variable}\) takes the given (value), then restores the former value of the (integer variable) if needed. We use it to ensure that the \texttt{\newlinechar} is 10, as needed for \texttt{\iow_newline}: to work, and that \texttt{\errorcontextlines} is \(-1\), to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the \texttt{\errmessage}, which go in the way of tokens which could be inserted by the user. This is unavoidable.

(End definition for \texttt{\__msg_interrupt:n}.)

\section{Displaying messages}
\LaTeX{} is handling error messages and so the \TeX{} ones are disabled. This is already done by the \LaTeX{} kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path
for a space.

\cs_new:Npn \msg_fatal_text:n #1
{ \text{Fatal} - \msg_error_text:n {#1} }
\cs_new:Npn \msg_critical_text:n #1
{ \text{Critical} - \msg_error_text:n {#1} }
\cs_new:Npn \msg_error_text:n #1
{ \__msg_text:nn {#1} { Error } }
\cs_new:Npn \msg_warning_text:n #1
{ \__msg_text:nn {#1} { Warning } }
\cs_new:Npn \msg_info_text:n #1
{ \__msg_text:nn {#1} { Info } }
\cs_new:Npn \__msg_text:nn #1#2
{ \exp_args:Nf \__msg_text:n { \msg_module_type:n {#1} } \msg_module_name:n {#1} ~ #2 }
\cs_new:Npn \__msg_text:n #1
{ \tl_if_blank:nF {#1} { #1 ~ } }

(End definition for \msg_fatal_text:n and others. These functions are documented on page 151.)

\g_msg_module_name_prop
\g_msg_module_type_prop
For storing public module information: the kernel data is set up in advance.
\prop_new:N \g_msg_module_name_prop
\prop_gput:Nnn \g_msg_module_name_prop { LaTeX } { LaTeX3 }
\prop_new:N \g_msg_module_type_prop
\prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

(End definition for \g_msg_module_name_prop and \g_msg_module_type_prop. These variables are documented on page 152.)

\msg_module_type:n
Contextual footer information, with the potential to give modules an alternative name.
\cs_new:Npn \msg_module_type:n #1
{ \prop_if_in:NnTF \g_msg_module_type_prop {#1} { \prop_item:Nn \g_msg_module_type_prop {#1} } { } }

(End definition for \msg_module_type:n. This function is documented on page 152.)
Contextual footer information, with the potential to give modules an alternative name.

\texttt{\msg_module_name:n}

(End definition for \texttt{\msg_module_name:n} and \texttt{\msg_see_documentation_text:n}. These functions are
documented on page 152.)

\texttt{\_\_\_msg_class_new:nnn}

(group_begin:

\texttt{\cs_set_protected:Npn \_\_\_msg_class_new:nnn \#1\#2
{
\prop_new:c { l\_\_\_msg_redirect_ \#1 \_prop }
\cs_new_protected:cpn { \_\_\_msg_ \#1 \_code:nnnnnn } ##1##2##3##4##5##6 {\#2}
\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnnnnn } ##1##2##3##4##5##6
{ \
\use:x
{
\exp_not:n { \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} }
{ \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
{ \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
}
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnnnn } ##1##2##3##4
{ \exp_not:N \exp_not:n { \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnnn } ##1##2##3##4
{ \exp_not:N \exp_not:n { \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnn } ##1##2##3##4
{ \exp_not:N \exp_not:n { \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nn } ##1##2##3##4
{ \exp_not:N \exp_not:n { \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnxxxx } ##1##2##3##4##5##6
{ \
\use:x
{
\exp_not:N \exp_not:n
{ \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} }
{\#3} {\#4} {\#5} {\#6}
}
}
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnxxx } ##1##2##3##4##5##6
{ \exp_not:N \exp_not:n
{ \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnnx } ##1##2##3##4##5##6
{ \exp_not:N \exp_not:n
{ \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}\cs_new_protected:cpn { \_\_\_msg_ \#1 :nnx } ##1##2##3##4##5##6
{ \exp_not:N \exp_not:n
{ \_\_\_msg_use:nnnnnnn \#1 \{\#1\} \{\#2\} \{\#3\} \{\#4\} \{\#5\} \{\#6\} }
}
For fatal errors, after the error message \TeX{} bails out. We force a bail out rather than using \end as this means it does not matter if we are in a context where normally the run cannot end.

\msg_fatal:nnnnnn
\msg_fatal:nnnxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnn
\msg_fatal:nn
\msg_fatal_exit:
\__msg_fatal_exit:
\cs_new_protected:Npn \__msg_fatal_exit:
\tex_batchmode:D
\tex_read:D -1 to \l__msg_internal_tl

(End definition for \__msg_class_new:nn.)

\msg_fatal:nnnnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nn
\__msg_fatal_exit:
\__msg_class_new:nn { fatal }
\__msg_interrupt:NnnnN
\msg_fatal_text:n {#1} {#2}
\msg_fatal_text:n {#3} {#4} {#5} {#6}
c__msg_fatal_text_tl
\__msg_fatal_exit:
\cs_new_protected:Npn \__msg_fatal_exit:
\tex_batchmode:D
\tex_read:D -1 to \l__msg_internal_tl

(End definition for \msg_fatal:nnnnnn and others. These functions are documented on page 153.)

\msg_critical:nnnnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nn
\msg_critical_exit:
\__msg_class_new:nn { critical }
\__msg_interrupt:NnnnN
\msg_critical_text:n {#1} {#2}
\msg_critical_text:n {#3} {#4} {#5} {#6}
c__msg_critical_text_tl
\msg_critical_exit:
\cs_new_protected:Npn \__msg_critical_exit:
\tex_endinput:D

(End definition for \msg_critical:nnnnnn and others. These functions are documented on page 153.)

\msg_error:nnnnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nn
\msg_error_exit:
\__msg_class_new:nn { error }
\__msg_interrupt:NnnnN
\msg_error_text:n {#1} {#2}
\msg_error_text:n {#3} {#4} {#5} {#6}
c_empty_tl
\msg_error_exit:
\cs_new_protected:Npn \__msg_error_exit:
\tex_endinput:D

(End definition for \msg_error:nnnnnn and others. These functions are documented on page 153.)

\msg_warning:nnnnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nn
\msg_warning_exit:
\__msg_class_new:nn { warning }
\str_set:Nx \l__msg_text_str { msg\_warning\_text:n {#1} }
\str_set:Nx \l__msg_name_str { msg\_module\_name:n {#1} }
iow_term:n { }
iow_wrap:nxN

(End definition for \msg_warning:nnnnnn and others. These functions are documented on page 153.)

Warnings are printed to the terminal.
Information only goes into the log.

\msg_info:nnnnnn \msg_info:nxxxx \msg_info:nxxxx \msg_info:nnxxx \msg_info:nnnxx
\msg_info:nnnxx \msg_info:nnnx \msg_info:nnnxx \msg_info:nnxx \msg_info:nnx
\msg_info:nnx \msg_info:nn { info }

\msg_class_new:nn { info }

\str_set:Nx \_msg_text_str \msg_info_text:n {#1} 
\str_set:Nx \_msg_text_str \msg_module_name:n {#1} 
\iow_log:n { }
\iow_wrap:nnnN
{ \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
{ } { } \iow_log:n
\iow_log:n { }

(End definition for \msg_warning:nnnnnn and others. These functions are documented on page 153.)

\msg_info:nnnnnn
\msg_info:nxxxx
\msg_info:nxxxx
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn { info }

\msg_class_new:nn { log }

\iow_wrap:nnnN
{ \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
{ } { } \iow_log:n
\iow_log:n { }

(End definition for \msg_info:nnnnnn and others. These functions are documented on page 154.)

\msg_log:nnnnnn
\msg_log:nxxxx
\msg_log:nnnxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn { log }

\msg_class_new:nn { log }

\iow_wrap:nnnN
{ \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
{ } { } \iow_log:n
\iow_log:n { }

(End definition for \msg_log:nnnnnn and others. These functions are documented on page 154.)
The `none` message type is needed so that input can be gobbled.

\begin{verbatim}
\msg_none:nnnnnn
\msg_none:nnxxxx
\msg_none:nnnnnn
\msg_none:nxxxx
\msg_none:nxxxx
\msg_show:nnnnnn
\msg_show:nnxxxx
\msg_show:nnnnn
\msg_show:nnxxx
\msg_show:nnnn
\msg_show:nnxx
\msg_show:nnn
\msg_show:nnx
\msg_show:nn
\__msg_class_new:nn { none } { }
\end{verbatim}

(End definition for `\msg_none:nnnnnn` and others. These functions are documented on page 154.)

The `show` message type is used for `\seq_show:N` and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to `\__msg_show:n`. If there is \texttt{\textbackslash >~} (or if the whole thing starts with \texttt{\textbackslash >}) we split there, print the first part and show the second part using `\showtokens` (the `\exp_after:wN` ensure a nice display).

Note that this primitive adds a leading \texttt{\textbackslash >} and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no \texttt{\textbackslash >} do the same but with an empty second part which adds a spurious but inevitable \texttt{\textbackslash >}.

\begin{verbatim}
\__msg_class_new:nn \msg_show:nn
\__msg_class_new:nn \msg_show:n
\__msg_class_new:nn \msg_show:w
\__msg_class_new:nn \msg_show_dot:w
\__msg_class_new:nn \msg_show:n
\msg_show:nn
\__msg_class_new:nn { show }
\msg_show:nn
\__msg_class_new:nn { } { }
\__msg_show:n
\cs_new_protected:Npn \__msg_show:n #1
\tl_if_in:nnTF { ^^J #1 } { ^^J \textgreater~ } { \__msg_show:nn { ? #1 } { } }
\__msg_class_new:nn \msg_show:nn #1
\tl_if_in:nnTF { \textgreater~ } { \textgreater~ } { \__msg_show:nn { #1 } { } }
\cs_new_protected:Npn \__msg_show:nn #1#2
\tl_if_empty:nF {#1} \exp_args:No \iow_term:n { \use_none:n #1 }
\tl_set:Nn \l__msg_internal_tl {#2}
\tl_if_empty:n { #1 #2 }
\iow_term:n { \use_none:n #1 } #2
\cs_new_protected:Npn \__msg_show:nn #1
\tl_if_empty:nF { #1 } \exp_args:No \iow_term:n { \use_none:n #1 }
\tl_set:Nn \l__msg_internal_tl \l__msg_internal_tl #2
\__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
\__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
\__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
\\\exp_after:wN \exp_after:wN \exp_after:wN
\l__msg_internal_tl
\end{verbatim}

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 263.)

End the group to eliminate `\__msg_class_new:nn`.

\begin{verbatim}
\__msg_class_chk_exist:nT
\end{verbatim}

Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

616
\cs_new:Npn \__msg_class_chk_exist:nT #1
{
  \cs_if_free:cTF { __msg_ #1 _code:nnnnn }
  { \__kernel_msg_error:nnx { kernel } { message-class-unknown } {#1} }
}

\Enddefinitionfor \__msg_class_chk_exist:nT

\l__msg_class_tl \l__msg_current_class_tl Support variables needed for the redirection system.
\tl_new:N \l__msg_class_tl \tl_new:N \l__msg_current_class_tl
\Enddefinitionfor \l__msg_class_tl and \l__msg_current_class_tl

\l__msg_redirect_prop For redirection of individually-named messages
\prop_new:N \l__msg_redirect_prop
\Enddefinitionfor \l__msg_redirect_prop

\l__msg_hierarchy_seq During redirection, split the message name into a sequence: {/module/submodule}, {/module}, and { }. \seq_new:N \l__msg_hierarchy_seq
\Enddefinitionfor \l__msg_hierarchy_seq

\l__msg_class_loop_seq Classes encountered when following redirections to check for loops. \seq_new:N \l__msg_class_loop_seq
\Enddefinitionfor \l__msg_class_loop_seq

\__msg_use:nnnnnnn \__msg_use_redirect_name:n \__msg_use_hierarchy:nwwN \__msg_use_redirect_module:n \__msg_use_code:
Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to \__msg_use_code: is similar to \tl_set:Nn. The message is eventually produced with whatever \l__msg_class_tl is when \__msg_use_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.
\cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
{
  ⟨package⟩ \cs_if_exist_use:N \conditionally@traceoff
  \msg_if_exist:nnTF {#2} {#3}
  { \__msg_class_chk_exist:nT {#1}
    \tl_set:Nn \l__msg_current_class_tl {#1}
    \cs_set_protected:Npx \__msg_use_code:
    {
      \exp_not:n
      \use:c { __msg_ \l__msg_class_tl _code:nnnnn }
      {#2} {#3} {#4} {#5} {#6} {#7}
    }
  }
  \__msg_use_redirect_name:n { #2 / #3 }
}
\Enddefinitionfor \__msg_use:nnnnnnn

617
The first check is for an individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into ⟨module⟩, ⟨submodule⟩ and ⟨message⟩ (with an arbitrary number of slashes), and store ⟨/module/submodule⟩, ⟨/module⟩ and ⟨⟩ into \l__msg_hierarchy_seq. We then map through this sequence, applying the most specific redirection.

At this point, the items of \l__msg_hierarchy_seq are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of \__msg_use_redirect_module:n are not attempted. This argument is empty for a class redirection, /module for a module redirection, etc. Loop through the sequence to find the most specific redirection, with module #1. The loop is interrupted after testing for a redirection for #1 equal to the argument #1 (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as #1.
Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in $\l__msg_current_class_tl$.
Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, \#1, and not submodules. After some initialization above, follow redirections with \l\_msg_class_t1, and keep track in \l\_msg_class_loop_seq of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from \#2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

\begin{verbatim}
\cs_new_protected:Npn \_msg_redirect_loop_chk:nnn #1#2#3
\{
    \seq_put_right:Nn \l__msg_class_loop_seq {#1}
    \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
    \{
        \str_if_eq:VnF \l__msg_class_tl {#1}
        \{
            \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
            \{
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
                \_kernel_msg_warning:nnxxxx
                \{ kernel \} \{ message-redirect-loop \}
                \{ \seq_item:Nn \l__msg_class_loop_seq { 1 } \}
                \{ \seq_item:Nn \l__msg_class_loop_seq { 2 } \}
                \{#3\}
            \}
            \{
                \seq_map_function:NN \l__msg_class_loop_seq \_msg_redirect_loop_list:n
                \{ \seq_item:Nn \l__msg_class_loop_seq { 1 } \}
            \}
        \}
        \seq_map_function:NN \l__msg_class_loop Seq \_msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} 
    \}
\}
\cs_generate_variant:Nn \_msg_redirect_loop_chk:nnn { o }
\cs_new:Npn \_msg_redirect_loop_list:n #1 { {#1} ~ => ~ }
\end{verbatim}

(End definition for \msg_redirect_class:nn and others. These functions are documented on page 155.)

\section{Kernel-specific functions}

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.
All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to \_msg_class_new:nn. This auxiliary is destroyed at the end of the group.

\begin{verbatim}
\__msg_kernel_class_new:nN
\__msg_kernel_class_new_aux:nN
\end{verbatim}

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the "\LaTeX" module name. Three functions are already defined by l3basics; we need to undefine them to avoid errors.

\begin{verbatim}
\__msg_kernel_class_new:nN \__msg_fatal_code:nnnnnn
\cs_undefine:N \__kernel_msg_error:nnnnn
\end{verbatim}

621
Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “\LaTeX.”

Error messages needed to actually implement the message system itself.

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.
LaTeX has been asked to define a function ‘#1’ with #2 arguments.

TeX allows between 0 and 9 arguments for a single function.

\__kernel_msg_new:nnn { kernel } { char-active }
\{ Cannot-generate-active-chars. \}
\__kernel_msg_new:nnn { kernel } { char-invalid-catcode }
\{ Invalid-catcode-for-char-generation. \}
\__kernel_msg_new:nnn { kernel } { char-null-space }
\{ Cannot-generate-null-char-as-a-space. \}
\__kernel_msg_new:nnn { kernel } { char-out-of-range }
\{ Charcode-requested-out-of-engine-range. \}
\__kernel_msg_new:nnn { kernel } { char-space }
\{ Cannot-generate-space-chars. \}
\__kernel_msg_new:nnn { kernel } { command-already-defined }
\{ Control-sequence ‘#1’-already-defined. \}
\__kernel_msg_new:nnn { kernel } { command-not-defined }
\{ Control-sequence ‘#1’-undefined. \}
\__kernel_msg_new:nnn { kernel } { empty-search-pattern }
\{ Empty-search-pattern. \}
\__kernel_msg_new:nnn { kernel } { out-of-registers }
\{ No-room-for-a-new-‘#1’. \}
\__kernel_msg_new:nnn { kernel } { non-base-function }
\{ Function ‘#1’-is-not-a-base-function \}
\__kernel_msg_new:nnn { kernel } { command-not-defined }
\{ Control-sequence ‘#1’-undefined. \}
\_\_kernel_msg_new:nnn { kernel } { char-active }
\{ Cannot-generate-active-chars. \}
\_\_kernel_msg_new:nnn { kernel } { char-invalid-catcode }
\{ Invalid-catcode-for-char-generation. \}
\_\_kernel_msg_new:nnn { kernel } { char-null-space }
\{ Cannot-generate-null-char-as-a-space. \}
\_\_kernel_msg_new:nnn { kernel } { char-out-of-range }
\{ Charcode-requested-out-of-engine-range. \}
\_\_kernel_msg_new:nnn { kernel } { char-space }
\{ Cannot-generate-space-chars. \}
\_\_kernel_msg_new:nnn { kernel } { command-already-defined }
\{ Control-sequence ‘#1’-already-defined. \}
\{ \_\_kernel_msg_new:nnn { kernel } { command-not-defined } \}
\{ Control-sequence ‘#1’-undefined. \}
\{ \_\_kernel_msg_new:nnn { kernel } { empty-search-pattern } \}
\{ Empty-search-pattern. \}
\{ \_\_kernel_msg_new:nnn { kernel } { out-of-registers } \}
\{ No-room-for-a-new-‘#1’. \}
\text{TeX-only-supports} `\int_use:N \c_max_register_int \% ` of each type. All the ‘#1’-registers have been used. `\text{This run will be aborted now.}
\{ \_\_kernel_msg_new:nnn { kernel } { non-base-function } \}
\{ Function ‘#1’-is-not-a-base-function \}
\{ \_\_kernel_msg_new:nnn { kernel } { command-not-defined } \}
\{ Control-sequence ‘#1’-undefined. \}
\_\_kernel_msg_new:nnn { kernel } { char-active }
\{ Cannot-generate-active-chars. \}
\_\_kernel_msg_new:nnn { kernel } { char-invalid-catcode }
\{ Invalid-catcode-for-char-generation. \}
\_\_kernel_msg_new:nnn { kernel } { char-null-space }
\{ Cannot-generate-null-char-as-a-space. \}
\_\_kernel_msg_new:nnn { kernel } { char-out-of-range }
\{ Charcode-requested-out-of-engine-range. \}
\_\_kernel_msg_new:nnn { kernel } { char-space }
\{ Cannot-generate-space-chars. \}
\_\_kernel_msg_new:nnn { kernel } { command-already-defined }
\{ Control-sequence ‘#1’-already-defined. \}
\{ \_\_kernel_msg_new:nnn { kernel } { command-not-defined } \}
\{ Control-sequence ‘#1’-undefined. \}
\{ \_\_kernel_msg_new:nnn { kernel } { empty-search-pattern } \}
\{ Empty-search-pattern. \}
\{ \_\_kernel_msg_new:nnn { kernel } { out-of-registers } \}
\{ No-room-for-a-new-‘#1’. \}
\text{TeX-only-supports} `\int_use:N \c_max_register_int \% ` of each type. All the ‘#1’-registers have been used. `\text{This run will be aborted now.}
\{ \_\_kernel_msg_new:nnn { kernel } { non-base-function } \}
\{ Function ‘#1’-is-not-a-base-function \}
\{ \_\_kernel_msg_new:nnn { kernel } { command-not-defined } \}
Code-level functions must contain ‘:’ to separate the argument specification from the function name. This is needed when defining conditionals or variants, or when building a parameter text from the number of arguments of the function.

Integers larger than \(2^{30}-1\) cannot be stored in arrays. An attempt was made to store \#3 at position \#2 in the array ‘\#1’. The largest allowed value \#4 will be used instead.

Access to an entry beyond an array’s bounds.

LaTeX has been asked to define ‘\#1’ as a protected predicate. Only expandable tests can have a predicate version.

Bounds ordered backwards in \texttt{\textbackslash int.rand:nn} {\#1}-{\#2}.

Conditional form ‘\#1’ for function ‘\#2’ unknown.

LaTeX has been asked to define the conditional form ‘\#1’ of the function ‘\#2’, but only ‘TF’, ‘T’, ‘F’, and ‘p’ forms exist.

No property given in definition of key ‘\#1’.

Inside ‘\texttt{\textbackslash keys_define:nn}’ each key name needs a property: \\ \\
LaTeX did not find a ‘.’ to indicate the start of a property.

The property ‘\#1’ accepts boolean values only.

The property ‘\#1’ only accepts the values ‘true’ and ‘false’.

LaTeX has been asked to define ‘\#1’ as a protected predicate.

Only expandable tests can have a predicate version.

LaTeX has been asked to define ‘\#1’ as a protected predicate.
The property '#1' requires a value.

LaTeX was asked to set property '#1' for key '#2'.
No value was given for the property, and one is required.

The key property '#1' is unknown.
LaTeX has been asked to set the property '#1' for key '#2': this property is not defined.

Quotes in shell command '#1'.
Shell commands cannot contain quotes (').

Quark test function '#1' is invalid.
LaTeX has been asked to create quark test function '#1' but that name with signature '#2', but that signature is not valid.

LaTeX has been asked to create a new scan mark '#1' but this name has already been used for a scan mark.

The sequence '#1' is too long to be shuffled by TeX.
TeX has \int_eval:n { \c_max_register_int + 1 } - toks registers: this only allows to shuffle up to \int_use:N \c_max_register_int \items.- The list will not be shuffled.

Variable '#1' undefined.

Variant form '#1' longer than base signature of '#2'.

variable not defined.

variant too long.
LaTeX has been asked to create a variant of the function '#2' with a signature starting with '#1', but that is longer than the signature (part after the colon) of '#2'.

\__kernel_msg_new:nnnn { kernel } { invalid-variant }
{ Variant-form:'#1'-invalid-for-base-form:'#2'. }
{
    \c__msg_coding_error_text_tl
LaTeX has been asked to create a variant of the function '#2' with a signature starting with '#1', but cannot change an argument from type '#3' to type '#4'.
}
\__kernel_msg_new:nnnn { kernel } { invalid-exp-args }
{ Invalid-variant-specifier:'#1'-in'#2'. }
{
    \c__msg_coding_error_text_tl
LaTeX has been asked to create an \iow_char:N\exp_args:N... function with signature '#2' but '#1' is not a valid argument specifier.
}
\__kernel_msg_new:nnnn { kernel } { deprecated-variant }
{ Variant-form:'#1'-deprecated-for-base-form:'#2'.
One-should-not-change-an-argument-from-type:'#3'-to-type:'#4'
\str_case:nnF {#3}
{
    { n } { :-use-a-\token_if_eq_charcode:NNTF \#4 \c \ntv \variant? } 
    { N } { :-base-form-only-accepts-a-single-token-argument. } 
    {#4} { :-base-form-is-already-a-variant. } 
} . 
}

Some errors are only needed in package mode if debugging is enabled by one of the options enable-debug, check-declarations, log-functions, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

\{package\}
\__kernel_msg_new:nnnn { kernel } { enable-debug }
{ To-use:'#1'-load-expl3-with-the-'enable-debug'-option. }
{
    The-function:'#1'-will-be-ignored-because-it-can-only-work-if-some-internal-functions-in-expl3-have-been-appropriately-defined.-This-only-happens-if-one-of-the-options-'enable-debug',-'check-declarations'-or-'log-functions'-was-given-when-loading-expl3.
}
\{package\}
\{initex\}
\__kernel_msg_new:nnnn { kernel } { enable-debug }
{ '#1'-cannot-be-used-in-format-mode. }
{
    The-function:'#1'-will-be-ignored-because-it-can-only-work-if-some-internal-functions-in-expl3-have-been-appropriately-defined.-This-only-happens-in-package-mode-(and-only-if-one-of-the-options-'enable-debug',-'check-declarations'-or-'log-functions')-

626
Some errors only appear in expandable settings, hence don’t need a “more-text” argument.

Messages used by the “show” functions.

```latex
\__kernel_msg_new:nnn { kernel } { show-clist }
{ The-comma-list- \tl_if_empty:nF {#1} { #1 - } \tl_if_empty:nTF {#2} { is-empty \> . } { contains-the-items-(without-outer-braces): #2 . } }
\__kernel_msg_new:nnn { kernel } { show-intarray }
{ The-integer-array-#1-contains-#2-items: \tl_if_empty:nF {#2} { is-empty \> . } { contains-the-pairs-(without-outer-braces): #2 . } }
\__kernel_msg_new:nnn { kernel } { show-prop }
{ The-property-list-#1- \tl_if_empty:nTF {#2} { is-empty \> . } { contains-the-pairs-(without-outer-braces): #2 . } }
\__kernel_msg_new:nnn { kernel } { show-seq }
{ The-sequence-#1- \tl_if_empty:nTF {#2} { is-empty \> . } { contains-the-items-(without-outer-braces): #2 . } }
\__kernel_msg_new:nnn { kernel } { show-streams }
{ The-following- }
```

627
19.7 Expandable errors

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed \TeX{} an undefined control sequence, \LaTeX{} error:. It is thus interrupted, and shows the context, which thanks to the odd-looking \use:n is

\begin{verbatim}
<argument> \LaTeX{} error:
The error message.
\end{verbatim}

In other words, \TeX{} is processing the argument of \use:n, which is \LaTeX{} error: \langle error message\rangle. Then \__msg_expandable_error:w cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until \exp_end:. The \exp_end: prevents losing braces around the user-inserted text if any, and stops the expansion of \exp:w. The group is used to prevent \LaTeX{}-error: from being globally equal to \scan_stop:.

\begin{verbatim}
\group_begin:
\cs_set_protected:Npn \__msg_tmp:w #1#2
{ \cs_new:Npn \__msg_expandable_error:n ##1
{ \exp:w \exp_after:wN \exp_after:wN
\exp_after:wN \exp_after:wN \exp_after:wN \exp_end: \use:n { #1 #2 ##1 } #2 }
\cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 { ##1 }
\exp_args:Ncx \__msg_tmp:w { \LaTeX{}-error: }
\group_end:
\end{verbatim}
The command built from the csname \c__msg_text_prefix_tl LaTeX / #1 / #2 takes four arguments and builds the error text, which is fed to \__msg_expandable_error:n with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded.

\exp_args_generate:n { oooo }
\cs_new:Npn \__kernel_msg_expandable_error:nnnnnn #1#2#3#4#5#6
{\exp_args:Ne \__msg_expandable_error:n
{\exp_args:Nc \exp_args:Noooo
  \c__msg_text_prefix_tl LaTeX / #1 / #2
  \tl_to_str:n {#3}
  \tl_to_str:n {#4}
  \tl_to_str:n {#5}
  \tl_to_str:n {#6}}}
\cs_new:Npn \__kernel_msg_expandable_error:nnnnn #1#2#3#4#5
{\__kernel_msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} {}}
\cs_new:Npn \__kernel_msg_expandable_error:nnnn #1#2#3#4
{\__kernel_msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {}}
\cs_new:Npn \__kernel_msg_expandable_error:nnn #1#2#3
{\__kernel_msg_expandable_error:nnnnnn {#1} {#2} {#3} {}}
\cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnnn { nnnnnn }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnn { nnnn }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnn { nnnf }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:nnn { nnn }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:nn { nff }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:n { nnf }
\cs_generate_variant:Nn \__kernel_msg_expandable_error:w { nnf }

(End definition for \__kernel_msg_expandable_error:nnnnnn and others.)

20 l3file implementation

The following test files are used for this code: m3file001.
20.1 Input operations

20.1.1 Variables and constants

\l__ior_internal_tl Used as a short-term scratch variable.
\tl_new:N \l__ior_internal_tl

(End definition for \l__ior_internal_tl.)

\c__ior_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.
\int_const:Nn \c__ior_term_ior { 16 }

(End definition for \c__ior_term_ior.)

\g__ior_streams_seq A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to \LaTeXe as they are needed (initially none are needed), so the starting point varies!
\seq_new:N \g__ior_streams_seq
\seq_gset_split:Nnn \g__ior_streams_seq { , } { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }

(End definition for \g__ior_streams_seq.)

\l__ior_stream_tl Used to recover the raw stream number from the stack.
\tl_new:N \l__ior_stream_tl

(End definition for \l__ior_stream_tl.)

\g__ior_streams_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For \LaTeXe and plain \TeX this data is stored in \count16: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In Con\TeXt, we need to look at \count38 but there is no subtraction: like the original plain \TeX/\LaTeXe mechanism it holds the value of the last stream allocated.
\prop_new:N \g__ior_streams_prop
\int_step_inline:nnn \{ 0 \} { \cs_if_exist:NTF \normalend \{ \tex_count:D 38 - \} \{ \tex_count:D 16 - \% \cs_if_exist:NT \loccount \{ - 1 \} \} \} { \prop_gput:Nnn \g__ior_streams_prop \{1\} \{ Reserved-by-format \} }

(End definition for \g__ior_streams_prop.)
20.1.2 Stream management

Reserving a new stream is done by defining the name as equal to using the terminal.
\ior_new:N \ior_new:c

(End definition for \ior_new:N. This function is documented on page 156.)

\g_tmpa_ior \g_tmpb_ior

The usual scratch space.
\ior_new:N \g_tmpa_ior \ior_new:N \g_tmpb_ior

(End definition for \g_tmpa_ior and \g_tmpb_ior. These variables are documented on page 163.)

\ior_open:Nn \ior_open:cn

Use the conditional version, with an error if the file is not found.
\ior_open:Nn #1#2
{ \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
\cs_generate_variant:Nn \ior_open:Nn { c }

(End definition for \ior_open:Nn. This function is documented on page 156.)

\l__ior_file_name_tl

Data storage.
\tl_new:N \l__ior_file_name_tl

(End definition for \l__ior_file_name_tl.)

\ior_open:NnTF \ior_open:cnTF

An auxiliary searches for the file in the \TeX{}, \LaTeX{} 2ε and \LaTeX{}3 paths. Then pass the
file found to the lower-level function which deals with streams. The full_name is empty
when the file is not found.
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
{ \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
  { \__kernel_ior_open:No #1 \l__ior_file_name_tl
    \prg_return_true:
  }

  \prg_return_false: }
\cs_if_exist:NT \normalend
{ \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

(End definition for \ior_open:NnTF. This function is documented on page 157.)

\_ior_new:N

In package mode, streams are reserved using \newread before they can be managed by
ior. To prevent ior from being affected by redefinitions of \newread (such as done by
the third-party package morewrites), this macro is saved here under a private name. The
complicated code ensures that \_ior_new:N is not \outer despite plain \TeX{}'s \newread
being \outer. For Con\TeX{}t, we have to deal with the fact that \newread works like our
own: it actually checks before altering definition.
\package
\exp_args:NNf \cs_new_protected:Npn \__ior_new:N
{ \exp_args:NNc \exp_after:wN \exp_stop_f: { \newread } }
\cs_if_exist:NT \normalend
{ \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
  \cs_set_protected:Npn \__ior_new:N #1

631
The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, life gets more complex as it’s important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask plain TeX or LaTeX2ε for a new stream and use that number (after a bit of conversion).

Here, we act defensively in case LuaTeX is in use with an extensionless file name.

Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range \([0,15]\)), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.
Show the property lists, but with some “pretty printing”. See the \texttt{l3msg} module. The first argument of the message is \texttt{ior} (as opposed to \texttt{iow}) and the second is empty if no read stream is open and non-empty (the list of streams formatted using \texttt{\msg_show_item_unbraced:nn}) otherwise. The code of the message \texttt{show-streams} takes care of translating ior/iow to English.

\begin{verbatim}
\cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nnnnnn }
\cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nnnnnn }
\cs_new_protected:Npn \__ior_list:N #1
{ #1 { \LaTeX / kernel } { show-streams } 
 { ior } 
 { \prop_map_function:NN \g__ior_streams_prop \msg_show_item_unbraced:nn }
 { } { } }
\end{verbatim}

(End definition for \texttt{\ior_show_list:}, \texttt{\ior_log_list:}, and \texttt{\__ior_list:N}. These functions are documented on page 157.)

\section*{20.1.3 Reading input}

\begin{verbatim}
\tex_closein:D \#1
\prop_gremove:NV \g__ior_streams_prop \#1
\seq_if_in:NVF \g__ior_streams_seq \#1
\begin{verbatim}
{ \seq_gpush:NV \g__ior_streams_seq \#1 \c__ior_term_ior }
\end{verbatim}
\cs_gset_eq:NN #1 \c__ior_term_ior }
\end{verbatim}

(End definition for \texttt{\ior_close:N}. This function is documented on page 157.)

\ior_if_eof_p:N \ior_if_eof:N \TF
\begin{verbatim}
\prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
{ \cs_if_exist:NTF #1
\int_compare:nTF { -1 < #1 < \c__ior_term_ior } \if_eof:w #1 \prg_return_true: \else: \prg_return_false: \fi: }
\end{verbatim}

To test if some particular input stream is exhausted the following conditional is provided. The primitive test can only deal with numbers in the range \[0, 15\] so we catch outliers (they are exhausted).

\begin{verbatim}
\prg_new_conditional:Npnn \ior_if_eof:N #1 \{ p , T , F , TF \}
{ \cs_if_exist:NTF \int_compare:nTF { -1 < #1 < \c__ior_term_ior } \if_eof:w #1 \prg_return_true: \else: \prg_return_false: \fi: }
\end{verbatim}

(End definition for \texttt{\ior_if_eof:w}. This function is documented on page 163.)

\begin{verbatim}
\end{verbatim}

633
And here we read from files.

\ior_get:NN
\__ior_get:NN
\ior_get:NN\TF

(End definition for \ior_if_eof:NTF. This function is documented on page 160.)

\ior_str_get:NN
\__ior_str_get:NN
\ior_str_get:NN\TF

Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

\c__ior_term_noprompt_ior

For reading without a prompt.

(End definition for \ior_str_get:NN, \__ior_str_get:NN, and \ior_str_get:NTF. These functions are documented on page 158.)

\c__ior_term_noprompt_ior
Getting from the terminal is better with pretty-printing.

```
\ior_get_term:nN \ior_str_get_term:nN \__ior_get_term:NnN
```

13192 \cs_new_protected:Npn \ior_get_term:nN #1#2
13193 \{ \__ior_get_term:NnN \__ior_get:NN {#1} #2 \}
13194 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
13195 \{ \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 \}
13196 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
13197 \{
13198 \group_begin:
13199 \\tex_escapechar:D = -1 \scan_stop:
13200 \tl_if_blank:nTF {#2}
13201 \{ \exp_args:NNc #1 \c__ior_term_noprompt_ior \}
13202 \{ \exp_args:NNc #1 \c__ior_term_ior \}
13203 \{#2\}
13204 \exp_args:NNv \group_end:
13205 \tl_set:Nn #3 {#2}
13206 \}

(End definition for \ior_get_term:nN, \ior_str_get_term:nN, and \__ior_get_term:NnN. These functions are documented on page 262.)

```
\ior_map_break: \ior_map_break:n
```

Usual map breaking functions.

```
\ior_map_inline:Nn \ior_str_map_inline:Nn \__ior_map_inline:NNn \__ior_map_inline:NNNn \__ior_map_inline_loop:NNN
```

Mapping to an input stream can be done on either a token or a string basis, hence the
set up. Within that, there is a check to avoid reading past the end of a file, hence the two
applications of \ior_if_eof:N and its lower-level analogue \if_eof:w. This mapping
cannot be nested with twice the same stream, as the stream has only one “current line”.

```
\cs_new_protected:Npn \ior_map_inline:Nn \ior_str_map_inline:Nn \__ior_map_inline:NNn \__ior_map_inline:NNNn \__ior_map_inline_loop:NNN
```

635
\ior_map_variable:NNn
\ior_str_map_variable:NNn
\__ior_map_variable:NNNn
\__ior_map_variable_loop:NNNn

Since the TeX primitive (\read or \readline) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

\cs_new_protected:Npn \ior_map_variable:NNn { \__ior_map_variable:NNNn \ior_get:NN }
\cs_new_protected:Npn \ior_str_map_variable:NNn { \__ior_map_variable:NNNn \ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
{ \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} } \prg_break_point:Nn \ior_map_break: { } }
\cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
{ #1 #2 #3 \if_eof:w #2 \exp_after:wN \ior_map_break: \fi: #4 \__ior_map_variable_loop:NNNn #1#2#3 {#4} }

(End definition for \ior_map_variable:NNn and others. These functions are documented on page 159.)

20.2 Output operations

\@@=iow

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

20.2.1 Variables and constants

Here we allocate two output streams for writing to the transcript file only (\c_log_iow) and to both the terminal and transcript file (\c_term_iow). Recent LuaTeX provide 128 write streams; we also use \c_term_iow as the first non-allowed write stream so its value depends on the engine.

\int_const:Nn \c_log_iow { -1 }
\int_const:Nn \c_term_iow
{ \bool_lazy_and:nnTF \sys_if_engine_luatex_p: \int_compare_p:nNn \tex_luatexversion:D > { 80 } \{ 128 \} \{ 16 \}}

(End definition for \c_log_iow and \c_term_iow. These variables are documented on page 163.)
\g__iow_streams_seq A list of the currently-available output streams to be used as a stack. The stream 18 is special, as \write18 is used to denote commands to be sent to the OS.

\l__iow_stream_tl Used to recover the raw stream number from the stack.

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

20.2.2 Internal auxiliaries

\s__iow_mark Internal scan marks.
\s__iow_stop Functions to gobble up to a scan mark.

\q__iow_nil Internal quarks.
20.3 Stream management

Reserving a new stream is done by defining the name as equal to writing to the terminal:

\iohnewN \newc

Odd but at least consistent.

\csnewprotectedNpN \iohnewN #1 { \csneweqNN #1 \ctermioh }
\csgeneratevariantNN \iohnewN { c }

(End definition for \iohnewN. This function is documented on page 156.)

The usual scratch space.

\gtmpaiow \gtmpbionw
\iohnewN \gtmpaiow
\iohnewN \gtmpbionw

(End definition for \gtmpaiow and \gtmpbionw. These variables are documented on page 163.)

As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

\csnewprotectedNpN \iohopenNn #1#2
\iohopenNc
\csnewprotectedNpN \iohopenNn #1#2
\tlsetNnx \l__ioh_filename_tl { \__kernel_file_name_sanitize:n {#2} }
\iowcloseN #1
\seqgpopNNTF \g__ioh_streams_seq \l__ioh_stream_tl
\l__ioh_open_stream:NV #1 \l__ioh_filename_tl
\__iow_new:N #1
\tlsetNnx \l__ioh_stream_tl { \intevaln {#1} }
\__iow_open_stream:NV #1 \l__ioh_filename_tl

\csgeneratevariantNn \iohopenNn { c }
\csnewprotectedNpN \__ioh_open_stream:Nn \#1
\csnewprotectedNpN \__ioh_open_stream:NV \#2
\texglobalD \texchardefD #1 = \l__ioh_stream_tl \scanstop
\propgputNNn \g__ioh_streams_prop #1 \{#2\}
\teximmediateD \texopenoutD #1 \l__kernel_file_name_quote:n \{#2\} \scanstop

638
Closing a stream is not quite the reverse of opening one. First, the close operation is
easier than the open one, and second as the stream is actually a number we can use it
directly to show that the slot has been freed up.

\begin{verbatim}
\cs_new_protected:Npn \iow_close:N #1
  {
    \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
    {
      \tex_immediate:D \tex_closeout:D #1
      \prop_gremove:NV \g__iow_streams_prop #1
      \seq_if_in:NVF \g__iow_streams_seq #1
      { \seq_gpush:NV \g__iow_streams_seq #1 }
      \cs_gset_eq:NN #1 \c_term_iow
    }
  }
\end{verbatim}

(End definition for \iow_close:N. This function is documented on page 157.)

\begin{verbatim}
\cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nnxxxx }
\cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nnxxxx }
\cs_new_protected:Npn \__iow_list:N #1
  {
    \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
    {
      \prop_map_function:NN \g__iow_streams_prop
      \msg_show_item_unbraced:nn
    }
  }
\end{verbatim}

(End definition for \iow_show_list:, \iow_log_list:, and \__iow_list:N. These functions are docu-
mented on page 157.)

20.3.1 Deferred writing

\begin{verbatim}
\cs_new_protected:Npm \iow_shipout_x:Nn \iow_shipout_x:Nx \iow_shipout_x:cn \iow_shipout_x:cx
\cs_new_protected:Npm \iow_shipout_x:Nn #1#2
{ \tex_write:D #1 \exp_not:n {#2} }
\cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
\end{verbatim}

(End definition for \iow_shipout_x:Nn. This function is documented on page 161.)

\begin{verbatim}
\cs_new_protected:Npm \iow_shipout:Nn \iow_shipout:Nx \iow_shipout:cn \iow_shipout:cx
\cs_new_protected:Npm \iow_shipout:Nn #1#2
{ \tex_write:D #1 \exp_not:n {#2} }
\cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
\end{verbatim}

(End definition for \iow_shipout:Nn. This function is documented on page 161.)
20.3.2 Immediate writing

If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

\begin{verbatim}
\cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
\{ \int_compare:nNnTF {#1} = {#2}
\{ \use:n \}
\{ \exp_args:No \__iow_with:nNnn \{ \int_use:N #1 \} #1 {#2} \}
\}
\cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
\{ \int_set:Nn #2 {#3} \#4 \int_set:Nn #2 {#1} \}
\}
\end{verbatim}

(End definition for \__kernel_iow_with:Nnn and \__iow_with:nNnn.)

\iow_now:Nn This routine writes the second argument onto the output stream without expansion. If this stream isn’t open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don’t use the expansion done by \write to get the \rm variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled. We set the \newlinechar to 10 using \__kernel_iow_with:Nnn to support formats such as plain \TeX: otherwise, \iow_newline: would not work. We do not do this for \iow_shipout:Nn or \iow_shipout_x:Nn, as \TeX looks at the value of the \newlinechar at shipout time in those cases.

\begin{verbatim}
\cs_new:Npn \iow_newline: { ^\hfill J }
\end{verbatim}

(End definition for \iow_newline:. This function is documented on page 161.)

\iow_log:n Writing to the log and the terminal directly are relatively easy.

\begin{verbatim}
\cs_set_protected:Npn \iow_log:x \{ \iow_now:Nx \c_log_iow \}
\cs_new_protected:Npn \iow_log:n \{ \iow_now:Nn \c_log_iow \}
\cs_set_protected:Npn \iow_term:x \{ \iow_now:Nx \c_term_iow \}
\cs_new_protected:Npn \iow_term:n \{ \iow_now:Nn \c_term_iow \}
\end{verbatim}

(End definition for \iow_log:n and \iow_term:n. These functions are documented on page 160.)

20.3.3 Special characters for writing

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

\begin{verbatim}
\cs_new:Nn \iow_newline: { ^\hfill J }
\end{verbatim}

(End definition for \iow_newline:. This function is documented on page 161.)
Function to write any escaped char to an output stream.
\cs_new_eq:NN \iow_char:N \cs_to_str:N

(End definition for \iow_char:N. This function is documented on page 161.)

20.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

\l_iow_line_count_int

This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TeXLive and MiKTeX.
\int_new:N \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }

(End definition for \l_iow_line_count_int. This variable is documented on page 162.)

\l_iow_newline_tl

The token list inserted to produce a new line, with the \textit{run-on text}.
\tl_new:N \l__iow_newline_tl

(End definition for \l__iow_newline_tl.)

\l__iow_line_target_int

This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.
\int_new:N \l__iow_line_target_int

(End definition for \l__iow_line_target_int.)

\__iow_set_indent:n
\__iow_unindent:w
\l__iow_one_indent_tl
\l__iow_one_indent_int

The one_indent variables hold one indentation marker and its length. The \texttt{\__iow_unindent:w} auxiliary removes one indentation. The function \texttt{\__iow_set_indent:n} (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int
\cs_new:Npn \__iow_one_indent_int { { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } } \int_set:Nn \l__iow_one_indent_int { \str_count:N \l__iow_one_indent_tl } \exp_last_unbraced:NNo \cs_set:Npn \__iow_one_indent_tl { } \exp_args:Nx \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } } (End definition for \__iow_set_indent:n and others.)

\l__iow_one_indent_tl
\l__iow_one_indent_int

The current indentation (some copies of \texttt{\l__iow_one_indent_tl} and its number of characters.
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int

(End definition for \l__iow_one_indent_tl and \l__iow_one_indent_int.)

641
These hold the current line of text and a partial line to be added to it, respectively.

(End definition for \__iow_line_tl and \__iow_line_part_tl.)

Indicates whether the line was broken precisely at a chunk boundary.

(End definition for \__iow_line_break_bool.)

Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

(End definition for \__iow_wrap_tl.)

Every special action of the wrapping code is starts with the same recognizable string, \c__iow_wrap_marker_tl. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of \escapechar here is not very important, but makes \c__iow_wrap_marker_tl look marginally nicer.

(End definition for \c__iow_wrap_marker_tl and others.)

We set \iow_allow_break:n to produce an error when outside messages. Within wrapped message, it is set to \__iow_allow_break: when valid and otherwise to \__iow_allow_break_error:. The second produces an error expandably.
\iow_indent:n
\__iow_indent:n
\__iow_indent_error:n

We set \iow_indent:n to produce an error when outside messages. Within wrapped message, it is set to \__iow_indent:n when valid and otherwise to \__iow_indent_error:n.

The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably.

Note that there are no forced line-break, so the indentation only changes when the next line is started.

\iow_wrap:nnnN
\iow_wrap:nxnN

The main wrapping function works as follows. First give $\backslash$, $\text{\textbackslash}$, and other formatting commands the correct definition for messages and perform the given setup #3.

The definition of $\text{\textbackslash}$ uses an “other” space rather than a normal space, because the latter might be absorbed by TEX to end a number or other $f$-type expansions. Use \texttt{\textbackslashconditionally@traceoff} if defined; it is introduced by the \texttt{trace} package and suppresses uninteresting tracing of the wrapping code.
Then fully-expand the input: in package mode, the expansion uses \protect mechanism in the same way as \typeout. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset \iow_indent:n to its error definition: it only works in the first argument of \iow_wrap:nnnN.

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count \l_iow_line_count_int instead).

Sanity check.

There is then a loop over the input, which stores the wrapped result in \l__iow_wrap_tl. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The \tl_to_str:N step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from \l__iow_wrap_tl.

Escape spaces and change newlines to \c__iow_wrap_newline_marker_tl. Set up a few variables, in particular the initial value of \l__iow_wrap_tl: the space stops the f-expansion of the main wrapping function and \use_none:n removes a newline marker inserted by later code. The main loop consists of repeatedly calling the chunk auxiliary to wrap chunks delimited by (newline or indentation) markers.

(End definition for \iow_wrap:nnnN. This function is documented on page 162.)

644
\__iow_wrap_: The \__iow_wrap_chunk: and \__iow_wrap_next: auxiliaries are defined indirectly to obtain the expansions of \c-_catcode_other_space_tl and \c__iow_wrap_marker_tl in their definition. The next auxiliary calls a function corresponding to the type of marker (its ##2), which can be newline or indent or unindent or end. The first argument of the \__iow_wrap: auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call next. Otherwise, set up a call to \__iow_wrap_line: , including the indentation if the current line is empty, and including a trailing space (#1) before the \__iow_wrap_end_chunk: auxiliaries.
This is followed by \{\langle\text{string}\rangle\ \langle\text{intexpr}\rangle\ ;\}. It stores the \langle\text{string}\rangle\ up to \langle\text{intexpr}\rangle\ characters from the current chunk into \l\_\_iow\_line\_part\_tl. Characters are grabbed 8 at a time and left in \l\_\_iow\_line\_part\_tl by the \text{line\_loop} auxiliary. When \(k < 8\) remain to be found, the \text{line\_aux} auxiliary calls the \text{line\_end} auxiliary followed by (the single digit) \(k\), then \(7 - k\) empty brace groups, then the chunk’s remaining characters. The \text{line\_end} auxiliary leaves \(k\) characters from the chunk in the line part, then ends the assignment. Ignore the \text{use\_none:nnnnn} line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the \text{end\_chunk} auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments \#2–\#9 of the \text{line\_loop} auxiliary or as one of the arguments \#2–\#8 of the \text{line\_end} auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a \text{exp\_stop\_f} has to be inserted to stop the \text{exp\_w}. The weird \text{use\_none:nnnnn} ensures that the required data is in the right place.
or: \use_none:nnnnnn
\or: \__iow_wrap_line_seven:nnnnnn
\fi:
{ } { } { } { } { } { } #3
}
\cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
{
#2 #3 #4 #5 #6 #7 #8
\use_none:nnnn \int_eval:w 8 - ; #9
\token_if_eq_charcode:NNTF \c_space_token #9
{ \__iow_wrap_line_end:nw { } }
{ \if_false: { \fi: } \__iow_wrap_break:w #9 }
}
\cs_new:Npn \__iow_wrap_line_end:nw #1
{
\if_false: { \fi: }
\__iow_wrap_store_do:n {#1}
\__iow_wrap_next_line:w
}
\cs_new:Npn \__iow_wrap_end_chunk:w
#1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
{
\if_false: { \fi: }
\exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
}
(End definition for \__iow_wrap_line:nw and others.)
\__iow_wrap_break:w
\__iow_wrap_break_first:w
\__iow_wrap_break_none:w
\__iow_wrap_break_loop:w
\__iow_wrap_break_end:w

Functions here are defined indirectly: \__iow_tmp:w is eventually called with an “other” space as its argument. The goal is to remove from \l__iow_line_part_tl the part after the last space. In most cases this is done by repeatedly calling the \break_loop auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument ##3 is ? \__iow_wrap_break_end:w instead of a single token, and that \break_end auxiliary leaves in the assignment the line until the last space, then calls \__iow_wrap_line_end:nw to finish up the line and move on to the next. If there is no space in \l__iow_line_part_tl then the \break_first auxiliary calls the \break_none auxiliary. In that case, if the current line is empty, the complete word (including ##4, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).
\begin{verbatim}
\cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
  {
    \use_none:nn #1 \__iow_wrap_break_none:w \__iow_wrap_break_loop:w ##1 #1 #2
  }
\cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 #3 \s__iow_mark ##4 #1
  {
    \tl_if_empty:NTF \l__iow_line_tl
      { ##2 #1 #3 \__iow_wrap_line_end:nw { } }
      { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
  }
\cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 #3
  {
    \use_none:n ##3
    #1 #1
    \__iow_wrap_break_loop:w ##2 #1 ##3
  }
\cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
  { ##1 \__iow_wrap_line_end:nw { } ##3 }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
\end{verbatim}

(End definition for \__iow_wrap_break:w and others.)

\__iow_wrap_next_line:w The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call \__iow_wrap_line:nw to find characters for the next line (remembering to account for the indentation).

\begin{verbatim}
\cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
  {
    \tl_clear:N \l__iow_line_tl
    \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
      { \tl_clear:N \l__iow_line_tl
        \bool_set_true:N \l__iow_line_break_bool
        \__iow_wrap_next:nw { \l__iow_line_target_int } }
      { \__iow_wrap_line:nw
        { \l__iow_indent_tl }
        \l__iow_line_target_int - \l__iow_indent_int ; #1 #2 \s__iow_stop }
  }
\end{verbatim}

(End definition for \__iow_wrap_next_line:w.)

\__iow_wrap_allow_break:n This is called after a chunk has been wrapped. The \l__iow_line_part_tl typically ends with a space (except at the beginning of a line?), which we remove since the allow_break marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

\begin{verbatim}
\cs_new_protected:Npn \__iow_wrap_allow_break:n #1
  {
    \tl_set:Nx \l__iow_line_tl
    { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }

\end{verbatim}

648
These functions are called after a chunk has been wrapped, when encountering
indent/unindent markers. Add the line part (last line part of the previous chunk)
to the line so far and reset a boolean denoting the presence of a line-break. Most impor-
tantly, add or remove one indent from the current indent (both the integer and the token
list). Finally, continue wrapping.

These functions are called after a chunk has been line-wrapped, when encountering a
newline/end marker. Unless we just took a line-break, store the line part and the line
so far into the whole \l__iow_wrap_tl, trimming a trailing space. In the newline case
look for a new line (of length \l__iow_line_target_int) in a new chunk.
First add the last line part to the line, then append it to \l__iow_line_tl with the appropriate new line (with “run-on” text), possibly with its last space removed (#1 is empty or \l__iow_line_tl is empty).

\cs_new_protected:Npn \__iow_wrap_store_do:n #1 \l__iow_line_tl \l__iow_line_part_tl \l__iow_newline_tl #1 \l__iow_line_tl \tl_set:Nx \l__iow_line_tl { \l__iow_line_tl \l__iow_line_part_tl } \tl_set:Nx \l__iow_wrap_tl { \l__iow_wrap_tl \l__iow_newline_tl \l__iow_line_tl } \tl_clear:N \l__iow_line_tl \tl_clear:N \l__iow_wrap_tl(End definition for \__iow_wrap_store_do:n.)

Remove one trailing “other” space from the argument if present.

\cs_set_protected:Npn \__iow_tmp:w #1 \exp_after:wN \__iow_wrap_trim:N ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark \s__iow_stop\cs_new:Npn \__iow_wrap_trim:w ##1 \s__iow_mark { \__iow_wrap_trim_aux:w ##1 \s__iow_mark #1 \s__iow_mark #2 \s__iow_stop {##1} } \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl(End definition for \__iow_wrap_trim:N, \__iow_wrap_trim:w, and \__iow_wrap_trim_aux:w.)

\tl_new:N \l__file_internal_tl(End definition for \l__file_internal_tl.)

\str_new:N \g_file_curr_dir_str \str_new:N \g_file_curr_ext_str \str_new:N \g_file_curr_name_str(End definition for \g_file_curr_dir_str, \g_file_curr_ext_str, and \g_file_curr_name_str. These variables are documented on page 163.)

The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by \LaTeX2ε (we must be in the preamble and loaded using \usepackage or \RequirePackage). As \LaTeX2ε doesn’t store directory and name separately, we stick to the same convention here. In pre-loading, \@currnamestack is empty so is skipped.
\seq_new:N \g__file_stack_seq
\group_begin:
\cs_set_protected:Npn \__file_tmp:w #1#2#3
{\tl_if_blank:nTF {#1}
{\cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
{ { } {##2} { } }
\seq_gput_right:Nx \g__file_stack_seq
{ \exp_after:wN \__file_tmp:w \tex_jobname:D
" \tex_jobname:D " \s__file_stop
}
}
\seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
\__file_tmp:w
}
\cs_if_exist:NT \@currnamestack
{\tl_if_empty:NF \@currnamestack
{ \exp_after:wN \__file_tmp:w \@currnamestack }
}
\group_end:
\langle/package\rangle

\g__file_record_seq The total list of files used is recorded separately from the current file stack, as nothing
is ever popped from this list. The current file name should be included in the file list.
In format mode, this is done at the very start of the \TeX run. In package mode we will
eventually copy the contents of \@filelist.
\seq_new:N \g__file_record_seq
\group_begin:
\tex_everyjob:D \exp_after:wN
\tex_the:D \tex_everyjob:D
\seq_gput_right:NW \g__file_record_seq \g_file_curr_name_str
\group_end:
\langle/intex\rangle

\End definition for \g__file_stack_seq

\l__file_base_name_tl For storing the basename and full path whilst passing data internally.
\l__file_full_name_tl
\tl_new:N \l__file_base_name_tl
\tl_new:N \l__file_full_name_tl
\End definition for \l__file_base_name_tl and \l__file_full_name_tl

\l__file_dir_str \l__file_ext_str \l__file_name_str
Used in parsing a path into parts: in contrast to the above, these are never used outside
of the current module.
\str_new:N \l__file_dir_str
\str_new:N \l__file_ext_str
\str_new:N \l__file_name_str
\End definition for \l__file_dir_str, \l__file_ext_str, and \l__file_name_str
\l_file_search_path_seq  The current search path.
\l__file_dir_str, \l__file_ext_str, and \l__file_name_str.
\l_file_search_path_seq
\seq_new:N \l_file_search_path_seq
(End definition for \l_file_search_path_seq. This variable is documented on page 164.)
\l__file_tmp_seq  Scratch space for comma list conversion in package mode.
\seq_new:N \l__file_tmp_seq
(End definition for \l__file_tmp_seq.)

20.4.1 Internal auxiliaries
\s__file_stop  Internal scan marks.
\scan_new:N \s__file_stop
(End definition for \s__file_stop.)
\q__file_nil  Internal quarks.
\quark_new:N \q__file_nil
(End definition for \q__file_nil.)
\__file_quark_if_nil_p:n  Branching quark conditional.
\__file_quark_if_nil:nTF
(End definition for \__file_quark_if_nil:nTF.)
\q__file_recursion_tail \q__file_recursion_stop  Internal recursion quarks.
\quark_new:N \q__file_recursion_tail
\quark_new:N \q__file_recursion_stop
(End definition for \q__file_recursion_tail and \q__file_recursion_stop.)
\__file_if_recursion_tail_break:NN \__file_if_recursion_tail_stop_do:Nn
Functions to query recursion quarks.
\__kernel_quark_new_test:N \__file_if_recursion_tail_break:NN \__file_if_recursion_tail_stop_do:NN
(End definition for \__file_if_recursion_tail_break:NN and \__file_if_recursion_tail_stop_-do:NN.)
\__kernel_file_name_sanitize:n \__kernel_file_name_expand_loop:w \__kernel_file_name_expand_N_type:Wn
\__kernel_file_name_expand_group:Wn \__kernel_file_name_expand_space:Wn
\__kernel_file_name_strip_quotes:n \__kernel_file_name_strip_quotes:nnn \__kernel_file_name_strip_quotes:nnw
\__kernel_file_name_trim_spaces:n \__kernel_file_name_trim_spaces:w \__kernel_file_name_trim_spaces_aux:n
\__kernel_file_name_trim_spaces_aux:w
Expanding the file name without expanding active characters is done using the same
token-by-token approach as for example case changing. The finale outcome only need be
e-type expandable, so there is no need for the shuffling that is seen in other locations.
\cs_new:Npn \__kernel_file_name_sanitize:n #1
\exp_args:Ne \__kernel_file_name_trim_spaces:n
\exp_args:Ne \__kernel_file_name_strip_quotes:n
\exp_args:Ne \__kernel_file_name_expand_loop:w #1
\q__file_recursion_tail \q__file_recursion_stop
652
Quoting file name uses basically the same approach as for \texttt{luaquotejobname}: count the " tokens and remove them.
Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

(End definition for `\__kernel_file_name_sanitize:n` and others.)

(End definition for `\__kernel_file_name_quote:n` and `\__kernel_file_name_quote:nw`.)

The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.
The approach here is similar to that for `\tl_set_rescan:Nnn`. The file contents are grabbed as an argument delimited by `\c__file_marker_tl`. A few subtleties: braces in `\if_false: ... \fi:` to deal with possible alignment tabs, `\tracingnesting` to avoid a warning about a group being closed inside the `\scantokens`, and `\prg_return_true:` is placed after the end-of-file marker.

```latex
\cs_new_protected:Npn \file_get:nnN #1#2#3
\begin{verbatim}
\file_get:nnNF {#1} {#2} #3
{ \tl_set:Nn #3 { \q_no_value } }
\end{verbatim}
```

```latex
\prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
\begin{verbatim}
\file_get_full_name:nNTF {#1} \l__file_full_name_tl
{ \exp_args:NV \__file_get_aux:nnN \l__file_full_name_tl {#2} #3
\prg_return_true:
}
{ \prg_return_false: }
\end{verbatim}
```

```latex
\cs_new_protected:Npx \__file_get_aux:nnN #1#2#3
\begin{verbatim}
\exp_not:N \if_false: { \exp_not:N \fi:
\group_begin:
\int_set_eq:NN \tex_tracingnesting:D \c_zero_int
\exp_not:N \exp_args:No \tex_everyeof:D
{ \exp_not:N \c__file_marker_tl }
#2 \scan_stop:
\exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
\exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
\exp_not:N \tex_input:D
\sys_if_engine_luatex:T
{ {#1} }
{ \exp_not:N \\kernel_file_name_quote:n {#1} \scan_stop: }
\exp_not:N \if_false: } \exp_not:N \fi:
\end{verbatim}
```

```latex
\cs_new_protected:Npx \__file_get_do:Nw #1#2
\begin{verbatim}
\group_end:
\tl_set:No #1 {#2}
\end{verbatim}
```

(End definition for `\c__file_marker_tl`)

\__file_size:n A copy of the primitive where it’s available, or the LuaTeX equivalent if relevant.

```latex
\cs_new_eq:NN \__file_size:n \tex_filesize:D
\sys_if_engine_luatex:T
{ }
```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 164.)
\cs_gset:Npn \__file_size:n #1
{
  \lua_now:e
  \l3kernel_filesize ( \lua_escape:e {#1} )
}
\end {definition for \__file_size:n}

File searching can be carried out if the \pdffilesize primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

First, we check of the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. For package mode, \input@path is a token list not a sequence.

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.
As \TeX automatically adds \texttt{.tex} if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

\begin{verbatim}
\cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
\cs_new:Npn \__file_name_end: { }
\As \TeX automatically adds \texttt{.tex} if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

\begin{verbatim}
\cs_new:Npn \__file_ext_check:n #1 { \__file_ext_check:nw { / } #1 / \q__file_nil / \s__file_stop }
\cs_new:Npn \__file_ext_check:nw #1 #2 / #3 / #4 \s__file_stop { \__file_quark_if_nil:nTF {#3} { \exp_args:No \__file_ext_check:nnw { \use_none:n #1 } {#2} . \q__file_nil . \s__file_stop } { \__file_ext_check:nnw { #1 #2 / } #3 / #4 \s__file_stop } }
\cs_new:Npx \__file_ext_check:nnw #1#2#3 . #4 . \s__file_stop { \exp_not:N \__file_quark_if_nil:nTF {#4} { \exp_not:N \__file_ext_check:nn { #1 #2 } { #1 #2 \tl_to_str:n { .tex } } } { #1 #2 } }
\cs_new:Npn \__file_ext_check:nn #1#2 { \tl_if_blank:eTF { \__file_size:n {#2} } {#1} { \int_compare:nNnTF { \__file_size:n {#1} } = { \__file_size:n {#2} } {#2} {#1} } }
\end{verbatim}

Deal with the fact that the primitive might not be available.

\begin{verbatim}
\bool_lazy_or:nnF { \cs_if_exist_p:N \texfilesize:D } { \sys_if_engine_luatex_p: } { \cs_gset:Npn \file_full_name:n #1 { \__kernel_msg_expandable_error:nnn { kernel } { primitive-not-available } { (pdf)filesize } } \__kernel_msg_new:nnnn { kernel } { primitive-not-available } { Primitive\textbackslash token_to_str:N #1 not-available } }
\end{verbatim}

657
The version of your TeX engine does not provide functionality equivalent to the \#1 primitive.

(End definition for \file_full_name:n and others. This function is documented on page 164.)

These functions pre-date using \tex_filesize:D for file searching, so are get functions with protection. To avoid having different search set ups, they are simply wrappers around the code above.

\cs_new_protected:Npn \file_get_full_name:nN #1#2
  { \file_get_full_name:nNF {#1} #2
    \tl_set:Nn #2 { \q_no_value } }
\cs_generate_variant:Nn \file_get_full_name:nN { V }
\prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
  { \tl_set:Nx #2 { \file_full_name:n {#1} } }
\tl_if_empty:NTF #2
  { \prg_return_false: }
  { \prg_return_true: }
\cs_generate_variant:Nn \file_get_full_name:nNT { V }
\cs_generate_variant:Nn \file_get_full_name:nNF { V }
\cs_generate_variant:Nn \file_get_full_name:nNTF { V }

If \tex_filesize:D is not available, the way to test if a file exists is to try to open it: if it does not exist then TeX reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to \prg_break_point:. If nothing is found, \#2 is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with .tex extension in that directory, and if it exists we include the .tex extension in the result.

\bool_lazy_or:nnF
  { \cs_if_exist_p:N \tex_filesize:D }
  { \sys_if_engine_luatex_p: }
  { \prg_set_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
    { \tl_set:Nx \l__file_base_name_tl
      { \__kernel_file_name_sanitize:n {#1} }
      \__file_get_full_name_search:nN {#1} #2
      \seq_map_inline:Nn \l_file_search_path_seq
        { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
      \tl_set:Nn \l__file_full_name_tl { \q_no_value }
    }\package
\cs_if_exist:NT \input@path
  { \tl_map_inline:Nn \input@path
    { \__file_get_full_name_search:nN {##1} \tl_map_break:n }
  }\package
\tl_set:Nn \l__file_full_name_tl { \q_no_value }
\prg_break_point:
\quark_if_no_value:NTF \l__file_full_name_tl
{
  \ior_close:N \g__file_internal_ior
  \prg_return_false:
}
{
  \file_parse_full_name:VNNN \l__file_full_name_tl
  \l__file_dir_str \l__file_name_str \l__file_ext_str
  \str_if_empty:NT \l__file_ext_str
  {
    \__kernel_ior_open:No \g__file_internal_ior
    { \l__file_full_name_tl .tex }
    \ior_if_eof:NF \g__file_internal_ior { \tl_put_right:Nn \l__file_full_name_tl { .tex } }
  }
  \ior_close:N \g__file_internal_ior
  \tl_set_eq:NN #2 \l__file_full_name_tl
  \prg_return_true:
}
\cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
{
  \tl_set:Nx \l__file_full_name_tl { \tl_to_str:n {#1} \l__file_base_name_tl }
  \__kernel_ior_open:No \g__file_internal_ior \l__file_full_name_tl
  \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
}

\bool_lazy_or:nnF
{ \cs_if_exist_p:N \tex_filesize:D }
{ \sys_if_engine_luatex_p: }
{ \ior_new:N \g__file_internal_ior }

(End definition for \file_get_full_name:nN, \file_get_full_name:NTF, and \__file_get_full_name_search:nN. These functions are documented on page 164.)

\g__file_internal_ior
A reserved stream to test for file existence, if required.
\bool_lazy_or:nnF
{ \cs_if_exist_p:N \tex_filesize:D }
{ \sys_if_engine_luatex_p: }
{ \ior_new:N \g__file_internal_ior }

(End definition for \g__file_internal_ior.)

\file_mdfive_hash:n
\file_size:n
\file_timestamp:n
\__file_details:nn
\__file_details_aux:nn
\__file_mdfive_hash:n

Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.
\cs_new:Npn \file_mdfive_hash:n #1
{ \__file_details:nn {#1} { mdfivesum } }
\cs_new:Npn \file_size:n #1
{ \__file_details:nn {#1} { size } }
\cs_new:Npn \file_timestamp:n #1
{ \__file_details:nn {#1} { moddate } }
\cs_new:Npn \__file_details:nn #1#2
{ \file_full_name:n {#1} } {#2}

659
\cs_new:Npn \_file_details_aux:nn #1#2
{\tl_if_blank:nF {#1}
{ \use:c { tex_file #2 :D } {#1} }
}
\sys_if_engine_luatex:TF
{ \cs_gset:Npn \_file_details_aux:nn #1#2
{\lua_now:e
{ l3kernel.file#2 ( " \lua_escape:e { \#1 } " ) } }
}
{ \cs_gset:Npn \file_mdfive_hash:n #1
{ \exp_args:Ne \__file_mdfive_hash:n { \file_full_name:n {#1} } }
\cs_new:Npn \__file_mdfive_hash:n #1
{ \tex_mdfivesum:D file {#1} }
}

(End definition for \file_mdfive_hash:n and others. These functions are documented on page 165.)

\file_hex_dump:nnn
\__file_hex_dump_auxi:nnn
\__file_hex_dump_auxii:nnnn
\__file_hex_dump_auxiii:nnnn
\file_hex_dump:n
\__file_hex_dump:n

These are separate as they need multiple arguments or the file size. For \LaTeX, the emulation does not need the file size so we save a little on expansion.
\cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
\int_compare:nNnTF {#4} > 0
{ \__file_hex_dump_auxiv:nnn {#4} }
{ \exp_args:Ne \__file_hex_dump_auxiv:nnn
\int_eval:n { #2 + #4 } }
{#1} {#3}
\cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
\tex_filedump:D
offset \int_eval:n { #2 - 1 } -
length \int_eval:n { #1 - #2 + 1 }
{#3}
\sys_if_engine_luatex:T
{ \cs_gset:Npn \__file_hex_dump_auxiv:nnn #1#2#3
\lua_now:e
{ l3kernel.filedump
\int_eval:n { #2 - 1 },
\int_eval:n { #1 - #2 + 1 }
\lua_escape:e {#3} ",
\lua_now:e
}
}
\cs_new:Npn \file_hex_dump:n #1
\tl_if_blank:nF {#1}
{ \tex_filedump:D length \tex_filesize:D {#1} {#1} }
\sys_if_engine_luatex:T
{ \cs_gset:Npn \__file_hex_dump:n #1
\lua_now:e
{ l3kernel.filedump ( " \lua_escape:e { #1 } " ) }
}
\cs_new:Npn \__file_hex_dump:n #1
\sys_if_engine_luatex:T
{ \cs_gset:Npn \__file_hex_dump:n #1
\lua_now:e
{ l3kernel.filedump ( " \lua_escape:e { #1 } " ) }
}
\sys_if_engine_luatex:T
{ \cs_gset:Npn \__file_hex_dump:n #1
\lua_now:e
{ l3kernel.filedump ( " \lua_escape:e { #1 } " ) }
}
\sys_if_engine_luatex:T
{ \cs_gset:Npn \__file_hex_dump:n #1
\lua_now:e
{ l3kernel.filedump ( " \lua_escape:e { #1 } " ) }
}

(End definition for \file_hex_dump:nnn and others. These functions are documented on page 165.)

Non-expandable wrappers around the above in the case where appropriate primitive support exists.
Where the primitive is not available, issue an error: this is a little more conservative than absolutely needed, but does work.

(End definition for \texttt{\file_get_hex_dump:nNTF} and others. These functions are documented on page 165.)

Custom code due to the additional arguments.

\file_get_hex_dump:nN
\file_get_hex_dump:nNTF
As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

\begin{verbatim}
cs_new:Npn \_file_str_cmp:nn #1#2 { \tex_strcmp:D {#1} {#2} }
sys_if_engine_luatex:T
{ \cs_set:Npn \_file_str_cmp:nn #1#2
{ \lua_now:e
{ l3kernel.strcmp
( " \_file_str_escape:n {#1}",
 " \_file_str_escape:n {#2}" )
}
}
\cs_new:Npn \_file_str_escape:n #1
{ \lua_escape:e
{ \use:e { {#1} } }
}
\end{verbatim}

(End definition for \_file_str_cmp:nn and \_file_str_escape:n.)

Comparison of file date can be done by using the low-level nature of the string comparison functions.

\begin{verbatim}
\prg_new_conditional:Nnn \_file_compare_timestamp:nnN #1#2#3
{ \exp_args:Nee \_file_compare_timestamp:nnN
\exp_args:Nee \_file_full_name:n {#1}
\exp_args:Nee \_file_full_name:n {#3}
\tl_if_blank:nTF #2
\cs_new:Npn \_file_compare_timestamp:nnN #1#2#3
{ \tl_if_blank:nTF {#1}
\end{verbatim}

663
\file_if_exist:nTF \file_if_exist:nTF The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located
then the return value is empty.

\begin{verbatim}
\prg_new_protected_conditional:Npnn \file_if_exist:n \{ T, F, TF \}
\{
    \file_get_full_name:nNTF \(#1\) \l__file_full_name_tl
    \{
        \prg_return_true: }
    \{
        \prg_return_false: }
\}
\end{verbatim}

(End definition for \file_if_exist:nTF. This function is documented on page 164.)

\begin{verbatim}
\file_if_exist_input:n
\file_if_exist_input:nF
\end{verbatim}

Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the ⟨true code⟩ would be inconsistent with other conditionals.

\begin{verbatim}
\cs_new_protected:Npn \file_if_exist_input:n \#1
\{
    \file_get_full_name:nNT \{\#1\} \l__file_full_name_tl
    \{
        \__file_input:V \l__file_full_name_tl }
\}
\cs_new_protected:Npn \file_if_exist_input:nF \#1\#2
\{
    \file_get_full_name:nNTF \{\#1\} \l__file_full_name_tl
    \{
        \__file_input:V \l__file_full_name_tl }
    \{\#2\}
\}
\end{verbatim}

(End definition for \file_if_exist_input:n and \file_if_exist_input:nF. These functions are documented on page 167.)

\begin{verbatim}
\file_input_stop:
\end{verbatim}

A simple rename.

\begin{verbatim}
\cs_new_protected:Npn \file_input_stop: \{ \tex_endinput:D \}
\end{verbatim}

(End definition for \file_input_stop:. This function is documented on page 167.)

\begin{verbatim}
\__kernel_file_missing:n
\end{verbatim}

An error message for a missing file, also used in \ior_open:Nn.

\begin{verbatim}
\cs_new_protected:Npm \__kernel_file_missing:n \#1
\{
    \__kernel_msg_error:nnx \{ kernel \} \{ file-not-found \}
    \{ \__kernel_file_name_sanitize:n \{\#1\} \}
\}
\end{verbatim}

(End definition for \__kernel_file_missing:n.)

\begin{verbatim}
\file_input:n
\__file_input:n
\__file_input:V
\__file_input_push:n
\__kernel_file_input_push:n
\__file_input_pop:
\__kernel_file_input_pop:
\__file_input_pop:nnn
\end{verbatim}

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g__file_stack_seq, and add it to the file list, either \g__file_record_seq, or \@filelist in package mode.

\begin{verbatim}
\cs_new_protected:Npm \file_input:n \#1
\{
    \file_get_full_name:nNTF \{\#1\} \l__file_full_name_tl
    \{
        \__file_input:V \l__file_full_name_tl }
    \{ \__kernel_file_missing:n \{\#1\} \}
\}
\end{verbatim}
Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

\begin{verbatim}
\cs_new_protected:Npn \_file_input_push:n #1
\seq_gpush:Nx \g__file_stack_seq
\{ #1 \}
\end{verbatim}

\begin{verbatim}
\cs_new_eq:NN \__kernel_file_input_push:n \_file_input_push:n
\end{verbatim}

\begin{verbatim}
\cs_new_protected:Npn \_file_input_pop:nnn #1#2#3
\str_gset:Nn \g_file_curr_dir_str {#1}
\str_gset:Nn \g_file_curr_name_str {#2}
\str_gset:Nn \g_file_curr_ext_str {#3}
\end{verbatim}

(End definition for \_file_input:n and others. This function is documented on page 166.)

Parsing starts by stripping off any surrounding quotes. Then find the directory \#4 by splitting at the last / (The auxiliary returns true/false depending on whether it found the delimiter.) We correct for the case of a file in the root /, as in that case we wish to keep the trailing (and only) slash. Then split the base name \#5 at the last dot. If there was indeed a dot, \#5 contains the name and \#6 the extension without the dot, which we
add back for convenience. In the special case of no extension given, the auxiliary stored
the name into #6, we just have to move it to #5.

```latex
\cs_new_protected:Npn \file_parse_full_name:nNNN #1#2#3#4
\exp_after:wN \__file_parse_full_name_auxi:w
  \tl_to_str:n { #1 " #1 " } \s__file_stop #2#3#4
\}
\cs_generate_variant:Nn \file_parse_full_name:nNNN { V }
\cs_new_protected:Npn \__file_parse_full_name_auxi:w
  \tl_to_str:n { #1 " #2 " #3 } \s__file_stop #4#5#6
\}
\cs_new_protected:Npn \__file_parse_full_name_split:nNNNTF {#1} / #4 #5
\exp_args:No \__file_parse_full_name_split:nNNNTF {#5} . #5 #6
\exp_after:wN \__file_parse_full_name_split:nNNNTF {#2} / #4 #5
\}
\cs_new_protected:Npn \file_show_list: {
  \__file_list:N \msg_show:nnxxxx }
\cs_new_protected:Npn \file_log_list: {
  \__file_list:N \msg_log:nnxxxx }
\cs_new_protected:Npn \__file_list:N #1
  \tl_if_empty:nTF {##1}
    \tl_if_empty:nTF {##3}
      \str_set:Nn #4 {##2}
      \tl_if_empty:nTF {##1}
        \str_clear:N #3
        \use_ii:nn
      \else
        \str_set:Nx #3 { \str_tail:n {##1} }
        \use_i:nn
      \fi
    \else
      \str_set_eq:NN #5 #6
      \str_clear:N #6
    \fi
  \else
    \__file_list:N #1 #2 #3 \s__file_stop 
  \fi
\}
`End definition for \file_parse_full_name:nNNN, \__file_parse_full_name_auxi:w, and \__file_parse_full_name_split:nNNNTF. This function is documented on page 165.`

A function to list all files used to the log, without duplicates. In package mode, if \@filelist is still defined, we need to take this list of file names into account (we
capture it \AtBeginDocument into \g__file_record_seq), turning it to a string (this
does not affect the commas of this comma list).
When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `@filelist` must be turned to strings before being added to `g__file_record_seq`.

(End definition for `file_show_list` and others. These functions are documented on page 167.)

20.5 `GetIdInfo`

As documented in `expl3.dtx` this function extracts file name etc from an `svn` `Id` line. This used to be how we got version number and so on in all modules, so it had to be defined in `l3bootstrap`. Now it’s more convenient to define it after we have set up quite a lot of tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard `svn` `Id` line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the `Id` keyword!

```
\cs_new_protected:Npn \GetIdInfo
\__file_id_info_auxi:w
\__file_id_info_auxii:w
\__file_id_info_auxiii:w
```

\GetIdInfo
A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using SVN cp but has not been checked in. That leaves a special marker -1 version, which has no further data. Dealing correctly with that is the reason for the space in the line to use __file_id_info_auxii:w.

\cs_new_protected:Npn __file_id_info_auxii:w #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
\{\tl_set:Nn \ExplFileName {#2} \tl_set:Nn \ExplFileExtension {#3} \tl_set:Nn \ExplFileVersion {#4} \str_if_eq:nnTF {#4} {-1} {\tl_set:Nn \ExplFileDate { 0000/00/00 } \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop} \}

Here, #1 is Id, #2 is the file name, #3 is the extension, #4 is the version, #5 is the check in date and #6 is the check in time and user, plus some trailing spaces. If #4 is the marker -1 value then #5 and #6 are empty.

\cs_new_protected:Npm __file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
\{\tl_set:Nn \ExplFileDate { #1/#2/#3 } \}

(End definition for \GetIdInfo and others. This function is documented on page 7.)

20.6 Messages

\__kernel_msg_new:nnnn { kernel } { file-not-found } { File-'#1'-not-found. }
\{ \{The-requested-file-could-not-be-found-in-the-current-directory,-\in-the-T\e X-search-path-or-in-the-LaTeX-search-path.\} \}
\__kernel_msg_new:nnn { kernel } { file-list } { >-File-List<- \#1 \} \}
.............
20.7 Functions delayed from earlier modules

\c_sys_platform_str Detecting the platform on Lua\TeX \ is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package platform), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

\sys_if_engine_luatex:TF

\cs_if_exist:NF \c_sys_platform_str

<@@=sys>
We can now set up the tests.

\begin{verbatim}
\clist_map_inline:nn { unix , windows }
{ \__file_const:nn { sys_if_platform_ #1 } \\
\{ \str_if_eq_p:Vn \c_sys_platform_str { #1 } \}
}
\end{verbatim}

(End definition for \sys_if_platform_unix:TF and \sys_if_platform_windows:TF. These functions are documented on page 115.)

\section{\texttt{l3skip} implementation}

\subsection{Length primitives renamed}

Primitives renamed.

\begin{verbatim}
\if_dim:w \__dim_eval:w \__dim_eval_end:
\cs_new_eq:NN \if_dim:w \tex_ifdim:D \\
\cs_new_eq:NN \__dim_eval:w \tex_dimexpr:D \\
\cs_new_eq:NN \__dim_eval_end: \tex_relax:D
\end{verbatim}

(End definition for \if_dim:w, \__dim_eval:w, and \__dim_eval_end:. This function is documented on page 182.)

\subsection{Internal auxiliaries}

Internal scan marks.

\begin{verbatim}
\scan_new:N \s__dim_mark \\
\scan_new:N \s__dim_stop
\end{verbatim}

(End definition for \s__dim_mark and \s__dim_stop.)

Functions to gobble up to a scan mark.

\begin{verbatim}
\cs_new:Npn \__dim_use_none_delimit_by_s_stop:w #1 \s__dim_stop { }
\end{verbatim}

(End definition for \__dim_use_none_delimit_by_s_stop:w.)

\subsection{Creating and initialising \texttt{dim} variables}

Allocating (\texttt{dim}) registers ...

\begin{verbatim}
\dim_new:N \dim_new:c
\end{verbatim}

(End definition for \dim_new:N. This function is documented on page 168.)
Contrarily to integer constants, we cannot avoid using a register, even for constants. We
cannot use \texttt{dim_gset:Nn} because debugging code would complain that the constant is
not a global variable. Since \texttt{dim_const:Nn} does not need to be fast, use \texttt{dim_eval:n}
to avoid needing a debugging patch that wraps the expression in checking code.

```latex
\texttt{\cs_new_protected:Npn \dim_const:Nn \#1\#2}
\begin{verbatim}
 \dim_new:N \#1
 \tex_global:D \#1 - \dim_eval:n \{\#2\} \scan_stop:
\end{verbatim}
\texttt{\cs_generate_variant:Nn \dim_const:Nn \{ c \}}
```

(End definition for \texttt{dim_const:Nn}. This function is documented on page 168.)

Reset the register to zero. Using \texttt{c_zero_skip} deals with the case where the variable
passed is incorrectly a skip (for example a \LaTeX\ length).

```latex
\texttt{\cs_new_protected:Npn \dim_zero:N \#1}
\begin{verbatim}
 \dim_if_exist:NTF \#1 { \dim_zero:N \#1 } { \dim_new:N \#1 }
\end{verbatim}
```

(End definition for \texttt{dim_zero:N} and \texttt{dim_gzero:N}. These functions are documented on page 168.)

Create a register if needed, otherwise clear it.

```latex
\texttt{\cs_new_protected:Npn \dim_zero_new:N \#1}
\begin{verbatim}
 \dim_if_exist:NTF \#1 { \dim_zero:N \#1 } { \dim_new:N \#1 }
\end{verbatim}
```

(End definition for \texttt{dim_zero_new:N} and \texttt{dim_gzero_new:N}. These functions are documented on page 168.)

Copies of the \texttt{cs} functions defined in \texttt{l3basics}.

```latex
\texttt{\prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N}
\begin{verbatim}
 \TF , \T , \F , \p
\end{verbatim}
```

(End definition for \texttt{dim_if_exist:NTF}. This function is documented on page 168.)

Setting dimensions is easy enough but when debugging we want both to check that the
variable is correctly local/global and to wrap the expression in some code. The \texttt{\scan_stop:}
deals with the case where the variable passed is a skip (for example a \LaTeX\ length).

```latex
\texttt{\cs_new_protected:Npn \dim_set:Nn \#1\#2}
\begin{verbatim}
 \dim_if_exist:NTF \#1 { \dim_set:Nn \#1 } { \dim_new:N \#1 }
\end{verbatim}
```

(End definition for \texttt{dim_set:Nn} and \texttt{dim_gset:Nn}. These variables are documented on page 168.)

21.4 Setting dim variables
(End definition for \texttt{dim\_set:Nn} and \texttt{dim\_gset:Nn}. These functions are documented on page 169.)

\texttt{\dim\_set_eq:NN} \dim\_set_eq:cN \dim\_set_eq:Nc \dim\_set_eq:cc
\texttt{\dim\_gset_eq:NN} \dim\_gset_eq:cN \dim\_gset_eq:Nc \dim\_gset_eq:cc

All straightforward, with a \texttt{\scan\_stop:} to deal with the case where \texttt{#1} is (incorrectly) a skip.

\begin{verbatim}
\cs_new_protected:Npn \dim\_set_eq:NN #1#2 \scan\_stop: }
\cs_generate_variant:Nn \dim\_set_eq:NN { c , Nc , cc }
\end{verbatim}

(End definition for \texttt{dim\_set_eq:NN} and \texttt{dim\_gset_eq:NN}. These functions are documented on page 169.)

\texttt{\dim\_add:Nn} \dim\_add:cn \dim\_gadd:Nn \dim\_gadd:cn
\texttt{\dim\_sub:Nn} \dim\_sub:cn \dim\_gsub:Nn \dim\_gsub:cn

Using by here deals with the (incorrect) case \texttt{dimen123}. Using \texttt{\scan\_stop:} deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as \texttt{\tex\_global:D} followed by the local versions. The debugging code is inserted by \texttt{\_\_dim\_tmp:w}.

\begin{verbatim}
\cs_new_protected:Npn \dim\_add:Nn #1#2 \scan\_stop: }
\cs_generate_variant:Nn \dim\_add:Nn { c }
\cs_new_protected:Npn \dim\_gadd:Nn #1#2 \scan\_stop: }
\cs_generate_variant:Nn \dim\_gadd:Nn { c }
\cs_new_protected:Npn \dim\_sub:Nn #1#2 \scan\_stop: }
\cs_generate_variant:Nn \dim\_sub:Nn { c }
\cs_new_protected:Npn \dim\_gsub:Nn #1#2 \scan\_stop: }
\cs_generate_variant:Nn \dim\_gsub:Nn { c }
\end{verbatim}

(End definition for \texttt{dim\_add:Nn} and others. These functions are documented on page 169.)

\texttt{\dim\_abs:n} \texttt{\__dim\_abs:N} \texttt{\dim\_max:nn} \texttt{\dim\_min:nn} \texttt{\__dim\_maxmin:wwN}

Functions for \texttt{min}, \texttt{max}, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

\begin{verbatim}
\cs_new:Npn \dim\_abs:n #1 \scan\_stop: }
\exp\_after:wN \__dim\_abs:N \__dim\_eval:w #1 \__dim\_eval\_end: \scan\_stop:
\cs_new:Npn \__dim\_abs:N #1 { \if\_meaning:w - \texttt{#1} \else: \exp\_after:wN \texttt{#1} \fi: }
\cs_new:Npn \dim\_max:nn \__dim\_maxmin:wwN \__dim\_eval:w #1 \__dim\_eval\_end: \scan\_stop:
\cs_new:Npn \dim\_min:nn \__dim\_maxmin:wwN \__dim\_eval:w #1 \__dim\_eval\_end: \scan\_stop:
\end{verbatim}

21.5 Utilities for dimension calculations

Functions for \texttt{min}, \texttt{max}, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.
\dim_use:N \__dim_eval:w #2 ;
> 
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:
}
\cs_new:Npn \dim_min:nn #1#2
{
\dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
\dim_use:N \__dim_eval:w #1 \exp_after:wN ;
\dim_use:N \__dim_eval:w #2 ;
<
\__dim_eval_end:

(End definition for \dim_abs:n and others. These functions are documented on page 169.)

\dim_ratio:nn
\__dim_ratio:n
With dimension expressions, something like 10 pt * ( 5 pt / 10 pt ) does not work. Instead, the ratio part needs to be converted to an integer expression. Using \int_-value:w forces everything into sp, avoiding any decimal parts.

\cs_new:Npn \dim_ratio:nn #1#2
{ \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
\cs_new:Npn \__dim_ratio:n #1
{ \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

(End definition for \dim_ratio:nn and \dim_ratio:n. This function is documented on page 170.)

21.6 Dimension expression conditionals

\dim_compare_p:nNn
\dim_compare:nNnTF
Simple comparison.

\prg_new_conditional:Npnnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
{ \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}

(End definition for \dim_compare:nNnTF. This function is documented on page 170.)

\dim_compare_p:n
\dim_compare:nTF
\__dim_compare:w
\__dim_compare:wNN
\__dim_compare:wN
\__dim_compare:w
\__dim_compare:wN
\__dim_compare:w
\__dim_compare:wN
\__dim_compare:w
\__dim_compare:wN
\__dim_error:
\__dim_compare_error:
\exp_after:wN \__dim_compare:w

This code is adapted from the \int_compare:nTF function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing \__-dim_compare_error:. Just like for integers, the looping auxiliary \__dim_compare:wNN closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with pt (with category other). Thus we do not need specific auxiliaries for the three “simple” relations <, =, and >.

\prg_new_conditional:Npnnn \dim_compare:n #1#2#3 { p , T , F , TF }
{ \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
  \prg_return_true: \else: \prg_return_false: \fi:
}

(End definition for \dim_compare:nNnTF. This function is documented on page 170.)
\dim_case:nn
\dim_case:nnT
\dim_case:nnF
\dim_case:nnTF
\dim_case:nnnTF
\__dim_case:nnTF
\__dim_case:nnnTF
\__dim_case_end:nw
\dim_case:nw
\dim_case:nwT
\dim_case:nwF
\__dim_case:nwTF
\__dim_case:nwF
\__dim_case:nwT
\__dim_case:nwF
\__dim_case_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
\__dim_compare_error:
21.7 Dimension expression loops

\texttt{\textbackslash dim\_while\_do:nn} \texttt{\textbackslash dim\_until\_do:nn} \texttt{\textbackslash dim\_do\_while:nn} \texttt{\textbackslash dim\_do\_until:nn} \texttt{while\_do} and \texttt{do\_while} functions for dimensions. Same as for the \texttt{int} type only the names have changed.

(End definition for \texttt{\textbackslash dim\_case:nnTF} and others. This function is documented on page 172.)
while_do and do_while functions for dimensions. Same as for the int type only the names have changed.

\begin{verbatim}
\cs_new:Npn \dim_while_do:nNnn #1#2#3#4
\{\dim_compare:nNnT {#1} #2 {#3}
\{#4\}
dim_while_do:nNnn {#1} #2 {#3} {#4}\}
\cs_new:Npn \dim_until_do:nNnn #1#2#3#4
{\dim_compare:nNnF {#1} #2 {#3}
\{#4\}
dim_until_do:nNnn {#1} #2 {#3} {#4}\}
\cs_new:Npn \dim_do_while:nNnn #1#2#3#4
{#4\\dim_compare:nNnT {#1} #2 {#3}\{ \dim_do_while:nNnn {#1} #2 {#3} {#4}\}}
\cs_new:Npn \dim_do_until:nNnn #1#2#3#4
{#4\\dim_compare:nNnF {#1} #2 {#3}\{ \dim_do_until:nNnn {#1} #2 {#3} {#4}\}}
\end{verbatim}

(End definition for \dim_while_do:nNnn and others. These functions are documented on page 173.)

\subsection*{21.8 Dimension step functions}

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

\begin{verbatim}
\cs_new:Npn \dim_step_function:nnnN #1#2#3
\{\exp_after:wN \__dim_step:wwwN
\exp_after:wN \tex_the:D \__dim_eval:w #1 \exp_after:wN ;\exp_after:wN \tex_the:D \__dim_eval:w #2 \exp_after:wN ;\exp_after:wN \tex_the:D \__dim_eval:w #3 ;\}
\cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
\{\dim_compare:nNnTF {#1} #2 {#3}
\{ \dim_step_function:nnnN \__dim_step:NnnnN #1#2#3\}
\}
\cs_new:Npn \__dim_step:NnnnN #1; #2; #3; #4
\{ \dim_step_function:nnnN \__dim_step:nnnN #1#2#3\}
\end{verbatim}

(End definition for \dim_step_function:nnnN and others. These functions are documented on page 173.)
The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using \dim_step_function:nnnN. We put a \prg_break_point:Nn so that map_break functions from other modules correctly decrement \g__kernel_prg_map_int before looking for their own break point. The first argument is \scan_stop:, so that no breaking function recognizes this break point as its own.

\dim_step_inline:nnnn \dim_step_variable:nnnNn \__dim_step:Nnnnn

\cs_new_protected:Npn \dim_step_inline:nnnn
\cs_new_protected:Npn \dim_step_variable:nnnNn
\cs_new_protected:Nnn \__dim_step:Nnnnn

\%dim_compare:nNnTF {#2} = \c_zero_dim
\%__kernel_msg_expandable_error:nnn { kernel } { zero-step } {#4}
\%use_none:nnnn
\% { \__dim_step:NnnnN < }
\%{#1} {#2} {#3} #4
\%}
\%cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
\%{#2} \dima \dima \dima \dima \dima \dima \dima \dima \dima
\%\dima \dima \dima \dima \dima \dima \dima \dima \dima
\%(End definition for \dim_step_function:nnnN, \__dim_step:wwwN, and \__dim_step:NnnnN. This function is documented on page 173.)

End definition for \dim_step_function:nnnN, \__dim_step:wwwN, and \__dim_step:NnnnN. This function is documented on page 173.)

\prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
21.9 Using dim expressions and variables

\texttt{\dim_eval:n}

Evaluating a dimension expression expandably.

\begin{verbatim}
\cs_new:Npn \dim_eval:n #1 { \dim_use:N \_\_\_dim_eval:w #1 \_\_\_\_dim_eval_end: }
\end{verbatim}

(End definition for \texttt{\dim_eval:n}. This function is documented on page 174.)

\texttt{\dim_sign:n \_\_\_dim_sign:Nw}

See \texttt{\dim_abs:n}. Contrarily to \texttt{\int_sign:n} the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since 0.2pt and 0pt start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

\begin{verbatim}
\cs_new:Npn \dim_sign:n #1 { \int_value:w \exp_after:wN \_\_\_dim_sign:Nw \dim_use:N \_\_\_dim_eval:w #1 \_\_\_\_dim_eval_end: ; \exp_stop_f: }
\cs_new:Npn \_\_\_dim_sign:Nw #1#2 ; { \if_dim:w #1#2 > \c_zero_dim 1 \else: \if_meaning:w - #1 -1 \else: \else: 0 \fi: \fi: }
\end{verbatim}

(End definition for \texttt{\dim_sign:n} and \texttt{\_\_\_dim_sign:Nw}. This function is documented on page 174.)

\texttt{\dim_use:N \_\_\_dim_use:c}

Accessing a \texttt{\langle dim \rangle}.

\begin{verbatim}
\cs_new:Npn \_\_\_dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
\end{verbatim}

(End definition for \texttt{\dim_use:N}. This function is documented on page 174.)

\texttt{\dim_to_decimal:n \_\_\_dim_to_decimal:w}

A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression \#1 then remove the trailing pt. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

\begin{verbatim}
\cs_new:Npn \dim_to_decimal:n #1 { \exp_after:wN \_\_\_dim_to_decimal:w \dim_use:N \_\_\_\_dim_eval:w #1 \_\_\_\_dim_eval_end: }
\end{verbatim}
\use:x
{
\cs_new:Npn \exp_not:N \__dim_to_decimal:w
##1 . ##2 \tl_to_str:n { pt }
}
\int_compare:nNnTF {#2} > { 0 }
{ #1 . #2 }
{ #1 }

(End definition for \dim_to_decimal:n and \__dim_to_decimal:w. This function is documented on page 174.)

\dim_to_decimal_in_bp:n
Conversion to big points is done using a scaling inside \__dim_eval:w as \TeX{} does
that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a
common case so is hand-coded for accuracy (and speed).
\cs_new:Npn \dim_to_decimal_in_bp:n #1
{ \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

(End definition for \dim_to_decimal_in_bp:n. This function is documented on page 175.)

\dim_to_decimal_in_sp:n
Another hard-coded conversion: this one is necessary to avoid things going off-scale.
\cs_new:Npn \dim_to_decimal_in_sp:n #1
{ \int_value:w \__dim_eval:w #1 \__dim_eval_end: }

(End definition for \dim_to_decimal_in_sp:n. This function is documented on page 175.)

\dim_to_decimal_in_unit:nn
An analogue of \dim_ratio:nn that produces a decimal number as its result, rather than
a rational fraction for use within dimension expressions.
\cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
{ \dim_to_decimal:n
{ \dim_ratio:nn {#1} {#2} }
}

(End definition for \dim_to_decimal_in_unit:nn. This function is documented on page 175.)

\dim_to_fp:n
Defined in \texttt{l3fp-convert}, documented here.

(End definition for \dim_to_fp:n. This function is documented on page 175.)

21.10  Viewing \texttt{dim} variables

\dim_show:N
Diagnostics.
\cs_new_eq:NN \dim_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \dim_show:N { c }

(End definition for \dim_show:N. This function is documented on page 175.)
Diagnostics. We don’t use the TeX primitive \showthe to show dimension expressions: this gives a more unified output.

14912 \cs_new_protected:Npn \dim_show:n
14913 \{ \msg_show_eval:Nn \dim_eval:n \}

(End definition for \dim_show:n. This function is documented on page 176.)

Diagnostics. Redirect output of \dim_show:n to the log.

14914 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
14915 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
14916 \cs_new_protected:Npn \dim_log:n
14917 \{ \msg_log_eval:Nn \dim_eval:n \}

(End definition for \dim_log:N and \dim_log:n. These functions are documented on page 176.)

21.11 Constant dimensions

\c_zero_dim
\c_max_dim

14918 \dim_const:Nn \c_zero_dim \{ 0 pt \}
14919 \dim_const:Nn \c_max_dim \{ 16383.99999 pt \}

(End definition for \c_zero_dim and \c_max_dim. These variables are documented on page 176.)

21.12 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
\g_tmpa_dim
\g_tmpb_dim

We provide two local and two global scratch registers, maybe we need more or less.

14920 \dim_new:N \l_tmpa_dim
14921 \dim_new:N \l_tmpb_dim
14922 \dim_new:N \g_tmpa_dim
14923 \dim_new:N \g_tmpb_dim

(End definition for \l_tmpa_dim and others. These variables are documented on page 176.)

21.13 Creating and initialising skip variables

\s__skip_stop

Internal scan marks.

14924 \scan_new:N \s__skip_stop

(End definition for \s__skip_stop.)

\skip_new:N
\skip_new:c

Allocation of a new internal registers.

14925 \{ \package
14926 \cs_new_protected:Npn \skip_new:N \#1
14927 \{ \__kernel_chk_if_free_cs:N \#1
14928 \cs:w newskip \cs_end: \#1
14929 \}
14930 \} \package
14931 \cs_generate_variant:Nn \skip_new:N \{ c \}

(End definition for \skip_new:N. This function is documented on page 176.)
Contrarily to integer constants, we cannot avoid using a register, even for constants. See \dim_const:Nn for why we cannot use \skip_gset:Nn.

\skip_zero:N Reset the register to zero.
\skip_zero:c
\skip_zero:N
\skip_gzero:N
\skip_gzero:c

Create a register if needed, otherwise clear it.
\skip_zero_new:N
\skip_zero_new:c
\skip_zero_new:N
\skip_zero_new:c

Copies of the cs functions defined in \texttt{l3basics}.
\skip_if_exist_p:N
\skip_if_exist_p:c
\skip_if_exist:N
\skip_if_exist:c

(End definition for \skip_zero:N and \skip_gzero:N. These functions are documented on page 177.)

21.14 Setting skip variables

Much the same as for dimensions.
\skip_set:Nn
\skip_set:cn
\skip_set:Nn
\skip_set:cn

All straightforward.
Using by here deals with the (incorrect) case \skip123.

As a result, only equality is tested.

With ε-T\TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string \fil which characterizes infinite glue.
21.16 Using skip expressions and variables

\skip_eval:n \skip_eval:n Evaluating a skip expression expandably.
\cs_new:Npn \skip_eval:n #1 \{ \skip_use:N \tex_glueexpr:D #1 \scan_stop: \}

(End definition for \skip_eval:n. This function is documented on page 178.)

\skip_use:N \skip_use:c Accessing a \langle skip\rangle.
\cs_new_eq:NN \skip_use:N \tex_the:D
\cs_generate_variant:Nn \skip_use:N { c }
\cs_new:Npn \skip_use:c #1 \{ \tex_the:D \cs:w #1 \cs_end: \}

(End definition for \skip_use:N. This function is documented on page 178.)

21.17 Inserting skips into the output

\skip_horizontal:N \skip_horizontal:c \skip_horizontal:n \skip_vertical:N \skip_vertical:c \skip_vertical:n Inserting skips.
\cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\cs_new:Npn \skip_horizontal:n #1 \{ \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: \}
\cs_generate_variant:Nn \skip_horizontal:N { c }
\cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\cs_new:Npn \skip_vertical:n #1 \{ \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: \}
\cs_generate_variant:Nn \skip_vertical:N { c }

(End definition for \skip_horizontal:N and others. These functions are documented on page 179.)

21.18 Viewing skip variables

\skip_show:N \skip_show:c Diagnostics.
\cs_new_eq:NN \skip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \skip_show:N { c }

(End definition for \skip_show:N. This function is documented on page 178.)

\skip_show:n Diagnostics. We don’t use the \TeX primitive \showthe to show skip expressions: this gives a more unified output.
\cs_new_protected:Npn \skip_show:n \{ \msg_show_eval:Nn \skip_eval:n \}

(End definition for \skip_show:n. This function is documented on page 178.)

\skip_log:N \skip_log:c \skip_log:n Diagnostics. Redirect output of \skip_show:n to the log.
\cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\cs_new_eq:NN \skip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \skip_log:n \{ \msg_log_eval:Nn \skip_eval:n \}

(End definition for \skip_log:N and \skip_log:n. These functions are documented on page 179.)
21.19  Constant skips
\c_zero_skip  Skips with no rubber component are just dimensions but need to terminate correctly.
\c_max_skip
\skip_const:Nn \c_zero_skip { \c_zero_dim }
\skip_const:Nn \c_max_skip { \c_max_dim }

(End definition for \c_zero_skip and \c_max_skip. These functions are documented on page 179.)

21.20  Scratch skips
\l_tmpa_skip \l_tmpb_skip \g_tmpa_skip \g_tmpb_skip
We provide two local and two global scratch registers, maybe we need more or less.
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip

(End definition for \l_tmpa_skip and others. These variables are documented on page 179.)

21.21  Creating and initialising muskip variables
\muskip_new:N \muskip_new:c
And then we add muskips.
\cs_new_protected:Npn \muskip_new:N #1
{ \__kernel_chk_if_free_cs:N #1 \cs:w newmuskip \cs_end: #1 }
\cs_generate_variant:Nn \muskip_new:N { c }

\muskip_const:Nn \muskip_const:cn
See \skip_const:Nn.
\cs_new_protected:Npm \muskip_const:Nn #1#2
{ \muskip_new:N #1 \text_global:D #1 - \muskip_eval:n {#2} \scan_stop: }
\cs_generate_variant:Nn \muskip_const:Nn { c }

(End definition for \muskip_const:Nn. This function is documented on page 180.)

\muskip_zero:N \muskip_zero:c \muskip_gzero:N \muskip_gzero:c
Reset the register to zero.
\cs_new_protected:Npm \muskip_zero:N #1
{ \c_zero_muskip }
\cs_new_protected:Npm \muskip_gzero:N #1
{ \c_zero_muskip }
\cs_generate_variant:Nn \muskip_zero:N { c }
\cs_generate_variant:Nn \muskip_gzero:N { c }

(End definition for \muskip_zero:N and \muskip_gzero:N. These functions are documented on page 180.)
Create a register if needed, otherwise clear it.

\cs_new_protected:Npn \muskip_zero_new:N #1
  \{ \muskip_if_exist:NTF #1 \{ \muskip_zero:N #1 \} \}
\cs_new_protected:Npn \muskip_gzero_new:N #1
  \{ \muskip_if_exist:NTF #1 \{ \muskip_gzero:N #1 \} \}
\cs_generate_variant:Nn \muskip_zero_new:N { c }
\cs_generate_variant:Nn \muskip_gzero_new:N { c }

(End definition for \muskip_zero_new:N and \muskip_gzero_new:N. These functions are documented on page 180.)

\prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
  \{ TF , T , F , p \}
\prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
  \{ TF , T , F , p \}

(End definition for \muskip_if_exist:NTF. This function is documented on page 180.)

21.22 Setting \texttt{muskip} variables

This should be pretty familiar.

\cs_new_protected:Npn \muskip_set:Nn #1#2
  \{ #1 \tex_muexpr:D #2 \scan_stop: \}
\cs_new_protected:Npn \muskip_gset:Nn #1#2
  \{ \tex_global:D #1 \tex_muexpr:D #2 \scan_stop: \}
\cs_generate_variant:Nn \muskip_set:Nn { c }
\cs_generate_variant:Nn \muskip_gset:Nn { c }

(End definition for \muskip_set:Nn and \muskip_gset:Nn. These functions are documented on page 181.)

\prg_new_eq_conditional:NNn \muskip_set_eq:NN \cs_if_exist:N
  \{ #1 = #2 \}
\prg_new_eq_conditional:NNn \muskip_set_eq:cN \cs_if_exist:c
  \{ #1 , #2 \}
\prg_new_eq_conditional:NNn \muskip_set_eq:Nc \cs_if_exist:NTF
  \{ #1 , #2 \}
\prg_new_eq_conditional:NNn \muskip_set_eq:cc \cs_if_exist:TF
  \{ #1 , #2 \}
\prg_new_eq_conditional:NNn \muskip_gset_eq:NN \cs_if_exist:N
  \{ #1 = #2 \}
\prg_new_eq_conditional:NNn \muskip_gset_eq:cN \cs_if_exist:c
  \{ #1 , #2 \}
\prg_new_eq_conditional:NNn \muskip_gset_eq:Nc \cs_if_exist:NTF
  \{ #1 , #2 \}
\prg_new_eq_conditional:NNn \muskip_gset_eq:cc \cs_if_exist:TF
  \{ #1 , #2 \}

(End definition for \muskip_set_eq:NN and \muskip_gset_eq:NN. These functions are documented on page 181.)

Using by here deals with the (incorrect) case \texttt{\textbackslash muskip123}.

\cs_new_protected:Npn \muskip_add:Nn #1#2
  \{ \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: \}
\cs_new_protected:Npn \muskip_gadd:Nn #1#2
  \{ \tex_global:D \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: \}
\cs_generate_variant:Nn \muskip_add:Nn { c }
\cs_generate_variant:Nn \muskip_gadd:Nn { c }
\cs_new_protected:Npn \muskip_sub:Nn #1#2
  \{ \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: \}
\cs_new_protected:Npn \muskip_gsub:Nn #1#2
  \{ \tex_global:D \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: \}
\cs_generate_variant:Nn \muskip_sub:Nn { c }
\cs_generate_variant:Nn \muskip_gsub:Nn { c }
\cs_new_protected:Npn \muskip_add:cc #1#2
  \{ \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: \}
\cs_new_protected:Npn \muskip_gadd:cc #1#2
  \{ \tex_global:D \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: \}
\cs_generate_variant:Nn \muskip_add:cc { c }
\cs_generate_variant:Nn \muskip_gadd:cc { c }
\cs_new_protected:Npn \muskip_sub:cc #1#2
  \{ \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: \}
\cs_new_protected:Npn \muskip_gsub:cc #1#2
  \{ \tex_global:D \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: \}
\cs_generate_variant:Nn \muskip_sub:cc { c }
\cs_generate_variant:Nn \muskip_gsub:cc { c }

(End definition for \muskip_add:cc and others. These functions are documented on page 180.)
21.23 Using muskip expressions and variables

\muskip_eval:n Evaluating a muskip expression expandably.

\cs_new:Npn \muskip_eval:n #1
{ \muskip_use:N \tex_muexpr:D #1 \scan_stop: }

(End definition for \muskip_eval:n. This function is documented on page 181.)

\muskip_use:N Accessing a \langle muskip \rangle.

\cs_new_eq:NN \muskip_use:N \tex_the:D
\cs_generate_variant:Nn \muskip_use:N { c }

(End definition for \muskip_use:N. This function is documented on page 181.)

21.24 Viewing muskip variables

\muskip_show:N, \muskip_show:c Diagnostics.

\cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \muskip_show:N { c }

(End definition for \muskip_show:N. This function is documented on page 181.)

\muskip_show:n Diagnostics. We don’t use the \TeX\ primitive \showthe to show muskip expressions: this gives a more unified output.

\cs_new_protected:Npn \muskip_show:n
{ \msg_show_eval:Nn \muskip_eval:n }

(End definition for \muskip_show:n. This function is documented on page 182.)

\muskip_log:N, \muskip_log:c, \muskip_log:n Diagnostics. Redirect output of \muskip_show:n to the log.

\cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \muskip_log:n
{ \msg_log_eval:Nn \muskip_eval:n }

(End definition for \muskip_log:N and \muskip_log:n. These functions are documented on page 182.)

21.25 Constant muskips

\c_zero_muskip \c_max_muskip Constant muskips given by their value.

\cs_new:Nn \c_zero_muskip { 0 \mu }
\cs_new:Nn \c_max_muskip { 16383.99999 \mu }

(End definition for \c_zero_muskip and \c_max_muskip. These functions are documented on page 182.)

21.26 Scratch muskips

\l_tmpa_muskip \l_tmpb_muskip \g_tmpa_muskip \g_tmpb_muskip We provide two local and two global scratch registers, maybe we need more or less.

\cs_new:Nn \l_tmpa_muskip
\cs_new:Nn \l_tmpb_muskip
\cs_new:Nn \g_tmpa_muskip
\cs_new:Nn \g_tmpb_muskip

(End definition for \l_tmpa_muskip and others. These variables are documented on page 182.)
22 \texttt{i3keys} Implementation

\subsection*{22.1 Low-level interface}

The low-level key parser's implementation is based heavily on \texttt{expkv}. Compared to \texttt{keyval} it adds a number of additional "safety" requirements and allows to process the parsed list of key–value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as \texttt{keyval} to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level \texttt{exp3} interfaces.

\newpage

\verbatiminput{\jobname-i3keys.tex}

\texttt{\__keyval_nil, \__keyval_mark, \__keyval_stop, \__keyval_tail} (End definition for \texttt{\__keyval_nil} and others.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro \texttt{#1} will be the active comma and \texttt{#2} will be the active equals sign.

\verbatiminput{\jobname-i3keys.tex}

\texttt{\keyval_parse:NNn} The main function starts the first of two loops. The outer loop splits the key–value list at active commas, the inner loop will do so at other commas. The use of \texttt{\__keyval_mark} here prevents loss of braces from the key argument.

\verbatiminput{\jobname-i3keys.tex}

\texttt{\__keyval_loop_active:NNw} First a fast test for the end of the loop is done, it'll gobble everything up to a \texttt{\__keyval_tail}. The loop ending macro will gobble everything to the last \texttt{\__keyval_mark} in this definition. If the end isn’t reached yet, start the second loop splitting at other commas, and after that one iterate the current loop.

\verbatiminput{\jobname-i3keys.tex}

\texttt{\__keyval_loop_active:NNw} (End definition for \texttt{\__keyval_loop_active:NNw}. This function is documented on page 195.)
These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following \s__keyval_mark that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```
cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
{ ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
{ ##3 #1 \s__keyval_stop \s__keyval_mark ##2 }
```

(End definition for \__keyval_split_other:w and \__keyval_split_active:w)

The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using \__keyval_split_active:w. The \s__keyval_nil prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```
cs_new:Npn \__keyval_loop_other:NNw ##1 ##2 ##3 ,
{\__keyval_if_recursion_tail:w ##3\__keyval_end_loop_other:w \s__keyval_tail\__keyval_split_active:w ##3 \s__keyval_nil\s__keyval_mark\__keyval_split_active_auxi:w#2 \s__keyval_mark\__keyval_clean_up_active:w\s__keyval_stop##1 ##2\__keyval_loop_other:NNw ##1 ##2 \s__keyval_mark}
```

(End definition for \__keyval_loop_other:NNw.)

After \__keyval_split_active:w the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. \#1 will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via \__keyval_misplaced_equal_after_active_error:w. If none was found we forward the key to \__keyval_split_active_auxii:w.

```
cs_new:Npn \__keyval_split_active_auxi:w #1 \s__keyval_stop{\__keyval_split_active_other:w #1 \s__keyval_nil= \s__keyval_mark\__keyval_misplaced_equal_after_active_error:w\s__keyval_stop\s__keyval_mark\__keyval_split_active_auxiii:w}
```

\__keyval_split_active_auxii:w gets the correct key name with a leading \s__keyval_mark as \#1. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to \__keyval_split_active_auxii:w.

```
cs_new:Npn \__keyval_split_active_auxii:w #1 \s__keyval_nil \s__keyval_mark\__keyval_misplaced_equal_after_active_error:w
\s__keyval_mark\__keyval_split_active_auxi:w\s__keyval_stop\s__keyval_stop}
```

(End definition for \__keyval_split_active_auxii:w.)

\__keyval_split_active_auxiii:w
Next we test for a misplaced active equals sign in the value, if none is found \_\_keyval\_split_active_auxiv:w will be called.

\begin{verbatim}
\cs_new:Npn \_\_keyval_split_active_auxiv:w ##1 ##2 \s__keyval_nil
{ \_\_keyval_split_active:w ##2 \s__keyval_nil \_\_keyval_mark \_\_keyval_misplaced_equal_in_split_error:w #2 \s__keyval_mark \_\_keyval_split_active_auxiv:w \s__keyval_stop
{ #1 }
}
\end{verbatim}

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

\begin{verbatim}
\cs_new:Npn \_\_keyval_split_active_auxiv:w
{ \_\_keyval_trim:nN { ##1 } \_\_keyval_pair:nnNN }
\end{verbatim}

(End definition for \_\_keyval_split_active_auxi:w and others.)

\_\_keyval_clean_up_active:w

The following is the branch taken if the key–value pair doesn’t contain an active equals sign. The remainder of that test will be cleaned up by \_\_keyval_clean_up_active:w which will then split at an equals sign of category other.

\begin{verbatim}
\cs_new:Npn \_\_keyval_clean_up_active:w
{ \_\_keyval_split_other:ww ##1 \s__keyval_nil \_\_keyval_mark \_\_keyval_split_other_auxi:w \_\_keyval_clean_up_other:w \s__keyval_stop
{ \_\_keyval_trim:nN { #1 } \_\_keyval_pair:nnNN }
}
\end{verbatim}

(End definition for \_\_keyval_clean_up_active:w.)

\_\_keyval_split_other_auxi:w

\_\_keyval_split_other_auxii:w

\_\_keyval_split_other_auxiii:w

This is executed if the key–value pair doesn’t contain an active equals sign but at least one other. \#1 of \_\_keyval_split_other_auxi:w will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

\begin{verbatim}
\cs_new:Npn \_\_keyval_split_other_auxi:w
{ \_\_keyval_trim:nN { #1 } \_\_keyval_split_other_auxii:w }
\end{verbatim}
We know that the value doesn’t contain misplaced active equals signs but we have to test
for others.

```latex
\cs_new:Npn \__keyval_split_other_auxii:w \#1 \#2 \s__keyval_nil
\{
  \__keyval_split_other:w \#2 \s__keyval_nil
  \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
  \s__keyval_mark \__keyval_split_other_auxiii:w
  \s__keyval_stop
  \{ \#1 \}
\}
\__keyval_split_other_auxiii:w
\__keyval_split_other_auxii:w
\__keyval_pair:nnNN.
\cs_new:Npn \__keyval_clean_up_other:w \#1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \s__keyval_mark
\{
  \__keyval_if_blank:w \#1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
    \s__keyval_mark \__keyval_stop \use:n
    \{ \__keyval_trim:nN \{ \#1 \} \__keyval_key:nNN \}
\}
\__keyval_clean_up_other:w
\__keyval_clean_up_other:w
\__keyval_pair:nnNN.
\cs_new:Npn \__keyval_misplaced_equal_after_active_error:w
\s__keyval_mark \s__keyval_stop \s__keyval_mark \s__keyval_nil \s__keyval_nil \s__keyval_nil \s__keyval_nil \s__keyval_stop \__kernel_msg_expandable_error:nn
\{ kernel \} \{ misplaced-equals-sign \}
\cs_new:Npn \__keyval_misplaced_equal_in_split_error:w
\s__keyval_mark \s__keyval_stop \s__keyval_mark \s__keyval_nil \s__keyval_nil \s__keyval_nil \s__keyval_nil \s__keyval_stop \__kernel_msg_expandable_error:nn
\{ kernel \} \{ misplaced-equals-sign \}
\end{definition}
```

All these two macros do is gobble the remainder of the current other loop execution and
throw an error.
All that’s left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call.

\[\text{\_keyval\_end\_loop\_other:w}\]
\[\text{\_keyval\_end\_loop\_active:w}\]

\[\text{\cs_new:Npn \_keyval\_end\_loop\_other:w}\]
\[\text{\_keyval\_tail}\]
\[\text{\_keyval\_split\_active:w ##1 \_keyval\_nil}\]
\[\text{\_keyval\_mark \_keyval\_split\_active\_auxi:w}\]
\[\text{\_keyval\_mark \_keyval\_clean\_up\_active:w}\]
\[\text{\_keyval\_stop}\]
\[\text{\_keyval\_loop\_other:NNw ##4 \_keyval\_mark}\]
\[\{\}\]
\[\text{\cs_new:Npn \_keyval\_end\_loop\_active:w}\]
\[\text{\_keyval\_tail}\]
\[\text{\_keyval\_loop\_other:NNw ##1 \_keyval\_tail,}\]
\[\text{\_keyval\_loop\_active:NNw ##2 \_keyval\_mark}\]
\[\{\}\]

(End definition for \_keyval\_end\_loop\_other:w and \_keyval\_end\_loop\_active:w.)

The parsing loops are done, so here ends the definition of \_keyval\_tmp:NN, which will finally set up the macros.

\[\text{\char_set_catcode_active:n \{ ', \}\char_set_catcode_active:n \{ '=' \}\_keyval\_tmp:NN , =}\]

\[\text{\group_end:}\]

\[\text{\_keyval\_pair:nnNN}\]
\[\text{\_keyval\_key:nnNN}\]

These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to \keyval\_parse:NNn inside of \exp_not:n with the correct arguments.

\[\text{\cs_new:Npn \_keyval\_pair:nnNN #1 #2 #3 #4}\]
\[\text{\{\}\_keyval\_if\_blank:w \_keyval\_mark \_keyval\_stop \_keyval\_blank:w}\]
\[\text{\_keyval\_mark \_keyval\_nil \_keyval\_stop}\]
\[\text{\exp_not:n \{ #4 \{ #2 \} \{ #1 \}\}}\]
\[\text{\}\}\]
\[\text{\cs_new:Npn \_keyval\_key:nnNN #1 #2 #3}\]
\[\text{\{\}\_keyval\_if\_blank:w \_keyval\_mark \_keyval\_nil \_keyval\_stop \_keyval\_blank:w}\]
\[\text{\_keyval\_mark \_keyval\_stop}\]
\[\text{\exp_not:n \{ #2 \{ #1 \}\}}\]
\[\text{\}\}\]

(End definition for \_keyval\_pair:nnNN and \_keyval\_key:nnNN.)

\[\text{\_keyval\_if\_empty:w}\]
\[\text{\_keyval\_if\_blank:w}\]
\[\text{\_keyval\_if\_recursion\_tail:w}\]

All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading \_keyval\_mark.

\[\text{\cs_new:Npn \_keyval\_if\_empty:w #1 \_keyval\_mark \_keyval\_stop \{\}}\]
\[\text{\cs_new:Npn \_keyval\_if\_blank:w #1 \_keyval\_mark \_keyval\_mark \_keyval\_nil \_keyval\_stop \{\text{\_keyval\_if\_empty:w \_keyval\_mark}\}}\]
\[\text{\cs_new:Npn \_keyval\_if\_recursion\_tail:w \_keyval\_mark \_keyval\_mark \_keyval\_tail \{\}}\]

(End definition for \_keyval\_if\_empty:w, \_keyval\_if\_blank:w, and \_keyval\_if\_recursion\_tail:w.)
These macros will be called if the tests above didn’t gobble them, they execute the branching.

\cs_new:Npn \__keyval_blank_true:w \s__keyval_mark \s__keyval_stop \use:n #1 #2 #3 { }
\cs_new:Npn \__keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop \exp_not:n #1
{
  \__kernel_msg_expandable_error:nn
  { kernel } { blank-key-name }
}

(End definition for \__keyval_blank_true:w and \__keyval_blank_key_error:w.)

Two messages for the low level parsing system.

\__kernel_msg_new:nnn { kernel } { misplaced-equals-sign }
{ Misplaced equals sign in key-value input \msg_line_context: }
\__kernel_msg_new:nnn { kernel } { blank-key-name }
{ Blank key name in key-value input \msg_line_context: }

And an adapted version of \__tl_trim_spaces:nn which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won’t comment on it here. The speed gain by using this instead of \tl_trim_spaces_apply:nN is about 10% of the total time for \keyval_parse:NNn with one key and one key-value pair, so I think it’s worth it.

\group_begin:
\cs_set_protected:Npn \__keyval_tmp:n #1
{
\cs_new:Npn \__keyval_trim:nN ##1
{
  \__keyval_trim_auxi:w
  ##1
  \s__keyval_nil
  \s__keyval_mark #1 {##1}
  \s__keyval_mark \__keyval_trim_auxii:w
  \__keyval_trim_auxiii:w
  \__keyval_trim_auxiv:w
  \s__keyval_stop
}
\cs_new:Npn \__keyval_trim_auxi:w
#1
\s__keyval_nil
\s__keyval_mark #1 { }
\s__keyval_mark \__keyval_trim_auxii:w
\__keyval_trim_auxiii:w
#1 \s__keyval_nil
\__keyval_trim_auxiv:w
\s__keyval_stop
}
\cs_new:Npn \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval_mark \s__keyval_mark #1 #2 \s__keyval_mark #3
{
  ##3
  \__keyval_trim_auxi:w
  \s__keyval_mark #2
  \s__keyval_mark #1{##1}
}
\cs_new:Npn \__keyval_trim_auxiii:w \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval_mark \s__keyval_mark
{
  \__keyval_trim_auxii:w
  ##1
}
\cs_new:Npn \__keyval_trim_auxiv:w \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval NIL #1 #2
{
  ##2
  #1 \s__keyval_nil
}
\group_end:
This is the one macro which differs from the original definition.

\cs_new:Npn \__keyval_trim_auxiv:w \s__keyval_mark ##1 \s__keyval_nil ##2 \s__keyval_stop ##3 { ##3 { ##1 } }

(End definition for \__keyval_trim:nN and others.)

22.2 Constants and variables

\c__keys_code_root_str \str_const:Nn { key~code~>~ }
\c__keys_default_root_str \str_const:Nn { key~default~>~ }
\c__keys_groups_root_str \str_const:Nn { key~groups~>~ }
\c__keys_inherit_root_str \str_const:Nn { key~inherit~>~ }
\c__keys_type_root_str \str_const:Nn { key~type~>~ }
\c__keys_validate_root_str \str_const:Nn { key~validate~>~ }

(End definition for \c__keys_code_root_str and others.)

\c__keys_props_root_str \str_const:Nn { key~prop~>~ }

(End definition for \c__keys_props_root_str.)

\l_keys_choice_int \l_keys_choice_tl \int_new:N \l_keys_choice_int \tl_new:N \l_keys_choice_tl

(End definition for \l_keys_choice_int and \l_keys_choice_tl. These variables are documented on page 189.)

\l__keys_groups_clist \clist_new:N \l__keys_groups_clist

(End definition for \l__keys_groups_clist.)

\l_keys_key_str \l_keys_key_tl \str_new:N \l_keys_key_str \tl_new:N \l_keys_key_tl

(End definition for \l_keys_key_str and \l_keys_key_tl. These variables are documented on page 191.)

\l_keys_module_str \str_new:N \l_keys_module_str
A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.
\bool_new:N \l__keys_no_value_bool
(End definition for \l__keys_no_value_bool.)

Used to track if only “known” keys are being set.
\bool_new:N \l__keys_only_known_bool
(End definition for \l__keys_only_known_bool.)
The “path” of the current key is stored here: this is available to the programmer and so is public. The older version is deprecated but has to be handled manually.
\str_new:N \l_keys_path_str
\tl_new:N \l_keys_path_tl
(End definition for \l_keys_path_str and \l_keys_path_tl. These variables are documented on page 191.)

The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.
\tl_new:N \l__keys_relative_tl
\tl_set:Nn \l__keys_relative_tl { \q__keys_no_value }
(End definition for \l__keys_relative_tl.)
The “property” begin set for a key at definition time is stored here.
\str_new:N \l__keys_property_str
(End definition for \l__keys_property_str.)

Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”).
\bool_new:N \l__keys_selective_bool
\bool_new:N \l__keys_filtered_bool
(End definition for \l__keys_selective_bool and \l__keys_filtered_bool.)
The list of key groups being filtered in or out during selective setting.
\seq_new:N \l__keys_selective_seq
(End definition for \l__keys_selective_seq.)

Used when setting only some keys to store those left over.
\tl_new:N \l__keys_unused_clist
(End definition for \l__keys_unused_clist.)
\l_keys_value_tl The value given for a key: may be empty if no value was given.

\l__keys_tmp_bool Scratch space.
\l__keys_tmpa_tl
\l__keys_tmpb_tl

(End definition for \l_keys_value_tl. This variable is documented on page 191.)

22.2.1 Internal auxiliaries

\s__keys_stop Internal scan marks.

(End definition for \s__keys_stop.)

\q__keys-nil Internal quarks.
\q__keys-no_value

(End definition for \q__keys-nil and \q__keys-no_value.)

\__keys_quark_if-nil_p:n TF Branching quark conditional.
\__keys_quark_if-nil:nTF

(End definition for \__keys_quark_if-nil:nTF.)

\q__keys-recursion_tail Internal recursion quarks.
\q__keys-recursion_stop

(End definition for \q__keys-recursion_tail and \q__keys-recursion_stop.)

\__keys_if-recursion_tail-stop:n Functions to query recursion quarks.

(End definition for \__keys_if-recursion_tail-stop:n.)
22.3 The key defining mechanism

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

\keys_define:nn
\__keys_define:nnn
\__keys_define:onn

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

\__keys_define:n
\__keys_define:nn
\__keys_define_aux:nn

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

\__keys_property_find:n
\__keys_property_find:w

(End definition for \keys_define:nn and \__keys_define:nnn. This function is documented on page 184.)
\_keys\_define\_code:n
\_keys\_define\_code:w

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it was missing the earlier tests would have failed.
22.4 Turning properties into actions

Boolean keys are really just choices, but all done by hand. The second argument here is
the scope: either empty or \texttt{g} for global.

Inverse boolean setting is much the same.

(End definition for \texttt{\_keys_bool_set:Nn} and \texttt{\_keys_bool_set:cn}.)
To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

\begin{verbatim}
\cs_new_protected:Npn \__keys_choice_make:
  { \__keys_choice_make:N \__keys_choice_find:n }
\cs_new_protected:Npn \__keys_multichoice_make:
  { \__keys_choice_make:N \__keys_multichoice_find:n }
\cs_new_protected:Npn \__keys_choice_make:N \l_keys_path_str
  { \__keys_choice_make_aux:N \l_keys_path_str #1 }
\cs_new_protected:Npn \__keys_choice_make_aux:N \l_keys_path_str
  { \__keys_cmd_set:nn { \l_keys_path_str } { \l_keys_path_str }#1 }
\cs_new_protected:Npn \__keys_choice_make_aux:N \l_keys_path_str
  { \__keys_cmd_set:nn { \l_keys_path_str } { unknown } #1 }
\end{verbatim}

(End definition for \__keys_choice_make: and others.)

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

\begin{verbatim}
\cs_new_protected:Npn \__keys_choices_make:nn
  { \__keys_choices_make:Nnn \__keys_choice_make: }
\cs_new_protected:Npn \__keys_multichoices_make:nn
  { \__keys_choices_make:Nnn \__keys_multichoice_make: }
\cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
  { #1 \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#2} }
\int_zero:N \l_keys_choice_int
\clist_map_inline:nn {#2}
\int_incr:N \l_keys_choice_int
\__keys_cmd_set:nx
  \l_keys_path_str / \__keys_trim_spaces:n {#1}
\tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
\end{verbatim}
\int_set:Nn \exp_not:N \l_keys_choice_int
\{ \int_use:N \l_keys_choice_int \}
\exp_not:n (#3)
\}
\}
\}
\)

(End definition for \_keys\_choices\_make:nn, \_keys\_multichoices\_make:nn, and \_keys\_choices\_-make:Nnn.)
\)

\_keys\_cmd\_set:nn\_keys\_cmd\_set:nx\_keys\_cmd\_set:Vn\_keys\_cmd\_set:Vo

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.
\cs\_new\_protected:Npn \_keys\_cmd\_set:nn \#1\#2
\cs\_generate\_variant:Nn \_keys\_cmd\_set:nn \{ nx , Vn , Vo \}

(End definition for \_keys\_cmd\_set:nn.)
\)

\_keys\_cs\_set:NNpn\_keys\_cs\_set:Ncpn

Creating control sequences is a bit more tricky than other cases as we need to pick up the p argument. To make the internals look clearer, the trailing n argument here is just for appearance.
\cs\_new\_protected:Npn \_keys\_cs\_set:NNpn \#1\#2\#3#
\cs\_generate\_variant:Nn \_keys\_cs\_set:NNpn \{ nx \}

(End definition for \_keys\_cs\_set:NNpn.)
\)

\_keys\_default\_set:n

Setting a default value is easy. These are stored using \cs_set:cpx as this avoids any worries about whether a token list exists.
\cs\_new\_protected:Npn \_keys\_default\_set:n \#1
\}

(End definition for \_keys\_default\_set:n.)
\)

\_keys\_groups\_set:n

Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the check-declarations code.
\cs_new_protected:Npn \__keys_groups_set:n #1
\clist_set:Nn \l__keys_groups_clist {#1}
\clist_if_empty:NTF \l__keys_groups_clist
\cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
\tex_undefined:D
}{
\cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
\l__keys_groups_clist
}

(End definition for \__keys_groups_set:n.)

\__keys_inherit:n Inheritance means ignoring anything already said about the key: zap the lot and set up.
\cs_new_protected:Npn \__keys_inherit:n #1
\__keys_undefine:
\cs_set_nopar:cpn { \c__keys_inherit_root_str \l_keys_path_str } {#1}

(End definition for \__keys_inherit:n.)

\__keys_initialise:n A set up for initialisation: just run the code if it exists.
\cs_new_protected:Npn \__keys_initialise:n #1
\cs_if_exist:cTF
{ \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
{ \__keys_execute_inherit: }
{ \str_clear:N \l__keys_inherit_str
\cs_if_exist_use:cT { \c__keys_code_root_str \l_keys_path_str } { {#1} } }

(End definition for \__keys_initialise:n.)

\__keys_meta_make:n To create a meta-key, simply set up to pass data through.
\__keys_meta_make:nn
\cs_new_protected:Npn \__keys_meta_make:n #1
\{ \__keys_cmd_set:Vo \l_keys_path_str
\exp_after:wN \keys_set:nn
\exp_after:wN { \l__keys_module_str } {#1}
\}
\cs_new_protected:Npn \__keys_meta_make:nn #1#2
{ \__keys_cmd_set:Vn \l_keys_path_str { \keys_set:nn {#1} {#2} } }

(End definition for \__keys_meta_make:n and \__keys_meta_make:nn.)
Much the same as other variables, but needs a dedicated auxiliary.

```latex
\cs_new_protected:Npn \__keys_prop_put:Nn #1 #2
\exp_after:wN \__keys_find_key_module:NNw
\exp_after:wN \l__keys_tmpa_tl
\exp_after:wN \l__keys_tmpb_tl
\l_keys_path_str / \s__keys_stop
\__keys_cmd_set:nx { \l_keys_path_str }
\exp_not:c { prop_ #2 put:Nnn }
\exp_not:N #1
\exp_not:n { {##1} }
```

(End definition for \__keys_prop_put:Nn.)

Undefining a key has to be done without \cs_undefine:c as that function acts globally.

```latex
\cs_new_protected:Npn \__keys_undefine:
\clist_map_inline:nn
\str_case:nnF {#2} {
\cs_set_eq:cc
\cs_if_eq:ccT
\\exp_not:c { \c__keys_validate_root_str \l_keys_path_str }
\\l__keys_undefine:D
}
```

(End definition for \__keys_undefine:)
Setting a variable takes the type and scope separately so that it is easy to make a new
variable if needed.
\begin{Verbatim}
\cs_new_protected:Npn \_keys_variable_set:NnnN \#1#2#3#4
\begin{Verbatim}
\cs_new_protected:Npn \_keys_variable_set_required:NnnN \#1#2#3#4
\cs_generate_variant:Nn \_keys_variable_set:NnnN { c } \end{Verbatim}
\cs_generate_variant:Nn \_keys_variable_set_required:NnnN { c } \end{Verbatim}

(End definition for \_keys_variable_set:NnnN and \_keys_variable_set_required:NnnN.)
22.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

One function for this.

(End definition for \bool_set:N and \bool_gset:N. These functions are documented on page 185.)

One function for this.

(End definition for \bool_set_inverse:N and \bool_gset_inverse:N. These functions are documented on page 185.)

Making a choice is handled internally, as it is also needed by \generate_choices:n.

(End definition for \choice:. This function is documented on page 185.)

For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

(End definition for \choices:nn. This function is documented on page 185.)
Creating code is simply a case of passing through to the underlying set function.

```
\cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
{ \c__keys_cmd_set:nn { \l_keys_path_str } {#1} }
```

(End definition for .code:n. This function is documented on page 185.)

```
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
{ \__keys_variable_set:NnnN #1 { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
{ \__keys_variable_set:cnnN {#1} { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
{ \__keys_variable_set:NnnN #1 { clist } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
{ \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for .clist_set:N and .clist_gset:N. These functions are documented on page 185.)

```
\cs_set:Np \cs_set:cp \cs_set_protected:Np \cs_set_protected:cp
\cs_gset:Np \cs_gset:cp \cs_gset_protected:Np \cs_gset_protected:cp
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
{ \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
{ \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
{ \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1
{ \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
{ \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
{ \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
{ \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
{ \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }
```

(End definition for .cs_set:Np and others. These functions are documented on page 185.)

```
\default:n \default:V \default:o \default:x
\cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
{ \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
{ \exp_args:NV \__keys_default_set:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
{ \exp_args:No \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
{ \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for .default:n. This function is documented on page 186.)

```
\dim_set:N \dim_set:c \dim_gset:N \dim_gset:c
\cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
{ \__keys_variable_set_required:NnnN #1 { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
{ \__keys_variable_set_required:cnnN {#1} { dim } { } n }
```

(End definition for .dim_set:N. This function is documented on page 186.)
\cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
{ \__keys_variable_set_required:NnnN #1 { dim } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
{ \__keys_variable_set_required:nnnN {#1} { dim } { g } n }

(End definition for \texttt{.dim_set:N} and \texttt{.dim_gset:N}. These functions are documented on page 186.)

\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1
{ \__keys_variable_set_required:NnnN #1 { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1
{ \__keys_variable_set_required:cnnN {#1} { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1
{ \__keys_variable_set_required:NnnN #1 { fp } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1
{ \__keys_variable_set_required:cnnN {#1} { fp } { g } n }

(End definition for \texttt{.fp_set:N} and \texttt{.fp_gset:N}. These functions are documented on page 186.)

\cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1
{ \__keys_groups_set:n {#1} }

(End definition for \texttt{.groups:n}. This function is documented on page 186.)

\cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1
{ \__keys_inherit:n {#1} }

(End definition for \texttt{.inherit:n}. This function is documented on page 186.)

\cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1
{ \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1
{ \exp_args:NV \__keys_initialise:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1
{ \exp_args:No \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1
{ \exp_args:Nx \__keys_initialise:n {#1} }

(End definition for \texttt{.initial:n}. This function is documented on page 186.)

\cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1
{ \__keys_variable_set_required:NnnN #1 { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1
{ \__keys_variable_set_required:nnnN {#1} { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1
{ \__keys_variable_set_required:NnnN {#1} { int } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1
{ \__keys_variable_set_required:nnnN {#1} { int } { g } n }

(End definition for \texttt{.int_set:N} and \texttt{.int_gset:N}. These functions are documented on page 187.)
Making a meta is handled internally.

Meta with path: potentially lots of variants, but for the moment no so many defined.

The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

Setting a variable is very easy: just pass the data along.

Setting a variable is very easy: just pass the data along.
Setting a variable is very easy: just pass the data along.

```latex
\cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
{ \__keys_variable_set_required:NnnN #1 { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
{ \__keys_variable_set_required:cnnN {#1} { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
{ \__keys_variable_set_required:NnnN #1 { skip } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
{ \__keys_variable_set_required:cnnN {#1} { skip } { g } n }
```

(End definition for \texttt{.skip\_set:N} and \texttt{.skip\_gset:N}. These functions are documented on page 188.)

Setting a variable is very easy: just pass the data along.

```latex
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
{ \__keys_variable_set:NnnN #1 { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
{ \__keys_variable_set:cnnN {#1} { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
{ \__keys_variable_set:NnnN #1 { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
{ \__keys_variable_set:cnnN {#1} { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1
{ \__keys_variable_set:NnnN #1 { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
{ \__keys_variable_set:cnnN {#1} { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
{ \__keys_variable_set:NnnN #1 { tl } { g } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
{ \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for \texttt{.tl\_set:N} and others. These functions are documented on page 188.)

\texttt{.undefine:} Another simple wrapper.

```latex
\cs_new_protected:Npn \keys_set:nn #1#2
{ \use:x { \__keys_set:nn { c__keys_props_root_str .undefine: } {#1} {#2} } }
```

(End definition for \texttt{.undefine:}. This function is documented on page 188.)

\texttt{.value\_forbidden:n} and \texttt{.value\_required:n} These are very similar, so both call the same function.

```latex
\cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
{ \__keys_value_requirement:nn { forbidden } {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
{ \__keys_value_requirement:nn { required } {#1} }
```

(End definition for \texttt{.value\_forbidden:n} and \texttt{.value\_required:n}. These functions are documented on page 188.)

22.6 Setting keys

```
\keys_set:nn
\keys_set:nV
\keys_set:nv
\keys_set:no
\keys_set:xn
\keys_set:xn
```

A simple wrapper allowing for nesting.
\bool_set_false:N \exp_not:N \l__keys_only_known_bool
\bool_set_false:N \exp_not:N \l__keys_filtered_bool
\bool_set_false:N \exp_not:N \l__keys_selective_bool
\tl_set:Nn \exp_not:N \l__keys_relative_tl
\q__keys_no_value
\__keys_set:nn \exp_not:n { {#1} {#2} }
\bool_if:NT \l__keys_only_known_bool
\bool_set_true:N \exp_not:N \l__keys_only_known_bool
\bool_if:NT \l__keys_filtered_bool
\bool_set_true:N \exp_not:N \l__keys_filtered_bool
\bool_if:NT \l__keys_selective_bool
\bool_set_true:N \exp_not:N \l__keys_selective_bool
\tl_set:Nn \exp_not:N \l__keys_relative_tl
\exp_not:o \l__keys_relative_tl
}

(End definition for \keys_set:nn, \__keys_set:nn, and \__keys_set:nnn. This function is documented on page 191.)

\cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
\cs_new_protected:Npn \__keys_set:nnn { \__keys_trim_spaces:n {#2} }
\keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
\str_set:Nn \l__keys_module_str {#1}

\keys_set_known:nnN \keys_set_known:nVN \keys_set_known:nvN \keys_set_known:noN \keys_set_known:nnnN \keys_set_known:nVnN \keys_set_known:nvnN \keys_set_known:nonN \__keys_set_known:nnnnN
\keys_set_known:nn \keys_set_known:nV \keys_set_known:nv \keys_set_known:no \keys_set_known:nn

Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of \l__keys_unused_clist is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a \tl operation to set the clist here!
The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\___keys_set_known:nnn` also apply here. We have a bit more shuffling to do to keep everything nestable.

```latex
\keys_set_filter:nnn
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
\keys_set_filter:nnnnN
\keys_set_filter:nnVnN
\keys_set_filter:nnvnN
\keys_set_filter:nnonN
\___keys_set_filter:nnnnn
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
\keys_set_groups:nnnN
\___keys_set_selective:nnn
\___keys_set_selective:nnnn
```

(End definition for `\keys_set_known:nnn` and others. These functions are documented on page 192.)
\bool_set_true:N \exp_not:N \l__keys_filtered_bool
\bool_set_true:N \exp_not:N \l__keys_selective_bool
\tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
\__keys_set_selective:nnn \exp_not:n { (#2) (#3) (#4) }
\bool_if:NT \l__keys_only_known_bool
{ \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
\bool_if:NF \l__keys_filtered_bool
{ \bool_set_false:N \exp_not:N \l__keys_filtered_bool }
\bool_if:NF \l__keys_selective_bool
{ \bool_set_false:N \exp_not:N \l__keys_selective_bool }
\tl_set:Nn \exp_not:N \l__keys_relative_tl
{ \exp_not:o \l__keys_relative_tl }
}
\cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
{
\use:x
{
\bool_set_false:N \exp_not:N \l__keys_only_known_bool
\bool_set_false:N \exp_not:N \l__keys_filtered_bool
\bool_set_true:N \exp_not:N \l__keys_selective_bool
\tl_set:Nn \exp_not:N \l__keys_relative_tl
{ \exp_not:N \q__keys_no_value }
\__keys_set_selective:nnn \exp_not:n { {#1} {#2} {#3} }
\bool_if:NT \l__keys_only_known_bool
{ \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
\bool_if:NF \l__keys_filtered_bool
{ \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
\bool_if:NF \l__keys_selective_bool
{ \bool_set_false:N \exp_not:N \l__keys_selective_bool }
\tl_set:Nn \exp_not:N \l__keys_relative_tl
{ \exp_not:o \l__keys_relative_tl }
}
\cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
\cs_new_protected:Npn \__keys_set_keyval:n #1
{
\bool_set_true:N \l__keys_no_value_bool
\__keys_set_keyval:onn \l__keys_module_str {#1} { }
}
\__keys_set_keyval:n
\__keys_set_keyval:nn
\__keys_set_keyval:nnn
\__keys_set_keyval:onn
\__keys_find_key_module:NNw
\__keys_set_selective:

\end{verbatim}

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

\cs_new_protected:Npm \__keys_set_keyval:n #1
{
\bool_set_true:N \l__keys_no_value_bool
\__keys_set_keyval:onn \l__keys_module_str {#1} { }
}

(End definition for \verb|\keys_set_filter:nnn| and others. These functions are documented on page 193.)
The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.
In the case where selective setting requires a comparison of the list of groups which apply

to a key with the list of those which have been set active. That requires two mappings,
and again a different outcome depending on whether opt-in or opt-out is set.

\cs_new_protected:Npn \__keys_check_groups:
\bool_set_false:N \l__keys_tmp_bool
\seq_map_inline:Nn \l__keys_selective_seq
{\clist_map_inline:Nn \l__keys_groups_clist
{\str_if_eq:nnT {##1} {####1}
\bool_set_true:N \l__keys_tmp_bool
\clist_map_break:n {\seq_map_break:}
}
\bool_if:NTF \l__keys_tmp_bool
{\bool_if:NTF \l__keys_filtered_bool
{\__keys_store_unused:}
{\__keys_execute:}
}
{\bool_if:NTF \l__keys_filtered_bool
{\__keys_execute:}
{\__keys_store_unused:}
}
}(End definition for \__keys_set_keyval:n and others.)

If a value is given, return it as #1, otherwise send a default if available.
\cs_new_protected:Npn \__keys_value_or_default:n #1
\bool_if:NTF \l__keys_no_value_bool
{\bool_if:NTF \l__keys_filtered_bool
{\__keys_store_unused:}
{\__keys_execute:}
}
{\bool_if:NTF \l__keys_filtered Bool
{\__keys_execute:}
{\__keys_store_unused:}
}

(End definition for \__keys_set_keyval:n and others.)
Actually executing a key is done in two parts. First, look for the key itself, then look for the unknown key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

To deal with the case where there is no hit, we leave \__keys_execute_unknown: in the input stream and clean it up using the break function: that avoids needing a boolean.
When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can’t happen earlier as we need to store \q__keys_no_value. Then, use a standard delimited approach to fish out the partial path.
Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That always exists, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

\cs_new:Npn \__keys_choice_find:n #1
\__keys_choice_find:nn
\__keys_multichoice_find:n

(End definition for \__keys_execute: and others.)
\cs_new:Npn \__keys_choice_find:nn #1 #2
{
\cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
{ \use:c { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} } {#2} }
{ \use:c { \c__keys_code_root_str #1 / unknown } {#2} }
}

\cs_new:Npn \__keys_multichoice_find:n #1
{ \clist_map_function:nN {#1} \__keys_choice_find:n }

(End definition for \__keys_choice_find:n, \__keys_choice_find:nn, and \__keys_multichoice_find:n.)

22.7 Utilities

\__keys_parent:n
\__keys_parent:o
\__keys_parent:w

Used to strip off the ending part of the key path after the last /.

\cs_new:Npn \__keys_parent:n #1
{ \__keys_parent:w #1 / / \s__keys_stop { } }
\cs_generate_variant:Nn \__keys_parent:n { o }
\cs_new:Npn \__keys_parent:w #1 / #2 / #3 \s__keys_stop #4
{\tl_if_blank:nTF {#2}
{ \tl_if_blank:nF {#4}
{ \use_none:n #4 }
}
{ \__keys_parent:w #2 / #3 \s__keys_stop { #4 / #1 }
}
}

(End definition for \__keys_parent:n and \__keys_parent:w.)

\__keys_trim_spaces:n
\__keys_trim_spaces_auxi:w
\__keys_trim_spaces_auxii:w
\__keys_trim_spaces_auxiii:w

Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part.

\cs_new:Npn \__keys_trim_spaces:n #1
{ \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n {#1} / \q__keys_nil \s__keys_stop }
\cs_new:Npn \__keys_trim_spaces_auxi:w #1 / #2 \s__keys_stop
{ \__keys_quark_if_nil:nTF {#2}
{ \tl_trim_spaces:n {#1} }
{ \__keys_trim_spaces_auxii:w #1 / #2 }
}
\cs_new:Npn \__keys_trim_spaces_auxii:w #1 / #2 / \q__keys_nil
A utility for others to see if a key exists.

\keys_if_exist_p:nn
\keys_if_exist:nnTF

Just an alternative view on \keys_if_exist:nnTF.
\keys_if_choice_exist_p:nnn
\keys_if_choice_exist:nnnTF

To show a key, show its code using a message.
\keys_show:nn
\keys_log:nn
__keys_show:Nnn

(End definition for `\keys_show`, `\keys_log`, and `\_keys_show`. These functions are documented on page 194.)

### 22.8 Messages

For when there is a need to complain.

```latex
\_\_kernel_msg_new:nnn { kernel } { bad-relative-key-path }
  { The-key-\#1\'-is-not-inside-the-\#2\'-path. }
\_\_kernel_msg_new:nnn { kernel } { boolean-values-only }
  { Key-\#1\'-accepts-boolean-values-only. }
\_\_kernel_msg_new:nnn { kernel } { key-choice-unknown }
  { Key-\#1\'-accepts-only-a-fixed-set-of-choices. }
  { The-key-\#1\'-only-accepts-predefined-values,-}
    and-\#2\'-is-not-one-of-these. }
\_\_kernel_msg_new:nnn { kernel } { key-unknown }
  { The-key-\#1\'-is-unknown-and-is-being-ignored. }
  { The-module-\#2\'-does-not-have-a-key-called-\#1.\}
    Check-that-you-have-spelled-the-key-name-correctly. }
\_\_kernel_msg_new:nnn { kernel } { nested-choice-key }
  { Attempt-to-define-\#1\'-as-a-nested-choice-key. }
  { The-key-\#1\'-cannot-be-defined-as-a-choice-as-the-parent-key-\#2\'-is-}
    itself-a-choice. }
\_\_kernel_msg_new:nnn { kernel } { value-forbidden }
  { The-key-\#1\'-does-not-take-a-value. }
  { The-key-\#1\'-should-be-given-without-a-value.\}
    The-value-\#2\'-was-present:-the-key-will-be-ignored. }
\_\_kernel_msg_new:nnn { kernel } { value-required }
  { The-key-\#1\'-requires-a-value. }
  { The-key-\#1\'-must-have-a-value.\}
    No-value-was-present:-the-key-will-be-ignored. }
\_\_kernel_msg_new:nnn { kernel } { show-key }
  { The-key-\#1-
    \tl_if_empty:nTF {\#2}
      { is-undefined. }
      { has-the-properties: \#2 . }
  }
```
\section*{23 \texttt{l3intarray} implementation}

23.1 Allocating arrays

\begin{verbatim}
\newcommand*{\_intarray_entry:w}{\tex_fontdimen:D}
\newcommand*{\_intarray_count:w}{\tex_hyphenchar:D}
\end{verbatim}

(End definition for \_intarray_entry:w and \_intarray_count:w.)

\begin{verbatim}
\newcommand*{\l_intarray_loop_int}{\int_new:N}{\l_intarray_loop_int}
\end{verbatim}

A loop index.

\begin{verbatim}
\newcommand*{\c_intarray_sp_dim}{1\sp}
\end{verbatim}

(End definition for \c_intarray_sp_dim.)

\begin{verbatim}
\newcommand*{\g_intarray_font_int}{\int_new:N}{\g_intarray_font_int}
\end{verbatim}

Used to assign one font per array.

\begin{verbatim}
\cs_new_protected:Npn \_intarray_new:N #1
\end{verbatim}

Declare #1 to be a font (arbitrarily \texttt{cmr10} at a never-used size). Store the array’s size as the \texttt{hyphenchar} of that font and make sure enough \texttt{fontdimen} are allocated, by setting the last one. Then clear any \texttt{fontdimen} that \texttt{cmr10} starts with. It seems \texttt{LuaT\textsc{ex}}’s \texttt{cmr10} has an extra \texttt{fontdimen} parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every \texttt{intarray} must be global; it’s enough to run this check in \texttt{intarray_new:Nn}.

\begin{verbatim}
\cs_new_protected:Npn \intarray_new:Nn \intarray_new:cn \_intarray_new:N
\end{verbatim}

\begin{verbatim}
\intarray_new:Nn \intarray_new:cn \_intarray_new:N
\end{verbatim}

\begin{verbatim}
\intarray_new:Nn \intarray_new:cn \_intarray_new:N
\end{verbatim}

\begin{verbatim}
\intarray_new:Nn \intarray_new:cn \_intarray_new:N
\end{verbatim}
\_intarray\_count:N
Size of an array.
\_intarray\_count:c
(End definition for \intarray\_count:N. This function is documented on page 196.)

23.2 Array items
Used when an item to be stored is larger than \c\_max\_dim in absolute value; it is replaced by ±\c\_max\_dim.
\__intarray\_signed\_max\_dim:n
The functions \intarray\_gset:Nnn and \intarray\_item:Nn share bounds checking. The T branch is used if #3 is within bounds of the array #2.
\__intarray\_bounds:NNnTF
\__intarray\_bounds\_error:NNn
(End definition for \__intarray\_signed\_max\_dim:n.)

\intarray\_gset:Nnn
\intarray\_gset:cnn
\__kernel\_intarray\_gset:Nnn
\__intarray\_gset:Nnn
\__intarray\_gset\_overflow:Nnn
Set the appropriate \fontdimen. The \__kernel\_intarray\_gset:Nnn function does not use \int\_eval:n, namely its arguments must be suitable for \int\_value:w. The user version checks the position and value are within bounds.
\cs\_new\_protected:Npm \__kernel\_intarray\_gset:Nnn #1#2#3

722
\intarray_gzero:N Set the appropriate \fontdimen to zero. No bound checking needed. The \prg_replicate:nn possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an \int_step_inline:nn loop.
Get the appropriate \fontdimen and perform bound checks. The \__kernel__intarray_item:Nn function omits bound checks and omits \int_eval:n; namely its argument must be a \TeX{} integer suitable for \int_value:w.

\cs_new:Npn \__kernel_intarray_item:Nn #1#2
\{ \exp_after:wN \__intarray_item:Nw \exp_after:wN #1 \int_value:w \int_eval:n {#2} ; \}
\cs_generate_variant:Nn \intarray_item:Nn { c }
\cs_new:Npn \__intarray_item:Nw #1#2 ;
\{ \__intarray_bounds:NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2} \{ \__kernel_intarray_item:Nn #1 {#2} \} \{ 0 \} \}
\cs_generate_variant:Nn \intarray_item:Nn { c }

Importantly, \intarray_item:Nn only evaluates its argument once.

\cs_new:Npn \intarray_rand_item:N #1
\{ \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } \}
\cs_generate_variant:Nn \intarray_rand_item:N { c }

Similar to \intarray_new:Nn (which we don’t use because when debugging is enabled that function checks the variable name starts with \texttt{g}.). We make use of the fact that \TeX{} allows allocation of successive \fontdimen as long as no other font has been declared: no need to count the comma list items first. We need the code in \intarray_gset:Nnn that checks the item value is not too big, namely \__intarray_gset_overflow_test:nw, but not the code that checks bounds. At the end, set the size of the intarray.

\cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
\{ \__intarray_new:N #1 \int_zero:N \l__intarray_loop_int
\clist_map_inline:nn {#2} \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 \}
\__intarray_count:w #1 \l__intarray_loop_int \}
\cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
\cs_new_protected:Npn \__intarray.const_from_clist:Nn #1#2
\{ \exp_args:Nf \__intarray.const_from_clist:nN { \int_eval:n {##1} } \}
\__intarray_count:w #1 \l__intarray_loop_int
\}
\cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
\cs_new_protected:Npn \__intarray.const_from_clist:cn #1#2
\{ \exp_args:Nf \__intarray.const_from_clist:cn { \int_eval:n {##1} } \}
\__intarray_count:w #1 \l__intarray_loop_int
\}
\cs_generate_variant:Nn \intarray_const_from_clist:cn { c }
\intarray_to_clist:N \intarray_to_clist:c \intarray_to_clist:Nn \intarray_to_clist:w

Loop through the array, putting a comma before each item. Remove the leading comma with f-expansion. We also use the auxiliary in \intarray_show:N with argument comma, space.

\cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
\cs_generate_variant:Nn \intarray_to_clist:N { c }
\cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
{ \if_int_compare:w #1 > \__intarray_count:w #2 \prg_break:n
\fi:
#3 \__kernel_intarray_item:Nn #2 {#1}
\exp_after:wN \__intarray_to_clist:w
\prg_break_point: \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3} }

\intarray_show:N \intarray_show:c \intarray_log:N \intarray_log:c

Convert the list to a comma list (with spaces after each comma)

\cs_new_protected:Npn \intarray_show:N { \__intarray_show:NN \msg_show:nnxxxx }
\cs_generate_variant:Nn \intarray_show:N { c }
\cs_new_protected:Npn \intarray_log:N { \__intarray_show:NN \msg_log:nnxxxx }
\cs_generate_variant:Nn \intarray_log:N { c }
\cs_new_protected:Npn \__kernel chk_defined:NT #2
{ \LaTeX/kernel } { \msg_intarray }{ \token_to_str:N #2 }
{ \intarray_count:N #2 }
{ >- \__intarray_to_clist:Nn #2 { , - } }
}

(End definition for \intarray_to_clist:N, \intarray_to_clist:Nn, and \intarray_to_clist:w. This function is documented on page 262.)

(End definition for \intarray_show:N and \intarray_log:N. These functions are documented on page 197.)
23.4 Random arrays

We only perform the bounds checks once. This is done by two \texttt{\_\_intarray_gset\_\_overflow_test:nw}, with an appropriate empty argument to avoid a spurious “at position #1” part in the error message. Then calculate the number of choices: this is at most $2^{31}-1 = 2^{31}-1$, which just barely does not overflow. For small ranges use \texttt{\_\_kernel_randint:nn} (making sure to subtract 1 before adding the random number to the \langle min \rangle), to avoid overflow when \langle min \rangle or \langle max \rangle are $\pm \cmaxint$, otherwise \texttt{\_\_kernel_randint:nn}. Finally, if there are no random numbers do not define any of the auxiliaries.

\begin{verbatim}
\cs_new_protected:Npn \intarray_gset_rand:Nn #1
\intarray_gset_rand:cn \intarray_gset_rand:Nnn \intarray_gset_rand:cnn
\__intarray_gset_rand:Nnn \__intarray_gset_rand:Nff
\__intarray_gset_rand_auxi:Nnnn \__intarray_gset_rand_auxii:Nnnn
\__intarray_gset_all_same:Nn
\end{verbatim}
\begin{verbatim}
\exp_stop_f:
\int_eval:n { \_kernel_randint:n {#2} - 1 + #3 }
\}
\cs_new_protected:Npn \_\intarray_gset_all_same:Nn #1#2
  \int_zero:N \l__intarray_loop_int
  \prg_replicate:nn { \intarray_count:N #1 }
  { \int_incr:N \l__intarray_loop_int
    \_\intarray_gset:Nnn #1 \l__intarray_loop_int {#2} }
\}
\cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
  \__kernel_msg_error:nnn { kernel } { fp-no-random } { \intarray_gset_rand:Nnn #1 {#2} {#3} }
\cs_generate_variant:Nn \intarray_gset_rand:Nnn { c }
\end{verbatim}

(End definition for \intarray_gset_rand:Nn and others. These functions are documented on page 262.)

24 \texttt{l3fp} implementation

Nothing to see here: everything is in the subfiles!

25 \texttt{l3fp-aux} implementation

Largely for performance reasons, we need to directly access primitives rather than use \int_eval:n. This happens a lot, so we use private names. The same is true for \romannumeral, although it is used much less widely.

(End definition for \_\fp_int_eval:w, \_\fp_int_eval_end:, and \_\fp_int_to_roman:w.)
25.2 Internal representation

Internally, a floating point number \( X \) is a token list containing

\[
\text{s__fp \_fp_chk:w \text{\langle case\rangle \langle sign\rangle \langle body\rangle ;}}
\]

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to \( f \)-expansion. They must leave a recognizable mark after \( f \)-expansion, to prevent the floating point number from being re-parsed. Thus, \text{s__fp} is simply another name for \( \text{relax} \).

When used directly without an accessor function, floating points should produce an error: this is the role of \text{\_fp_chk:w}. We could make floating point variables be protected to prevent them from expanding under \( x \)-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their \langle case \rangle, which is a single digit:

0 zeros: +0 and −0,
1 “normal” numbers (positive and negative),
2 infinities: +inf and −inf,
3 quiet and signalling \( \text{nan} \).

The \langle sign \rangle is 0 (positive) or 2 (negative), except in the case of \( \text{nan} \), which have \langle sign \rangle = 1. This ensures that changing the \langle sign \rangle digit to \( 2 - \langle sign \rangle \) is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

\[
\text{s__fp \_fp_chk:w \langle case\rangle \langle sign\rangle \text{s__fp... ;}}
\]

where \text{s__fp...} is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers (\langle case \rangle = 1) have the form

\[
\text{s__fp \_fp_chk:w 1 \langle sign\rangle \{\langle exponent\\} \{\langle X_1\rangle \} \{\langle X_2\rangle \} \{\langle X_3\rangle \} \{\langle X_4\rangle \} ;}
\]

Here, the \langle exponent \rangle is an integer, between −10000 and 10000. The body consists in four blocks of exactly 4 digits, 0000 ≤ \langle X_i \rangle ≤ 9999, and the floating point is

\[
(-1)^{\langle sign\rangle/2} X_1 X_2 X_3 X_4 \cdot 10^{\langle exponent\rangle - 16}
\]

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the \langle exponent \rangle is minimal, in other words, 1000 ≤ \langle X_1 \rangle ≤ 9999.

Calculations are done in base 10000, \textit{i.e.} one myriad.
Table 3: Internal representation of floating point numbers.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 \s__fp_... ;</td>
<td>Positive zero.</td>
</tr>
<tr>
<td>0 2 \s__fp_... ;</td>
<td>Negative zero.</td>
</tr>
<tr>
<td>1 0 {\langle exponent\rangle } {\langle X_1\rangle } {\langle X_2\rangle } {\langle X_3\rangle } {\langle X_4\rangle } ;</td>
<td>Positive floating point.</td>
</tr>
<tr>
<td>1 2 {\langle exponent\rangle } {\langle X_1\rangle } {\langle X_2\rangle } {\langle X_3\rangle } {\langle X_4\rangle } ;</td>
<td>Negative floating point.</td>
</tr>
<tr>
<td>2 0 \s__fp_... ;</td>
<td>Positive infinity.</td>
</tr>
<tr>
<td>2 2 \s__fp_... ;</td>
<td>Negative infinity.</td>
</tr>
<tr>
<td>3 1 \s__fp_... ;</td>
<td>Quiet nan.</td>
</tr>
<tr>
<td>3 1 \s__fp_... ;</td>
<td>Signalling nan.</td>
</tr>
</tbody>
</table>

25.3 Using arguments and semicolons

\_\_fp\_use\_none\_stop\_f:n
This function removes an argument (typically a digit) and replaces it by \exp\_stop\_f:, a marker which stops f-type expansion.
\cs_new:Npn \_\_fp\_use\_none\_stop\_f:n #1 { \exp\_stop\_f: }
(End definition for \_\_fp\_use\_none\_stop\_f:n.)

\_\_fp\_use\_s:n
\_\_fp\_use\_s:nn Those functions place a semicolon after one or two arguments (typically digits).
\cs_new:Npn \_\_fp\_use\_s:n #1 { #1; }
\cs_new:Npn \_\_fp\_use\_s:nn #1#2 { #1; #2; }
(End definition for \_\_fp\_use\_s:n and \_\_fp\_use\_s:nn.)

\_\_fp\_use\_none\_until\_s:w
\_\_fp\_use\_i\_until\_s:nnw
\_\_fp\_use\_ii\_until\_s:nnwnw Those functions select specific arguments among a set of arguments delimited by a semicolon.
\cs_new:Npn \_\_fp\_use\_none\_until\_s:w #1; { } \cs_new:Npn \_\_fp\_use\_i\_until\_s:nnw #1#2; { #1; }
\cs_new:Npn \_\_fp\_use\_ii\_until\_s:nnwnw #1#2#3; { #1; #2; #3; }
(End definition for \_\_fp\_use\_none\_until\_s:w, \_\_fp\_use\_i\_until\_s:nnw, and \_\_fp\_use\_ii\_until\_s:nnwnw.)

\_\_fp\_reverse\_args:Nww
Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.
\cs_new:Npn \_\_fp\_reverse\_args:Nww #1 #2; { #1 #3; #2; }
(End definition for \_\_fp\_reverse\_args:Nww.)

\_\_fp\_rrot:www
Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.
\cs_new:Npn \_\_fp\_rrot:www #1; #2; #3; { #2; #3; #1; }
(End definition for \_\_fp\_rrot:www.)

\_\_fp\_use\_i:ww
\_\_fp\_use\_i:www Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.
\cs_new:Npn \_\_fp\_use\_i:ww #1; #2; { #1; }
\cs_new:Npn \_\_fp\_use\_i:www #1; #2; #3; { #1; }
(End definition for \_\_fp\_use\_i:ww and \_\_fp\_use\_i:www.)
25.4 Constants, and structure of floating points

\_\_fp\_misused:n
This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an \texttt{fp} variable is left in the input stream and its contents reach \TeX's stomach.

\cs_new_protected:Npn \_\_fp\_misused:n #1
{ \_\_kernel_msg_error:nnx { \kernel } { misused-fp } { \fp\_to\_tl:n \{#1\} } }

(End definition for \_\_fp\_misused:n.)

\_\_fp\_chk:w
Floating points numbers all start with \_\_fp \_\_fp\_chk:w, where \_\_fp is equal to the \TeX primitive \relax, and \_\_fp\_chk:w is protected. The rest of the floating point number is made of characters (or \relax). This ensures that nothing expands under \texttt{f}-expansion, nor under \texttt{x}-expansion. However, when typeset, \_\_fp does nothing, and \_\_fp\_chk:w is expanded. We define \_\_fp\_chk:w to produce an error.

\scan_new:N \_\_fp
\cs_new_protected:Npn \_\_fp\_chk:w #1 ;
{ \_\_fp\_misused:n \{ \_\_fp \_\_fp\_chk:w #1 ; \} }

(End definition for \_\_fp and \_\_fp\_chk:w.)

\_\_fp\_expr\_mark \_\_fp\_expr\_stop
Aliases of \texttt{tex}\_\relax:D, used to terminate expressions.

\scan_new:N \_\_fp\_expr\_mark
\scan_new:N \_\_fp\_expr\_stop

(End definition for \_\_fp\_expr\_mark and \_\_fp\_expr\_stop.)

\_\_fp\_mark \_\_fp\_stop
Generic scan marks used throughout the module.

\scan_new:N \_\_fp\_mark
\scan_new:N \_\_fp\_stop

(End definition for \_\_fp\_mark and \_\_fp\_stop.)

\_fp\_use\_i\_delimit\_by\_s\_stop:nw
Functions to gobble up to a scan mark.

\cs_new_protected:N \_fp\_use\_i\_delimit\_by\_s\_stop:nw \_\_fp\_stop #1 \_\_fp\_stop \{ #1 \}

(End definition for \_fp\_use\_i\_delimit\_by\_s\_stop:nw.)

\_\_fp\_invalid \_\_fp\_underflow \_\_fp\_overflow \_\_fp\_division \_\_fp\_exact
A couple of scan marks used to indicate where special floating point numbers come from.

\scan_new:N \_\_fp\_invalid
\scan_new:N \_\_fp\_underflow
\scan_new:N \_\_fp\_overflow
\scan_new:N \_\_fp\_division
\scan_new:N \_\_fp\_exact

(End definition for \_\_fp\_invalid and others.)

\_c\_zero\_fp \_c\_minus\_zero\_fp \_c\_inf\_fp \_c\_minus\_inf\_fp \_c\_nan\_fp
The special floating points. We define the floating points here as “exact”.

\tl_const:Nn \_c\_zero\_fp \{ \_\_fp \_\_fp\_chk:w 0 0 \_\_fp\_exact ; \}
\tl_const:Nn \_c\_minus\_zero\_fp \{ \_\_fp \_\_fp\_chk:w 0 2 \_\_fp\_exact ; \}
\tl_const:Nn \_c\_inf\_fp \{ \_\_fp \_\_fp\_chk:w 2 0 \_\_fp\_exact ; \}
\tl_const:Nn \_c\_minus\_inf\_fp \{ \_\_fp \_\_fp\_chk:w 2 2 \_\_fp\_exact ; \}
\tl_const:Nn \_c\_nan\_fp \{ \_\_fp \_\_fp\_chk:w 3 1 \_\_fp\_exact ; \}

(End definition for \_c\_zero\_fp and others. These variables are documented on page 205.)
\c__fp_prec_int \c__fp_half_prec_int \c__fp_block_int

The number of digits of floating points.

\int_const:Nn \c__fp_prec_int { 16 }
\int_const:Nn \c__fp_half_prec_int { 8 }
\int_const:Nn \c__fp_block_int { 4 }

(End definition for \c__fp_prec_int, \c__fp_half_prec_int, and \c__fp_block_int.)

\c__fp_myriad_int

Blocks have 4 digits so this integer is useful.

\int_const:Nn \c__fp_myriad_int { 10000 }

(End definition for \c__fp_myriad_int.)

\c__fp_minus_min_exponent_int \c__fp_max_exponent_int

Normal floating point numbers have an exponent between \( -\text{minus_min_exponent} \) and \( \text{max_exponent} \) inclusive. Larger numbers are rounded to \( \pm\infty \). Smaller numbers are rounded to \( \pm0 \). It would be more natural to define a \text{min_exponent} with the opposite sign but that would waste one \LaTeX{} count.

\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }
\int_const:Nn \c__fp_max_exponent_int { 10000 }

(End definition for \c__fp_minus_min_exponent_int and \c__fp_max_exponent_int.)

\c__fp_max_exp_exponent_int

If a number’s exponent is larger than that, its exponential overflows/underflows.

\int_const:Nn \c__fp_max_exp_exponent_int { 5 }

(End definition for \c__fp_max_exp_exponent_int.)

\c__fp_overflowing_fp

A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

\cs_new:Npn \__fp_zero_fp:N #1
{ \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
\cs_new:Npn \__fp_inf_fp:N #1
{ \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }

(End definition for \c__fp_zero_fp:N and \c__fp_inf_fp:N.)

\__fp_exponent:w

For normal numbers, the function expands to the exponent, otherwise to 0. This is used in \l3str-format.

\cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
{ \if_meaning:w 1 #1 \exp_after:wN \__fp_use_ii_until_s:nnw \else: \exp_after:wN \__fp_use_i_until_s:nw \exp_after:wN 0 \fi: }

(End definition for \__fp_exponent:w.)

\__fp_zero_fp:N \__fp_inf_fp:N

In case of overflow or underflow, we have to output a zero or infinity with a given sign.

\cs_new:Npn \__fp_zero_fp:N #1
{ \s__fp \__fp_chk:w 0 \s__fp_use_ii_until_s:nnw \exp_after:wN \__fp_use_i_until_s:nw \exp_after:wN 0 \s__fp_underflow ; }
\cs_new:Npn \__fp_inf_fp:N #1
{ \s__fp \__fp_chk:w 2 \s__fp_use_ii_until_s:nnw \exp_after:wN \__fp_use_i_until_s:nw \exp_after:wN 0 \s__fp_overflow ; }

(End definition for \__fp_zero_fp:N and \__fp_inf_fp:N.)
\__fp\_neg\_sign:N  
When appearing in an integer expression or after \int\_value:w, this expands to the sign opposite to \#1, namely 0 (positive) is turned to 2 (negative), 1 (nan) to 1, and 2 to 0.

\begin{verbatim}
cs_new:Npn \__fp\_neg\_sign:N #1 
  {
    \__fp\_int\_eval:w 2 - #1 \__fp\_int\_eval\_end:
  }
\end{verbatim}

\__fp\_kind:w  
Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for NaN, 4 for tuples.

\begin{verbatim}
cs_new:Npn \__fp\_kind:w #1 
  {
    \__fp\_if\_type\_fp:NTwFw
    \if_int_compare:w #1 \__fp\_max\_exponent_int ~ \else:
      \if_int_compare:w #1 \__fp\_minus\_min\_exponent_int ~ \else:
        \if_meaning:w 1 #1 \__fp\_underflow:w
        \__fp\_overflow:w
        \__fp\_sanitize\_zero:w
      \fi:
    \fi:
  }
\end{verbatim}

\__fp\_sanitize:Nw, \__fp\_sanitize:wn, and \__fp\_sanitize\_zero:w  
Expects the sign and the exponent in some order, then the significand (which we don’t touch). Outputs the corresponding floating point number, possibly underflowed to ±0 or overflowed to ±∞. The functions \__fp\_underflow:w and \__fp\_overflow:w are defined in l3fp-traps.

\begin{verbatim}
cs_new:Npn \__fp\_sanitize:Nw #1 #2; 
  {
    \if_case:w #1 
      \if_int_compare:w #2 \__fp\_max\_exponent_int - \else:
        \if_int_compare:w #2 \__fp\_minus\_min\_exponent_int - \else:
          \if_meaning:w 1 #1 \__fp\_underflow:w
          \__fp\_overflow:w
          \__fp\_sanitize\_zero:w
        \fi:
      \fi:
    \fi:
    \__fp\_sanitize\_chk:w 1 #1 {#2}
  }
cs_new:Npn \__fp\_sanitize:wn #1 { \__fp\_sanitize:Nw \__fp\_sanitize\_zero:w }
\end{verbatim}

\__fp\_exp\_after\_o:w, \__fp\_exp\_after\_f:nu  
Places \langle\textit{tokens}\rangle (empty in the case of \__fp\_exp\_after\_o:w) between the \langle\textit{floating point}\rangle and the following tokens, then hits those tokens with o or f-expansion, and leaves the floating point number unchanged.
We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

\cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
\{    \if_meaning:w 1 #1
\exp_after:wN \__fp_exp_after_normal:nNNw
\else:
\exp_after:wN \__fp_exp_after_special:nNNw
\fi:
\{ \}
#1
\}
\cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
\{    \if_meaning:w 1 #2
\exp_after:wN \s__fp \exp_after:wN \__fp_chk:w \exp_after:wN #2
\exp_after:wN \exp_after:wN \int_value:w #3 \exp_after:wN ;
\int_value:w 1 #4 \exp_after:wN ;
\int_value:w 1 #5 \exp_after:wN ;
\int_value:w 1 #6 \exp_after:wN ;
\int_value:w 1 #7 \exp_after:wN ;
#1
\}
\cs_new:Npn \__fp_exp_after_normal:nNNw #1 #2 #3 #4 #5 #6 #7;
\{    \exp_after:wN \__fp_exp_after_normal:Nwwwww
\exp_after:wN \__fp_exp_after_normal:nNNw #1 1 #2 1 #3 1 #4 #5 1 #6 #7;
\}
\cs_new:Npn \__fp_exp_after_special:nNNw \{after\} \{case\} \{sign\} \{scan mark\};
Special floating point numbers are easy to jump over since they contain few tokens.
\cs_new:Npn \__fp_exp_after_special:nNNw #1 #2 #3 #4;
\{    \exp_after:wN \s__fp \exp_after:wN \__fp_chk:w
\exp_after:wN \exp_after:wN \exp_after:wN #2
\exp_after:wN \exp_after:wN \int_value:w #3 \exp_after:wN ;
\int_value:w 1 #4 \exp_after:wN ;
\int_value:w 1 #5 \exp_after:wN ;
\int_value:w 1 #6 \exp_after:wN ;
\int_value:w 1 #7 \exp_after:wN ;
#1
\}
\cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 1 #3 1 #4 #5 1 #6 #7;
\{    \exp_after:wN \__fp_exp_after_normal:Nwwwww
\exp_after:wN \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4 #5 #6 #7;
\}
For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by ,). That may be changed some day.
25.7 Other floating point types

Floating point tuples take the form \( \texttt{\_fp_tuple \_fp_tuple_chk:w \{ (fp 1) (fp 2) \ldots \} ; } \); where each \( (fp) \) is a floating point number or tuple, hence ends with \( ; \) itself. When a tuple is typeset, \( \texttt{\_fp_tuple_chk:w} \) produces an error, just like usual floating point numbers. Tuples may have zero or one element.

\[
\texttt{\_fp_tuple_count:w} \quad \text{Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to \texttt{\_fp_tuple_count:n}, but with the loop built-in. Checking for the end of the loop is done with the \texttt{use_none:n} \#1 construction.}
\]

\[
\texttt{\_fp_if_type_fp:NTwFw} \quad \text{Used as \texttt{\_fp_if_type_fp:NTwFw \{marker\} \{true code\}} \texttt{\_fp_stop} \texttt{\{false code\}} \texttt{\_fp_stop}, this test whether the \{marker\} is \texttt{\_fp} or not and runs the appropriate \{code\}. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.}
\]

\[
\texttt{\_fp_array_if_all_fp:nTF} \quad \text{True if all items are floating point numbers. Used for \texttt{min}.}
\]
\_fp \{ \prg_break:n \use_iii:nnn \}
\_fp_stop

(End definition for \_fp_array_if_all_fp:NTF and \_fp_array_if_all_fp_loop:u.)

Used as \_fp_type_from_scan:N \langle token \rangle. Grabs the pieces of the stringified \langle token \rangle which lies after the first \_fp. If the \langle token \rangle does not contain that string, the result is _?_.

\cs_new:Npn \_fp_type_from_scan:N #1
\{ \_fp_if_type_fp:NTwFw
#1 \{
\_fp \{ \_fp_type_from_scan_other:N #1 \}
\_fp_stop
\}
\}
\cs_new:Npx \_fp_type_from_scan_other:N #1
\{ \exp_not:N \exp_after:wN \exp_not:N \_fp_type_from_scan:w
\exp_not:N \token_to_str:N #1 \s__fp_mark
\tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
\}

\_fp_change_func_type:NNN
\_fp_change_func_type:aux:w
\_fp_change_func_type:chk:NNN

Arguments are \langle type marker \rangle \langle function \rangle \langle recovery \rangle. This gives the function obtained by placing the type after @. If the function is not defined then \langle recovery \rangle \langle function \rangle is used instead; however that test is not run when the \langle type marker \rangle is \_fp.

\cs_new:Npn \_fp_change_func_type:NNN #1#2#3
\{ \_fp_if_type_fp:NTwFw
#1 #2
\_fp
\{ \exp_after:wN \_fp_change_func_type:chk:NNN
\_fp \_fp_type_from_scan:other:N #1
\exp_after:wN \_fp_change_func_type:aux:w \token_to_str:N #2
\cs_end:
#2 #3
\}
\_fp_stop
\}
\exp_last_unbraced:NNNNo
\cs_new:Npn \_fp_change_func_type:aux:w #1 \{ \tl_to_str:n { __fp } \} \{ \}
\cs_new:Npn \_fp_change_func_type:chk:NNN #1#2#3
\{ \if_meaning:w \scan_stop: #1 \exp_after:wN \exp_after:wN #3 \exp_after:wN #2 \else:

735
The \texttt{Nnw} function simply dispatches to the appropriate \texttt{\_fp_exp_after..._f:nw} with "..." (either empty or \texttt{\_⟨type⟩}) extracted from \texttt{#1}, which should start with \texttt{\s__fp}. If it doesn't start with \texttt{\s__fp} the function \texttt{\_fp_exp_after_?_f:nw} defined in \texttt{l3fp-parse} gives an error; another special \texttt{(type)} is \texttt{stop}, useful for loops, see below. The \texttt{nw} function has an important optimization for floating points numbers; it also fetches its type marker \texttt{#2} from the floating point.

The loop works by using the \texttt{n} argument of \texttt{\_fp_exp_after_any_f:nw} to place the loop macro after the next item in the tuple and expand it.


25.8 Packing digits

When a positive integer \#1 is known to be less than \(10^8\), the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

\[
\text{\textbackslash{cs}}\text{\textunderscore{new}}:\text{\textbackslash{N}}\text{\textbackslash{p}}\text{\textbackslash{n}}\text{\textbackslash{\{\textpack{\textbackslash{NNNNNw}} \ #1 \ #2#3#4#5 \ #6; \ \{ \#2#3#4#5 \ \} \#6 \}}}
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\{\textpack{\textbackslash{NNNNNw}} \ \text{\textbackslash{\_fp\_int\_value}}:\text{\textbackslash{\_fp\_int\_eval}}:w \ 1 \ 0000 \ 0000 \ + \ #1}}\text{\textbackslash{;}}
\text{\textbackslash{}}
\]

The idea is that adding \(10^8\) to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by \TeX{}'s integers). This method is very heavily relied upon in \texttt{13fp-basics}.

More specifically, the auxiliary inserts \(+ \ #1#2#3#4#5 \ ; \ \{\#6} \), which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute \(12345 \times 6677 \ 8899\). With simplified names, we would do

\[
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\post\_processing:w}}
\text{\textbackslash{\_fp\_int\_value}}:w \ \text{\textbackslash{\_fp\_int\_eval}}:w \ - \ 5 \ 0000
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\pack:NNNNNw}}
\text{\textbackslash{\_fp\_int\_value}}:w \ \text{\textbackslash{\_fp\_int\_eval}}:w \ 4 \ 9995 \ 0000
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\pack:NNNNNw}}
\text{\textbackslash{\_fp\_int\_value}}:w \ \text{\textbackslash{\_fp\_int\_eval}}:w \ 5 \ 0000 \ 0000
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\pack:NNNNNw}}
\text{\textbackslash{\_fp\_int\_value}}:w \ \text{\textbackslash{\_fp\_int\_eval}}:w \ 12345 \ * \ 6677
\text{\textbackslash{exp\_after}}:\text{\textbackslash{wN}}\text{\textbackslash{\pack:NNNNNw}}
\text{\textbackslash{\_fp\_int\_value}}:w \ \text{\textbackslash{\_fp\_int\_eval}}:w \ 12345 \ * \ 8899
\]

The \texttt{\exp\_after:wN} triggers \texttt{\int\_value:w \ \texttt{\_fp\_int\_eval:w}} which starts a first computation, whose initial value is \(-5\ 0000\) (the “leading shift”). In that computation appears an \texttt{\exp\_after:wN}, which triggers the nested computation \texttt{\int\_value:w \ \texttt{\_fp\_int\_-eval:w}} with starting value \texttt{4\ 9995\ 0000} (the “middle shift”). That, in turn, expands \texttt{\exp\_after:wN} which triggers the third computation. The third computation's value is \texttt{5\ 0000\ 0000 \ + \ 12345 \times \ 8899}, which has 9 digits. Adding \(5 \cdot 10^8\) to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The \texttt{\pack} function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a \(+\), which combines the carry with the previous computation. The shifts nicely combine into \texttt{5\ 0000\ 0000/10^4 + 4\ 9995\ 0000 = 5\ 0000\ 0000}. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding \texttt{\pack} function, expanded after the result is computed, braces the last 4 digits, and leaves \(+\langle\ 5\ \text{digits}\rangle\) for the initial computation. The “leading shift” cancels the combination of the other shifts, and the \texttt{\post\_processing:w} takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making \texttt{13fp} as fast as other pure \TeX{} floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

This set of shifts allows for computations involving results in the range \([-4 \cdot 10^8, 5 \cdot 10^8 - 1]\). Shifted values all have exactly 9 digits.

\[
\text{\texttt{\int\_const:Nn \ \texttt{\_fp\_leading\_shift\_int} \ \{ - 5 \ 0000 \ \}}}
\text{\texttt{\int\_const:Nn \ \texttt{\_fp\_middle\_shift\_int} \ \{ 5 \ 0000 \ * \ 9999 \ \}}}
\text{\texttt{\int\_const:Nn \ \texttt{\_fp\_trailing\_shift\_int} \ \{ 5 \ 0000 \ * \ 10000 \ \}}}
\text{\texttt{\cs\_new:Npn \ \texttt{\_fp\_pack:NNNNNw \ #1 \ #2#3#4#5 \ #6; \ \{ + \ #1#2#3#4#5 \ ; \ \{\#6} \}}}\]

737
This set of shifts allows for computations involving results in the range \([-5 \cdot 10^8, 6 \cdot 10^8 - 1]\) (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to \(\LaTeX\)'s limit of \(2^{31} - 1\) on integers. The shifts are chosen to be roughly the midpoint of \(10^9\) and \(2^{31}\), the two bounds on 10-digit integers in \(\LaTeX\).

\begin{verbatim}
\cs_new:Npn \_fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
{ \int_const:Nn \c__fp_big_leading_shift_int { \times 0000 } \int_const:Nn \c__fp_big_middle_shift_int { #1#2#3#4#5#6 \times 0000 } \int_const:Nn \c__fp_big_trailing_shift_int { #7 } }
\end{verbatim}

(End definition for \_fp_pack_big:NNNNNNw and others.)

This set of shifts allows for computations with results in the range \([-1 \cdot 10^9, 147483647]\); the end-point is \(2^{31} - 2 \cdot 10^9 \approx 1.47 \cdot 10^9\). Shifted values all have exactly 10 digits.

\begin{verbatim}
\cs_new:Npn \_fp_pack_Bigg:NNNNNNw #1#2#3#4#5 #6#7#8#9
{ \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 } \int_const:Nn \c__fp_Bigg_middle_shift_int { #1#2#3#4#5#6 \times 0000 } \int_const:Nn \c__fp_Bigg_trailing_shift_int { #7 } }
\end{verbatim}

(End definition for \_fp_pack_Bigg:NNNNNNw and others.)

\_fp_pack_twice_four:NNNNNNNN
\_fp_pack_twice_four:NNNNNNNN
\_fp_pack_eight:NNNNNNNN
\_fp_pack_eight:NNNNNNNN
\_fp_basics_pack_low:NNNNNw
\_fp_basics_pack_high:NNNNNw
\_fp_basics_pack_high_carry:w

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, \_fp_basics_pack_high_carry:w is always followed by four times \{0000\}.

This is used in \l3fp-basics and \l3fp-extended.

\begin{verbatim}
\cs_new:Npn \_fp_pack_twice_four:NNNNNNNN #1; #2#3#4#5 #6#7#8#9
{ \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 } \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 } \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 } \_fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7; }
\end{verbatim}

(End definition for \_fp_pack_twice_four:NNNNNNNN.)

\begin{verbatim}
\cs_new:Npn \_fp_pack_eight:NNNNNNNN #1; #2#3#4#5 #6#7#8#9
{ \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 } \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 } \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 } \_fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7; }
\end{verbatim}

(End definition for \_fp_pack_eight:NNNNNNNN.)

\begin{verbatim}
\cs_new:Npn \_fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
{ + #1#2#3#4#5#6 ; \{#7\} }
\end{verbatim}

(End definition for \_fp_basics_pack_low:NNNNNw.)

\begin{verbatim}
\cs_new:Npn \_fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
{ + #1#2#3#4#5#6 ; \{#7\} }
\end{verbatim}

(End definition for \_fp_basics_pack_high:NNNNNw.)

\begin{verbatim}
\cs_new:Npn \_fp_basics_pack_high_carry:w #1 #2#3#4#5#6#7#8#9
{ \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 } \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 } \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 } \_fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7; }
\end{verbatim}

(End definition for \_fp_basics_pack_high_carry:w.)
This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence
the name.

\cs_new:Npn \__fp_basics_pack_weird_low:NNNNw \#1 \#2 \#3 \#4 \#5; 
\if_meaning:w 2 \#1 + 1 \fi: \__fp_int_eval_end: \#2 \#3 \#4; \{ \#5 \};

(End definition for \__fp_basics_pack_weird_low:NNNNw.

\cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw \#1\#2\#3\#4 \#5\#6\#7\#8 \#9; 
\{ \#1\#2\#3\#4 \#5\#6\#7\#8 \#9 \} )

(End definition for \__fp_basics_pack_weird_low:NNNNw and \__fp_basics_pack_weird_high:NNNNNNNNw.)

25.9 Decimate (dividing by a power of 10)

\__fp_decimate:nNnnnn \langle \#1 \rangle \langle \#2 \rangle \langle \#3 \rangle \langle \#4 \rangle \langle \#5 \rangle ;

\langle X_1 \rangle \{ X_2 \} \{ X_3 \} \{ X_4 \}

Each \langle X_i \rangle consists in 4 digits exactly, and 1000 \leq \langle X_1 \rangle < 9999. The first argument determines by how much we shift the digits. \langle f_1 \rangle is called as follows:

\langle f_1 \rangle \langle \text{rounding} \rangle \langle X'_1 \rangle \{ X'_2 \} \langle \text{extra-digits} \rangle ;

where 0 \leq \langle X'_1 \rangle < 10^8 - 1 are 8 digit integers, forming the truncation of our number. In other words,

\[ \sum_{i=1}^{4} 10^{-4i} \cdot 10^{-\langle \text{shift} \rangle} \cdot 10^{-8} \cdot \langle X'_1 \rangle + 10^{-16} \cdot \langle X'_2 \rangle = 0. \langle \text{extra-digits} \rangle \cdot 10^{-16} \in [0, 10^{-16}) \]

To round properly later, we remember this information about the difference. The \langle \text{rounding} \rangle digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly 0.5 \cdot 10^{-16}. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, \langle \text{rounding} \rangle is 1 (not 0), and \langle X'_1 \rangle and \langle X'_2 \rangle are both zero.

If the shift is 1, the \langle \text{rounding} \rangle digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the \langle \text{rounding} \rangle digit to be placed after the \langle X'_1 \rangle, but the choice we make involves less reshuffling.

Note that this function treats negative \langle \text{shift} \rangle as 0.

\cs_new:Npn \__fp_decimate:nNnnnn \#1 

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for $\langle f_2 \rangle$.

If the $\langle \text{shift} \rangle$ is zero, or too big, life is very easy.

(End definition for $\langle f_3 \rangle$.

Shifting happens in two steps: compute the $\langle \text{rounding} \rangle$ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through $\langle \text{tmp} \rangle$. The arguments are as follows: #1 indicates which function is being defined; after one step of expansion, #2 yields the "extra digits" which are then converted by $\langle \text{fp_round_digit} \rangle$ to the $\langle \text{rounding} \rangle$ digit (note the + separating blocks of digits to avoid overflowing \TeX{}'s integers). This triggers the $\text{f}$-expansion of $\langle \text{fp_decimate_pack} \rangle$, responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.
The computation of the (rounding) digit leaves an unfinished \texttt{\texttt{int_value:w}}, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

\begin{verbatim}
\cs_new:Npn \__fp_decimate_pack:nnnnnnnnnnw #1#2#3#4#5
{ \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
\cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
{ {#1} {#2#3#4#5#6} }
\end{verbatim}

(End definition for \texttt{\texttt{\_fp_decimate_pack:nnnnnnnnnnw}} and others.)

25.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one \texttt{\fi:} as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in \texttt{l3fp} must perform tests on the type of floating points that they receive. This is often done in an \texttt{\if_case:w} statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

\begin{verbatim}
\if_case:w {integer} \exp_stop_f:
  \_\_fp_case_return_o:Nw {fp var}
\or: \_\_fp_case_use:nw {some computation}
\or: \_\_fp_case_return_same_o:w
\or: \_\_fp_case_return:nw {something}
\fi:

{junk}
{floating point}
\end{verbatim}

In this example, the case 0 returns the floating point \texttt{\langle fp var \rangle}, expanding once after that floating point. Case 1 does \texttt{\langle some computation \rangle} using the \texttt{(floating point)} (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the \texttt{(floating point)} without modifying it, removing the \texttt{\langle junk \rangle} and expanding once after. Case 3 closes the conditional, removes the \texttt{\langle junk \rangle} and the \texttt{\langle floating point \rangle}, and expands \texttt{\langle something \rangle} next. In other cases, the \texttt{\langle junk \rangle} is expanded, performing some other operation on the \texttt{\langle floating point \rangle}. We provide similar functions with two trailing \texttt{(floating points)}.  

741
\_fp\_case\_use:nw
This function ends a \TeX{} conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

\begin{verbatim}
cs_new:Npn \_fp\_case\_use:nw \#1\#2 \fi: \#3 \s__fp { \fi: \#1 \s__fp }
\end{verbatim}

(End definition for \_fp\_case\_use:nw.)

\_fp\_case\_return:nw
This function ends a \TeX{} conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don’t define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the \langle junk \rangle may not contain semicolons.

\begin{verbatim}
cs_new:Npn \_fp\_case\_return:nw \#1\#2 \fi: \#3 ; { \fi: \#1 }
\end{verbatim}

(End definition for \_fp\_case\_return:nw.)

\_fp\_case\_return\_o:W
This function ends a \TeX{} conditional, removes junk and a floating point, and returns its first argument (an \langle fp var \rangle) then expands once after it.

\begin{verbatim}
cs_new:Npn \_fp\_case\_return\_o:W \#1\#2 \fi: \#3 \s__fp \#4 ; { \fi: \exp\_after:wN \#1 }
\end{verbatim}

(End definition for \_fp\_case\_return\_o:W.)

\_fp\_case\_return\_same\_o:W
This function ends a \TeX{} conditional, removes junk, and returns the following floating point, expanding once after the result.

\begin{verbatim}
cs_new:Npn \_fp\_case\_return\_same\_o:W \#1 \fi: \#2 \s__fp \#3 ; \s__fp \#4 ; { \fi: \__fp\_exp\_after:o:w \s__fp \#3 ; }
\end{verbatim}

(End definition for \_fp\_case\_return\_same\_o:W.)

\_fp\_case\_return\_o:W
Same as \_fp\_case\_return\_o:W but with two trailing floating points.

\begin{verbatim}
cs_new:Npn \_fp\_case\_return\_o:W \#1\#2 \fi: \#3 \s__fp \#4 ; \#5 ; { \fi: \_fp\_exp\_after:o:w \s__fp \#4 \#5 ; }
\end{verbatim}

(End definition for \_fp\_case\_return\_o:W.)

\_fp\_case\_return\_i:W
Similar to \_fp\_case\_return\_same\_o:W, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

\begin{verbatim}
cs_new:Npn \_fp\_case\_return\_i:W \#1 \fi: \#2 \s__fp \#3 ; \s__fp \#4 ; { \fi: \_fp\_exp\_after:o:w \s__fp \#3 \#4 ; }
cs_new:Npn \_fp\_case\_return\_ii:W \#1 \fi: \#2 \s__fp \#3 ; \s__fp \#4 ; { \fi: \_fp\_exp\_after:o:w }
\end{verbatim}

(End definition for \_fp\_case\_return\_i:W and \_fp\_case\_return\_ii:W.)
25.11 Integer floating points
\__fp_int.p:w
\__fp_int:wTF
Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from ___fp_decimate:nNnnnn is 0.

\prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
{ TF , T , F , p }
{
  \if_case:w #1 \exp_stop_f:
  \prg_return_true:
  \or:
  \if_charcode:w 0 \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
  \__fp_use_i_until_s:nw #4
  \prg_return_true:
  \else:
  \prg_return_false:
  \fi:
  \else:
  \prg_return_false:
  \fi:
}
(End definition for \__fp_int:wTF.)

25.12 Small integer floating points
\__fp_small_int:wTF
\__fp_small_int_true:wTF
\__fp_small_int_normal:NnwTF
\__fp_small_int_test:NnnwNTF
Tests if the floating point argument is an integer or ±∞. If so, it is clipped to an integer in the range \([-10^8, 10^8]\) and fed as a braced argument to the ⟨true code⟩. Otherwise, the ⟨false code⟩ is performed.

First filter special cases: zeros and infinities are integers, \texttt{nan} is not. For normal numbers, decimate. If the rounding digit is not 0 run the ⟨false code⟩. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise \(10^8\).

\cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1\#2
{ \if_case:w #1 \exp_stop_f:
  \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
  \or: \exp_after:wN \__fp_small_int_normal:NnwTF
  \or:
  \__fp_case_return:nw
  \exp_after:wN \__fp_small_int_true:wTF \int_value:w
  \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
  \}
  \else: \__fp_case_return:nw \use_ii:nn
  \fi:
  \#2
}
\cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
\cs_new:Npn \__fp_small_int_normal:NnwTF \s__fp \__fp_chk:w #1#2#3;
{ \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
  \__fp_small_int_test:NnnwNw
  \#3 \#1
}

25.13 Fast string comparison

\__fp_str_if_eq:nn

A private version of the low-level string comparison function. As the nature of the arguments is restricted and as speed is of the essence, this version does not seek to deal with # tokens. No l3sys or l3luatex just yet so we have to define in terms of primitives.

25.14 Name of a function from its l3fp-parse name

\__fp_func_to_name:N \__fp_func_to_name_aux:w

The goal is to convert for instance \__fp_sin_o:w to sin. This is used in error messages hence does not need to be fast.
25.15 Messages

Using a floating point directly is an error.

\[ \text{\_kernel\_msg\_new:nnnn \{ kernel \} \{ misused-fp \}} \]

\[ \{ \text{A floating-point with value ‘#1’ was misused.} \} \]

\[ \{ \text{To obtain the value of a floating-point variable, use ‘\text{\_token\_to\_str:N \_fp\_to\_decimal:N}’, ‘\text{\_token\_to\_str:N \_fp\_to\_tl:N}’, or other conversion functions.} \} \]

\[ \langle /initex | package \rangle \]

26 l3fp-traps Implementation

\[ \langle *initex | package \rangle \]

\[ \langle @@=fp \rangle \]

Exceptions should be accessed by an n-type argument, among

- invalid_operation
- division_by_zero
- overflow
- underflow
- inexact (actually never used).

26.1 Flags

Flags to denote exceptions.

\[ \text{\flag\_new:n \{ fp\_invalid\_operation \}} \]

\[ \text{\flag\_new:n \{ fp\_division\_by\_zero \}} \]

\[ \text{\flag\_new:n \{ fp\_overflow \}} \]

\[ \text{\flag\_new:n \{ fp\_underflow \}} \]

\[ \langle /initex | package \rangle \]

26.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an n-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions
• \_\_fp_invalid_operation:nw,
• \_\_fp_invalid_operation_o:Nww,
• \_\_fp_invalid_operation_tl_o:ff,
• \_\_fp_division_by_zero_o:Nnw,
• \_\_fp_division_by_zero_o:NNww,
• \_\_fp_overflow:w,
• \_\_fp_underflow:w.

Rather than changing them directly, we provide a user interface as \fp_trap:nn \langle\text{exception}\rangle \langle\text{way of trapping}\rangle, where the \langle\text{way of trapping}\rangle is one of error, flag, or none.

We also provide \_\_fp_invalid_operation_o:nw, defined in terms of \_\_fp_invalid_operation:nw.

\fp_trap:nn
\begin{verbatim}
\cs_new_protected:Npn \fp_trap:nn #1#2
\end{verbatim}

(End definition for \fp_trap:nn. This function is documented on page 207.)

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don’t even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.
We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don’t even raise the flag. In all cases, the function must produce a result, namely its first argument, ±∞ or NaN.
Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding \texttt{\prg_do_nothing}, \texttt{\use_none:nnnn} or \texttt{\use_none:nnnnnn} to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the \texttt{\_fp_overflow:w} and \texttt{\_fp_underflow:w} functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as $10^{**1e9999}$, the exponent would be too large for \TeX{}, and \texttt{\_fp_overflow:w} receives $\pm \infty$ (\texttt{\_fp_underflow:w} would receive $\pm 0$); then we cannot do better than simply say an overflow or underflow occurred.

```
\cs_new_protected:Npn \__fp_trap_overflow_set_error: { \__fp_trap_overflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_overflow_set_flag: { \__fp_trap_overflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set_none: { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set:N #1 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
\cs_new_protected:Npn \__fp_trap_underflow_set_error: { \__fp_trap_underflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_underflow_set_flag: { \__fp_trap_underflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set_none: { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set:N #1 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
\cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4 { \exp_args:Nno \use:n { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; } #1 \__fp_error:nffn { fp-flow \if_meaning:w 1 ##1 -to \fi: } \__fp_to_tl:n { \s__fp \__fp_chk:w #1##2##3; } \token_if_eq_meaning:NNF 0 ##2 { - } #4 }{#2} \flag_raise_if_clear:n { fp_#2 } #3 ##2}
```

(End definition for \_fp_trap_division_by_zero_set_error: and others.)

```
\__fp_invalid_operation:nnw \__fp_invalid_operation_o:Nww \__fp_invalid_operation_tl_o:ff \__fp_division_by_zero_o:Nww \__fp_invalid_operation_1:ff \__fp_overflow:w \__fp_underflow:w
```

Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
\cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
\cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
\cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
```

(End definition for \_fp_trap_overflow_set_error: and others.)
Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

\cs_new:Npn \__fp_invalid_operation_o:nw
\cs_new:Npn \__fp_invalid_operation_o:fw
(End definition for `\__fp_invalid_operation_o:nw`.
\__fp_invalid_operation_o:nw
\__fp_invalid_operation_o:fw

26.3 Errors

\cs_new:Npn \__fp_error:nnnn
\cs_new:Npn \__fp_error:nnfn
\cs_new:Npn \__fp_error:nffn
\cs_new:Npn \__fp_error:nfff
\cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }
(End definition for `\__fp_error:nnnn`.)

26.4 Messages

Some messages.

\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\__kernel_msg_new:nnnn \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-flow \}
{ An - \#3 - occurred. }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-flow-to \}
{ \#1 - \#3 ed - to - \#2 . }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-zero-div \}
{ Division-by-zero-in- \#1 (\#2) }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-zero-div-ii \}
{ Division-by-zero-in- (\#1) \#3 (\#2) }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-invalid \}
{ Invalid-operation- \#1 (\#2) }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-invalid-ii \}
{ Invalid-operation- (\#1) \#3 (\#2) }
\_\_kernel\_msg\_new:\nnn \{ kernel \} \{ fp-unknown-type \}
{ Unknown-type-for-'\#1' }
(\textit{\textless} \textit{initext \| \textgreater\textless} package\textit{\textgreater})

\_\_fp\_parse\_word\_trunc:N
\_\_fp\_parse\_word\_floor:N
\_\_fp\_parse\_word\_ceil:N
\_\_fp\_parse\_word\_round:N
\_\_fp\_parse\_round:N
(\textit{End definition for \_\_fp\_parse\_word\_trunc:N, \_\_fp\_parse\_word\_floor:N, and \_\_fp\_parse\_word\_ceil:N.})

\_\_fp\_parse\_word\_round:N
\_\_fp\_parse\_round:N
\_\_fp\_parse\_five\_int
This is used as the half-point for which numbers are rounded up/down.
\int\_const:Nn \c\_fp\_five\_int \{ 5 \}

27 I3fp-round implementation

27.1 Rounding tools

\c\_fp\_five\_int
Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- **Round to nearest**: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.

- **Round towards negative infinity**: round to the greatest floating point number not larger than the exact result.

- **Round towards zero**: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.

- **Round towards positive infinity**: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function \_\_fp\_round\:NNN, which expands to 0\exp_stop_f: or 1\exp_stop_f: depending on whether the final result should be rounded up or down.

- \_\_fp\_round\:NNN \:\langle sign \rangle \:\langle digit1 \rangle \:\langle digit2 \rangle can expand to 0\exp_stop_f: or 1\exp_stop_f:.

- \_\_fp\_round_s\:NNN\:w \:\langle sign \rangle \:\langle digit1 \rangle \:\langle digit2 \rangle \:\langle more digits \rangle can expand to 0\exp_stop_f:; or 1\exp_stop_f:;.

- \_\_fp\_round_neg\:NNN \:\langle sign \rangle \:\langle digit1 \rangle \:\langle digit2 \rangle can expand to 0\exp_stop_f: or 1\exp_stop_f:.

See implementation comments for details on the syntax.

\_\_fp\_round\:NNN \:\langle final sign \rangle \:\langle digit1 \rangle \:\langle digit2 \rangle

If rounding the number \:\langle final sign \rangle\:\langle digit1 \rangle\:\langle digit2 \rangle to an integer rounds it towards zero (truncates it), this function expands to 0\exp_stop_f:, and otherwise to 1\exp_stop_f:. Typically used within the scope of an \_\_fp\_int_eval:w, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that \:\langle final sign \rangle be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that \:\langle final sign \rangle is 0 for positive, and 2 for negative.

By default, the functions below return 0\exp_stop_f:, but this is superseded by \_\_fp\_round\:NNN \:\langle final sign \rangle \:\langle digit1 \rangle \:\langle digit2 \rangle, which instead returns 1\exp_stop_f:, expanding everything and removing 0\exp_stop_f: in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the \:\langle digit2 \rangle is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that \:\langle digit1 \rangle plus the result is even. In other words, round up if \:\langle digit1 \rangle is odd.
The “round to nearest” mode has three variants, which differ in how ties are rounded:
down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

\begin{verbatim}
\cs_new:Npn \__fp_round_return_one:
  { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
\cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
  { \if_meaning:w 2 #1
    \if_int_compare:w #3 > 0 \exp_stop_f:
      \__fp_round_return_one:
    \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
\cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
  { \if_meaning:w 0 #1
    \if_int_compare:w #3 > 0 \exp_stop_f:
      \__fp_round_return_one:
    \fi:
    \fi:
    0 \exp_stop_f:
  }
\cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
  { \if_int_compare:w #3 > \c__fp_five_int
    \__fp_round_return_one:
  \else:
    \if_meaning:w 5 #3
      \if_int_odd:w #2 \exp_stop_f:
        \__fp_round_return_one:
      \fi:
      \fi:
      0 \exp_stop_f:
    \fi:
    \fi:
    \fi:
  }
\cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
  { \if_int_compare:w #3 > \c__fp_five_int
    \__fp_round_return_one:
  \else:
    \if_meaning:w 5 #3
      \if_meaning:w 2 #1
        \__fp_round_return_one:
      \fi:
      \fi:
      \fi:
      0 \exp_stop_f:
    \fi:
    \fi:
    \fi:
  }
\cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
  { \if_int_compare:w #3 > \c__fp_five_int
    \__fp_round_return_one:
  \fi:
\end{verbatim}
\begin{verbatim}
\cs_new:Npn \_fp_round_to_nearest_pinf:NNN #1 #2 #3 {
    \if_int_compare:w #3 > \c__fp_five_int
        \_fp_round_return_one:
    \else:
        \if_meaning:w 5 #3
            \if_meaning:w 0 #1
                \_fp_round_return_one:
            \fi:
        \fi:
    \fi:
}
\cs_new_eq:NN \_fp_round:NNN \_fp_round_to_nearest:NNN

(End definition for \_fp_round:NNN and others.)

\_fp_round_s:NNNw
\_fp_round_s:NNNw \langle final sign \rangle \langle digit \rangle \langle more digits \rangle ;
Similar to \_fp_round:NNN, but with an extra semicolon, this function expands to 0\exp_stop_f; if rounding \langle final sign\rangle\langle digit\rangle\langle more digits\rangle to an integer truncates, and to 1\exp_stop_f; otherwise. The \langle more digits \rangle part must be a digit, followed by something that does not overflow a \texttt{\int_use:N} \_fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

\cs_new:Npn \_fp_round_s:NNNw #1 #2 #3 #4; {
    \exp_after:wN \_fp_round:NNN
    \exp_after:wN #1
    \exp_after:wN #2
    \int_value:w \_fp_int_eval:w
    \if_int_odd:w \if_meaning:w 0 #1 1 \else:
        \if_meaning:w 5 #1 1 \fi:
    \fi:
    \exp_stop_f:
    \if_int_compare:w \_fp_int_eval:w #4 > 0 \exp_stop_f:
        1 +
    \fi:
    \if: \fi:
    \if: \fi:
    \_fp_int_eval:w
    \fi:
    \_fp_int_eval:w
    \fi:
    #3
}

(End definition for \_fp_round_s:NNNw.)

\_fp_round_digit:Nw
\_fp_round_digit:Nw \langle digit \rangle \langle intexpr \rangle ;
This function should always be called within an \texttt{\int_value:w} or \_fp_int_eval:w expansion; it may add an extra \_fp_int_eval:w, which means that the integer or integer expression should not be ended with a synonym of \texttt{\relax}, but with a semi-colon for instance.

\cs_new:Npn \_fp_round_digit:Nw #1 #2; {
    \if_int_odd:w \if_meaning:w 0 #1 1 \else:
        \if_meaning:w 5 #1 1 \else:
            0 \fi: \fi: \exp_stop_f:
}
\end{verbatim}
\if_int_compare:w \__fp_int_eval:w #2 > 0 \exp_stop_f:
  \__fp_int_eval:w 1 +
\fi:
\fi:
#1}

(End definition for \_fp_round_digit:Nw.)


This expands to \_exp_stop_f: or 1\_exp_stop_f: after doing the following test. Starting from a number of the form \_{final sign}0\_{15 digits}00...0\_{digit2}, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns 1\_exp_stop_f:. Otherwise, i.e., if the result is rounded back to the first operand, then this function returns 0\_exp_stop_f:

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

27.2 The round function

First check that all arguments are floating point numbers. The \_trunc, \_ceil and \_floor functions expect one or two arguments (the second is 0 by default), and the round function also accepts a third argument (nan by default), which changes #1 from \_fp_round_to_nearest:NNN to one of its analogues.

(End definition for \_fp_round_neg:NNN and others.)
Having three arguments is only allowed for \texttt{round}, not \texttt{trunc}, \texttt{ceil}, \texttt{floor}, so check for that case. If all is well, construct one of \texttt{\_fp_round\_to\_nearest:NNN}, \texttt{\_fp_round\_to\_nearest\_zero:NNN}, \texttt{\_fp_round\_to\_nearest\_ninf:NNN}, \texttt{\_fp_round\_to\_nearest\_pinf:NNN} and act accordingly.

\[\__fp\_round\_no\_arg\_o:Ww\]

\[\__fp\_round\_Nww\]

\[\__fp\_round\_aux\_o:Ww\]

\[\__fp\_round\_no\_arg\_o:Ww\]
\__fp_error:nffn { fp-num-args }
{ \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
\exp_after:wN \c_nan_fp
}

(End definition for \__fp_round:Nwww.)

\__fp_round_name_from_cs:N
\cs_new:Npn \__fp_round_name_from_cs:N #1
{ \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { \cs_randomname:Nw \cs_zip:Nn \c_zero_fp }
  { \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { \cs_randomname:Nw \cs_zip:Nn \c_neginf_fp }
    { \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { \cs_randomname:Nw \cs_zip:Nn \c_posinf_fp }
      \cs_if_eq:NNTF #1 \cs_randomname:Nw \cs_zip:Nn \c_nan_fp }
    }
  }
}

(End definition for \__fp_round_name_from_cs:N.)

\__fp_round:Nww \__fp_round:Nwn \__fp_round_normal:Nww
\__fp_round_normal:Nwww \__fp_round_pack:Nw
\__fp_round_normal_end:www \__fp_round_special:Nww
\__fp_round_special_aux:Nw

\__fp_round:Nww
\__fp_round:Nwn
\__fp_round_normal:Nww
\__fp_round_normal:Nwww
\__fp_round_pack:Nw
\__fp_round_normal_end:www
\__fp_round_special:Nww
\__fp_round_special_aux:Nw

If the number of digits to round to is an integer or infinity all is good; if it is \texttt{nan} then just produce a \texttt{nan}; otherwise invalid as we have something like \texttt{round(1,3.14)} where the number of digits is not an integer.

\__fp_round:Nww
\__fp_round:Nwn
\__fp_round_normal:Nww
\__fp_round_normal:Nwww
\__fp_round_pack:Nw
\__fp_round_normal_end:www
\__fp_round_special:Nww
\__fp_round_special_aux:Nw

\__fp_round:Nww #1 #2 ; #3 ;
{ \__fp_small_int:wTF #3; { \__fp_round:Nwn #1 #2; } 
  { \if:w 3 \__fp_kind:w #3 ;
    \exp_after:wN \use_i:nn
    \else:
    \exp_after:wN \use_ii:nn
    \fi: 
    \__fp_small_int:wTF #3; { \__fp_round:Nwn #1 #2; } 
  { \__fp_invalid_operation_tl_o:ff 
    { \__fp_round_name_from_cs:N #1 }
    { \__fp_array_to_clist:n { #2; #3; } }
  } 
}

\__fp_round:Nww #1 \s__fp \__fp_chk:w #2 #3 #4; #5
{ \if_meaning:w 1 #2
  \exp_after:wN \__fp_round_normal:Nwww
  \exp_after:wN \__fp_round_normal:Nww
  \int_value:w #5
  \else: 
  \exp_after:wN \__fp_exp_after:o:w 
  \s__fp \__fp_chk:w #2 #3 #4; 
  \fi: 

756
\fi:
\fi:
\fi:
\fi:

\cs_new:Npn \__fp_round_special_aux:Nw #1#2;
{\__fp_exp_after:wN \__fp_exp_after_o:w \exp:w \exp_continue_f:w
\__fp_sanitize:Nw #1#2; \{1000}\{0000}\{0000}\{0000};
}

(End definition for \__fp_round:Nww and others.)

\section{l3fp-parse implementation}

\subsection{Work plan}

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a \textit{floating point object} is a floating point number or tuple. This can be extended to anything that starts with \texttt{\_\_fp} or \texttt{\_\_fp__\langle type\rangle} and ends with \texttt{;} with some internal structure that depends on the \texttt{\langle type\rangle}.

\__fp_parse:n
\__fp_parse:n {\langle fpexpr\rangle}
Evaluates the \textit{floating point expression} and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully f-expanded.

\__fp_parse:o:n does the same but expands once after its result.

\TeXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as \texttt{\int_use:N}. Invalid tokens remaining after f-expansion lead to unrecoverable low-level \TeX errors.

(End definition for \__fp_parse:n.)

\c__fp_prec_func_int
\c__fp_prec_hatii_int
\c__fp_prec_hat_int
\c__fp_prec_not_int
\c__fp_prec_juxt_int
\c__fp_prec_times_int
\c__fp_prec_plus_int
\c__fp_prec_minus_int
\c__fp_prec_comp_int
\c__fp_prec_and_int
\c__fp_prec_or_int
\c__fp_prec_quest_int
\c__fp_prec_colon_int
\c__fp_prec_comma_int
\c__fp_prec_tuple_int
\c__fp_prec_end_int

Floating point expressions are composed of numbers, given in various forms, infix operators, such as $+$, $\ast$, or , (which joins two numbers into a list), and prefix operators, such as the unary $-$, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary $\ast\ast$ and $\sim$ (right to left).

12 Unary $+$, $-$, ! (right to left).

11 Juxtaposition (implicit $\ast$) with no parenthesis.

10 Binary $\ast$ and $/$.
Binary + and -.

Comparisons.

Logical and, denoted by &&.

Logical or, denoted by ||.

Ternary operator ?;, piece ?.

Ternary operator ?:, piece :.

Commas.

Place where a comma is allowed and generates a tuple.

Start and end of the expression.

(End definition for \c__fp_prec_func_int and others.)

28.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to f-expand tokens one by one in the expression (as \int_eval:n does), and with this approach, expanding the next unread token forces us to jump with \exp_after:wN over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the \exp_after:wN is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be
present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored before the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w ⟨stuff⟩

One step of expansion expands \exp_after:wN, which triggers the primitive \int_-value:w, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second \exp_after:wN to expand, and to trigger the construction \exp:w, which expands \operand:w, defined to read what follows and make a number out of it, then leave \exp_end:, the number, and a semicolon in the input stream. Once \operand:w is done expanding, we obtain essentially

\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;

where in fact \exp_after:wN has already been expanded, \int_value:w has already seen 12345, and \exp:w is still looking for a number. It finds \exp_end:, hence expands to nothing. Now, \int_value:w sees the ;, which cannot be part of a number. The expansion stops, and we are left with

\add:ww 12345 ; 333444 ;

which can safely perform the addition by grabbing two arguments delimited by ;.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as \add:ww. Just like \int_value:w 12345 \exp_after:wN ; expanded what follows once, we need \add:ww to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form \__fp_..._o:ww expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

28.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: \(1 + 2 \times 3 = 1 + (2 \times 3)\) because \(\times\) has a higher precedence than \(+\).

The true analog of our macro \operand:w must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called \operand:w. This means that \operand:w must know what the previous binary operator is, or rather, its precedence: we thus re-name it \operand:Nw. Let us describe as an example how we plan to do the calculation

\(41 - 2^3 \times 4 + 5\). More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of \operand:Nw should be an integer constant (\c__fp_prec_plus_int, \ldots) equal to the precedence of the given operator, not directly the operator itself.
• Clean up 41 and find -. We call \( \text{operand}:Nw \) to find the second operand.

• Clean up 2 and find \(^\) .

• Compare the precedences of - and \(^\). Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to \( \text{operand}:Nw \) .

• Clean up 3 and find *.

• Compare the precedences of - and *. Since the former is higher, \( \text{operand}:Nw \) has found the second operand of the exponentiation, which is computed: \( 2^3 = 8 \).

• We now have 41-8*4+5, and \( \text{operand}:Nw \) is still looking for a second operand for the subtraction. Is it 8?

• Compare the precedences of - and *. Since the latter is higher, we are not done with 8. Call \( \text{operand}:Nw \) * to find the second operand of the multiplication.

• Clean up 4, and find +.

• Compare the precedences of * and +. Since the former is higher, \( \text{operand}:Nw \) * has found the second operand of the multiplication, which is computed: \( 8 * 4 = 32 \).

• We now have 41-32+5, and \( \text{operand}:Nw \) is still looking for a second operand for the subtraction. Is it 32?

• Compare the precedences of - and +. Since they are equal, \( \text{operand}:Nw \) has found the second operand for the subtraction, which is computed: \( 41 - 32 = 9 \).

• We now have 9+5.

The procedure above stops short of performing all computations, but adding a surrounding call to \( \text{operand}:Nw \) with a very low precedence ensures that all computations are performed before \( \text{operand}:Nw \) is done. Adding a trailing marker with the same very low precedence prevents the surrounding \( \text{operand}:Nw \) from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the \textit{l3fp-parse} functions correspond to each step above.

First, \( \text{\textunderscore{fp_parse_operand}}:Nw \) is the \( \text{operand}:Nw \) function above, with small modifications due to expansion issues discussed later. We denote by \( \langle \text{precedence} \rangle \) the argument of \( \text{\textunderscore{fp_parse_operand}}:Nw \), that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by \( \text{\textunderscore{fp_parse_one}}:Nw \). A first approximation of this function is that it reads one \( \langle \text{number} \rangle \), performing no computation, and finds the following binary \( \langle \text{operator} \rangle \). Then it expands to

\[
\langle \text{number} \rangle \\
\text{\textunderscore{fp_parse_infix}}(\langle \text{operator} \rangle):N \langle \text{precedence} \rangle
\]
expanding the \texttt{infix} auxiliary before leaving the above in the input stream.

We now explain the \texttt{infix} auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as $1-2-3$ being computed as $(1-2)-3$, but $2^3^4$ should be evaluated as $2^{(3^4)}$ instead. For this reason, and to support the equivalence between $\texttt{**}$ and $\texttt{^}$ more easily, each binary operator is converted to a control sequence \texttt{\_\_fp_parse_infix\_\{operator\}:N} when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the \texttt{\langle precedence\rangle} (of the earlier operator) to the \texttt{infix} auxiliary for the following \texttt{\langle operator\rangle}, to know whether to perform the computation of the \texttt{\langle operator\rangle}. If it should not be performed, the \texttt{infix} auxiliary expands to

\begin{verbatim}
\use_none:n \_fp_parse_infix_\{operator\}:N
\end{verbatim}

and otherwise it calls \texttt{\_\_fp_parse_operand:Nw} with the precedence of the \texttt{\langle operator\rangle} to find its second operand \texttt{\langle number\rangle} and the next \texttt{\langle operator\rangle}, and expands to

\begin{verbatim}
\use_none:n \_fp_parse_apply_binary:NwNwN
\__fp_parse_infix_\{operator\}:N\n\end{verbatim}

The \texttt{infix} function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand \texttt{\langle number\rangle} is before the \texttt{infix} function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of \texttt{\_\_fp_parse_operand:Nw \langle precedence\rangle} with some of the expansion control removed is

\begin{verbatim}
\exp_after:wN \_fp_parse_continue:NwNwN
\exp_after:wN \langle precedence\rangle
\exp:w \exp_end_continue_f:w
\_fp_parse_one:Nw \langle precedence\rangle
\end{verbatim}

This expands \texttt{\_\_fp_parse_one:Nw \langle precedence\rangle} completely, which finds a number, wraps the next \texttt{\langle operator\rangle} into an \texttt{infix} function, feeds this function the \texttt{\langle precedence\rangle}, and expands it, yielding either

\begin{verbatim}
\_fp_parse_continue:Nw \langle precedence\rangle
\langle number\rangle \@ \_fp_parse_infix_\{operator\}:N
\end{verbatim}

or

\begin{verbatim}
\_fp_parse_continue:Nw \langle precedence\rangle
\langle number\rangle \@ \_fp_parse_apply_binary:NwNwN
\langle operator\rangle \langle number\rangle
\_fp_parse_infix_\{operator\}:N
\end{verbatim}

The definition of \texttt{\_\_fp_parse_continue:NwN} is then very simple:

\begin{verbatim}
\cs_new:Npn \_fp_parse_continue:NwN { \#1\#2\#3 { \#3 \#1 \#2 \@ } }
\end{verbatim}

In the first case, \texttt{\#3} is \texttt{\use_none:n}, yielding

762
28.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary -, +, !) and parentheses are taken care of by the same mechanism, and functions (\texttt{sin}, \texttt{exp}, etc.) as well. Finding the argument of the unary -, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary -, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function \_\_\_fp\_parse\_one:Nw earlier. This function is responsible for reading in the input stream the first possible \langle\texttt{number}\rangle and the next infix \langle\texttt{operator}\rangle. If what follows \_\_\_fp\_parse\_one:Nw \langle\texttt{precedence}\rangle is a prefix operator, then we must find the operand of this prefix operator through a nested call to \_\_\_fp\_parse\_operand:Nw with the appropriate precedence, then apply the operator to the operand found to yield the result of \_\_\_fp\_parse\_one:Nw. So far, all is simple.

The unary operators +, -, ! complicate things a little bit: \(-3**2\) should be \(-(3^2) = -9\), and not \((-3)^2 = 9\). This would easily be done by giving \(-\) a lower precedence, equal to that of the infix + and -. Unfortunately, this fails in cases such as \(3**-2*4\), yielding \(3^{-2*4}\) instead of the correct \(3^{-2} \times 4\). A second attempt would be to call \_\_\_fp\_parse\_operand:Nw with the \langle\texttt{precedence}\rangle of the previous operator, but \(0**-2+3\) is then parsed as \(0>(-2+3)\): the addition is performed because it binds more tightly than the comparison which precedes -. The correct approach is for a unary \(-\) to perform

\begin{verbatim}
\use_none:n \langle\texttt{precedence}\rangle \langle\texttt{number}\rangle \empty
\_\_fp\_parse\_infix\langle\texttt{operator}\rangle:Nw
then \langle\texttt{number}\rangle \empty \_\_fp\_parse\_infix\langle\texttt{operator}\rangle:Nw
In the second case, \#3 is \_\_fp\_parse\_apply\_binary:NwNwN, whose role is to compute \langle\texttt{number}\rangle \langle\texttt{operator}\rangle \langle\texttt{number}\rangle and to prepare for the next comparison of precedences: first we get
\_\_fp\_parse\_apply\_binary:NwNwN \langle\texttt{precedence}\rangle \langle\texttt{number}\rangle \empty \langle\texttt{operator}\rangle \langle\texttt{number}\rangle \empty \_\_fp\_parse\_infix\langle\texttt{operator}\rangle:Nw
then
\exp_after:wN \_\_fp\_parse\_continue:NwN
\exp_after:wN \langle\texttt{precedence}\rangle \exp:w \exp_end_continue_f:w
\_\_fp\_\langle\texttt{operator}\rangle_o:w \langle\texttt{number}\rangle \langle\texttt{number}\rangle \empty \exp:w \exp_end_continue_f:w \_\_fp\_parse\_infix\langle\texttt{operator}\rangle:Nw
\end{verbatim}
operations whose precedence is greater than both that of the previous operation, and that of the unary - itself. The unary - is given a precedence higher than multiplication and division. This does not lead to any surprising result, since \((-x/y) = (-x)/y\) and similarly for multiplication, and it reduces the number of nested calls to \_\_fp_parse_operand:Nw. Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the infix functions discussed earlier, the prefix functions do perform tests on the previous (precedence) to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

28.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form \langle\text{significand}\rangle e \langle\text{exponent}\rangle, where the \langle\text{significand}\rangle is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent \langle e\langle\text{exponent}\rangle\rangle is optional and is composed of an exponent mark \text{e} followed by a possibly empty string of signs + or − and a non-empty string of decimal digits. The \langle\text{significand}\rangle can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the \langle\text{exponent}\rangle can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as \text{nan}, \text{inf} or \text{pi}. We may add more types in the future.

When \_\_fp_parse_one:Nw is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of \text{scan_stop}, it can be: \text{\s__fp}, in which case our job is done, as what follows is an internal floating point number, or \text{\s__fp_expr_mark}, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from c-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with \text{\tex_the:D}, and use its value (in pt for dimensions and skips, \text{mu} for muskips) as the \langle\text{significand}\rangle of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as \text{asin}, a constant such as \text{pi} or be unknown. In the first case, we call \_\_fp_parse_operand:Nw to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value \text{nan} for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case \_\_fp_parse_operand:Nw finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, \_\_fp_parse_one:Nw also finds an infix operator. This goes as follows.
• If the next token is a control sequence, it could be the special marker `\s__-fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.

• If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.

• Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\__fp_infix⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f: is a digit
\else:
   not a digit
\fi:

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `'#1` lies in `[65, 90]` (uppercase letters) or `[97, 112]` (lowercase letters)

\if_int_compare:w \__fp_int_eval:w ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f: is a letter
\else:
   not a letter
\fi:

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes `{3, 6, 7, 8, 11, 12}` should work without trouble, but not `{1, 2, 4, 10, 13}`, and of course `{0, 5, 9}` cannot become tokens.

Floating point expressions should behave as much as possible like \(\varepsilon\)-\TeX-based integer expressions and dimension expressions. In particular, \(f\)-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop \(f\)-expansion: for instance, the macro `\X` below would not be expanded if we simply performed \(f\)-expansion.

\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then \(f\)-expand it. This is not a complete solution, since a macro’s expansion could contain leading spaces which would stop the \(f\)-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The \(f\)-expansion is performed by `\__fp_parse_expand:w`.  

765
28.2 Main auxiliary functions

\__fp_parse_operand:Nw \exp:w \__fp_parse_operand:Nw \exp:w \__fp_parse_operand:Nw \exp:w

Reads the "...", performing every computation with a precedence higher than \langle precedence \rangle, then expands to

\langle result \rangle \exp:w \__fp_parse_infix\langle operation\rangle:N \exp:w ...

where the \langle operation \rangle is the first operation with a lower precedence, possibly end, and the "..." start just after the \langle operation \rangle.

(End definition for \__fp_parse_operand:Nw.)

\__fp_parse_infix_+\exp:w

\__fp_parse_infix_+\exp:w \__fp_parse_infix_+\exp:w \__fp_parse_infix_+\exp:w

If + has a precedence higher than the \langle precedence \rangle, cleans up a second \langle operand \rangle and finds the \langle operation2 \rangle which follows, and expands to

\langle operand \rangle \__fp_parse_apply_binary:Nw\langle operation \rangle \__fp_parse_infix\langle operation2 \rangle:N \__fp_parse_infix_+\exp:w ...

Otherwise expands to

\__fp_parse_infix_+\exp:w ...

A similar function exists for each infix operator.

(End definition for \__fp_parse_infix_+\exp:w.)

\__fp_parse_one:Nw \exp:w \__fp_parse_one:Nw \exp:w \__fp_parse_one:Nw

Cleans up one or two operands depending on how the precedence of the next operation compares to the \langle precedence \rangle. If the following \langle operation \rangle has a precedence higher than \langle precedence \rangle, expands to

\langle operand \rangle \__fp_parse_apply_binary:Nw\langle operation \rangle \__fp_parse_infix\langle operation2 \rangle:N \__fp_parse_one:Nw ...

and otherwise expands to

\__fp_parse_one:Nw ...

(End definition for \__fp_parse_one:Nw.)

28.3 Helpers

\__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w

This function must always come within a \exp:w expansion. The \langle tokens \rangle should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

\cs_new:Npn \__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w

\cs_new:Npn \__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w \__fp_parse_expand:w \exp:w

This very odd function swaps its position with the following \fi: and removes \__fp_parse_expand:w normally responsible for expansion. That turns out to be useful.

\cs_new:Npn \__fp_parse_return_semicolon:w \exp:w \__fp_parse_return_semicolon:w \exp:w \__fp_parse_return_semicolon:w \exp:w

\cs_new:Npn \__fp_parse_return_semicolon:w \exp:w \__fp_parse_return_semicolon:w \exp:w \__fp_parse_return_semicolon:w

766
These functions must be called within an \int_value:w or __fp_int_eval:w construction. The first token which follows must be f-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

\langle digits \rangle ; \langle filling 0 \rangle ; \langle length \rangle

where \langle filling 0 \rangle is a string of zeros such that \langle digits \rangle \langle filling 0 \rangle has the length given by the index of the function, and \langle length \rangle is the number of zeros in the \langle filling 0 \rangle string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through \token_to_str:N to normalize their category code.

\cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
\cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
{ \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
\else:
\__fp_parse_return_semicolon:w #3 ##1
\fi:
\__fp_parse_expand:w }
\__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 000000 ; 7 }
\__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
\__fp_tmp:w {v} \__fp_parse_digits_iv:N { 000000 ; 5 }
\__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 000000 ; 4 }
\__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000000 ; 3 }
\__fp_tmp:w {ii} \__fp_parse_digits_i:N { 000000 ; 2 }
\__fp_tmp:w {i} \__fp_parse_digits_:N { 000000 ; 1 }
\cs_new:Npn \__fp_parse_one:Nw #1 #2
{ \if_catcode:w \scan_stop: \exp_not:N #2
\exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
\exp_after:wN \reverse_if:N
\exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
\exp_after:wN \__fp_parse_one_fp:NN
\fi:
\__fp_parse_one:NN #3 #1 ; #2 ;
\fi:
\__fp_parse_one:NN #3 #1 ; #2 ;
}
\cs_new:Npn \__fp_parse_one:NN
This function finds one number, and packs the symbol which follows in an \__fp_parse_infix..., csname. #1 is the previous \\lset펜, and #2 the first token of the operand. We distinguish four cases: #2 is equal to \scan_stop: in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by \exp_not:N, as may happen with the \hset펜X2e command \protect. Using a well placed \reverse_if:N, this case is sent to \__fp_parse_one_fp:NN which deals with it robustly.
This function receives a ⟨precedence⟩ and a control sequence equal to \scan_stop: in meaning. There are three cases.

- \__fp starts a floating point number, and we call \__fp_exp_after_f:nw, which f-expands after the floating point.

- \__fp_expr_mark is a premature end, we call \__fp_exp_after_expr_mark_f:nw, which triggers an fp-early-end error.

- For a control sequence not containing \__fp, we call \__fp_exp_after?f:nw, causing a bad-variable error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form \__fp⟨type⟩ and defining \__fp_exp_after_{⟨type⟩}_f:nw. In all cases, we make sure that the second argument of \__fp_parse_infix:NN is correctly expanded. A special case only enabled in \LaTeX2ε is that if \protect is encountered then the error message mentions the control sequence which follows it rather than \protect itself. The test for \LaTeX2ε uses \unexpandable\protect rather than \protect because \protect is often \scan_stop: hence “does not exist”.

(End definition for \__fp_parse_one:Nw.)
This is called whenever \#2 is a control sequence other than \scan_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex_the:D within braces. First, we find the exponent following \#2. Then we unpack \#2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert \textlangle\textlangle value\textrangle\textrangle e\textlangle\textlangle exponent\textrangle\textrangle to a floating point number with
\_fp\_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by \textit{TplX} does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with \texttt{\int_value:w \dim_to_decimal_in_sp:n \{ \langle decimal value \rangle pt \}}, and use an auxiliary of \texttt{\dim_to_fp:n}, which performs the multiplication by $2^{-16}$, correctly rounded.

\begin{verbatim}
\cs_new:Npn \__fp_parse_one_register:NN #1#2
{\exp_after:wN \__fp_parse_infix_after_operand:NwN
 \exp_after:wN #1
\exp:w \exp_end_continue_f:w
\__fp_parse_one_register_special:N #2
\exp_after:wN \__fp_parse_one_register_aux:Nw
\exp:w \__fp_parse_expand:w}
\cs_new:Npx \__fp_parse_one_register_aux:Nw #1
{\exp_not:n {\exp_after:wN \use:nn
\exp_after:wN \__fp_parse_one_register_auxii:wwwNw
\exp_not:N \__fp_parse_one_register_dim:ww
\s__fp_stop}
\exp_args:Nno \use:nn
\{ \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 
\{ \exp_after:wN \use:nn
\exp_after:wN \__fp_parse_one_register_aux:Nw
\{ \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop \}
\exp_after:wN \use:nn
\{ \cs_new:Npn \__fp_parse_one_register_aux:Nw \{ pt \} #3 ; #4#5 \s__fp_stop \}
\exp_after:wN \use:nn
\{ \cs_new:Npn \__fp_parse_one_register_dim:ww #1 ; #2
\{ \tl_to_str:n { pt } ; #2 \}
\cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
\__fp_parse:n { #1 e #3 }
\cs_new:Npn \__fp_parse_one_register_dim:ww #1 ; #2;
\{ \exp_after:wN \__fp_from_dim_test:ww
\exp_after:wN \dim_to_decimal_in_sp:n \{ #1 pt \};
\\int_value:w #2 \exp_after:wN ,
\\int_value:w \dim_to_decimal_in_sp:n \{ #1 pt \};
\}
\}
\end{verbatim}

\textit{(End definition for \_fp\_parse\_one\_register:NN and others.)}

\texttt{\_fp\_parse\_one\_register\_special:N}
\texttt{\_fp\_parse\_one\_register\_math:Nw}
\texttt{\_fp\_parse\_one\_register\_wd:w}
\texttt{\_fp\_parse\_one\_register\_vd:Nw}

The \texttt{\wd, \dp, \ht} primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker \texttt{e}. Once that “exponent” is found, use \texttt{\tex_the:D} to find the box dimension and then copy what we did for dimensions.

770
\cs_new:Npn \__fp_parse_one_register_special:N #1
{
  \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
  \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
  \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
  \if_meaning:w \infty #1 \__fp_parse_one_register_math:NNw \infty #1 \fi:
  \if_meaning:w \pi #1 \__fp_parse_one_register_math:NNw \pi #1 \fi:
}
\cs_new:Npn \__fp_parse_one_register_math:NNw #1#2#3#4 \__fp_parse_expand:w
{
  #3 \str_if_eq:nnTF {#1} {#2}
  {
    \__kernel_msg-expandable_error:nnn
    \{ kernel \} \{ fp-infty-pi \} \{#1\}
    \c_nan_fp
  }
  { #4 \__fp_parse_expand:w }
}
\cs_new:Npn \__fp_parse_one_register_wd:w #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
{
  #1 \exp_after:wN \__fp_parse_one_register_wd:Nw #4 \__fp_parse_expand:w e
}
\cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
{
  \exp_after:wN \__fp_from_dim_test:ww
  \exp_after:wN 0 \exp_after:wN ,
  \int_value:w \dim_to_decimal_in_sp:n \{ #1 #2 \} ;
}
(End definition for \__fp_parse_one_register_special:N and others.)
\__fp_parse_one_digit:NN A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with \__fp_sanitize:wN, then \__fp_parse_infix_after_operand:NwN expands \__fp_parse_infix:NN after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.
\cs_new:Npn \__fp_parse_one_digit:NN #1
{
  \exp_after:wN \__fp_parse_infix_after_operand:NwN
  \exp_after:wN \exp_end_continue_f:w
  \exp_after:wN \__fp_sanitize:wN
  \int_value:w \__fp_int_eval:w \__fp_parse_trim_zeros:N
}
(End definition for \__fp_parse_one_digit:NN.)

771
\_fp_parse_one_other:NN\ For this function, \#2 is a character token which is not a digit. If it is an ASCII letter, \_fp_parse_letters:N beyond this one and give the result to \_fp_parse_word:Nw. Otherwise, the character is assumed to be a prefix operator, and we build \_fp_parse_prefix_(operator):Nw.

\cs_new:Npn \_fp_parse_one_other:NN #1 #2
\begin{verbatim}
  \if_int_compare:w \_fp_int_eval:w \texttt{#2} \texttt{#2 > } \texttt{Z - 32 \fi: } / 26
  = 3 \exp_stop_f:
  \exp_after:wN \_fp_parse_word:Nw
  \exp_after:wN #1
  \exp_after:wN #2
  \exp:w \exp_after:wN \_fp_parse_letters:N
  \exp:w
  \else:
  \exp_after:wN \_fp_parse_prefix:NNN
  \exp_after:wN #1
  \exp_after:wN #2
  \cs:w\_fp_parse_prefix_ \token_to_str:N #2 :Nw
  \exp:w
  \fi:
  \_fp_parse_expand:w
\end{verbatim}

(End definition for \_fp_parse_one_other:NN.)

\_fp_parse_word:Nw\ \_fp_parse_letters:N\ Finding letters is a simple recursion. Once \_fp_parse_letters:N has done its job, we try to build a control sequence from the word \#2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield \c_nan_fp, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every \l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

\cs_new:Npn \_fp_parse_word:Nw #1#2;\begin{verbatim}
\cs_if_exist_use:cF { \_fp_parse_word_#2:N }
\cs_if_exist_use:cF
  { \_fp_parse_caseless_ \str_foldcase:n {#2} :N }
  \_kernel_msg_expandable_error:nnn
    { kernel } { unknown-fp-word } {#2}
\exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
\_fp_parse_infix:NN
\end{verbatim}

(End definition for \_fp_parse_word:Nw.)
\cs_new:Npn \__fp_parse_letters:N #1
{
\exp_end_continue_f:w
\if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1 0 \else:
\__fp_int_eval:w (#1 \if_int_compare:w #1 > 'Z - 32 \fi: ) / 26 \fi:
= 3 \exp_stop_f:
\exp_after:wN #1 \exp:w \exp_after:wN \__fp_parse_letters:N
\else:
\__fp_parse_return_semicolon:w #1 \fi:
\__fp_parse_expand:w
}
(End definition for \__fp_parse_word:Nw and \__fp_parse_letters:N.)
\__fp_parse_prefix:NNN \__fp_parse_prefix_unknown:NNN
For this function, #1 is the previous (precedence), #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put nan, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from \__fp_parse_one:Nw.
\cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
{
\if_meaning:w \scan_stop: #3 \exp_after:wN \__fp_parse_prefix_unknown:NNN \exp_after:wN #2 \fi:
#3 #1
}
\cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
{
\cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
{\_kernel_msg_expansible_error:nnn
 \{ kernel \} \{ fp-missing-number \} \{#1\}
 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
 \__fp_parse_infix:NN #3 #1
}
{\_kernel_msg_expansible_error:nnn
 \{ kernel \} \{ fp-unknown-symbol \} \{#1\}
 \__fp_parse_one:Nw #3
}
(End definition for \__fp_parse_prefix:NNN and \__fp_parse_prefix_unknown:NNN.)
28.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand \( \geq 1 \) with the set of functions \( \__fp_parse_large... \); if it is a period, the significand is \( < 1 \); and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift \( \langle \text{exp} \rangle < 0 \), then read the significand with the set of functions \( \__fp_parse_small... \). Once the significand is read, read the exponent if \( \text{e} \) is present.

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call \( \__fp_parse_large:N \) (the significand is \( \geq 1 \)); if it is \( . \), then continue trimming zeros with \( \__fp_parse_strim_zeros:N \); otherwise, our number is exactly zero, and we call \( \__fp_parse_zero: \) to take care of that case.

\[
\begin{align*}
\texttt{cs_new:Npn} & \quad \__fp_parse_trim_zeros:N \ #1 \\
\texttt{1771} & \quad \{ \\
\texttt{1772} & \quad \texttt{\textbf{if:w} 0 \ exp_not:N \ #1} \\
\texttt{1773} & \quad \texttt{\exp_after:wN} \quad \__fp_parse_trim_zeros:N \\
\texttt{1774} & \quad \texttt{\exp:w} \\
\texttt{1775} & \quad \texttt{\textbf{else:}} \\
\texttt{1776} & \quad \texttt{\textbf{if:w} . \ \exp_not:N \ #1} \\
\texttt{1777} & \quad \texttt{\exp_after:wN} \quad \__fp_parse_strim_zeros:N \\
\texttt{1778} & \quad \texttt{\exp:w} \\
\texttt{1779} & \quad \texttt{\textbf{else:}} \\
\texttt{1780} & \quad \texttt{\textbf{\_fp_parse_trim_end:w} \ #1} \\
\texttt{1781} & \quad \texttt{\textbf{fi:}} \\
\texttt{1782} & \quad \texttt{\textbf{fi:}} \\
\texttt{1783} & \quad \texttt{\textbf{\_fp_parse_expand:w}} \\
\texttt{1784} & \quad \}
\end{align*}
\]

(End definition for \( \__fp_parse_trim_zeros:N \) and \( \__fp_parse_trim_end:w \).

If we have removed all digits until a period (or if the body started with a period), then enter the “small trim” loop which outputs \(-1\) for each removed 0. Those \(-1\) are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call \( \__fp_parse_small:N \) (our significand is smaller than 1), and otherwise, the number is an exact zero. The name \texttt{strim} stands for “small trim”.

\[
\begin{align*}
\texttt{cs_new:Npn} & \quad \__fp_parse_strim_zeros:N \ #1 \\
\texttt{1790} & \quad \{ \\
\texttt{1791} & \quad \texttt{\textbf{if:w} 0 \ \exp_not:N \ #1} \\
\texttt{1792} & \quad - \ 1 \\
\texttt{1793} & \quad \texttt{\exp_after:wN} \quad \__fp_parse_strim_zeros:N \ \\exp:w
\end{align*}
\]
After reading a significand of 0, find any exponent, then put a sign of 1 for \_fp_\_sanitize:w, which removes everything and leaves an exact zero.

\_fp_\_parse_zero: We leave ⟨digits⟩⟨zeros⟩⟨number of zeros⟩ in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack further up.

\_fp_\_parse_small:N This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because \int_value:w (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since \#1 is a digit, read seven more digits using \_fp_\_parse_digits_vii:N. The small_leading auxiliary leaves those digits in the \int_value:w, and grabs some more, or stops if there are no more digits. Then the pack_leading auxiliary puts the various parts in the appropriate order for the processing further up.

We leave (digits) (zeros) in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack
8 more digits, with an exponent shift of zero (this shift is used in the case of a large significant). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significant, then look for an exponent.

```
\cs_new:Npn \__fp_parse_small_leading:wwNN #1 ; #2; #3 #4
{ #1 #2 \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww \exp_after:wN 0 \int_value:w \__fp_int_eval:w 1 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f: \token_to_str:N #4 \exp_after:wN \__fp_parse_small_trailing:wwNN \int_value:w 1 \exp_after:wN \__fp_parse_digits_vi:N \exp:w \else: 0000 0000 \__fp_parse_exponent:Nw #4 \fi: \__fp_parse_expand:w }
```

(End definition for \__fp_parse_small_leading:wwNN.)

```
\__fp_parse_small_trailing:wwNN
\__fp_parse_small_trailing:wwNN 1 (digits) ; (zeros) ; (number of zeros) \(\text{next token}\)
\if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f: \token_to_str:N #4 \exp_after:wN \__fp_parse_small_round:NN \exp_after:wN #4 \exp:w \else: 0 \__fp_parse_exponent:Nw #4 \fi: \__fp_parse_expand:w }
```

(End definition for \__fp_parse_small_trailing:wwNN.)

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The trailing function has an exponent shift as its first argument, which we add to the exponent found in the e... syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this
carry all the way up, the function \_fp\_parse\_pack\_carry:w increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

\begin{verbatim}
\cs_new:Npn \_fp\_parse\_pack\_trailing:NNNNNNww #1 #2 #3 #4 #5 #6 #7; #8 ;
  \{\if_meaning:w 2 #2 + 1 \fi: ; #8 + #1 ; \{#3#4#5#6\} \{#7\};
\}
\cs_new:Npn \_fp\_parse\_pack\_leading:NNNNNww #1 #2 #3 #4 #5 #6; #7;\
  \{#7 + \if_meaning:w 2 #1 \_fp\_parse\_pack\_carry:w \fi: ; 0 \{#2#3#4#5\} \{#6\};\}
\cs_new:Npn \_fp\_parse\_pack\_carry:w \fi: ; 0 #1 {\fi: + 1 ; 0 \{1000\}}
\end{verbatim}

(End definition for \_fp\_parse\_pack\_trailing:NNNNNNww, \_fp\_parse\_pack\_leading:NNNNNww, and \_fp\_parse\_pack\_carry:w)

\subsection{Number: large significand}

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate parse\_small function in those cases.

\begin{verbatim}
\cs_new:Npn \_fp\_parse\_large:N #1\
  \exp_after:wN \_fp\_parse\_large\_leading:wwNN\
  \int_value:w 1 \token_to_str:N #1\
  \exp:w \_fp\_parse\_digits\_vii:N\
  \exp:w \_fp\_parse\_expand:w\
\end{verbatim}

(End definition for \_fp\_parse\_large:N.)

\begin{verbatim}
\_fp\_parse\_large\_leading:wwNN\_fp\_parse\_large\_leading:wwNN 1 \{digits\} ; \{zeros\} ; \{number of zeros\}\next\token\}
\end{verbatim}

We shift the exponent by the number of digits in \#1, namely the target number, 8, minus the \{number of zeros\} (number of digits missing). Then prepare to pack the 8 first digits. If the \{next token\} is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the small functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the \{zeros\} to complete the 8 first digits, insert 8 more, and look for an exponent.

\begin{verbatim}
\cs_new:Npn \_fp\_parse\_large\_leading:wwNN 1 \#1 ; \#2; \#3 \#4\
  \{ + \c\_fp\_half\_prec\_int - \#3\
  \exp_after:wN \_fp\_parse\_pack\_leading:NNNNNww\
\end{verbatim}
We have just read 15 digits. If the \langle\text{next token}\rangle is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the \text{pack_trailing} function. We keep the \langle\text{digits}\rangle and this 16-th digit, and find how this should be rounded using \text{\_\_fp\_parse\_large\_round}. Otherwise, the exponent shift is the number of \langle\text{digits}\rangle, 7 minus the \langle\text{number of zeros}\rangle, and we test for a decimal point. This case happens in 12345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate \text{small} auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the \langle\text{zeros}\rangle and providing a 16-th digit of 0.
\if:w \exp_not:N #4
\exp_after:wN \__fp_parse_small_trailing:wwNN
\int_value:w 1
\cs:w
\__fp_parse_digits_
\__fp_int_to_roman:w #3
:N \exp_after:wN
\cs_end:w
\exp:w
\else:
 #2 0 \__fp_parse_exponent:Nw #4
\fi:
\fi:
\__fp_parse_expand:w }

(End definition for \__fp_parse_large_trailing:wwNN.)

28.4.4 Number: beyond 16 digits, rounding

This loop is called when rounding a number (whether the mantissa is small or large). It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to round up at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

\cs_new:Npn \__fp_parse_round_loop:N #1
{\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f: + 1
\if:w 0 \token_to_str:N #1
\exp_after:wN \__fp_parse_round_loop:N \exp:w
\else:
 \exp_after:wN \__fp_parse_round_up:N \exp:w
\fi:
\fi:
\__fp_parse_return_semicolon:w 0 #1
\fi:
\__fp_parse_expand:w }

\cs_new:Npn \__fp_parse_round_up:N #1
{\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f: + 1
\exp_after:wN \__fp_parse_round_up:N \exp:w
\else:
 \__fp_parse_return_semicolon:w 1 #1
\fi:
\__fp_parse_expand:w }

779
After the loop \_\_fp\_parse\_round\_loop:N, this function fetches an exponent with \_\_fp\_parse\_exponent:N, and combines it with the number of digits counted by \_\_fp\_parse\_round\_loop:N. At the same time, the result 0 or 1 is added to the surrounding integer expression.

\cs_new:Npn \_\_fp\_parse\_round\_after:wN #1; #2
\exp_after:wN \int_value:w \__fp\_int\_eval:w #1 + \_\_fp\_parse\_exponent:N

Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to \langle exponent \rangle only. Otherwise, we expand to +0 or +1, then \langle exponent \rangle. To decide which, call \_\_fp\_round\_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by \_\_fp\_parse\_round\_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by \_\_fp\_parse\_round\_after:wN.

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with \_\_fp\_parse\_round\_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.
28.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.
The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function \texttt{erf} for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after \texttt{e}. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading 3.2, \texttt{\_c\_pi\_fp} is expanded to \texttt{\_s\_fp\_\_fp\_chk:w 1 0 {-1} \{3141\} \cdots}; and \texttt{\_s\_fp} stops the expansion. Expanding two tokens ahead would then force the expansion of \texttt{\_fp\_chk:} (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as \TeX{} does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is \texttt{e}. Then we expand (and possibly unpack) the second token.

\begin{verbatim}
\__fp_parse_exponent:Nw \texttt{This auxiliary is convenient to smuggle some material through \texttt{\_fi:} ending conditional processing. We place those \texttt{\_fi:} (argument \#2) at a very odd place because this allows us to insert \texttt{\_fp\_int\_eval:w} \ldots there if needed.}
\begin{verbatim}
17975 \cs_new:Npn \__fp_parse_exponent:Nw \#1 \#2 \__fp_parse_expand:w
17976 \{ \exp_after:wN ; \int_value:w \#2 \__fp_parse_exponent:N \#1 \}
\end{verbatim}
\end{verbatim}
\texttt{(End definition for \texttt{\_fp\_parse\_exponent:Nw}.)}
\end{verbatim}

\begin{verbatim}
\__fp_parse_exponent:N \__fp_parse_exponent_aux:N \texttt{This function should be called within an \texttt{\_int\_value:w} expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no \texttt{e} (or \texttt{E}), leave an exponent of 0. If there is an \texttt{e} or \texttt{E}, expand the next token to run some tests on it. The first rough test is that if the character code of \#1 is greater than that of \#9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.}
\begin{verbatim}
17981 \cs_new:Npn \__fp_parse_exponent:N \#1
17982 \{
17983 \if:w e \if:w E \exp_not:N \#1 \@ \else: \exp_not:N \#1 \fi:
17984 \exp_after:wN \__fp_parse_exponent_aux:N
17985 \exp:w
17986 \else:
17987 0 \__fp_parse_return_semicolon:w \#1
17988 \fi:
17989 \__fp_parse_expand:w
17990 \}
17991 \cs_new:Npn \__fp_parse_exponent_aux:N \#1
17992 \{
17993 \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N \#1
17994 0 \else: '9 \if:w \fi: > '9 \exp_stop_f:
17995 0 \exp_after:wN ; \exp_after:wN e
17996 \else:
17997 \end{verbatim}
782
\_\_fp\_parse\_exponent\_sign:N Read signs one by one (if there is any).
\cs_new:Npn \_\_fp\_parse\_exponent\_sign:N #1
{\if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1 \exp_after:wN \_\_fp\_parse\_exponent\_sign:N
\exp:w \exp_after:wN \_\_fp\_parse\_expand:w \else:
\exp_after:wN \_\_fp\_parse\_exponent\_body:N \_\_fp\_parse\_body:w #1
\exp_after:wN ; \exp:w \fi:
}\__fp\_parse\_expand:w

(End definition for \_\_fp\_parse\_exponent\_sign:N.)

\_\_fp\_parse\_exponent\_body:N An exponent can be an explicit integer (most common case), or various other things
(most of which are invalid).
\cs_new:Npn \_\_fp\_parse\_exponent\_body:N #1
{\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
\token_to_str:N #1 \exp_after:wN \_\_fp\_parse\_exponent\_digits:N \_\_fp\_parse\_digits:w
\exp:w \else:
\_\_fp\_parse\_exponent\_keep:NTF #1 {\_\_fp\_parse\_return\_semicolon:w #1 }
{ \_\_fp\_parse\_return\_semicolon:w #1 }
\exp_after:wN ; \exp:w \fi:
\_\_fp\_parse\_expand:w }

(End definition for \_\_fp\_parse\_exponent\_body:N.)

\_fp\_parse\_exponent\_digits:N Read digits one by one, and leave them behind in the input stream. When finding a
non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the
exponent, hence there can be a \TeX\ error. It is mostly harmless, except when parsing
0e9876543210, which should be a valid representation of 0, but is not.
\cs_new:Npn \_\_fp\_parse\_exponent\_digits:N #1
{\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
\token_to_str:N #1 \exp_after:wN \_\_fp\_parse\_exponent\_digits:N \_\_fp\_parse\_digits:w
\exp:w \else:
\_\_fp\_parse\_return\_semicolon:w #1 }

783
This is the last building block for parsing exponents. The argument \#1 is already fully expanded, and neither + nor − nor a digit. It can be:

- \s__fp, marking the start of an internal floating point, invalid here;
- another control sequence equal to \relax, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than +, − or digits, again, an error.

\prg_new_conditional:Npn \__fp_parse_exponent_keep:N \#1 { TF }
{ \if_catcode:w \scan_stop: \exp_not:N \#1
  \if_meaning:w \scan_stop: \#1
  \if_int_compare:w
    \__fp_str_if_eq:nn { \s__fp } { \exp_not:N \#1 } = 0 \exp_stop_f:
    0
    \__kernel_msg_expandable_error:nnn
    { kernel } { fp-after-e } { floating-point- }
    \prg_return_true:
  \else:
    0
    \__kernel_msg_expandable_error:nnn
    { kernel } { bad-variable } {\#1}
    \prg_return_false:
  \fi:
  \else:
    \if_int_compare:w
      \__fp_str_if_eq:nn { \int_value:w \#1 } { \tex_the:D \#1 } = 0 \exp_stop_f:
      \int_value:w \#1
      \else:
        0
        \__kernel_msg_expandable_error:nnn
        { kernel } { fp-missing } { exponent }
        \prg_return_true:
      \fi:
      \prg_return_false:
    \fi:
    \else:
      0
      \__kernel_msg_expandable_error:nnn
      { kernel } { fp-missing } { exponent }
      \prg_return_true:
    \fi:
  \else:
    0
    \__kernel_msg_expandable_error:nnn
    { kernel } { fp-after-e } { dimension-\#1 }
    \prg_return_true:
  \fi:
}
28.5 Constants, functions and prefix operators

28.5.1 Prefix operators

\__fp_parse_prefix_+:Nw

A unary + does nothing; we should continue looking for a number.

(End definition for \__fp_parse_prefix_+:Nw.)

\__fp_parse_apply_function:NNNw

Here, \#1 is a precedence, \#2 is some extra data used by some functions, \#3 is e.g., \__fp_sin_o:w, and expands once after the calculation, \#4 is the operand, and \#5 is a \__fp_parse_infix_...:N function. We feed the data \#2, and the argument \#4, to the function \#3, which expands \exp:w thus the infix function \#5.

(End definition for \__fp_parse_apply_function:NNNw.)

\__fp_parse_apply_unary:NNNw
\__fp_parse_apply_unary_chk:NwNw
\__fp_parse_apply_unary_type:NNN
\__fp_parse_apply_unary_error:Nw

In contrast to \__fp_parse_apply_function:NNNw, this checks that the operand \#4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like \s__fp. If there is no argument produce the fp-no-arg error; if there are at least two produce fp-multi-arg. For the error message extract the mathematical function name (such as \textit{sin}) from the expl3 function that computes it, such as \__fp_sin_o:w.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like \textit{sin}((1,2)) where it does not make sense to take the sine of a tuple.

(End definition for \__fp_parse_apply_unary:NNNw.)
The unary `−` and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence `#1` and the precedence `\c__fp_prec_not_int` of the unary operator, then call the appropriate `\_fp_⟨operation⟩_o:w` function, where the `⟨operation⟩` is `set_sign` or `not`.

\[\__fp_parse_prefix_-:Nw\]
\[\__fp_parse_prefix_!:Nw\]

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to `\_fp_parse_one_digit:NN` but calls `\_fp_parse_strim_zeros:N` to trim zeros after the decimal point, rather than the `trim_zeros` function for zeros before the decimal point.

\[\__fp_parse_prefix_.:Nw\]

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is `\c__fp_prec_func_int` we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: `\c__fp_prec_comma_int` for the case of arguments, `\c__fp_prec_tuple_int` for the case of tuples. Once the operand
is found, the lparen_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

\begin{verbatim}
\cs_new:cpn { __fp_parse_prefix_(:Nw } #1
\exp_after:wN \__fp_parse_lparen_after:NwN
\exp_after:wN #1
\exp:w
\if_int_compare:w #1 = \c__fp_prec_func_int
 \__fp_parse_operand:Nw \c__fp_prec_comma_int
\else:
 \__fp_parse_operand:Nw \c__fp_prec_tuple_int
\fi:
\__fp_parse_expand:w
\cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
\exp_not:N \token_if_eq_meaning:NNTF #3 \exp_not:c { __fp_parse_infix_):N }
{ \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
 \exp_not:N \exp_after:wN \exp_not:N \__fp_parse_infix_after_paren:NN
 \exp_not:N \exp_after:wN #1
 \exp_not:N \exp:w \exp_not:N \__fp_parse_expand:w
}
{ \exp_not:N \__kernel_msg_expandable_error:nnn
 { kernel } { fp-missing } { ) }
 \exp_after:wN \c_nan_fp \exp:w
 \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
 \exp_not:N \use_none:n #3
}
\end{verbatim}

(End definition for \__fp_parse_prefix_(:Nw and \__fp_parse_lparen_after:Ww)

\__fp_parse_prefix_):Ww The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in max(1,2,) or in rand().

\begin{verbatim}
\cs_new:cpn { __fp_parse_prefix_):Nw } #1
\if_int_compare:w #1 = \c__fp_prec_comma_int
\else:
 \if_int_compare:w #1 = \c__fp_prec_tuple_int
 \exp_after:wN \c__fp_empty_tuple_fp \exp:w
\else:
 \__kernel_msg_expandable_error:nnn
 { kernel } { fp-missing-number } { ) }
 \exp_after:wN \c_nan_fp \exp:w
\fi:
\exp_end_continue_f:w
\f1:
\end{verbatim}

787
28.5.2 Constants

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```latex
\begin{align*}
\text{\texttt{\__fp_parse_word_inf:N}} &\quad \text{\texttt{\__fp_parse_word_nan:N}} \\
\text{\texttt{\__fp_parse_word_pi:N}} &\quad \text{\texttt{\__fp_parse_word_deg:N}} \\
\text{\texttt{\__fp_parse_word_true:N}} &\quad \text{\texttt{\__fp_parse_word_false:N}}
\end{align*}
```

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```latex
\cs_set_protected:Npn \__fp_tmp:w #1 #2 
\{
\cs_new:cpn { \__fp_parse_word_#1:N } { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN } 
\}
\__fp_tmp:w \{ inf \} \c_inf_fp 
\__fp_tmp:w \{ nan \} \c_nan_fp 
\__fp_tmp:w \{ pi \} \c_pi_fp 
\__fp_tmp:w \{ deg \} \c_one_degree_fp 
\__fp_tmp:w \{ true \} \c_one_fp 
\__fp_tmp:w \{ false \} \c_zero_fp
```

Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```latex
\begin{align*}
\text{\texttt{\__fp_parse_word_pt:N}} &\quad \text{\texttt{\__fp_parse_word_in:N}} \\
\text{\texttt{\__fp_parse_word_pc:N}} &\quad \text{\texttt{\__fp_parse_word_cm:N}} \\
\text{\texttt{\__fp_parse_word_mm:N}} &\quad \text{\texttt{\__fp_parse_word_dd:N}} \\
\text{\texttt{\__fp_parse_word_cc:N}} &\quad \text{\texttt{\__fp_parse_word_nd:N}} \\
\text{\texttt{\__fp_parse_word_nc:N}} &\quad \text{\texttt{\__fp_parse_word_bp:N}} \\
\text{\texttt{\__fp_parse_word_sp:N}}
\end{align*}
```

```latex
\ cs_set_protected:Npm \__fp_tmp:w \{ inf \} \c_inf_fp 
\__fp_tmp:w \{ nan \} \c_nan_fp 
\__fp_tmp:w \{ pi \} \c_pi_fp 
\__fp_tmp:w \{ deg \} \c_one_degree_fp 
\__fp_tmp:w \{ true \} \c_one_fp 
\__fp_tmp:w \{ false \} \c_zero_fp
\cs_set_protected:Npn \__fp_tmp:w \{ pt \} \{ \begin{array}{c} 
\{ 1 \} \{ 1000 \} \{ 0000 \} \{ 0000 \} \{ 0000 \} \\
\{ 2 \} \{ 7227 \} \{ 0000 \} \{ 0000 \} \{ 0000 \} \\
\{ 2 \} \{ 1200 \} \{ 0000 \} \{ 0000 \} \{ 0000 \} \\
\{ 2 \} \{ 2845 \} \{ 2755 \} \{ 9055 \} \{ 1181 \} \\
\{ 1 \} \{ 2845 \} \{ 2755 \} \{ 9055 \} \{ 1181 \} \\
\{ 1 \} \{ 1070 \} \{ 0085 \} \{ 6496 \} \{ 0630 \} \\
\{ 2 \} \{ 1284 \} \{ 0102 \} \{ 7795 \} \{ 2756 \} \\
\{ 1 \} \{ 1066 \} \{ 9783 \} \{ 4645 \} \{ 6693 \} \\
\{ 2 \} \{ 1280 \} \{ 3740 \} \{ 1574 \} \{ 8031 \} \\
\{ 1 \} \{ 1003 \} \{ 7500 \} \{ 0000 \} \{ 0000 \} \\
\{ -4 \} \{ 1525 \} \{ 8789 \} \{ 0625 \} \{ 0000 \}
\end{array} \end{align*}
```

(End definition for `\__fp_parse_word_pt:N` and others.)

788
The font-dependent units \textit{em} and \textit{ex} must be evaluated on the fly. We reuse an auxiliary of \texttt{dim_to_fp:n}.

\begin{verbatim}
\tl_map_inline:nn { {em} {ex} }
{\cs_new:cpn { __fp_parse_word_#1:N }{
\exp_after:wN \__fp_from_dim_test:ww
\exp_after:wN 0 \exp_after:wN ,
\int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
\exp:w \exp_end_continue_f:w \__fp_parse_infix:NN}
}
\end{verbatim}

(End definition for \texttt{__fp_parse_word_em:n} and \texttt{__fp_parse_word_ex:N}.)

### 28.5.3 Functions

\begin{verbatim}
\cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3 {
\exp_after:wN \__fp_parse_apply_unary:NNNwN
\exp_after:wN #3 \exp_after:wN #2 \exp_after:wN #1 \exp:w
\__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
}
\cs_new:Npn \__fp_parse_function:NNN #1#2#3 {
\exp_after:wN \__fp_parse_apply_function:NNNwN
\exp_after:wN #3 \exp_after:wN #2 \exp_after:wN #1 \exp:w
\__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
}
\end{verbatim}

(End definition for \texttt{__fp_parse_unary_function:NNN} and \texttt{__fp_parse_function:NNN}.)

### 28.6 Main functions

Start an \texttt{exp:w} expansion so that \texttt{__fp_parse:n} expands in two steps. The \texttt{__fp_parse_operand:N} function performs computations until reaching an operation with precedence \texttt{c__fp_prec_end_int} or less, namely, the end of the expression. The marker \texttt{s__fp_expr_mark} indicates that the next token is an already parsed version of an infix operator, and \texttt{__fp_parse_infix_end:N} has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with \texttt{exp_end:}.

\begin{verbatim}
\cs_new:Npn \__fp_parse:n #1
{ \exp:w \exp_after:wN \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
\exp:w}
\cs_new:Npn \__fp_parse_o:n
{ \exp:w \exp_after:wN \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
\exp:w}
\cs_new:Npn \__fp_parse_after:ww
{ \exp:w \exp_after:wN \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
\exp:w}
\end{verbatim}

789
\__fp_parse_operand:Nw \c\_fp_prec_end_int
\__fp_parse_expand:w \#1
\s\_fp_expr_mark \__fp_parse_infix_end:N
\s\_fp_expr_stop
\exp_end:
\}
\cs_new:Npn \__fp_parse_after:ww
#1@ \__fp_parse_infix_end:N \s\_fp_expr_stop #2 { #2 #1 }
\cs_new:Npn \__fp_parse_o:n #1
{
\exp:w \exp_after:wN \__fp_parse_continue:NwN \exp_after:wN #1 \exp:w \exp_end_continue_f:w
\exp_after:wN \__fp_parse_one:Nw \exp_after:wN #1 \exp:w
\}
\exp:w \exp_after:wN \__fp_parse_after:ww
\exp:w \__fp_parse_operand:Nw \c\_fp_prec_end_int
\__fp_parse_expand:w \#1
\s\_fp_expr_mark \__fp_parse_infix_end:N
\s\_fp_expr_stop
\}
\exp_end_continue_f:w
\__fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
\}
\}
(End definition for \__fp_parse:o, \__fp_parse_o:n, and \__fp_parse_after:ww.)

This is just a shorthand which sets up both \__fp_parse_continue:NwN and \__fp_parse_one:Nw with the same precedence. Note the trailing \exp:w.
\cs_new:Npn \__fp_parse_operand:Nw \#1
{\exp_end_continue_f:w \exp_after:wN \__fp_parse_continue:NwN \exp_after:wN \__fp_parse_one:Nw \exp_after:wN \__fp_parse_one:Nw \exp:w \exp:w \exp_end_continue_f:w \exp_after:wN \__fp_parse_one:Nw \exp_after:wN \__fp_parse_one:Nw \exp:w \exp:w
\}
\cs_new:Npn \__fp_parse_continue:NwN \#1 \#2 \#3 \#4 \#5 \#6 \#7
{\exp_after:wN \__fp_type_from_scan:N \#2
(End definition for \__fp_parse_operand:Nw and \__fp_parse_continue:Nw.)

Receives \langle precedence\rangle \langle operand 1\rangle \circ \langle operation\rangle \langle operand 2\rangle \circ \langle infix command\rangle. Builds the appropriate call to the \langle operation\rangle \#3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.
\cs_new:Npn \__fp_parse_apply_binary:NwNwN \#1 \#2 \#3 \#4 \#5 \#6 \#7
{ \exp_after:wN \__fp_parse_apply_binary:NwNwN \exp_after:wN \__fp_parse_apply_binary:NwNwN \exp_after:wN \__fp_parse_apply_binary:NwNwN \exp_after:wN \__fp_parse_apply_binary:NwNwN \exp:w \exp:w \exp:w \exp:w \exp:w \exp:w \exp:w \exp:w
\}
(End definition for \__fp_parse_apply_binary:NwN and \__fp_parse_apply_binary:NwN.)

790
Applies the operator \#1 to its two arguments, dispatching according to their types, and expands once after the result. The \texttt{rev} version swaps its arguments before doing this.

(End definition for \texttt{\_fp_binary_type_o:Nww}, \texttt{\_fp_binary_rev_type_o:Nww}, \texttt{\_fp_parse_apply_binary:NN}, \texttt{\_fp_parse_apply_binary_chk:NN}, and \texttt{\_fp_parse_apply_binary_error:NNN}.)
28.7 Infix operators

\cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
  {
    \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 } #2;
  }
\cs_new:Npn \__fp_parse_infix:NN #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \if_int_compare:w
        \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
        = 0 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_mark:NNN
      \else:
      \exp_after:wN \exp_after:wN
      \exp_after:wN \__fp_parse_infix_juxt:N
      \fi:
    \else:
      \if_int_compare:w
        \__fp_int_eval:w
        ( #2 \if_int_compare:w #2 > 'Z - 32 \fi: ) / 26
        = 3 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_juxt:N
      \else:
        \exp_after:wN \__fp_parse_infix_check:NNN
        \cs:w
        \__fp_parse_infix_ \token_to_str:N #2 :N
        \exp_after:wN \exp_after:wN
        \exp_after:wN \cs_end:
      \fi:
    \fi:
    #1 #2 }
\cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
  {
    \if_meaning:w \scan_stop: #1
      \__kernel_msg_expandable_error:nnn
      { kernel } { fp-missing } { * }
      \exp_after:wN \__fp_parse_infix_mul:N
      \exp_after:wN #2
      \exp:w \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN \exp_after:wN
      \exp_after:wN \__fp_parse_infix_juxt:N
      \cs:w
      \__fp_parse_infix_ \token_to_str:N #2 :N
      \exp_after:wN \exp_after:wN
      \exp_after:wN \cs_end:
    \fi:
  }
(End definition for \__fp_parse_infix_after_operand:NwN.)
Variant of \_\_fp_parse_infix:NN for use after a closing parenthesis. The only difference is that \_\_fp_parse_infix_juxt:N is replaced by \_\_fp_parse_infix_mul:N.

\cs_new:Npn \_\_fp_parse_infix_after_paren:NN #1 #2
\begin{Verbatim}
\if_catcode:w \scan_stop: \exp_not:N #2
\if_int_compare:w
\__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
= 0 \exp_stop_f:
\exp_after:wN \exp_after:wN
\exp_after:wN \__fp_parse_infix_mark:NNN
\else:
\exp_after:wN \exp_after:wN
\exp_after:wN \__fp_parse_infix_mul:N
\fi:
\else:
\if_int_compare:w
\__fp_int_eval:w ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
= 3 \exp_stop_f:
\exp_after:wN \exp_after:wN
\exp_after:wN \__fp_parse_infix_mul:N
\else:
\exp_after:wN \__fp_parse_infix_check:NNN \cs:w
\_\_fp_parse_infix_ \token_to_str:N #2 :N
\exp_after:wN \exp_after:wN \exp_after:wN
\cs_end:
\fi:
#1
#2
\end{Verbatim}
\end{Verbatim}

(End definition for \_\_fp_parse_infix_after_paren:NN.)

\_\_fp_parse_infix_mark:NNN
As an infix operator, \_\_fp_expr_mark means that the next token (#3) has already gone through \_\_fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

\cs_new:Npn \_\_fp_parse_infix_end:N #1
\begin{Verbatim}
\@ \use_none:n \_\_fp_parse_infix_end:N
\end{Verbatim}

(End definition for \_\_fp_parse_infix_end:N.)

\_\_fp_parse_infix_end:N
This one is a little bit odd: force every previous operator to end, regardless of the precedence.

\cs_new:Npn \_\_fp_parse_infix_end:N #1
\begin{Verbatim}
\{ \@ \use_none:n \_\_fp_parse_infix_end:N \}
\end{Verbatim}

(End definition for \_\_fp_parse_infix_end:N.)

\_\_fp_parse_infix_end:NN
This is very similar to \_\_fp_parse_infix_end:N, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence \c__fp_prec_end_int.

793
As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call \_fp\_parse\_operand:Nw to read more comma-delimited arguments that \_fp\_parse\_infix\_comma:w simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call \_fp\_parse\_apply\_comma:Nw\_w\_w\_w whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to \_fp\_parse\_apply\_\_binary:Nw\_w\_w\_w this function’s operands are not single-object arrays.
As described in the “work plan”, each infix operator has an associated \..._infix... function, a computing function, and precedence, given as arguments to \__fp_tmp:w. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
\cs_new:Npn #1 ##1
\if_int_compare:w ##1 < #3
\exp_after:wN @\exp_after:wN \__fp_parse_apply_binary:NwNwN\exp_after:wN #2\exp:w\__fp_parse_operand:Nw #4\exp_after:wN \__fp_parse_expand:w
\else:\exp_after:wN @\exp_after:wN \use_none:n\exp_after:wN #1\fi:
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N } ^
\c__fp_prec_hatii_int \c__fp_prec_hat_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_juxt:N } *
\c__fp_prec_juxt_int \c__fp_prec_juxt_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:-N } /
\c__fp_prec_times_int \c__fp_prec_times_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
\c__fp_prec_times_int \c__fp_prec_times_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_-:N } -
\c__fp_prec_plus_int \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_+:N } +
\c__fp_prec_plus_int \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
\c__fp_prec_and_int \c__fp_prec_and_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N } |
\c__fp_prec_or_int \c__fp_prec_or_int

(End definition for \__fp_parse_infix_+:N and others.)
28.7.3 Juxtaposition

\__fp_parse_infix_(:N

When an opening parenthesis appears where we expect an infix operator, we compute
the product of the previous operand and the contents of the parentheses using \__fp_
parse_infix_mul:N.

\cs_new:cpn { \__fp_parse_infix_(:N } #1 \\{ \\__fp_parse_infix_mul:N #1 \ ( \ )

(End definition for \__fp_parse_infix_(:N.)

28.7.4 Multi-character cases

\__fp_parse_infix_*:N

\cs_set_protected:Npn \__fp_tmp:w #1 \\{ \cs_new:cpn { \__fp_parse_infix_*:N } ##1##2 \\{ \if:w * \exp_not:N ##2 \exp_after:wN #1 \exp_after:wN ##1 \exp:w \exp_after:wN \__fp_parse_expand:w \else: \exp_after:wN #3 \exp_after:wN ##1 \exp_after:wN ##2 \fi: \} \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_^:N }

(End definition for \__fp_parse_infix_*:N)

\__fp_parse_infix_|:Nw \__fp_parse_infix_&:Nw

\cs_set_protected:Npn \__fp_tmp:w #1#2#3 \\{ \cs_new:Npn #1 ##1##2 \\{ \if:w #2 \exp_not:N ##2 \exp_after:wN #1 \exp_after:wN ##1 \exp:w \exp_after:wN \__fp_parse_expand:w \else: \exp_after:wN #3 \exp_after:wN ##1 \exp_after:wN ##2 \fi: \} \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_|:N } | \__fp_parse_infix_or:N \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_&:N } & \__fp_parse_infix_and:N

(End definition for \__fp_parse_infix_|:Nw and \__fp_parse_infix_&:Nw.)
28.7.5 Ternary operator

\_fp_parse_infix:_?N
\_fp_parse_infix::_N
\cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
\cs_new:Npn #1 ##1
\if_int_compare:w ##1 < \c\_fp_prec_quest_int
\exp_after:wN @
\exp_after:wN #2
\exp:w
\_fp_parse_operand:Nw #3
\exp_after:wN \_fp_parse_expand:w
\else:
\exp_after:wN @
\exp_after:wN \use_none:n
\exp_after:wN #1
\fi:
\exp_args:Nc \_fp_tmp:w { \_fp_parse_infix:_?N }
\_fp_ternary:NwwN \c\_fp_prec_quest_int { }
\exp_args:Nc \_fp_tmp:w { \_fp_parse_infix:_::N }
\_fp_ternary_auxi:NwwN \c\_fp_prec_colon_int { }
\_kernel_msg_expansible_error:nnnn
{ kernel } { fp-missing } { ? } { ~for~?: }
(End definition for \_fp_parse_infix:_?N and \_fp_parse_infix:_::N.)

28.7.6 Comparisons

\_fp_parse_infix_<:N
\_fp_parse_infix_=:N
\_fp_parse_infix_>!:N
\_fp_parse_excl_error:
\_fp_parse_compare:NNNNNN
\_fp_parse_compare_auxi:NNNNNN
\_fp_parse_compare_end:NNNN
\_fp_compare:wNNNNw
\cs_new:cpn { \_fp_parse_infix_<:N } #1
\cs_new:cpn { \_fp_parse_compare:NNNNNN #1 1 0 0 0 < } #1
\cs_new:cpn { \_fp_parse_infix_=:N } #1
\cs_new:cpn { \_fp_parse_compare:NNNNNN #1 1 0 0 0 = } #1
\cs_new:cpn { \_fp_parse_infix_:>:N } #1
\cs_new:cpn { \_fp_parse_compare:NNNNNN #1 1 0 0 0 > } #1
\cs_new:cpn { \_fp_parse_infix_!:N } #1
\exp_after:wN \_fp_parse_compare:NNNNNN
\exp_after:wN #1
\exp_after:wN #1
\exp_after:wN #1
\exp_after:wN #1
\exp_after:wN #1
\exp_after:wN \_fp_parse_excl_error:
{ \_kernel_msg_expansible_error:nnnn

797
\cs_new:Npn \__fp_parse_compare:NNNNNNN #1
\{
\if_int_compare:w #1 < \c__fp_prec_comp_int
\exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
\exp_after:wN \__fp_parse_excl_error:
\else:
\exp_after:wN @
\exp_after:wN \use_none:n
\exp_after:wN \__fp_parse_compare:NNNNNNN
\fi:
\}
\cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
\{
\if_case:w
\__fp_int_eval:w \exp_after:wN ' \token_to_str:N #7 - '<
\__fp_int_eval_end:
\or: \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
\or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
\or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
\else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
\fi:
\}
\cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
\{
\exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
\exp_after:wN \prg_do_nothing:
\exp_after:wN \exp_after:wN #1
\exp_after:wN #2
\exp_after:wN #3
\exp_after:wN #4
\exp_after:wN #5
\exp:w \exp_after:wN \__fp_parse_expand:w
\}
\cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
\{
\fi:
\exp_after:wN @
\exp_after:wN \__fp_parse_apply_compare:NwNNNNN
\exp_after:wN \c_one_fp
\exp_after:wN #1
\exp_after:wN #2
\exp_after:wN #3
\exp_after:wN #4
\exp:w \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
\}
\cs_new:Npn \__fp_parse_apply_compare:NwNNNNN \#1 \#2 \#3 \#4\#5\#6 \#7 \#8 \#9
\{
\if_int_odd:w
\if_meaning:w \c_zero_fp \#3
\exp:w \c_exp:w \c__fp_prec_comp_int \__fp_parse_expand:w \#5
\}
\if_int_odd:w
\if_meaning:w \c_zero_fp \#3
\exp:w
28.8 Tools for functions

\_\_fp\_parse\_function\_all\_fp\_o:\texttt{fnw}

Followed by \{\texttt{function name}\} \{\texttt{code}\} \{\texttt{float array}\} \@ this checks all floats are floating point numbers (no tuples).

\cs_new:Npn \_\_fp\_parse\_function\_all\_fp\_o:\texttt{fnw} #1#2#3 @

\{ #2 #3 @ \}

\{ \_\_fp\_array\_if\_all\_fp:nTF \#3 \}

\{ #2 #3 @ \}

\{ \_\_fp\_error:nffn \{ \_\_fp\_bad\_args \} \}

\{ \_\_fp\_to\_tl:n \{ \_\_fp\_tuple \_\_\_fp\_tuple\_chk:w \#3 \} \}

\{ \}

\exp_after:wN \c_nan_fp

\exp_after:wN \c_zero_fp

(End definition for \_\_fp\_parse\_infix<:\texttt{N} and others.)
This is followed by \{\langle function name\rangle\} \{\langle code\rangle\} \langle float array\rangle \ 0. It checks that the \langle float array\rangle consists of one or two floating point numbers (not tuples), then leaves the \langle code\rangle (if there is one float) or its tail (if there are two floats) followed by the \langle float array\rangle. The \langle code\rangle should start with a single token such as \_\_fp_atan_default:w that deals with the single-float case.

The first \_\_fp_if_type_fp:NTwFw test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add \c_one_fp) from a tuple second argument. Finally check there is no further argument.

Apply \#1 to all items in the following tuple and expand once afterwards. The code \#1 should itself expand once after its result.

(End definition for \_\_fp_parse_function_one_two:nnw and others.)
Apply \#1 to pairs of items in the two following tuples and expand once afterwards.

(End definition for \_\_fp_tuple_map_loop_o:nw and \_\_fp_tuple_mapthread_o:nww.)
Floating point variables are initialized to be +0.

```latex
\fp_new:N
\fp_set:Nn
\fp_gset:Nn
\fp_const:Nn
```

Simply use \_\_fp_parse:n within various f-expanding assignments.

```latex
\cs_if_exist:cT { @unexpandable@protect }
\__kernel_msg_new:nnn { kernel } { fp-robust-cmd }
\{ Robust-command-#1 invalid-in-fp-expression! \}
\}
```

29 \textbf{l3fp-assign implementation}

29.1 Assigning values

```
\cs_new_protected:Npn \fp_new:N #1
\cs_generate_variant:Nn \fp_new:N {c}
\cs_generate_variant:Nn \fp_gset:Nn {c}
\cs_generate_variant:Nn \fp_const:Nn {c}
```

(End definition for \fp_new:N. This function is documented on page 199.)
Copying a floating point is the same as copying the underlying token list.

Setting a floating point to zero: copy \c_zero_fp.

Set the floating point to zero, or define it if needed.

For the sake of error recovery we should not simply set #1 to #1±(#2): for instance, if #2 is 0+2, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting parentheses. As an optimization we use \__fp_parse:n rather than \fp_eval:n, which would convert the result away from the internal representation and back.
29.3 Showing values

This shows the result of computing its argument by passing the right data to `\tl_show:n` or `\tl_log:n`.

\begin{verbatim}
\cs_new_protected:Npn \fp_show:N { \__fp_show:NN \tl_show:n }
\cs_generate_variant:Nn \fp_show:N { c }
\cs_new_protected:Npn \fp_log:N { \__fp_show:NN \tl_log:n }
\cs_generate_variant:Nn \fp_log:N { c }
\cs_new_protected:Npn \__fp_show:NN #1#2
{ \__kernel_chk_defined:NT #2
{ \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
}
\end{verbatim}

(End definition for `\fp_show:N`, `\fp_log:N`, and `\__fp_show:NN`. These functions are documented on page 207.)

\begin{verbatim}
\cs_new_protected:Npn \fp_show:n { \msg_show_eval:Nn \fp_to_tl:n }
\cs_new_protected:Npn \fp_log:n { \msg_log_eval:Nn \fp_to_tl:n }
\end{verbatim}

(End definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 207.)

29.4 Some useful constants and scratch variables

Some constants.

\begin{verbatim}
\fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
\fp_const:Nn \c_one_fp { 1 }
\end{verbatim}

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 205.)

\begin{verbatim}
\fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
\end{verbatim}

(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 206.)

\begin{verbatim}
\fp_new:N \l_tmpa_fp
\fp_new:N \l_tmpb_fp
\fp_new:N \g_tmpa_fp
\fp_new:N \g_tmpb_fp
\end{verbatim}

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 206.)

⟨/initex | package⟩
30 l3fp-logic Implementation

Those functions may receive a variable number of arguments.

\__fp_parse_word_max:N
\__fp_parse_word_min:N

(End definition for __fp_parse_word_max:N and __fp_parse_word_min:N.)

30.1 Syntax of internal functions

- \__fp_compare_npos:nwnw {⟨expo1⟩} ⟨body1⟩ ; {⟨expo2⟩} ⟨body2⟩ ;
- \__fp_minmax_o:Nw ⟨sign⟩ ⟨floating point array⟩
- \__fp_not_o:w ? ⟨floating point array⟩ (with one floating point number only)
- \__fp_&_o:ww ⟨floating point⟩ ⟨floating point⟩
- \__fp_|_o:ww ⟨floating point⟩ ⟨floating point⟩
- \__fp_ternary:NwwN, \__fp_ternary_auxi:NwwN, \__fp_ternary_auxii:NwwN have to be understood.

30.2 Tests

\fp_if_exist_p:N
\fp_if_exist_p:c
\fp_if_exist:NTF
\fp_if_nan:p:n
\fp_if_nan:nTF

Copies of the cs functions defined in l3basics.

(End definition for \fp_if_exist:NTF. This function is documented on page 202.)

Evaluate and check if the result is a floating point of the same kind as NaN.

(End definition for \fp_if_nan:nTF. This function is documented on page 261.)
30.3 Comparison

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with ±0. Tuples are true.

\texttt{\textbackslash fp\_compare\_p:n} \texttt{\textbackslash fp\_compare:nTF} \texttt{\_\_fp\_compare\_return:w}

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to \texttt{\_\_fp\_compare\_back\_any:ww}, defined below. Compare the result with '#2-=', which is −1 for <, 0 for =, 1 for > and 2 for ?.

\texttt{\textbackslash prg\_new\_conditional:Npnn \textbackslash fp\_compare:n \#1 \{ p , T , F , TF \}}
\texttt{\{ \textbackslash exp\_after:wN \_\_fp\_compare\_return:w \textbackslash exp:w \textbackslash exp\_end\_continue_f:w \_\_fp\_parse:n \{#1\} \}}
\texttt{\}}
\texttt{\cs\_new:Npn \_\_fp\_compare\_return:w \#1\#2\#3;}
\texttt{\{ \textbackslash if\_charcode:w 0 \_\_fp\_if\_type_fp:NTwFw}
\texttt{ \#1 \{ \_\_fp\_use\_i\_delimit\_by_s\_stop:nw \#3 \_\_fp\_stop \}}
\texttt{\_\_fp\_fp 1 \_\_fp\_stop}
\texttt{\prg\_return\_false:}
\texttt{\} \texttt{\else:}
\texttt{\prg\_return\_true:}
\texttt{\}}
\texttt{\prg\_return\_false:}
\texttt{\}}
\texttt{\textbackslash cs\_new:Npn \_\_fp\_compare\_aux:wn \#1; \#2}
\texttt{\{ \textbackslash exp\_after:wN \_\_fp\_compare\_aux:wn \textbackslash exp:w \textbackslash exp\_end\_continue_f:w \_\_fp\_parse:n \{#2\} \#1; \}}
\texttt{\}}

(End definition for \texttt{\_\_fp\_compare:nTF} and \texttt{\_\_fp\_compare\_return:w}. This function is documented on page 203.)

\texttt{\_\_fp\_compare\_back\_any:ww}
\texttt{\_\_fp\_compare\_back:ww}
\texttt{\_\_fp\_compare\_nan:w}

Expands (in the same way as \texttt{\_\_fp\_compare\_back:ww}) \texttt{\_\_fp\_compare\_back\_any:ww \{y\} \{x\}}; (x);

(End definition for \texttt{\_\_fp\_compare:nNaN} and \texttt{\_\_fp\_compare\_aux:wn}. This function is documented on page 202.)

806
are equal. If \( y \geq 0 \), but of a different type, the highest type is a larger number. Finally, if \( y \leq 0 \), then \( x > y \), unless both are zero.

Tuple and floating point numbers are not comparable so return 2 in mixed cases or when tuples have a different number of items. Otherwise compare pairs of items with \(--fp_compare_back__any:ww\) and if any don’t match return 2 (as \texttt{\int_value:w 02 \exp_stop_f:}).
Within an \int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, −1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.
30.4 Floating point expression loops

These are quite easy given the above functions. The do_until and do_while versions execute the body, then test. The until_do and while_do do it the other way round.

As above but not using the nNn syntax.

(End definition for \_fp_compare_npos:nwnw and \_fp_compare_significand:nnnnnn.)
The approach here is somewhat similar to \texttt{\int_step_function:nnnN}. There are two subleties: we use the internal parser \_\_\_\_fp_parse:n to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

\begin{verbatim}
\cs_new:Npn \fp_step_function:nnN #1#2#3
\begin{macrocode}
\exp_after:wN \__fp_step:wwwN
\exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
\exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
\exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
\end{macrocode}
\end{verbatim}

\begin{verbatim}
\cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
\begin{macrocode}
\end{macrocode}
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
\begin{macrocode}
\__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
\__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
\__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
\use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
\prg_break_point:
\use:n
\begin{macrocode}
\end{macrocode}
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_step.fp:wwwN \__fp_step:NnN \__fp_step:NfnnnN
\begin{macrocode}
\end{macrocode}
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \fp_step_function:nnc
\begin{macrocode}
\end{macrocode}
\end{verbatim}

\begin{verbatim}
\_\_fp_step.fp:wwwN \_\_fp_step:NnN \_\_fp_step:NfnnnN
\_\_fp_step_function:nnnc
\_\_fp_step_function:nnnnN
\_\_fp_step_function:nnnc
\_\_fp_step_function:nnnnN
\end{macrocode}

The approach here is somewhat similar to \texttt{\int_step_function:nnnN}. There are two subleties: we use the internal parser \_\_\_\_fp_parse:n to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases. This function has one more argument than its integer counterpart, namely the previous value, to catch the case where the loop has made no progress. Conversion to decimal is done just before calling the user’s function.
\fp_step_inline:nnnn \fp_step_variable:nnnNn

As for \int_step_inline:nnnn, create a global function and apply it, following up with a break point.

(End definition for \fp_step_function:nnnN and others. This function is documented on page 205.)
\[ \int_{g} \text{incr} : N \ \text{g\_kernel\_prg\_map\_int} \]
\[ \exp\text{args} : N Nc \ \text{\_fp\_step} : NNnnn \]
\[ \text{cs\_gset\_protected} : Npx \]
\[ \{ \text{\_fp\_map} _{\int\text{use}} : N \ \text{g\_kernel\_prg\_map\_int} : w \} \]
\[ \{ \#1 \} \{ \#2 \} \{ \#3 \} \]
\[ \{ \]
\[ \text{tl\_set} : Nn \ \text{exp\_not} : N \ #4 \{ \#1 \} \]
\[ \text{exp\_not} : n \{ \#5 \} \]
\[ \} \]
\[ \text{cs\_new\_protected} : Npn \ \text{\_fp\_step} : NNnnnn \ #1\#2\#3\#4\#5\#6 \]
\[ \{ \]
\[ \#1 \#2 \#1 \{ \#6 \} \]
\[ \text{\_fp\_step\_function} : nnnN \ \{ \#3 \} \{ \#4 \} \{ \#5 \} \#2 \]
\[ \text{prg\_break\_point} : Nn \ \text{\_scan\_stop} : \{ \int\text{gdecr} : N \ \text{g\_kernel\_prg\_map\_int} \} \]
\[ \} \]

(End definition for \text{\_fp\_step\_inline} : nnnnn, \text{\_fp\_step\_variable} : nnnNn, and \text{\_fp\_step} : NNnnnn. These functions are documented on page 205.)

\[ \text{\_kernel\_msg\_new} : nnn \ \{ \text{kernel} \} \ \{ \text{fp\_step\_tuple} \} \]
\[ \{ \text{tuple\_argument\_in\_fp\_step} : \ldots \{-\#1\}\{-\#2\}\{-\#3\}. \} \]
\[ \text{\_kernel\_msg\_new} : nnn \ \{ \text{kernel} \} \ \{ \text{fp\_bad\_step} \} \]
\[ \{ \text{Invalid\_step\_size\_\#2\_in\_step\_function\_\#3}. \} \]
\[ \text{\_kernel\_msg\_new} : nnn \ \{ \text{kernel} \} \ \{ \text{fp\_tiny\_step} \} \]
\[ \{ \text{Tiny\_step\_size\_\{-\#1\}_\#2\_\#1\}_\text{in\_step\_function\_\#3}. \} \]

### 30.5 Extrema

First check all operands are floating point numbers. The argument \#1 is 2 to find the maximum of an array \#2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance \pm 0), the first is kept. We append \(-\infty\) (\(\infty\)), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of \text{max}() and \text{min}() with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of \text{\_fp\_minmax\_loop} : Nww.

\[ \text{cs\_new} : Npn \ \text{\_fp\_minmax\_o} : Nw \ \#1 \]
\[ \{ \]
\[ \{ \text{\_fp\_parse\_function\_all\_fp\_o} : fnw \]
\[ \{ \text{\_fp\_minmax\_aux\_o} : Nw \ \#1 \} \]
\[ \} \]
\[ \text{cs\_new} : Npn \ \text{\_fp\_minmax\_aux\_o} : Nw \ \#1\#2 \ 0 \]
\[ \{ \]
\[ \text{\_fp\_minmax\_loop} : Nww \ \text{\_fp\_minmax\_o} : Nw \ \#1 \]
\[ \{ \]
\[ \{ \text{\_fp\_chk} : w \ 2 \ \#1 \ \text{\_fp\_exact} \} \]
\[ \{ \text{\_fp\_chk} : w \ { 3 \ \text{\_fp\_minmax\_break\_o} : w } \} ; \]
\__fp_minmax\_loop:Nww

The first argument is \- or \+ to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is \texttt{nan}, keep that as the extremum, unless that extremum is already a \texttt{nan}. Otherwise, compare the two numbers. If the new number is larger (in the case of \texttt{max}) or smaller (in the case of \texttt{min}), the test yields \texttt{true}, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

\begin{verbatim}
\cs_new:Npn \__fp_minmax\_loop:Nww
\end{verbatim}

\__fp_minmax\_auxi:ww

Keep the first/second number, and remove the other.

\begin{verbatim}
\cs_new:Npn \__fp_minmax\_auxi:ww
\end{verbatim}

\__fp_minmax\_auxii:ww

This function is called from within an \texttt{\if\meaning:w} test. Skip to the end of the tests, close the current test with \texttt{\fi:}, clean up, and return the appropriate number with one post-expansion.

\begin{verbatim}
\cs_new:Npn \__fp_minmax\_break:o:w
\end{verbatim}
30.6 Boolean operations

Return true or false, with two expansions, one to exit the conditional, and one to please \texttt{\textbackslash l3fp-parse}. The first argument is provided by \texttt{\textbackslash l3fp-parse} and is ignored.

\begin{verbatim}
\cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3; @
{ \if_meaning:w 0 #2 \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
  \else: \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
  \fi: }
\cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
\end{verbatim}

(End definition for \__fp_not_o:w and \__fp_tuple_not_o:w.)

For \texttt{and}, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For \texttt{or}, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking \__fp_&_o:ww, inserting an extra argument, \texttt{\else:}, before \texttt{s__fp}. In all cases, expand after the floating point number.

\begin{verbatim}
\group_begin:
\char_set_catcode_letter:N &
\char_set_catcode_letter:N |
\cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
{ \if_meaning:w 0 #2 #1 \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
  \else: \__fp_exp_after_o:w
  \fi: }
\cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
\cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }

\group_end:
\cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
{ \fi: \__fp_exp_after_o:w #1; }
\end{verbatim}

(End definition for \__fp_&_o:ww and others.)
30.7 Ternary operator

The first function receives the test and the true branch of the ?: ternary operator. It calls `__fp_ternary_auxii:NwwN` if the test branch is a floating point number ±0, and otherwise calls `__fp_ternary_auxi:NwwN`. These functions select one of their two arguments.

\begin{verbatim}
cs_new:Npn __fp_ternary:NwwN #1 #2#30 #40 #5
{ 
  \if_meaning:w __fp_parse_infix::N #5
  \if_charcode:w 0
  __fp_if_type_fp:NTwFw
#2 { use_i:nn __fp_use_i_delimit_by_s_stop:nw #3 \a__fp_stop }
  \a__fp 1 \a__fp_stop
  \exp_after:wN \exp_after:wN \exp_after:wN __fp_ternary_auxii:NwwN
  \else:
  \exp_after:wN \exp_after:wN \exp_after:wN __fp_ternary_auxi:NwwN
  \fi:
  \exp_after:wN #1
  \exp:w \exp_end_continue_f:w
  __fp_exp_after_array_f:w #4 \a__fp_expr_stop
  \exp_after:wN \__fp_parse_operand:Nw \c__fp_prec_colon_int
  \__fp_parse_expand:w
  \else:
  \__kernel_msg_expandable_error:nnnn
  \{ kernel \} \{ fp-missing \} \{ : \} \{ -for-?: \}
  \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #1
  \exp:w \exp_end_continue_f:w
  \__fp_exp_after_array_f:w #4 \a__fp_expr_stop
  \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #5
  \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary:NwwN #1
  \fi:
}
\cs_new:Npn __fp_ternary_auxi:NwwN #1#2@#3@#4
{ \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #1
  \exp:w \exp_end_continue_f:w
  \__fp_exp_after_array_f:w #3 \a__fp_expr_stop
  \__fp_exp_after_array_f:w #2 \a__fp_expr_stop
  \a__fp 1 \a__fp_stop
  \exp_after:wN \exp_after:wN \exp_after:wN __fp_ternary_auxii:NwwN
}
\cs_new:Npn __fp_ternary_auxii:NwwN #1#2@#3@#4
{ \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #1
  \exp:w \exp_end_continue_f:w
  \__fp_exp_after_array_f:w #4 \a__fp_expr_stop
  \exp_after:wN \__fp_parse_operand:Nw \c__fp_prec_colon_int
  \__fp_parse_expand:w

(End definition for __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, and __fp_ternary_auxii:NwwN.)
\end{verbatim}

815
The \l3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at http://cr.yp.to/2005-590/goldberg.pdf.

### Unary functions.

\begin{verbatim}
\cs_new:Npn \__fp_parse_word_abs:N { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
\cs_new:Npn \__fp_parse_word_logb:N { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
\cs_new:Npn \__fp_parse_word_sign:N { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
\cs_new:Npn \__fp_parse_word_sqrt:N { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
\end{verbatim}

(End definition for \__fp_parse_word_abs:N and others.)

### 31.1 Addition and subtraction

We define here two functions, \__fp_-_o:ww and \__fp_+_o:ww, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, \__fp_add_big_i_o:wNww, is used in \l3fp-expo.

The logic goes as follows:

- \__fp_-_o:ww calls \__fp_+_o:ww to do the work, with the sign of the second operand flipped;
- \__fp_+_o:ww dispatches depending on the type of floating point, calling specialized auxiliaries;
- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the \__fp_\-basics_pack_... functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.
31.1.1 Sign, exponent, and special numbers

The \_fp+_o:ww auxiliary has a hook: it takes one argument between the first \_s-_fp and \_fp chk:w, which is applied to the sign of the second operand. Positioning the hook there means that \_fp+_o:ww can still perform the sanity check that it was followed by \s_fp.

\cs_new:cpx { __fp_-_o:ww } \s_fp
{
\exp_not:c { __fp_+_o:ww }
\exp_not:n { \s_fp \__fp_neg_sign:N }
}

(End definition for \_fp_-_o:ww.)

This function is either called directly with an empty #1 to compute an addition, or it is called by \_fp_-_o:ww with \_fp neg_sign:N as #1 to compute a subtraction, in which case the second operand’s sign should be changed. If the ⟨types⟩ #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to \int_value:w, those receive the tweaked ⟨sign2⟩ (expansion of #1#5) as an argument. If the ⟨types⟩ are distinct, the result is simply the floating point number with the highest ⟨type⟩. Since case 3 (used for two nan) also picks the first operand, we can also use it when ⟨type1⟩ is greater than ⟨type2⟩. Also note that we don’t need to worry about ⟨sign2⟩ in that case since the second operand is discarded.

\cs_new:cpn { __fp_+_o:ww }
\s_fp #1 \_fp_chk:w #2 #3 \_fp chk:w #4 #5
{
\if_case:w
\if_meaning:w #2 #4
#2
\else:
\if_int_compare:w #2 > #4 \exp_stop_f:
3
\else:
4
\fi:
\fi:
\exp_stop_f:
\exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
\or: \exp_after:wN \__fp_add_normal_o:Nww \int_value:w
\or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
\or: \exp_after:wN \__fp_case_return_i_o:ww
\else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
\fi:
#1 #5
\_s_fp \_fp_chk:w #2 #3 ;
\_s_fp \_fp_chk:w #4 #5
}

(End definition for \_fp+_o:ww.)

\_fp_add_return_ii_o:Nww Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

\cs_new:fnp \_fp_add_return_ii_o:Nww #1 #2 ; \_s_fp \_fp chk:w #3 #4
{ \_fp_exp_after_o:w \s_fp \_fp chk:w #3 #1 }

817
Adding two zeros yields \( \mathbf{c_{zero}} \), except if both zeros were \(-0\).

\[
\text{\texttt{\_fp_add_zeros\_o:Nww}}
\]

Interpretation: If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked \( \langle \text{sign}_2 \rangle \) (#1) and the \( \langle \text{sign}_2 \rangle \) (#4) are identical.

\[
\text{\texttt{\_fp_add_normal\_o:Nww}}
\]

Interpretation: We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.
31.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```latex
\_\_fp_add_npos_o:NnwNnw (sign_1) (exp_1) (body_1) ; \s__fp \_\_fp_chk:w 1
\_\_fp_add_npos_o:NnwNnw (initial sign_2) (exp_2) (body_2) ;
```

Since we are doing an addition, the final sign is \langle sign_1 \rangle. Start an \_\_fp_int_eval:w, responsible for computing the exponent: the result, and the \langle final sign \rangle are then given to \_\_fp_sanitize:Nw which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by \_\_fp_add_big_i:o:wNww or \_\_fp_add_big_ii:o:wNww. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```latex
\cs_new:Npn \_\_fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \_\_fp_chk:w 1 #4 #5
\begin{verbatim}
{ \exp_after:wN \_\_fp_sanitize:Nw
\exp_after:wN #1
\int_value:w \_\_fp_int_eval:w
\if_int_compare:w #2 #5 \exp_stop_f:
\else:
#5
\exp_after:wN \_\_fp_add_big_i_o:wNww \int_value:w -
\else:
#5
\exp_after:wN \_\_fp_add_big_ii_o:wNww \int_value:w
\fi:
\_\_fp_int_eval:w #5 - #2 ; #1 #3;
}
\end{verbatim}
```

(End definition for \_\_fp_add_npos_o:NnwNnw.)

```latex
\_\_fp_add_big_i_o:wNww
\_\_fp_add_big_ii_o:wNww
\_\_fp_add_big_i:o:wNww \langle shift \rangle ; \langle final sign \rangle \langle body_1 \rangle ; \langle body_2 \rangle ;
```

Used in l3fp-expo. Shift the significand of the small number, then add with \_\_fp_add_significand_o:NnnwnnnN.

```latex
\cs_new:Npn \_\_fp_add_big_i:o:wNww #1; #2 #3; #4;
\begin{verbatim}
{ \_\_fp_decimate:nNnnnn \{#1
\_\_fp_add_significand_o:NnnwnnnN
#4
#3
#2 }
}
\cs_new:Npn \_\_fp_add_big_ii:o:wNww #1; #2 #3; #4;
\begin{verbatim}
{ \_\_fp_decimate:nNnnnn \{#1
\_\_fp_add_significand_o:NnnwnnnN
#3
#4
#2 }
}
\end{verbatim}
```

(End definition for \_\_fp_add_big_i:o:wNww and \_\_fp_add_big_ii:o:wNww.)

819
To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as 0.99...95 → 1.00...0, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

\__fp_add_significand_o:NnnwnnnnN (rounding digit) \{\langle Y_1 \rangle \} \{\langle Y_2 \rangle \} (extra-digits) ; \{\langle X_1 \rangle \} \{\langle X_2 \rangle \} \{\langle X_3 \rangle \} \{\langle X_4 \rangle \} (final sign)

If there’s no carry, grab all the digits again and round. The packing function \__fp_basics_pack_high:NNNNNw takes care of the case where rounding brings a carry.

\__fp_add_significand_no_carry_o:wwwNN (8d) ; (6d) ; (2d) ; (rounding digit) \langle sign \rangle

(End definition for \__fp_add_significand_o:NnnwnnnnN, \__fp_add_significand_pack:NNNNNNN, and \__fp_add_significand_test_o:N.)
The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don’t care.

\begin{verbatim}
\cs_new:Npn \__fp_add_significand_carry_o:wwwNN \#1; \#2; \#3#4; \#5#6
{ + 1 \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw \int_value:w \__fp_int_eval:w 1 1 \#1 \exp_after:wN \__fp_basics_pack_weird_low:NNNNw \int_value:w \__fp_int_eval:w 1 \#2\#3 + \exp_after:wN \__fp_round:NNN \exp_after:wN \#6 \exp_after:wN \#3 \int_value:w \__fp_round_digit:Nw \#4 \#5 ; \exp_after:wN ;}
\end{verbatim}

(End definition for \__fp_add_significand_carry_o:wwwNN.)

31.1.3 Absolute subtraction

\begin{verbatim}
\cs_new:Npn \__fp_sub_npos_o:NnwNnw \__fp_sub_npos_ii_o:Nnwnw
\__fp_sub_eq_o:Nnwnw \langle sign \rangle \langle exp \rangle \langle body \rangle \langle initial sign2 \rangle \langle exp2 \rangle \langle body2 \rangle ; \s__fp \__fp_chk:w 1 \langle final sign \rangle \langle exponent \rangle (delimited by ;).
\end{verbatim}

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call \__fp_sub_npos_-i_o:Nnwnw with the opposite of \langle sign \rangle.

\begin{verbatim}
\cs_new:Npn \__fp_sub_npos_o:NnwNnw \#1\#2\#3; \s__fp \__fp_chk:w 1 \#4\#5\#6;
{ \if_case:w \__fp_compare_npos:nwnw \#2; \#3; \#5; \#6; \exp_stop_f: \exp_after:wN \__fp_sub_eq_o:Nnwnw \\or: \exp_after:wN \__fp_sub_npos_i_o:Nnwnw \\else: \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw \fi: \#1 \{\#2\} \#3; \#5; \#6; }
\end{verbatim}

(End definition for \__fp_sub_npos_o:NnwNnw, \__fp_sub_eq_o:Nnwnw, and \__fp_sub_npos_ii_o:Nnwnw.)

\begin{verbatim}
\cs_new:Npn \__fp_sub_npos_i_o:Nnwnw \__fp_sub_eq_o:Nnwnw \#1\#2; \#3; \{ \exp_after:wN \c_zero_fp \} \cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw \#1 \#2; \#3; \{ \exp_after:wN \__fp_neg_sign:N \#1 \#3; \#2; \}
\end{verbatim}

After the computation is done, \__fp-sanitize:Nw checks for overflow/underflow. It expects the \langle final sign \rangle and the \langle exponent \rangle (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the near auxiliary. Otherwise, decimate \textit{y}, then call the far auxiliary to evaluate
the difference between the two significands. Note that we decimate by 1 less than one could expect.

\begin{verbatim}
\cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
{ \exp_after:wN \__fp_sanitize:Nw \exp_after:wN \int_value:w \__fp_int_eval:w #2
\if_int_compare:w #2 = #4 \exp_stop_f:
\exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
\else:
\exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
\exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN \fi:
#5 #3 #1}
\end{verbatim}

(End definition for \texttt{\__fp_sub_npos_i_o:Nnwnw}.)

\begin{verbatim}
\__fp_sub_back_near_o:nnnnnnnnN \{ (Y_1) \} \{ (Y_2) \} \{ (Y_3) \} \{ (Y_4) \} \{ (X_1) \}
\{ (X_2) \} \{ (X_3) \} \{ (X_4) \} \{ \textit{final sign} \}

In this case, the subtraction is exact, so we discard the \textit{final sign} $\#9$. The very large shifts of $10^9$ and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

\begin{verbatim}
\cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
{ \exp_after:wN \__fp_sub_back_near_after:wNNNNw \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11 \exp_after:wN }
\int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;}
\end{verbatim}

(End definition for \texttt{\__fp_sub_back_near_o:nnnnnnnnN}, \texttt{\__fp_sub_back_near_pack:NNNNNNw}, and \texttt{\__fp_sub_back_after:wNNNNw}.)

\begin{verbatim}
\__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5#6#7#8 #9
\end{verbatim}

This function is called with $\langle Z_1 \rangle \leq 999$. Act with \texttt{number} to trim leading zeros from $\langle Z_1 \rangle \langle Z_2 \rangle$ (we don’t do all four blocks at once, since non-zero blocks would then overflow \TeX{}’s integers). If the first two blocks are zero, the auxiliary receives an empty \#1 and trims \#2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from \#1 alone, yielding a result between 1 and 6. Once the
exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space
before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are
added so that digits can be grabbed safely.

\cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2#3
\exp_after:wN \__fp_sub_back_shift_ii:ww
\int_value:w #1 #2 0 ;
\}
\cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
\if_meaning:w @ #1 @ - 7
\exp_after:wN \use_i:nnn
\exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
\int_value:w #2#3 0 ~ 123456789;
\else:
- \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
\fi:
\exp_after:wN \__fp_sub_back_shift_iv:nnnnw
\exp_after:wN ;
\int_value:w #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
\}
\cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
\cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

(End definition for \__fp_sub_back_shift:wnnnn and others.)
\__fp_sub_back_far_o:NnnwnnnnN \__fp_sub_back_far_o:NnnwnnnnN \__fp_sub_back_far_o:NnnwnnnnN \__fp_sub_backFar_o:NNnnnnnn {rounding} \{\langle Y' 1 \rangle \} \{\langle Y' 2 \rangle \} \{\langle X 1 \rangle \} \{\langle X 2 \rangle \} \{\langle X 3 \rangle \} \{\langle X 4 \rangle \} \{final sign\}
If the difference is greater than 10^{\langle \text{expo} \rangle}, call the very_far auxiliary. If the result is
less than 10^{\langle \text{expo} \rangle}, call the not_far auxiliary. If it is too close a call to know yet, namely
if \langle Y' 1 \rangle \langle Y' 2 \rangle = \langle X 1 \rangle \langle X 2 \rangle \langle X 3 \rangle \langle X 4 \rangle 0, then call the quite_far auxiliary. We use the odd
combination of space and semi-colon delimiters to allow the not_far auxiliary to grab
each piece individually, the very_far auxiliary to use \__fp_pack_eight:wNNNNNNNN, and the quite_far to ignore the significands easily (using the ; delimiter).
\cs_new:Npn \__fp_sub_back_far_o:NNnnnnnn \#1 \#2 \#3 \#4; \#5\#6\#7\#8
\{ 
\if_case:w
\if_int_compare:w 1 \#2 = \#5\#6 \use_i:nnn \#7 \exp_stop_f:
\if_int_compare:w \#3 = \use_none:n \#7\#8 0 \exp_stop_f:
0
\else:
\if_int_compare:w \#3 > \use_none:n \#7\#8 0 \{-fi: 1
\fi:
\else:
\if_int_compare:w 1 \#2 > \#5\#6 \use_i:nnn \#7 \{-fi: 1
\fi:
\exp_stop:f;
\exp_after:wN \__fp_sub_back_quite_far_o:wWW
\or: \exp_after:wN \__fp_sub_back_very_far_o:wWWW
\}
823
The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of $x$ is 1, and all others vanish when subtracting $y$. Then the \textit{(rounding)} digit and the \textit{(final sign)} control whether we get 1 or 0.0000000000000001. In the usual round-to-nearest mode, we get 1 whenever the \textit{(rounding)} digit is less than or equal to 5 (remember that the \textit{(rounding)} digit is only equal to 5 if there was no further non-zero digit).

In the present case, $x$ and $y$ have different exponents, but $y$ is large enough that $x - y$ has a smaller exponent than $x$. Decrement the exponent (with -1). Then proceed in a way similar to the \textit{near} auxiliaries seen earlier, but multiplying $x$ by 10 ($\#30$ and $\#40$ below), and with the added quirk that the \textit{(rounding)} digit has to be taken into account. Namely, we may have to decrease the result by one unit if \texttt{\__fp_round_neg:NNN} returns 1.

This function expects the \textit{(final sign)} digit, last digit of $110000000000+\#40-\#2$, and the \textit{(rounding)} digit. Instead of redoing the computation for the second argument, we note that \texttt{\__fp_round_neg:NNN} only cares about its parity, which is identical to that of the last digit of #2. 

(End definition for \texttt{\__fp_sub_back_quite_far_o:wwNN} and \texttt{\__fp_sub_back_quite_far_ii:NN}.)
The case where $x - y$ and $x$ have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the $y$ significand by adding a leading 0. Then the logic is similar to the not_far functions above. Rounding is a bit more complicated: we have two \langle rounding \rangle digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first \texttt{int_value:w} triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

\begin{verbatim}
\cs_new:Npn \__fp_sub_back_very_far_o:wwwwNN #1#2#3#4#5#6#7
{ \__fp_pack_eight:wNNNNNNNN \__fp_sub_back_very_far_ii_o:nnNwwNN { 0 #1#2#3 #4#5#6#7 } ; }
\cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
{ \exp_after:wN \__fp_basics_pack_high:NNNNNw \int_value:w \__fp_int_eval:w 1#4 - #1 - 1 \exp_after:wN \__fp_basics_pack_low:NNNNNN \int_value:w \__fp_int_eval:w 2#5 - #2 - \exp_after:wN \__fp_round_neg:NNN \exp_after:wN #7 \int_value:w \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end: 1 \else: 2 \fi: \int_value:w \__fp_round_digit:Nw #3 #6 ; \exp_after:wN ; }
\end{verbatim}

(End definition for \__fp_sub_back_very_far_o:wwwwNN and \__fp_sub_back_very_far_ii_o:nnNwwNN.)

31.2 Multiplication

31.2.1 Signs, and special numbers

We go through an auxiliary, which is common with \__fp_/_o:ww. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in \__fp_/_o:ww.

\begin{verbatim}
\cs_new:cpn { __fp_*_o:ww }
{ \__fp_mul_cases_o:nNnnww \__fp_mul_cases_o:NnNwww \__fp_mul_cases_o:NnNwwww \__fp_mul_cases_o:NnNwwww \__fp_mul_cases_o:NnNwwwww \__fp_mul_cases_o:NnNwwwwww }
\end{verbatim}

(End definition for \__fp_*_o:ww.)

\__fp_mul_cases_o:nNnnww

Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call \__fp_mul_npos_o:Nww to do the work. If
the first operand is \texttt{nan}, go to case 2, in which the second operand is discarded; if the second operand is \texttt{nan}, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products \(0 \times 0 = 0 \times 1 = 1 \times 0 = 0\) to case 4 or 5 depending on the combined sign, the products \(0 \times \infty\) and \(\infty \times 0\) to case 6 or 7 (invalid operation), and the products \(1 \times \infty = \infty \times 1 = \infty \times \infty = \infty\) to cases 8 and 9. Note that the code for these two cases (which return \(\pm \infty\)) is inserted as argument \#4, because it differs in the case of divisions.

\begin{verbatim}
\cs_new:Npn \__fp_mul_cases_o:NnNnww
  \#1\#2\#3\#4 \s__fp \__fp_chk:w \#5\#6\#7; \s__fp \__fp_chk:w \#8\#9
  { \if_case:w \__fp_int_eval:w
        \if_int_compare:w \#5 \#8 = 11 ~
        \else:
        \if_meaning:w 3 \#8
        3
        \else:
        \if_meaning:w 3 \#5
        2
        \else:
        \if_int_compare:w \#5 \#8 = 10 ~
        9 \#2 - 2
        \else:
        (#5 \#8) / 2 * 2 + 7
        \fi:
        \fi:
        \fi:
        \fi:
        \if_meaning:w \#6 \#9 - 1 \fi:
        \__fp_int_eval_end:
        \__fp_case_use:nw { \#3 0 }
        \or: \__fp_case_use:nw { \#3 2 }
        \or: \__fp_case_return_i_o:ww
        \or: \__fp_case_return_ii_o:ww
        \or: \__fp_case_return_o:Nww \c_zero_fp
        \or: \__fp_case_return_o:Nww \c_minus_zero_fp
        \or: \__fp_case_return_o:Nww \c_inf_fp
        \or: \__fp_case_return_o:Nww \c_minus_inf_fp
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_return_o:Nww \c_inf_fp
        \or: \__fp_case_return_o:Nww \c_minus_inf_fp
        \or: \__fp_case_return_o:Nww \c_zero_fp
        \or: \__fp_case_return_o:Nww \c_minus_zero_fp
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_return_o:Nww \c_inf_fp
        \or: \__fp_case_return_o:Nww \c_minus_inf_fp
        \or: \__fp_case_return_o:Nww \c_zero_fp
        \or: \__fp_case_return_o:Nww \c_minus_zero_fp
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww \#1 }
        \or: \__fp_case_return_o:Nww \c_inf_fp
        \or: \__fp_case_return_o:Nww \c_minus_inf_fp
        \or: \__fp_case_return_o:Nww \c_zero_fp
        \or: \__fp_case_return_o:Nww \c_minus_zero_fp
       \fi:
  \end{verbatim}

\textit{(End definition for \texttt{\__fp_mul_cases_o:NnNnww}.)}

31.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

826
\__fp_mul_npos_o:Nww (final sign) \s__fp \__fp_chk:w 1 (sign1) {\exp1} (body1) ; \s__fp \__fp_chk:w 1 (sign2) {\exp2} (body2) ;

After the computation, \__fp_sanitze:Nw checks for overflow or underflow. As we did for addition, \__fp_int_eval:w computes the exponent, catching any shift coming from the computation in the significand. The (final sign) is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by \__fp_mul_significand_o:nnnnNnnnn.

This is also used in \fp-convert.

\cs_new:Npn \__fp_mul_npos_o:Nww #1 #2 #3 #4 #5 ; #6 #7 #8 #9 ;
{\exp_after:wN \__fp_mul_significand_test_f:NNN \exp_after:wN #5 \int_value:w \__fp_int_eval:w #1*#6 + #2*#7 + #3*#8 + #4*#9 ; \exp_after:wN ; 827

(End definition for \__fp_mul_npos_o:Nww.)

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_drop:NNNNNw \__fp_mul_significand_keep:NNNNNw

Note the three semicolons at the end of the definition. One is for the last \__fp_mul_significand_drop:NNNNNw; one is for \__fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an \__fp_int_eval:w), is used by \__fp_basics_pack_low:NNNNNw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of \__fp_int_eval:w.

\cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn \#1 \#2 \#3 \#4 \#5 \#6 \#7 \#8 \#9
{\exp_after:wN \__fp_mul_significand_test_f:NNN \exp_after:wN #5 \int_value:w \__fp_int_eval:w 99990000 + #1*#6 + #2*#7 + #3*#8 + #4*#9 ; \exp_after:wN ; 827

Note the three semicolons at the end of the definition. One is for the last \__fp_mul_significand_drop:NNNNNw; one is for \__fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an \__fp_int_eval:w), is used by \__fp_basics_pack_low:NNNNNw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of \__fp_int_eval:w.
\cs_new:Npn \__fp_mul_significand\_drop:NNNNNw \#1\#2\#3\#4\#5 \#6;
  \{ \#1\#2\#3\#4\#5 ; + \#6 \}
\cs_new:Npn \__fp_mul_significand\_keep:NNNNNw \#1\#2\#3\#4\#5 \#6;
  \{ \#1\#2\#3\#4\#5 ; \#6 ; \}

(End definition for \__fp_mul_significand\_o:nnnnNnnnn, \__fp_mul_significand\_drop:NNNNNw, and \__fp_mul_significand\_keep:NNNNNw.)

\__fp_mul_significand\_test\_f:NNN

\__fp_mul_significand\_test\_f:NNN (sign) 1 \langle digits 1–8 \rangle ; \langle digits 9–12 \rangle ;
  \langle digits 13–16 \rangle ; + \langle digits 17–20 \rangle + \langle digits 21–24 \rangle + \langle digits 25–28 \rangle + \langle digits 29–32 \rangle ; \exp\_after:WN ;

If the \langle digit 1 \rangle is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if \langle digit 1 \rangle is zero, we care about digits 17 and 18, and whether further digits are zero.

\cs_new:Npn \__fp_mul_significand\_large\_f:NwwNNNN

In this branch, \langle digit 1 \rangle is non-zero. The result is thus \langle digits 1–16 \rangle, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, \__fp_round_digit:Nw takes digits 17 and further (as an integer expression), and replaces it by a \langle rounding digit \rangle, suitable for \__fp_round:NNN.

\cs_new:Npn \__fp_mul_significand\_small\_f:NNwwwN

In this branch, \langle digit 1 \rangle is zero. Our result is thus \langle digits 2–17 \rangle, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the small\_pack auxiliary, by the next digit, to form a 9 digit number.
31.3 Division

31.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display / rather than *. In the formula for dispatch, we replace \(-2+\) by \(-\). The case of normal numbers is treated using \__/fp\_/\_o:ww\__/fp\_/\_mul\_/\_n\_/\_/o:ww rather than \__/fp\_/\_mul\_/\_n\_/\_/o:ww. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised; cases 10 and 11 of the \_/fp\_/\_mul\_/\_cases\_/\_/o:ww construction in \__/fp\_/\_mul\_/\_cases\_/\_/o:ww are provided as the fourth argument here.

We want to compute \(A/Z\). As for multiplication, \__/fp\_/\_sanitize:ww checks for overflow or underflow; we provide it with the \(\langle\text{final sign}\rangle\), and an integer expression in which we compute the exponent. We set up the arguments of \__/fp\_/\_div\_/\_significand\_/\_/i\_/\_/o:ww, namely an integer \((y)\) obtained by adding 1 to the first 5 digits of \(Z\) (explanation given soon below), then the four \(\langle A_i\rangle\), then the four \(\langle Z_i\rangle\), a semi-colon, and the \(\langle\text{final sign}\rangle\), used for rounding at the end.
31.3.2 Work plan

In this subsection, we explain how to avoid overflowing \TeX's integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of $A_1$ and of $Z_1$ are non-zero. To compute $A/Z$, we proceed as follows.

- Find an integer $Q_A \simeq 10^4A/Z$.
- Replace $A$ by $B = 10^4A - QA$.
- Find an integer $Q_B \simeq 10^4B/Z$.
- Replace $B$ by $C = 10^4B - QB$.
- Find an integer $Q_C \simeq 10^4C/Z$.
- Replace $C$ by $D = 10^4C - QC$.
- Find an integer $Q_D \simeq 10^4D/Z$.
- Consider $E = 10^4D - QDZ$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the $Q_i$ are integers, $B$, $C$, $D$, and $E$ are all exact multiples of $10^{-16}$, in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general $B$, $C$, $D$, and $E$ may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $QA \leq 10^4A/Z$ etc. A reasonable attempt would be to define $Q_A$ as

$$\intevaln\left\{A_1A_2\over Z_1+1 - 1\right\} \leq 10^4\overline{A/Z}$$

Subtracting 1 at the end takes care of the fact that \TeX's \texttt{\_fp_int_eval:w} rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers).

We add 1 to $Z_1$ because $Z_1 \leq 10^4Z < Z_1 + 1$ and we need $Q_A$ to be an underestimate. However, we are now underestimating $Q_A$ too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then $B$ could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since \TeX can only handle integers less than roughly $2 \cdot 10^9$. 

830
A better formula is to take
\[ Q_A = \text{\texttt{int\_eval}}:\left\{ \frac{10 \cdot A_1 A_2}{10^{-3} \cdot Z_1 Z_2} + 1 \right\}. \]
This is always less than \(10^9 A/(10^5 Z)\), as we wanted. In words, we take the 5 first digits of \(Z\) into account, and the 8 first digits of \(A\), using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the \(Q_i\) avoids any overflow. For convenience, let us denote
\[ y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1, \]
so that, taking into account the fact that e\TeX rounds ties away from zero,
\[ Q_A = \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor > \frac{A_1 A_2 0}{y} - \frac{3}{2}. \]
Note that \(10^4 < y \leq 10^5\), and \(999 \leq Q_A \leq 99989\). Also note that this formula does not cause an overflow as long as \(A < (2^{31} - 1)/10^9 \approx 2.147\cdots\), since the numerator involves an integer slightly smaller than \(10^9 A\).

Let us bound \(B\):
\[ 10^5 B = A_1 A_2 0 + 10 \cdot 0 A_3 A_4 - 10 \cdot Z_1 Z_2 Z_3 Z_4 \cdot Q_A \]
\[ < A_1 A_2 0 \cdot \left( 1 - 10 \cdot \frac{Z_1 Z_2 Z_3 Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1 Z_2 Z_3 Z_4 + 10 \]
\[ \leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \]
\[ \leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \]
At the last step, we hide 10 into the second term for later convenience. The same reasoning yields
\[ 10^5 B < 10^9 A/y + 1.6 y, \]
\[ 10^5 C < 10^9 B/y + 1.6 y, \]
\[ 10^5 D < 10^9 C/y + 1.6 y, \]
\[ 10^5 E < 10^9 D/y + 1.6 y. \]
The goal is now to prove that none of \(B, C, D,\) and \(E\) can go beyond \((2^{31} - 1)/10^9 = 2.147\cdots\).

Combining the various inequalities together with \(A < 1\), we get
\[ 10^5 B < 10^9/y + 1.6 y, \]
\[ 10^5 C < 10^{13}/y^2 + 1.6(y + 10^4), \]
\[ 10^5 D < 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \]
\[ 10^5 E < 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2). \]
All of those bounds are convex functions of $y$ (since every power of $y$ involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

\[ 10^5B < \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \]
\[ 10^5C < \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \]
\[ 10^5D < \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \]
\[ 10^5E < \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \]

All of those bounds are less than $2.147 \cdot 10^8$, and we are thus within \TeX{}’s bounds in all cases!

We later need to have a bound on the \( Q_i \). Their definitions imply that

\[ Q_A < 10^9A/y - 1/2 < 10^5A \]
and similarly for the other \( Q_i \). Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

\[ A/Z = \sum_{i=1}^{4} (10^{-4i}Q_i) + 10^{-16}E/Z \]

exactly. Furthermore, we know that the result is in \([0.1, 10)\), hence will be rounded to a multiple of $10^{-16}$ or of $10^{-15}$, so we only need to know the integer part of $E/Z$, and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

\[ 2E/Z = 2 \cdot 10^5E/10^5Z \leq 2 \cdot 10^5E/10^4 < 36, \]

this integer part is between 0 and 35 inclusive. We let \( \varepsilon \)-\TeX{} round

\[ P = \text{\texttt{int\_eval:n}} \left\{ \frac{2 \cdot E_1E_2}{Z_1Z_2} \right\}, \]

which differs from $2E/Z$ by at most

\[ \frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8}Z_1Z_2} \right| + 2 \left| \frac{10^8E - E_1E_2}{Z_1Z_2} \right| < 1, \]

(1/2 comes from \( \varepsilon \)-\TeX{}’s rounding) because each absolute value is less than $10^{-7}$. Thus \( P \) is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, \( P = 2E/Z \). We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of \( P \), and the rounding mode.

### 31.3.3 Implementing the significand division

\texttt{\_\_fp\_div\_significand\_i\_o:www} \( y \); \{\langle A_1 \rangle \} \{\langle A_2 \rangle \} \{\langle A_3 \rangle \} \{\langle A_4 \rangle \} \{\langle Z_1 \rangle \} \{\langle Z_2 \rangle \} \{\langle Z_3 \rangle \} \{\langle Z_4 \rangle \}; \langle \text{sign} \rangle

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace \( \langle A_1 \rangle \) and \( \langle A_2 \rangle \), and prepare the \langle continuation \rangle arguments for 4 consecutive calls to \texttt{\_\_fp\_div\_significand\_calc:wwwnnnnnn}. Each of these calls needs \( \langle y \rangle \) (#1), and it turns out that
we need post-expansion there, hence the \texttt{int_value:w}. Here, \#4 is six brace groups, which give the six first n-type arguments of the \texttt{calc} function.

\begin{verbatim}
\cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
  \exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 \\
\cs_new:Npn \__fp_div_significand_test_o:w
  \exp_after:wN \__fp_div_significand_calc:wwn\\nnnnn
\cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn
  \exp_after:wN \__fp_div_significand_i_o:wnnw \int_value:w #1 \\
\exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 \\
\exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 \\
\int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 \\
\exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
\exp_after:wN \__fp_div_significand_test_o:w
\int_value:w \__fp_int_eval:w
\exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
\end{verbatim}

(End definition for \texttt{\__fp_div_significand_i_o:wnnw}.)

\begin{verbatim}
\__fp_div_significand_calc:wwn\\nnnnn
\__fp_div_significand_calc_i:wnnnnnn
\__fp_div_significand_calc_ii:wnnnnnn
\end{verbatim}

We know that \(0 < Q_A < 1.8 \cdot 10^5\), so the product of \(Q_A\) with each \(Z_i\) is within \TeX's bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on \(Q_A\), implies that \(10^6 + Q_A\) starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a \texttt{continuation}, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

\[10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5 \#6) + 10^{-6}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7)
\]
\[+ 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),\]

where \(\langle i \rangle\) stands for the \(10^5\) digit of \(Q_A\), which is 0 or 1, and \#1, \#2, etc. are the parameters of either auxiliary. The factors of 10 come from the fact that \(Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1\). As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most \(10^8\) and the negative contributions can go up to \(10^9\). Indeed, for the auxiliary with \(\langle i \rangle = 1\), \#1 is at most 80000, leading to contributions of at worse \(-8 \cdot 10^4\) while the other negative term is very small \(< 10^6\) (except in the first expression, where we don’t care about the number of digits); for the auxiliary with \(\langle i \rangle = 0\), \#1 can go up to 99999, but there is no other negative term. Hence, a good choice is \(2 \cdot 10^9\), which produces totals in the range \([10^0, 2.1 \cdot 10^9]\). We are flirting with \TeX’s limits once more.

\begin{verbatim}
\cs_new:Npn \__fp_div_significand_calc:wwn\\nnnnnnn
\end{verbatim}

833
\begin{verbatim}
{  \if_meaning:w 1 #1
   \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
\else:
   \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
\fi:
\}
\cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn
#1; #2;#3#4 #5#6#7#8 #9
  {1 1 #1
   #9 \exp_after:wN ;
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
   + #2 - #1 * #5 - #5#60
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
   + #3 - #1 * #6 - #70
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
   + #4 - #1 * #7 - #80
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
   - #1 * #8 ;
   {#5}{#6}{#7}{#8}
  }
\cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn
#1; #2;#3#4 #5#6#7#8 #9
  {1 0 #1
   #9 \exp_after:wN ;
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
   + #2 - #1 * #5
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
   + #3 - #1 * #6
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
   + #4 - #1 * #7
   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
   - #1 * #8 ;
   {#5}{#6}{#7}{#8}
  }
\end{verbatim}

\__(\text{End definition for } \__fp_div_significand_calc:wwnnnnnnn, \__fp_div_significand_calc_i:wwnnnnnnn, and \__fp_div_significand_calc_ii:wwnnnnnnn.\)
For large $P \geq 3$, the rounding error on $P$, which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $#6/#7 \leq 10^8 \cdot Z$, hence $P \cdot #6/#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot #8/#9$ may overlap, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use + as a separator (ending integer expressions explicitly). $T$ is negative if the first character is $-$, it is positive if the first character is neither 0 nor $. It is also positive if the first character is $0$ and second argument of $\_\_fp_div_significand_vi:Nw$, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

Once more, we need to be careful and show that the calculation $#1 \cdot #6/#7$ below does not cause an overflow: naively, $P$ can be up to 35, and $#6/#7$ up to $10^8$, but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot #6/#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on $P$, which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $#6/#7 \leq 10^8 \cdot Z$, hence $P \cdot #6/#7 \leq 4E \cdot 10^8 < 10^9$.
\cs_new:Npn \_fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9 \exp_after:wN \__fp_div_significand_vi:Nw \int_value:w \__fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 + \exp_after:wN \__fp_div_significand_v:NN \int_value:w \__fp_int_eval:w 199980 + 2*#4 - #1*#8 + \exp_after:wN \__fp_int_eval:w 200000 + 2*#5 - #1*#9 ; 
\cs_new:Npn \__fp_div_significand_v:NN #1#2 \__fp_int_eval_end: + 
\cs_new:Npn \__fp_div_significand_vi:Nw #1#2; 
\if_meaning:w 0 #1 \if_int_compare:w \__fp_int_eval:w #2 > 0 + 1 \fi: \else: + \fi: 1 
\if_meaning:w #2 - #1 - \else: + \fi: 1 
\fi: 
\} 
(End definition for \_fp_div_significand_iv:wwnnnnnn, \_fp_div_significand_v:NN, and \_fp_div_significand_vi:Nw.)
\__fp_div_significand_pack:NNN
At this stage, we are in the following situation: \TeX{} is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.
\__fp_div_significand_test_o:w \__fp_div_significand_pack:NNN 10^6 + Q_A \_fp_div_significand_pack:NNN 10^6 + Q_B \_fp_div_significand_pack:NNN 10^6 + Q_C \_fp_div_significand_pack:NNN 10^6 + Q_D \_fp_div_significand_pack:NNN 10^6 + 10 \cdot Q_E + 5 \cdot P + \varepsilon; \langle \text{sign} \rangle
Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that $P$ was the correct value, but not with an exact quotient, and $-1$ if $2E < PZ$, i.e., $P$ was an overestimate. The packing function we define now does nothing special: it removes the $10^6$ and carries two digits (for the $10^5$'s and the $10^4$'s).
\__fp_div_significand_pack:NN
\_fp_div_significand_test_o:w \_fp_div_significand_test_o:w 10 \langle 5d \rangle; \langle 4d \rangle; \langle 4d \rangle; \langle 5d \rangle; \langle \text{sign} \rangle
The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence $Q_A$ (the tilde denoting the contribution from the other $Q_i$) is at most 99999, and $10^6 + Q_A = 10 \cdots$
It is now time to round. This depends on how many digits the final result will have.
\__fp_div_significand_test_o:w \_fp_div_significand_test_o:w 10 \#1
{ \if_meaning:w 0 #1 \exp_after:wN \_fp_div_significand_small_o:wwNNNNwN \else: \exp_after:wN \_fp_div_significand_large_o:wwNNNNwN \fi: #1 } 
(End definition for \_fp_div_significand_test_o:w.)

836
Standard use of the functions \_fp\_basics\_pack\_low:NNNNNW and \_fp\_basics\_pack\_high:NNNNNW. We finally get to use the (final sign) which has been sitting there for a while.

\begin{verbatim}
\cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN 0 \{ \exp_after:wN \_fp_basics_pack_high:NNNNNW
\int_value:w \_fp_int_eval:w 1 \#1\#2\#3\#4#5#6#7#8; \#9
\exp_after:wN ;
\}
\end{verbatim}

(End definition for \_fp\_div_significand_small_o:wwwNNNNwN.)

\begin{verbatim}
\cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN \{ \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
\int_value:w \_fp_int_eval:w 1 \#1\#2\#3\#4\#5#6\#7#8; \#9
\exp_after:wN \_fp_basics_pack_weird_low:NNNNw
\int_value:w \_fp_int_eval:w 1 \#3\#4\#5#6 + \_fp_round:NNN \#7 \#8 \#9 \exp_after:wN ;
\exp_after:wN ;
\}
\end{verbatim}

(End definition for \_fp\_div_significand_large_o:wwwNNNNwN.)

31.4 Square root

\_fp\_sqrt\_o:w

Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than $-0$) have no real square root. Positive infinity, and \texttt{nan}, are unchanged. Finally, for normal positive numbers, there is some work to do.

\begin{verbatim}
\cs_new:Npn \_fp_sqrt\_o:w \#1 \s__fp \_fp_chk:w \#2\#3#4; \#0
\{ \if_meaning:w 0 \#2 \_fp_case\_return\_same\_o:w \fi;
\exp_after:wN \_fp_case\_use\_nw \{ \_fp\_invalid\_operation\_o:nw \{ \_fp\_sqrt\}\}
\fi;
\exp_after:wN \_fp\_sqrt\_npos\_o:w
\end{verbatim}

837
Prepare \texttt{\_fp\_sanitize:Nw} to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent \#1 of the argument. If the exponent \#1 is even, find a first approximation of the square root of the significand \(10^8a_1 + a_2 = 10^8\#2\#3 + \#4\#5\) through Newton's method, starting at \(x = 57234133 \approx 10^{7.75}\). Otherwise, first shift the significand of the argument by one digit, getting \(a_1' \in [10^6, 10^7]\) instead of \([10^7, 10^8]\), then use Newton's method starting at \(17782794 \approx 10^{7.25}\).

Newton's method maps \(x \mapsto \frac{1}{2} \left[ (x + [10^8a_1/x]) / 2 \right]\) in each iteration, where \([b/c]\) denotes \texttt{\_fp\_div:TeX} division. This division rounds the real number \(b/c\) to the closest integer, rounding ties away from zero, hence when \(c\) is even, \(b/c - 1/2 + 1/2c \leq [b/c] \leq b/c + 1/2\) and when \(c\) is odd, \(b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)\). For all \(c\), \(b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2\).

Let us prove that the method converges when implemented with \texttt{\_fp\_div:TeX} division, for any \(10^6 \leq a_1 < 10^8\) and starting value \(10^6 \leq x < 10^8\). Using the inequalities above and the arithmetic–geometric inequality \((x + t)/2 \geq \sqrt{xt}\) for \(t = 10^8a_1/x\), we find

\[
x' = \left[ \frac{x + [10^8a_1/x]}{2} \right] \geq \frac{x + 10^8a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8a_1} - \frac{1}{4} + \frac{1}{4x}.
\]

After any step of iteration, we thus have \(\delta = x - \sqrt{10^8a_1} \geq -0.25 + 0.25 \cdot 10^{-8}\). The new difference \(\delta' = x' - \sqrt{10^8a_1}\) after one step is bounded above as

\[
x' - \sqrt{10^8a_1} \leq \frac{x + 10^8a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8a_1} \leq \frac{\delta}{2} \sqrt{10^8a_1} + \frac{3}{4}.
\]
For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence $\delta$ decreases at each step: since all $x$ are integers, $\delta$ must reach a value $-1/4 < \delta < 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2^{\sqrt{10^8}a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for $x$ may contain two integers, hence $x$ might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8a_1/x] \leq 2x - 2$$

hence $10^8a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8a_1/(x - 1)] \geq 2x - 1$$

hence $10^8a_1/(x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer $x$. To stop the iteration when two consecutive results are equal, the function \_fp_sqrt_Newton::wn receives the newly computed result as #1, the previous result as #2, and $a_1$ as #3. Note that \_fpX combines the computation of a multiplication and a following division, thus avoiding overflow in #3 * 10000000 / #1. In any case, the result is within $[10^7, 10^8]$.

\begin{verbatim}
1971 \_fp_sqrt_auxi_o::NNNNwnnN
1972 { if_int_compare:w #1 = #2 \exp_stop_f:
1973 \_fp_sqrt_auxii_o:NnnnnnnnN
1975 \_exp_after:wN \_fp_sqrt_auxi_o::NNNNwnnN
1976 \_int_value:w \_fp_int_eval:w 9999 9999 +
1977 \_exp_after:wN \_fp_use_none_until_s:w
1978 \fi:
1979 \_exp_after:wN \_fp_sqrt_Newton::wn
1980 \_int_value:w \_fp_int_eval:w (#1 + #3 + 1 0000 0000 / #1) / 2 ;
1981 #1; {#3}
1982 }
1983 \_fp_sqrt_auxi_o::NNNNwnnN
\end{verbatim}

(End definition for \_fp_sqrt_Newton::wn.)

This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then $\{a_1\} \{a_2\} \{a_3\}$. Here, $x \approx \sqrt{10^{8}a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a_3$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \leq \sqrt{a}.$$ 

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, \_fp_sqrt_auxii_o::NNNNwnnN is called several times to get closer and closer underestimates of $\sqrt{a}$. By construction, the underestimates $y$ are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$. 

839
This receives a continuation function \#1, then five blocks of 4 digits for \( y \), then two 8-digit blocks and a single digit for \( a \). A common estimate of \( \sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \) is \((a - y^2)/(2y)\), which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer \( j \leq 6 \) such that \( 10^{4j}(a - y^2) < 2 \cdot 10^8 \), then computes the integer (with \( \varepsilon \)-TEX's rounding division)

\[
10^{4j}z = \left\lfloor \frac{(10^{4j}(a - y^2)) - 257 \cdot 10^8}{10^8 y + 1} \right\rfloor.
\]

The choice of \( j \) ensures that \( 10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9 \), thus \( 10^9 + 10^{4j}z \) has exactly 10 digits, does not overflow \( \varepsilon \)-TEX's integer range, and starts with 1. Incidentally, since all \( a - y^2 \leq 3.2 \cdot 10^{-8} \), we know that \( j \geq 3 \).

Let us show that \( z \) is an underestimate of \( \sqrt{a} - y \). On the one hand, \( \sqrt{a} - y \leq 16 \cdot 10^{-8} \) because this holds for the initial \( y \) and values of \( y \) can only increase. On the other hand, the choice of \( j \) implies that \( \sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^9 - 4j \). For \( j = 3 \), the first bound is better, while for larger \( j \), the second bound is better. For all \( j \in [3, 6] \), we find \( \sqrt{a} - y < 16 \cdot 10^{-2j} \). From this, we deduce that

\[
10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2) - (\sqrt{a} - y)^2}{2y} \geq \left\lfloor \frac{10^{4j}(a - y^2)}{2 \cdot 10^8 [10^8 y + 1]} \right\rfloor + \frac{1}{2}
\]

where we have replaced the bound \( 10^{4j}(16 \cdot 10^{-2j}) = 256 \) by 257 and extracted the corresponding term \( 1/(2 \cdot 10^{-8} [10^8 y + 1]) \geq 1/2 \). Given that \( \varepsilon \)-TEX's integer division obeys \( \lfloor b/c \rfloor \leq b/c + 1/2 \), we deduce that \( 10^{4j}z \leq 10^{4j}(\sqrt{a} - y) \), hence \( y + z \leq \sqrt{a} \) is an underestimate of \( \sqrt{a} \), as claimed. One implementation detail: because the computation involves \(-#4\#4 - 2\#3\#3 - 2\#2\#2\#2\#2\#2\) which may be as low as \(-5 \cdot 10^6\), we need to use the \texttt{pack_big} functions, and the \texttt{big} shifts.
We receive here the difference $a - y^2 = \sum d_i \cdot 10^{-4i}$, as $(d_2) \ldots (d_{10})$, where each block has 4 digits, except $(d_2)$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller $\_\_fp_sqrt_auxii_o:NnnnnnnN$, which completes the expression

$$10^{4j}z = \left[ \left( \frac{10^{4j}(a - y^2) - 257 \cdot 0.5 \cdot 10^8}{10^8 y + 1} \right) \right]$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the auxiv auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4d_2 + 10^3d_3 + 10^4d_4 + d_5 \geq 1$. In this last case, we detect when $10^{2j}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded $\sqrt{a}$ with only one more call to $\_\_fp_sqrt_auxii_o:NnnnnnnN$. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{10^8 y + 1} \geq (20000 - 257) \cdot 0.5 > 0.$$
\begin{verbatim}
\int_value:w \__fp_int_eval:w (#1#2#3#4 %)
\else:
 \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
 \int_value:w \__fp_int_eval:w (#1#2#3#4#5 %)
\fi:
\fi:
\fi:
\fi:
\cs_new:Npn \__fp_sqrt_auxviii_o:NNNNNw 1#1#2#3#4#5#6;
 { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5} {#60000} }
\cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
 { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
\cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
 { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
\cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
 { \if_int_compare:w #1#2 = 0 \exp_stop_f:
 \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
 \fi:
 \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
}

(End definition for \__fp_sqrt_auxviii_o:nnnnnnn and others.)
\end{verbatim}

Simply add the two 8-digit blocks of \( z \), aligned to the last four of the five 4-digit blocks of \( y \), then call the auxii auxiliary to evaluate \( y' = (y + z)^2 \).

\begin{verbatim}
\cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn \__fp_sqrt_auxvii_o:nnnnnnn \__fp_sqrt_auxviii_o:wnnnn
 { \__fp_sqrt_auxii_o:Nnnnnnnn \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5} }
\end{verbatim}

(End definition for \__fp_sqrt_auxviii_o:nnnnnnn and \__fp_sqrt_auxvii_o:nnnnnnn)

At this stage, \( j = 6 \) and \( 10^{24}z < 10^7 \), hence
\[
10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) \big/ (10^8y + 1),
\]
then \( 10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8}) \), and
\[
10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),
\]
which finally implies \( 0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16} \). In particular, \( y \) is an underestimate of \( \sqrt{a} \) and \( y + 0.5 \cdot 10^{-16} \) is a (strict) overestimate. There is at exactly one multiple \( m \) of \( 0.5 \cdot 10^{-16} \) in the interval \([y, y + 0.5 \cdot 10^{-16}]\). If \( m^2 > a \), then the square root is inexact and
is obtained by rounding $m - \epsilon$ to a multiple of $10^{-16}$ (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly $m$, and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16}]$; rather, only the last 4 digits #8 of $y$ are considered, and we do not perform any carry yet. The auxxi auxiliary sets up auxxi with a continuation function auxxii instead of auxxi as before. To prevent auxxi from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$. 

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess $m$ is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the auxxiv function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

(End definition for \_fp_sqrt_auxx_o:NNNNNNN and \_fp_sqrt_auxxi_o:WWNN.)
This receives 9998, 9999 or 10000 as \#1 when \( m \) is an underestimate, exact, or an overestimate, respectively. Then comes \( m \) as five blocks of 4 digits, but where the last block \#6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless \( m \) is an overestimate (\#1 is then 10000). Then comes \( a \) as three arguments. Rounding is done by \texttt{\_fp_round:NNN}, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are "rounded to even", and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by \texttt{\_fp_round_digit:Nw}, which receives (after removal of the 10000's digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```latex
\cs_new:Npn \_fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
\{\exp_after:wN \_fp_basics_pack_high:NNNNNw \int_value:w \_fp_int_eval:w 1 0000 0000 + #2#3\exp_after:wN \_fp_basics_pack_low:NNNNNw \int_value:w \_fp_int_eval:w 1 0000 0000 + #4#5\if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:\exp_after:wN \_fp_round:NNN \exp_after:wN 0 \exp_after:wN 0 \int_value:w \_fp_int_eval:w #6 + 19999 - #1 \} \exp_after:wN ;
```

(End definition for \texttt{\_fp_sqrt_auxxiv_o:wnnnnnnnN}.

31.5 About the sign and exponent

The exponent of a normal number is its \langle exponent \rangle minus one.

```latex\begin{verbatim}
\cs_new:Npn \_fp_logb_o:w ? \s__fp \_fp_chk:w #1#2; 0 \{ \_fp_case:w #1 \exp_stop_f: \_fp_case_use:nnw \_fp_division_by_zero_o:Nnw \c_minus_inf_fp \{ \_fp_logb_o:w \_fp_logb_aux_o:w \_fp_case_return_o:Nw \c_inf_fp \_else: \_fp_case_return_same_o:w \fi: \s__fp \_fp_chk:w #1 #2; \}
\end{verbatim}
```
\cs_new:Npn \__fp_logb_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
\{ \exp_after:wN \__fp_parse:n \exp_after:wN \{ \int_value:w \int_eval:w #3 - 1 \exp_after:wN \}
\}

(End definition for \__fp_logb_o:w and \__fp_logb_aux_o:w.)

\__fp_sign_o:w \__fp_sign_aux_o:w

Find the sign of the floating point: \texttt{nan}, +0, −0, +1 or −1.
\cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1 #2; @
\{ \if_case:w #1 \exp_stop_f:
\__fp_case_return_same_o:w
\or: \exp_after:wN \__fp_sign_aux_o:w
\or: \exp_after:wN \__fp_sign_aux_o:w
\else: \__fp_case_return_same_o:w
\fi:
\s__fp \__fp_chk:w #1 #2;
\}
\cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
\{ \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ \}

(End definition for \__fp_sign_o:w and \__fp_sign_aux_o:w.)

\__fp_set_sign_o:w

This function is used for the unary minus and for \texttt{abs}. It leaves the sign of \texttt{nan} invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on \#1. It also expands after itself in the input stream, just like \__fp+_o:ww.
\cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2 #3 #4 ; @
\{ \exp_after:wN \__fp_exp_after_o:w \exp_after:wN \s__fp \__fp_chk:w #2 \int_value:w \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f: \#4; \}

(End definition for \__fp_set_sign_o:w.)

31.6 Operations on tuples

\__fp_tuple_set_sign_o:w \__fp_tuple_set_sign_aux_o:Nnw \__fp_tuple_set_sign_aux_o:w

Two cases: \texttt{abs(⟨tuple⟩)} for which \#1 is 0 (invalid for tuples) and −⟨tuple⟩ for which \#1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.
\cs_new:Npn \__fp_tuple_set_sign_o:w #1
\{ \if_meaning:w 2 #1 \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw \exp_after:wN \s__fp \__fp_chk:w #1 #2 #3 #4 ; \}
\cs_new:Npn \__fp_tuple_set_sign_aux_o:Nnw #1 #2 #3 #4 @
\{ \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw \exp_after:wN \s__fp \__fp_chk:w \int_value:w \if_case:w #3 \exp_stop_f: \#1 \or: 1 \or: 0 \fi: \exp_stop_f: \#4; \}

(End definition for \__fp_tuple_set_sign_o:w.)
\texttt{\__fp_tuple_map_o:nnw \__fp_tuple_set_sign_aux_o:w \#3}
\texttt{\cs_new:Npn \__fp_tuple_set_sign_aux_o:w \#1\#2 ;}
\texttt{\__fp_change_func_type:NNN \#1 \__fp_set_sign_o:w}
\texttt{\__fp_parse_apply_unary_error:Nww}
\texttt{2 \#1 \#2 ; @}

(End definition for \__fp_tuple_set_sign_o:w, \__fp_tuple_set_sign_aux_o:Nnw, and \__fp_tuple_set_sign_aux_o:w.)

\texttt{\__fp_*_tuple_o:ww}
\texttt{\__fp_tuple_*_o:ww}
\texttt{\__fp_tuple_/o:ww}

For \langle number\rangle*\langle tuple\rangle and \langle tuple\rangle*\langle number\rangle and \langle tuple\rangle/\langle number\rangle, loop through the \langle tuple\rangle some code that multiplies or divides by the appropriate \langle number\rangle. Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

\texttt{\cs_set_protected:Npn \__fp_tmp:w #1}
\texttt{\cs_new:cpn { \__fp_tuple_#1_tuple_o:ww }}
\texttt{\s__fp_tuple \__fp_tuple_chk:w ##1 ;}
\texttt{\s__fp_tuple \__fp_tuple_chk:w ##2 ;}
\texttt{\int_compare:nNnTF}
\texttt{\__fp_array_count:n {##1} = \__fp_array_count:n {##2} }
\texttt{\__fp_tuple_mapthread_o:nww \__fp_binary_type_o:Nww \#1 ;}
\texttt{\__fp_invalid_operation_o:nww \#1 ;}
\texttt{\s__fp_tuple \__fp_tuple_chk:w {##1} ;}
\texttt{\s__fp_tuple \__fp_tuple_chk:w {##2} ;}

(End definition for \__fp_*_tuple_o:ww, \__fp_tuple_*_o:ww, and \__fp_tuple_/o:ww.)

\texttt{\__fp_tuple+_tuple_o:ww}
\texttt{\__fp_tuple-_tuple_o:ww}

Check the two tuples have the same number of items and map through these a helper that dispatches appropriately depending on the types. This means \((1,2)+((1,1),2)\) gives \((\text{nan},4)\).

\texttt{\cs_set_protected:Npm \__fp_tmp:w \#1}
\texttt{\cs_new:cpn { \__fp_tuple_#1_tuple_o:ww }}
\texttt{\s__fp_tuple \__fp_tuple_chk:w ###1 ;}
\texttt{\s__fp_tuple \__fp_tuple_chk:w ###2 ;}
\texttt{\int_compare:nNnTF}
\texttt{\__fp_array_count:n {###1} = \__fp_array_count:n {###2} }
\texttt{\__fp_tuple_mapthread_o:nww \__fp_binary_type_o:Nww \#1 }\texttt{\__fp_invalid_operation_o:nww \#1}
\texttt{\s__fp_tuple \__fp_tuple_chk:w {###1} ;}
\texttt{\s__fp_tuple \__fp_tuple_chk:w {###2} ;}

(End definition for \__fp_tuple+_tuple_o:ww and \__fp_tuple-_tuple_o:ww.)

32 l3fp-extended implementation

(*intex package)
\langle@=fp\rangle
32.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

\[ \langle a_1 \rangle \langle a_2 \rangle \langle a_3 \rangle \langle a_4 \rangle \langle a_5 \rangle \langle a_6 \rangle ; \]

where each \( \langle a_i \rangle \) is exactly 4 digits (ranging from 0000 to 9999), except \( \langle a_1 \rangle \), which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number \( a \) corresponding to the representation above is

\[ a = \sum_{i=1}^{6} \langle a_i \rangle \cdot 10^{-4i}. \]

Most functions we define here have the form

\[ \langle \text{calculation} \rangle : w n \langle \text{operand}_1 \rangle ; \langle \text{operand}_2 \rangle ; \{ \langle \text{continuation} \rangle \} \]

They perform the \( \langle \text{calculation} \rangle \) on the two \( \langle \text{operands} \rangle \), then feed the result (6 brace groups followed by a semicolon) to the \( \langle \text{continuation} \rangle \), responsible for the next step of the calculation. Some functions only accept an \( \text{N-type} \) \( \langle \text{continuation} \rangle \). This allows constructions such as

\[ \langle \text{fp_fixed_add} \rangle : w n \langle X_1 \rangle ; \langle X_2 \rangle ; \langle \text{fp_fixed_mul} \rangle : w n \langle X_3 \rangle ; \langle \text{fp_fixed_add} \rangle : w n \langle X_4 \rangle ; \]

to compute \( (X_1 + X_2) \cdot X_3 + X_4 \). This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using \( \langle \text{fp_fixed_to_float} \rangle : w N \). This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

32.2 Helpers for numbers with extended precision

\c__fp_one_fixed_tl

The fixed-point number 1, used in l3fp-expo.

\c__fp_fixed_continue:wn

This function simply calls the next function.
This function adds 1 to the fixed point \( \langle a \rangle \), by changing \( a_1 \) to \( 10000 + a_1 \), then calls the \( \langle \text{continuation} \rangle \). This requires \( a_1 + 10000 < 2^{31} \).

\begin{verbatim}
\cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
{ \exp_after:wN #3 \exp_after:wN \{ \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 \} #2 ; }
\end{verbatim}

(End definition for \__fp_fixed_add_one:wN.)

\__fp_fixed_div_myriad:wn

Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group \#1 may have any number of digits, and we must split \#1 into the new first group and a second group of exactly 4 digits. The choice of shifts allows \#1 to be in the range \([0, 5 \cdot 10^8 - 1]\).

\begin{verbatim}
\cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6; #7#8#9;
{ \exp_after:wN \__fp_fixed_mul_after:wwn \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int \exp_after:wN \__fp_pack:NNNNNw \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int + #1 \exp_after:wN \__fp_pack:NNNNNw \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int + #2*#7 + #3*#8*#9; }
\end{verbatim}

(End definition for \__fp_fixed_div_myriad:wn.)

\__fp_fixed_mul_after:wwn

The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the \( \langle \text{continuation} \rangle \) \#3 in front.

\begin{verbatim}
\cs_new:Npn \__fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
\end{verbatim}

(End definition for \__fp_fixed_mul_after:wwn.)

### 32.3 Multiplying a fixed point number by a short one

\__fp_fixed_mul_short:wwn

\( \{ a_1 \} \{ a_2 \} \{ a_3 \} \{ a_4 \} \{ a_5 \} \{ a_6 \} ; \{ \text{continuation} \} \)

Computes the product \( c = ab \) of \( a = \sum_i (a_i)10^{-4i} \) and \( b = \sum_i (b_i)10^{-4i} \), rounds it to the closest multiple of \( 10^{-24} \), and leaves \( \langle \text{continuation} \rangle \) \{c_1\} \ldots \{c_6\} ; \) in the input stream, where each of the \( \{c_i\} \) are blocks of 4 digits, except \( \{c_1\} \), which is any \TeX integer. Note that indices for \( \{b\} \) start at 0: for instance a second operand of \( \{0001\} \{0000\} \{0000\} \) leaves the first operand unchanged (rather than dividing it by \( 10^4 \), as \__fp_fixed_mul:wwn would).

\begin{verbatim}
\cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
{ \exp_after:wN \__fp_fixed_mul_after:wwn \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int + #1*#7 \exp_after:wN \__fp_pack:NNNNNw \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int + #1*#8 + #2*#7 \exp_after:wN \__fp_pack:NNNNNw \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int }
\end{verbatim}

848
32.4 Dividing a fixed point number by a small integer

\[ \text{\_fp_fixed_div_int:wwN} \ (a \ ; \ \{n\} \ ; \ \{\text{continuation}\}) \]

Divides the fixed point number \( a \) by the (small) integer \( 0 < \langle n \rangle < 10^4 \) and feeds the result to the \{continuation\}. There is no bound on \( a_i \).

The arguments of the \( i \) auxiliary are 1: one of the \( a_i \), 2: \( n \), 3: the \( ii \) or the \( iii \) auxiliary. It computes a (somewhat tight) lower bound \( Q_i \) for the ratio \( a_i/n \).

The \( ii \) auxiliary receives \( Q_i \), \( n \), and \( a_i \) as arguments. It adds \( Q_i \) to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes \( a_i - n \cdot Q_i \), placing the result in front of the 4 digits of \( a_i+1 \). The resulting \( a_i+1 = 10^4(a_i - n \cdot Q_i) + a_i+1 \) serves as the first argument for a new call to the \( i \) auxiliary.

When the \( iii \) auxiliary is called, the situation looks like this:

\[ \text{\_fp_fixed_div_int_after:Nw} \ (\text{continuation}) \]
\[ -1 + Q_i \]
\[ \text{\_fp_fixed_div_int_pack:Nw} \ 9999 + Q_2 \]
\[ \text{\_fp_fixed_div_int_pack:Nw} \ 9999 + Q_3 \]
\[ \text{\_fp_fixed_div_int_pack:Nw} \ 9999 + Q_4 \]
\[ \text{\_fp_fixed_div_int_pack:Nw} \ 9999 + Q_5 \]
\[ \text{\_fp_fixed_div_int_pack:Nw} \ 9999 \]
\[ \text{\_fp_fixed_div_int_auxii:wwn} \ Q_6 \ ; \ \{\langle n\rangle\} \ \{\langle a_0\rangle\} \]

where expansion is happening from the last line up. The \( iii \) auxiliary adds \( Q_6 + 2 \approx a_0/n + 1 \) to the last 9999, giving the integer closest to \( 10000 + a_0/n \).

Each \texttt{pack} auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the \texttt{pack} auxiliary thus produces one brace group. The last brace group is produced by the \texttt{after} auxiliary, which places the \{continuation\} as appropriate.

\[ \text{\cs_new:Npn} \ \_fp_fixed_div_int:wwN \ #1\#2\#3\#4\#5\#6 \ ; \ #7 \ ; \ #8 \]
\[ \{ \]
\[ \exp_after:wN \ \_fp_fixed_div_int_after:Nw \]
\[ \exp_after:wN \ \#8 \]
\[ \int_value:w \ \_fp_int_eval:w - 1 \]
\[ \_fp_fixed_div_int:wwN \]
\[ \#1 \ ; \ \{\#7\} \ \_fp_fixed_div_int_auxii:wwn \]
32.5 Adding and subtracting fixed points

\begin{itemize}
  \item \texttt{\_fp_fixed_add:wn} \(a\); \(b\); \{\textit{continuation}\}
  \item \texttt{\_fp_fixed_sub:wn} \(a\); \(b\); \{\textit{continuation}\}
\end{itemize}

Computes \(a + b\) (resp. \(a - b\)) and feeds the result to the \textit{continuation}. This function requires \(0 \leq a_1, b_1 \leq 114748\), its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: \((a \pm b)_1 < 100000\). The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, \(a_1, \ldots, a_4\), the rest of \(a\), and \(b_1\) and \(b_2\). The second auxiliary receives the rest of \(a\), the sign multiplying \(b\), the rest of \(b\), and the \textit{continuation} as arguments. After going down through the various level, we go back up, packing digits and bringing the \textit{continuation} (#8, then #7) from the end of the argument list to its start.

\begin{itemize}
  \item \texttt{\_fp_fixed_add:wn} \{ \texttt{\_fp_fixed_add:Nnnnnnn} + \}
  \item \texttt{\_fp_fixed_add:wn} \{ \texttt{\_fp_fixed_add:Nnnnnnn} - \}
  \item \texttt{\_fp_fixed_sub:wn} \{ \texttt{\_fp_fixed_sub:Nnnnnnn} \}
\end{itemize}

(End definition for \texttt{\_fp_fixed_div_int:wn} and others.)
32.6 Multiplying fixed points

\__fp_fixed_mul:wwn (a) ; (b) ; ⟨continuation⟩

Computes \( a \times b \) and feeds the result to \( ⟨\text{continuation}⟩ \). This function requires \( 0 \leq a_1, b_1 < 10000 \). Once more, we need to play around the limit of 9 arguments for \TeX{} macros. Note that we don’t need to obtain an exact rounding, contrarily to the \( \ast \) operator, so things could be harder. We wish to perform carries in

\[
\begin{align*}
a \times b &= a_1 \cdot b_1 \cdot 10^{-8} \\
&\quad + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
&\quad + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
&\quad + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
&\quad + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\
&\quad \quad + a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1 \right) \cdot 10^4 \\
&\quad + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{align*}
\]

where the \( O(10^{-24}) \) stands for terms which are at most \( 5 \cdot 10^{-24} \); ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on \( a_1, \ldots, a_4 \) and \( b_1, \ldots, b_4 \), while the last 6 terms only depend on \( a_1, a_2, a_5, a_6 \), and the corresponding parts of \( b \). Hence, the first function grabs \( a_1, \ldots, a_4 \), the rest of \( a \), and \( b_1, \ldots, b_4 \), and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The \( i \) auxiliary receives \( a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6 \) and finally the \( ⟨\text{continuation}⟩ \) as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The \( ⟨\text{continuation}⟩ \) is finally placed in front of the 6 brace groups by \( \__fp_fixed_mul_after:wwn \).
Combining product and sum of fixed points

\[ a \times b + c = (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1 \right) \cdot \frac{10^4}{10^{24}} + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6) \cdot 10^{-24} + O(10^{-24}), \]

where \( c_1 c_2, c_3 c_4, c_5 c_6 \) denote the 8-digit number obtained by juxtaposing the two blocks of digits of \( c \), and \( \cdot \) denotes multiplication. The task is obviously tough because we have 18 branch groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to \( 10^{-4} \), is empty), with \( c_1 c_2 \) in the first level, calls the \( i \) auxiliary with arguments described later, and adds a trailing \( + c_5 c_6 ; \{ (continuation) \} \). The \( + c_5 c_6 \) piece, which is omitted for \( \_\_fp_fixed_one_minus_mul:wwn \), is taken in the integer expression for the \( 10^{-24} \) level.
\int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int + \#3 \#4
\_fp_fixed_mul_add:Nw
\_fp_int_eval:w \c__fp_big_middle_shift_int + \#3 \#4
+ \#5 \#6 ; \#2 ; \#1 ; \#2 ; +
+ \#7 \#8 ; ;
\}
\cs_new:Npn \_fp_fixed_mul_sub_back:wwwn #1; #2; #3\#4\#5\#6\#7\#8;
{
\exp_after:wN \_fp_fixed_mul_after:wwn
\int_value:w \_fp_int_eval:w \c__fp_big_leading_shift_int
\exp_after:wN \_fp_pack_big:NNNNNNw
\int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int + \#3 \#4
\_fp_fixed_mul_add:Nw
+ \#5 \#6 ; \#2 ; \#1 ; \#2 ; +
+ \#7 \#8 ; ;
}
\cs_new:Npn \_fp_fixed_one_minus_mul:wwn #1; #2;
{
\exp_after:wN \_fp_fixed_mul_after:wwn
\int_value:w \_fp_int_eval:w \c__fp_big_leading_shift_int
\exp_after:wN \_fp_pack_big:NNNNNNw
\int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int +
1 0000 0000
\_fp_fixed_mul_add:Nw
- ; \#2 ; \#1 ; \#2 ; -
;
;
}
(End definition for \_fp_fixed_mul_add:wwn, \_fp_fixed_mul_sub_back:wwwn, and \_fp_fixed_one_minus_mul:wwn.)

\_fp_fixed_mul_add:Nw
\_fp_fixed_mul_add:wwn (op) + ⟨c⟩ (ci) ;
⟨b⟩; ⟨a⟩; ⟨b⟩; ⟨op⟩
+ ⟨ci⟩ (ci) ;
Here, ⟨op⟩ is either + or -. Arguments #3, #4, #5 are ⟨b1⟩, ⟨b2⟩, ⟨b3⟩; arguments #7, #8, #9 are ⟨a1⟩; ⟨a2⟩, ⟨a3⟩. We can build three levels: ⟨a1⟩ · ⟨b1⟩ for 10^{-8}, ⟨a1⟩ · ⟨b2⟩ for 10^{-12}, and ⟨a1⟩ · ⟨b3⟩ + ⟨a2⟩ · ⟨b2⟩ + ⟨a3⟩ · ⟨b1⟩ + c_{34} for 10^{-16}. The a-b products use the sign #1. Note that #2 is empty for \_fp_fixed_one_minus_mul:wwn. We call the ii auxiliary for levels 10^{-20} and 10^{-24}, keeping the pieces of ⟨a⟩ we’ve read, but not ⟨b⟩, since there is another copy later in the input stream.
\cs_new:Npn \_fp_fixed_mul_add:Nw #1 \#2; #3\#4\#5\#6; #7\#8\#9
{
 #1 \#7\#3
 \exp_after:wN \_fp_pack_big:NNNNNNw
 \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
 #1 \#7\#4 \#1 \#8\#3
 \exp_after:wN \_fp_pack_big:NNNNNNw
 \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
 #1 \#7\#5 \#1 \#8\#4 \#1 \#9\#3 \#2
 \exp_after:wN \_fp_pack_big:NNNNNNw
 \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
 #1 \_fp_fixed_mul_add:nnnnw
 \{\#7\#8\#9\}
}
(End definition for \_fp_fixed_mul_add:Nw.)
Level $10^{-20}$ is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the $i$ auxiliary. Then we prepare level $10^{-24}$. We don’t have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_1 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_1 \cdot b_3 + a_3 \cdot b_4 + a_2.$$ 

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+c_5c_6$.

To do all this, we keep $a_1$, $a_5$, $a_6$, and the corresponding pieces of $\langle b \rangle$.

The second one is divided by 10000: this is the carry from level $10^{-28}$. The trailing $+c_5c_6$ is taken into the expression for level $10^{-24}$. Note that the total of level $10^{-24}$ is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See l3fp-aux for the definition of the shifts and packing auxiliaries.

Complete the $\langle \text{partial}_1 \rangle$ and $\langle \text{partial}_2 \rangle$ expressions as explained for the $ii$ auxiliary.

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.
Converting an extended-precision number with an exponent at most 4 and a first block less than $10^8$ to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

\begin{verbatim}
\cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
\{ \exp_after:wN \__fp_ep_to_fixed_auxi:www
\int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
\exp:w \exp_end_continue_f:w
\prg_replicate:nn { 4 - \int_max:nn {#1} {-32} } { 0 } ;
\}
\cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
\{ \__fp_pack_eight:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
#2 #1#3#4#5#6#7 0000 ! \}
\cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
\{ #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; \}
\end{verbatim}

(End definition for \__fp_ep_to_fixed:wwn, \__fp_ep_to_fixed_auxi:www, and \__fp_ep_to_fixed_auxii:nnnnnnnwn.)

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation \#8 is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The loop auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the end auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by 0, 0. Then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with \__fp_use_i:ww any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

\begin{verbatim}
\cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
\{ \exp_after:wN \__fp_ep_to_ep_loop:N
\int_value:w \__fp_int_eval:w #1 + 4 \exp_after:wN \use_i:nn \exp_after:wN \__fp_ep_to_ep_loop:N
\int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
#3#4#5#6#7 ; ! \}
\cs_new:Npn \__fp_ep_to_ep_loop:N #1
\{ \if_meaning:w 0 #1
\- 1 \else:
\__fp_ep_to_ep_end:www \#1 \fi:
\__fp_ep_to_ep_loop:N
\end{verbatim}

855
In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in $[1000, 9999]$.

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas $\#2$ and $\#4$ as fixed point numbers, and sum the exponents $\#1$ and $\#3$. The result’s first block is in $[100, 9999]$. 

(End definition for \_fp_ep_to_ep:wwN and others.)

(End definition for \_fp_ep_compare:www and \_fp_ep_compare_aux:www.)

856
Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in \texttt{l3fp-basics} for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call \(\langle n \rangle\) the numerator and \(\langle d \rangle\) the denominator. After a simple normalization step, we can assume that \(\langle n \rangle \in (0.1, 1)\) and \(\langle d \rangle \in (0.1, 1)\), and compute \(\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)\). In terms of the 6 blocks of digits \(\langle n_1 \rangle \cdots \langle n_6 \rangle\) and the 6 blocks \(\langle d_1 \rangle \cdots \langle d_6 \rangle\), the condition translates to \(\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]\).

We first find an integer estimate \(a \simeq 10^8 / \langle d \rangle\) by computing

\[
\alpha = \left\lfloor \frac{10^9}{\langle d_1 \rangle + 1} \right\rfloor
\]

\[
\beta = \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor
\]

\[
a = 10^3\alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,
\]

where \(\left\lfloor \frac{\alpha}{\beta} \right\rfloor\) denotes \(\varepsilon\)-\TeX’s rounding division, which rounds ties away from zero. The idea is to interpolate between \(10^3\alpha\) and \(10^3\beta\) with a parameter \(\langle d_2 \rangle / 10^4\), so that when \(\langle d_2 \rangle = 0\) one gets \(a = 10^3\beta - 1250 \simeq 10^{12} / \langle d_1 \rangle \simeq 10^9 / \langle d \rangle\), while when \(\langle d_2 \rangle = 9999\) one gets \(a = 10^3\alpha - 1250 \simeq 10^{12} / (\langle d_1 \rangle + 1) \simeq 10^8 / \langle d \rangle\). The shift by 1250 helps to ensure that \(a\) is an underestimate of the correct value. We shall prove that

\[
1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.
\]

We can then compute the inverse of \(\langle d \rangle a / 10^8 = 1 - \epsilon\) using the relation \(1 / (1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4\), which is correct up to a relative error of \(\epsilon^5 < 1.6 \cdot 10^{-24}\). This allows us to find the desired ratio as

\[
\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).
\]

Let us prove the upper bound first (multiplied by \(10^{15}\)). Note that \(10^7 \langle d \rangle < 10^3 \langle d_1 \rangle + 10^{-1} (\langle d_2 \rangle + 1)\), and that \(\varepsilon\)-\TeX’s division \(\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor\) underestimates \(10^{-1} (\langle d_2 \rangle + 1)\) by 0.5 at
most, as can be checked for each possible last digit of \(\langle d_2 \rangle\). Then,

\[
10^7\langle d \rangle a \lessdot \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \left(10^3 - \left[\frac{\langle d_2 \rangle}{10}\right]\right) \beta + \left[\frac{\langle d_2 \rangle}{10}\right] \alpha - 1250
\]

(1)

\[
< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10}\right]\right) \left(10^9 + 1\right) + \left[\frac{\langle d_2 \rangle}{10}\right] \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2}\right) - 1250\right)
\]

(2)

\[
< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \left(\frac{10^12}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10}\right] \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750\right)
\]

(3)

We recognize a quadratic polynomial in \(\langle d_2 \rangle / 10\) with a negative leading coefficient: this polynomial is bounded above, according to \((\langle d_2 \rangle/10 + a)(b - c(\langle d_2 \rangle/10)) \leq (b + ca)^2/(4c)\). Hence,

\[
10^7\langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle(\langle d_1 \rangle + 1)\right)^2
\]

Since \(\langle d_1 \rangle\) takes integer values within [1000, 9999], it is a simple programming exercise to check that the squared expression is always less than \(\langle d_1 \rangle(\langle d_1 \rangle + 1)\), hence \(10^7\langle d \rangle a < 10^{15}\). The upper bound is proven. We also find that \(d\) can be replaced by slightly smaller numbers, but nothing less than 0.374563... , and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

\[
10^7\langle d \rangle a > \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] - \frac{1}{2}\right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10}\right] \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 1750\right)
\]

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values \(y/10 = 0\) or \(y/10 = 100\), and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that \(a < 10^{15}/d \leq 10^8\), hence we can compute \(a\) safely as a \TeX{} integer, and even add \(10^9\) to it to ease grabbing of all the digits. The lower bound implies \(10^8 - 1755 < a\), which we do not care about.

\__fp_ep_div:wwww

Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range [100, 9999], and is placed after the \langle continuation \rangle once we are done. First normalize the inputs so that both first block lie in [1000, 9999], then call \__fp_ep_div_esti:wwww (denominator) \langle numerator \rangle, responsible for estimating the inverse of the denominator.

2383 \cs_new:Npn \__fp_ep_div:wwww \#1,#2; \#3,#4;
2389 \{
2390 \__fp_ep_to_ep:wwN \#1,#2;
2391 \__fp_fixed_continue:wn
2392 \{
2393 \__fp_ep_to_ep:wwN \#3,#4;
2394 \__fp_ep_div_esti:wwww
2395 \}
2396 \}

858
The \textit{esti} function evaluates $\alpha = 10^9/(\langle d \rangle + 1)$, which is used twice in the expression for $a$, and combines the exponents \#1 and \#4 (with a shift by 1 because we later compute $\langle n \rangle /(10\langle d \rangle)$). Then the \textit{estii} function evaluates $10^9 + a$, and puts the exponent \#2 after the continuation \#7: from there on we can forget exponents and focus on the mantissa. The \textit{estiii} function multiplies the denominator \#7 by $10^{-8}a$ (obtained as a split into the single digit \#1 and two blocks of 4 digits, \#2#3#4#5 and \#6). The result $10^{-8}a(d) = (1-\epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to \texttt{\_fp_ep_div_epsi:wnNNNNn}, which computes $10^{-9}a/(1-\epsilon)$, that is, $1/(10\langle d \rangle)$ and we finally multiply this by the numerator \#8.

\begin{verbatim}
\cs_new:Npn \__fp_ep_div_esti:wwwwn #1,#2#3; #4, {
    \exp_after:wN \__fp_ep_div_estii:wwnnwwn
    \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
    \exp_after:wN ;
    \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
    (#2) #3; }
\cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7 {
    \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
    \int_value:w \__fp_int_eval:w 10 0000 0000 - 1750
    + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
    (#3)#4#5; #6; { #7 #2, }
}
\cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7; {
    \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
    \__fp_ep_div_epsi:wnNNNNn (#1#2#3#4)#5#6
    \__fp_fixed_mul:wwn }
\end{verbatim}

The bounds shown above imply that the \textit{epsi} function’s first operand is $(1-\epsilon)$ with $\epsilon \in [0,1.755 \cdot 10^{-5}]$. The \textit{epsi} function computes $\epsilon$ as $1 - (1-\epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use \#1 (which is 9999). Then \textit{epsi} evaluates $10^{-9}a/(1-\epsilon)$ as $(1+\epsilon^2)(1+\epsilon)(10^{-9}a)/10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by $\epsilon$ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of \texttt{short_mul} and \texttt{div_myriad} is both faster and more precise than a simple mul.

\begin{verbatim}
\cs_new:Npn \__fp_ep_div_epsi:wnNNNNn #1#2#3#4#5#6; {
    \exp_after:wN \__fp_ep_div_epi:wwnnnnn
    \int_value:w \__fp_int_eval:w 1 9998 - #2
    \exp_after:wN \__fp_ep_div_epi:wwn
    \int_value:w \__fp_int_eval:w 1 9999 9998 - #3#4
    \exp_after:wN \__fp_ep_div_epi:wnNNNNw
    \int_value:w \__fp_int_eval:w 2 0000 0000 - #5#6 ;
}
\end{verbatim}
32.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have \( x \in [0.01, 1) \). Then find an integer approximation \( r \in [1001, 1003) \) of \( 10^2/\sqrt{x} \), starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace \( 10^8 \) by a slightly larger number which ensures that \( r^2x \geq 10^4 \). This also causes \( r \in [1001, 1003) \). Another correction to the above is that the input is actually normalized to \([0.1, 1)\), and we use either \( 10^8 \) or \( 10^9 \) in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation \( r \), we set \( y = 10^{-4}r^2x \) (or rather, the correct power of 10 to get \( y \approx 1 \)) and compute \( y^{-1/2} \) through another application of Newton’s method. This time, the starting value is \( z = 1 \), each step maps \( z \mapsto z(1.5 - 0.25z^2) \), and we perform a fixed number of steps. Our final result combines \( r \) with \( y^{-1/2} \) as \( x^{-1/2} = 10^{-2}ry^{-1/2} \).
\exp_after:wN \__fp_ep_isqrt_auxii:wwnnnwn
\int_value:w \__fp_int_eval:w
\int_if_odd:nTF {#1}
{ (1 - #1) / 2 , 535 , { 0 } { } }
{ 1 - #1 / 2 , 168 , { } { 0 } }
\cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
{\__fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
{#5} #6 ; { #7 #1 , }
\end{definition}
\__fp_ep_isqrt_estii:wwwnnwn
\__fp_ep_isqrt_estiii:NNNNNwwwn
\if_int_compare:w #1 = #2 \exp_stop_f:
\exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
\fi:
\exp_after:wN \__fp_ep_isqrt_esti:wwwnnwn
\int_value:w \__fp_int_eval:w
(\#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
#1, #3, {#4}
\exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
\int_value:w \__fp_int_eval:w
10000 0000 + #2 * #2 #5 * 5
\exp_after:wN , \int_value:w \__fp_int_eval:w
1000000 + #2 ;
\cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
{ \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
\__fp_ep_isqrt_epsii:wN
\__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
\end{definition}
\__fp_ep_isqrt_epsii:wN
Here, we receive a fixed point number y/2 with y ∈ [1,1.0201]. Starting from z = 1 we iterate z → z(3/2 − z²y/2). In fact, we start from the first iteration z = 3/2 − y/2 to avoid useless multiplications. The epsii auxiliary receives z as #1 and y as #2.
32.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

\begin{verbatim}
\cs_new:Npn \__fp_ep_to_float_o:wwN #1, \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wN }
\cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2; \__fp_ep_div:wwwwn 1,\{1000\}{0000}\{0000\}{0000}\{0000\}; #1,#2; \__fp_ep_to_float_o:wwN }
\end{verbatim}

(End definition for \__fp_ep_to_float_o:wwN and \__fp_ep_inv_to_float_o:wwN.)

\begin{verbatim}
\cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1; \__fp_fixed_mul:wwn #1; \{ \__fp_ep_to_float_o:wwN 2, \}
\end{verbatim}

(End definition for \__fp_fixed_to_float_rad_o:wN.)

\begin{verbatim}
\cs_new:Npn \__fp_fixed_inv_to_float_o:wN \__fp_ep_inv_to_float_o:wwN 0, \}
\end{verbatim}

(End definition for \__fp_fixed_inv_to_float_o:wN.)

\begin{verbatim}
\cs_new:Npn \__fp_fixed_to_float_rad_o:wN \__fp_fixed_to_float_o:wwN \__fp_fixed_mul:wwn \{5729\}{5779}\{5130\}{8232}\{0876\}{7981}; \__fp_ep_to_float_o:wwN 2, \}
\end{verbatim}

(End definition for \__fp_fixed_to_float_rad_o:wN.)
\__fp_fixed_to_float_o:wN \__fp_int_eval:w (exponent) \__fp_fixed_to_float_o:wN \{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\}; \langle \text{sign} \rangle \quad \text{yields} \\
\langle \text{exponent}' \rangle; \{\langle a_1' \rangle\} \{\langle a_2' \rangle\} \{\langle a_3' \rangle\} \{\langle a_4' \rangle\} \\
And the to_fixed version gives six brace groups instead of 4, ensuring that 1000 ≤ \langle a_1' \rangle ≤ 9999. At this stage, we know that \langle a_1 \rangle is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8.\footnote{Bruno: I must double check this assumption.}
\cs_new:Npn \__fp_fixed_to_float_pack:ww \#1 ; \#2 ; ;
{ \if_int_compare:w \#2 > 4 \exp_stop_f:
\exp_after:wN \__fp_fixed_to_float_round_up:wnnnw
\fi:
\fi:
; \#1 ;
}
\cs_new:Npn \__fp_fixed_to_float_round_up:wnnnw ; \#1 \#2 \#3 \#4 ;
{ \exp_after:wN \__fp_basics_pack_high:NNNNNw
\int_value:w \__fp_int_eval:w 1 \#1 \#2
\exp_after:wN \__fp_basics_pack_low:NNNNNw
\int_value:w \__fp_int_eval:w 1 \#3 \#4 + 1 ;
}

(End definition for \__fp_fixed_to_float_o:wN and \__fp_fixed_to_float_o:Nw.)

33 \texttt{l3fp-expo} implementation

\*\texttt{\texttt{\texttt{\texttt{\texttt{l3fp-expo}}} implementation}}

\cs_new:Npn \__fp_parse_word_exp:N \{ \__fp_parse_unary_function:NNN \__fp_exp_o:w ? \}
\cs_new:Npn \__fp_parse_word_ln:N \{ \__fp_parse_unary_function:NNN \__fp_ln_o:w ? \}
\cs_new:Npn \__fp_parse_word_fact:N \{ \__fp_parse_unary_function:NNN \__fp_fact_o:w ? \}

(End definition for \__fp_parse_word_exp:N, \__fp_parse_word_ln:N, and \__fp_parse_word_fact:N)

33.1 Logarithm

33.1.1 Work plan

As for many other functions, we filter out special cases in \__fp Ln o:w. Then \__fp Ln npos o:w receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in (0,1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1,10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1,1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of $\ln$ near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1 + t}{1 - t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$
where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \ldots,$$

is not too difficult to compute.

### 33.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don’t need the value of $\ln(5)$.

```
\c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl
\c__fp_ln_iii_fixed_tl
\c__fp_ln_iv_fixed_tl
\c__fp_ln_vi_fixed_tl
\c__fp_ln_vii_fixed_tl
\c__fp_ln_viii_fixed_tl
\c__fp_ln_ix_fixed_tl
\c__fp_ln_x_fixed_tl
```

End definition for `\c__fp_ln_i_fixed_tl` and others.

### 33.1.3 Sign, exponent, and special numbers

The logarithm of negative numbers (including $-\infty$ and $-0$) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `\_fp_ln_npos_o:w`.

```
\cs_new:Npn \_fp_ln_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
```

(End definition for `\_fp_ln_o:w`.

### 33.1.4 Absolute ln

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least $10^{-4}$, and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```
\cs_new:Npn \_fp_ln_npos_o:w \s__fp \_fp_chk:w #2#3#4;
```

\int_value:w % for the overall sign

865
This function expands to

\( Y = -\ln(X) \) as an extended fixed point.

We have thus found \( c \in [1, 10] \) such that \( 0.7 \leq ac < 1.4 \) in all cases. Compute \( 1 + x = 1 + ac \in [1.7, 2.4] \).
The Taylor series to be used is expressed in terms of \( t \equiv (x - 1)/(x + 1) = 1 - 2/(x+1) \).

We now compute the quotient with extended precision, reusing some code from \__fp_/o:ww. Note that \( 1 + x \) is known exactly.

To reuse notations from l3fp-basics, we want to compute \( A/Z \) with \( A = 2 \) and \( Z = x + 1 \). In l3fp-basics, we considered the case where both \( A \) and \( Z \) are arbitrary, in the range \([0.1, 1)\), and we had to monitor the growth of the sequence of remainders \( A, B, C, \) etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly \( 10^9 \cdot A/10^5 \cdot Z \): then \( A \) was bound to be below \( 2.147 \cdots \), and this limit was never far.

In our case, we can simply work with \( 10^8 \cdot A \) and \( 10^4 \cdot Z \), because our reason to work with higher powers has gone: we needed the integer \( y \approx 10^5 \cdot Z \) to be at least \( 10^4 \), and now, the definition \( y \approx 10^4 \cdot Z \) suffices.

Let us thus define \( y = [10^4 \cdot Z] + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4] \), and

\[
Q_1 = \left\lfloor \frac{10^8 \cdot A}{y} - \frac{1}{2} \right\rfloor.
\]

(The \( 1/2 \) comes from how \( \varepsilon \)-TeX rounds.) As for division, it is easy to see that \( Q_1 \leq 10^4 A/Z \), i.e., \( Q_1 \) is an underestimate.

Exactly as we did for division, we set \( B = 10^4 A - Q_1 Z \). Then

\[
10^4 B \leq A_1 A_2 A_3 A_4 - \left( \frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
\leq A_1 A_2 \left( 1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
\]
In the same way, and using \( 1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4 \), and convexity, we get
\[
\begin{align*}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &= 10^8 \frac{A}{y} + 1.6 y \leq 4.7 \cdot 10^4 \\
10^4 C &= 10^8 \frac{B}{y} + 1.6 y \leq 5.8 \cdot 10^4 \\
10^4 D &= 10^8 \frac{C}{y} + 1.6 y \leq 6.3 \cdot 10^4 \\
10^4 E &= 10^8 \frac{D}{y} + 1.6 y \leq 6.5 \cdot 10^4 \\
10^4 F &= 10^8 \frac{E}{y} + 1.6 y \leq 6.6 \cdot 10^4
\end{align*}
\]

Note that we compute more steps than for division: since \( t \) is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don’t need an exact rounding for transcendental functions, but just a faithful rounding).

The number is \( x \). Compute \( y \) by adding 1 to the five first digits.
We now have essentially

\[ \_\_fp\_ln\_div\_after: NW \{ \text{fixed \, tl} \} \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_1 \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_2 \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_3 \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_4 \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_5 \]
\[ \_\_fp\_div\_significand\_pack: NNN \ 10^6 + Q_6 ; \]
\[ \langle \text{exponent} \rangle ; \langle \text{continuation} \rangle \]

where \( \langle \text{fixed \, tl} \rangle \) holds the logarithm of a number in \([1, 10]\), and \( \langle \text{exponent} \rangle \) is the exponent. Also, the expansion is done backwards. Then \( \_\_fp\_div\_significand\_pack: NNN \) puts things in the correct order to add the \( Q_i \) together and put semicolons between each piece. Once those have been expanded, we get

\[ \_\_fp\_ln\_div\_after: NW \{ \text{fixed-tl} \} \ (1d) ; \ (4d) ; \ (4d) ; \]
\[ (4d) ; (4d) ; (4d) ; (4d) ; \langle \text{exponent} \rangle ; \langle \text{continuation} \rangle \]

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division \( 2/(x + 1) \), which is between roughly 0.8 and 1.2. We then compute \( 1 - 2/(x + 1) \), after testing whether \( 2/(x + 1) \) is greater than or smaller than 1.

\[ \_\_fp\_ln\_t\_small: NW \#1#2; \]
\[ \{ \]
\[ \text{if \ meaning: w 0 #2} \]
\[ \exp\_after:wN \_\_fp\_ln\_t\_small: NW \]
\[ \text{else:} \]
\[ \exp\_after:wN \_\_fp\_ln\_t\_large: NWW \]
\[ \exp\_after:wN = - \]
\[ \text{fi:} \]
\[ \#1 \]
\[ \} \]
\[ \_\_fp\_ln\_t\_small: NW \#1 \#2; \#3; \#4; \#5; \#6; \#7; \]
\[ \} \]
\[ \exp\_after:wN \_\_fp\_ln\_t\_large: NWW \]
\[ \exp\_after:wN + \times \langle \text{sign} \rangle \]
\[ \exp\_after:wN \#1 \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 9999 - #2 \ exp\_after:wN ;} \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 9999 - #3 \ exp\_after:wN ;} \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 9999 - #4 \ exp\_after:wN ;} \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 9999 - #5 \ exp\_after:wN ;} \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 9999 - #6 \ exp\_after:wN ;} \]
\[ \text{int\_value:w \_\_fp\_int\_eval:w 1 0000 - #7 ;} \]
\[ \} \]
\[ \_\_fp\_ln\_t\_large: NWW \langle \text{sign} \rangle \{ \text{fixed \, tl} \} \]
\[ \langle t_1 \rangle ; \langle t_2 \rangle ; \langle t_3 \rangle ; \langle t_4 \rangle ; \langle t_5 \rangle ; \langle t_6 \rangle ; \]
\[ \langle \text{exponent} \rangle ; \langle \text{continuation} \rangle \]

Compute the square \( t^2 \), and keep \( t \) at the end with its sign. We know that \( t < 0.1765 \), so every piece has at most 4 digits. However, since we were not careful in \( \_\_fp\_ln\_\_\_t\_small: w \), they can have less than 4 digits.
\__fp_ln_Taylor:wwNw  Denoting $T = t^2$, we get

\__fp_ln_Taylor:wwNw
{(T_1)} {(T_3)} {(T_5)} {(T_7)} {(T_9)} ; ;
{((2t)_1)} {((2t)_3)} {((2t)_5)} {((2t)_7)} {((2t)_9)} ; ;
{ \__fp_ln_c:NwW (sign) }  \{ fixed \} \{ exponent \}; \{ continuation \}

And we want to compute

$$\ln \left( \frac{1 + t}{1 - t} \right) = 2t \left( 1 + T \left( \frac{1}{3} + T \left( \frac{1}{5} + T \left( \frac{1}{7} + \cdots \right) \right) \right) \right)$$

The process looks as follows
This uses the routine for dividing a number by a small integer ($< 10^4$).

We are now reduced to finding $\ln(c)$ and ($exponent$) $\ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $b \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $-10^6$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $10^4$, since the final result is at least $\ln(10^7) \approx 0.35$. 

(End definition for \__fp\_ln\_Taylor:wwNW.)
\_fp\_ln\_exponent:wn
{(s_1)}\{(s_2)}\{(s_3)}\{(s_4)}\{(s_5)}\{(s_6)}
{(exponent)}

Compute \((exponent)\) times \(\ln(10)\). Apart from the cases where \(\langle exponent \rangle\) is 0 or 1, the result is necessarily at least \(\ln(10) \approx 2.3\) in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order \(10^4\). Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

\begin{verbatim}
\cs_new:Npn \_fp\_ln\_exponent:wn #1; #2
\{\if_case:w #2 \exp_stop_f:
  0 \_fp\_case\_return:nw \{ \_fp\_fixed\_to\_float\_o:Nw #1 2 \}
  \or:
  \exp_after:wN \_fp\_ln\_exponent\_one:ww \int_value:w \else:
  \if_int_compare:w #2 > 0 \exp_stop_f:
  \exp_after:wN \_fp\_ln\_exponent\_small:NNww
  \exp_after:wN 0 \exp_after:wN \_fp\_fixed\_sub:wwn \int_value:w \else:
  \exp_after:wN \_fp\_ln\_exponent\_small:NNww
  \exp_after:wN 2 \exp_after:wN \_fp\_fixed\_add:wwn \int_value:w - \fi:
\fi:
#2; #1;
\}
\end{verbatim}

Now we painfully write all the cases.\footnote{Bruno: do rounding.} No overflow nor underflow can happen, except when computing \(\ln(1)\).

\begin{verbatim}
\cs_new:Npn \_fp\_ln\_exponent\_one:ww #1; #2;
\{ 0 \exp_after:wN \_fp\_fixed\_sub:wwn \c\_fp\_ln\_x\_fixed_tl #1;
\_fp\_fixed\_to\_float\_o:wN #1 \}
\end{verbatim}

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

\begin{verbatim}
\cs_new:Npn \_fp\_ln\_exponent\_small:NNww #1#2#3; #4#5#6#7#8#9;
\{ \exp_after:wN \_fp\_fixed\_mul:wwn \c\_fp\_ln\_x\_fixed_tl
  \#3\{0000\}{0000}\{0000\}{0000}\{0000\};
\_fp\_fixed\_to\_float\_o:wN #1 \}
\end{verbatim}
33.2 Exponential

33.2.1 Sign, exponent, and special numbers

```latex
\cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
\begin{verbatim}
\if_case:w #2 \exp_stop_f:
  \__fp_case_return_o:Nw \c_one_fp
\or:
  \exp_after:wN \__fp_exp_normal_o:w
\or:
  \if_meaning:w 0 #3
    \exp_after:wN \__fp_case_return_o:Nw \c_inf_fp
  \else:
    \exp_after:wN \__fp_case_return_o:Nw \c_zero_fp
  \fi:
\or:
  \__fp_case_return_same_o:w
\fi:
\s__fp \__fp_chk:w #2#3#4; }
\end{verbatim}
```

(End definition for \__fp_exp_o:w.)

```latex
\cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
\begin{verbatim}
\if_meaning:w 0 #1
  \__fp_exp_pos_o:NNwnw + \__fp_fixed_to_float_o:wN
\else:
  \__fp_exp_pos_o:NNwnw - \__fp_fixed_inv_to_float_o:wN
\fi:
\end{verbatim}
```

(End definition for \__fp_exp_normal_o:w)

```latex
\cs_new:Npn \__fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
\begin{verbatim}
\if_int_compare:w #4 > \c__fp_max_exp_exponent_int
  \token_if_eq_charcode:NNTF + #1
    \{ \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp \}
    \{ \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp \}
  \exp:w
\else:
  \exp_after:wN \__fp_sanitize:Nw
  \exp_after:wN 0
  \int_value:w #1 \__fp_int_eval:w
  \if_int_compare:w #4 < 0 \exp_stop_f:
    \use_i:nn
  \else:
    \use_ii:nn
\end{verbatim}
```

873
This function is called for numbers in the range \([10^{-9}, 10^{-1})\). We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

\begin{verbatim}
\cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
  \__fp_pack_twice_four:wNNNNNNNN
  \__fp_pack_twice_four:wNNNNNNNN
  \__fp_pack_twice_four:wNNNNNNNN
  \__fp_exp_Taylor_ii:ww ; #2#3#4 0000 0000 ;
\cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2; #3; \s__fp_stop
  \__fp_fixed_div_int:wwN #3 ; #1 ;
  \__fp_fixed_add_one:wN
  \__fp_fixed_mul:wwn #2 ;
\end{verbatim}
The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \ldots, 9$ and $i = -1, \ldots, 4$. Each value is expressed as $\approx 10^p \times 10^{m_1} m_2 m_3$ with three 8-digit blocks $m_1$, $m_2$, $m_3$ and an integer exponent $p$ (one more than the scientific exponent), and these are stored in the integer array as four items: $p$, $10^8 + m_1$, $10^8 + m_2$, $10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```
\intarray_const_from_clist:Nn \c__fp_exp_intarray
{
  \begin{tabular}{l}
    1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
    1 , 1 1221 4027 , 1 1807 5647 , 1 6248 1171 ,
    1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
    1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
    1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
    1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
    1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
    1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
    1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
    1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
    1 , 1 1738 9060 , 1 9893 0650 , 1 2272 3043 ,
    2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
    2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
    3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
    3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
    4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
    4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
    4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
    4 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
    4 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
    14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
    18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
    22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
    27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
    31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
    35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
    40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
    44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
    87 , 1 1403 5922 , 1 6812 5749 , 1 2581 7748 ,
    131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
    174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
    218 , 1 1043 5922 , 1 1785 2837 , 1 4107 3977 ,
    261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
    305 , 1 1014 3230 , 1 5473 5004 , 1 5094 5533 ,
    348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
    391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
    435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
  \end{tabular}
}
\__fp_exp_intarray_aux:w
\__fp_exp_intarray:w
\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wwn
\__fp_exp_large:NwN
\__fp_exp_intarray:w
The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros).
The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0,5]. Remove leading zeros from the integer part: putting \#4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up \exp((digit)\cdot10^{\langle exponent\rangle}) in a table, and multiplying that to the current total. The loop is done by \__fp_exp_large:NwN, whose \#1 is the \langle exponent\rangle, \#2 is the current mantissa, and \#3 is the \langle digit\rangle. At the end, \__fp_exp_large_after:wwn moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.
\cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
\exp_after:wN \exp_after:wN \exp_after:wN \__fp_exp_large:NwN
\exp_after:wN \exp_after:wN \exp_after:wN \c__fp_one_fixed_tl
\int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
\fi:
\else:
\exp_after:wN \__fp_exp_intarray:w
\int_value:w \__fp_int_eval:w 36 * \#1 + 4 * \#3 \exp_after:wN ;
\fi:
\else:
\exp_after:wN \__fp_exp_large_after:wwn
\int_value:w \__fp_int_eval:w \#1 - 1 \exp_after:wN \scan_stop:
\fi:

33.3 Power

Raising a number \( a \) to a power \( b \) leads to many distinct situations.

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
a^b & -\infty & (-\infty, -0) & -\text{integer} & \pm0 & +\text{integer} & (0, \infty) & +\infty \\
\hline
+\infty & +0 & +0 & +1 & +\infty & +\infty & NaN \\
(1, \infty) & +0 & +|a|^b & +1 & +|a|^b & +\infty & NaN \\
+1 & +1 & +1 & +1 & +1 & +1 & 1 \\
\pm0 & +\infty & +|a|^b & +1 & +|a|^b & 0 & NaN \\
0 & +\infty & +\infty & +1 & +0 & 0 & NaN \\
−0 & NaN & (−1)^{b\infty} & +1 & (−1)^{b0} & +0 & 0 & NaN \\
(−1, 0) & +\infty & NaN & (−1)^b |a|^b & +1 & (−1)^b |a|^b & NaN & 0 & NaN \\
−1 & +1 & NaN & (−1)^b & +1 & (−1)^b & NaN & 1 & NaN \\
(−\infty, −1) & +0 & NaN & (−1)^b |a|^b & +1 & (−1)^b |a|^b & NaN & +\infty & NaN \\
−\infty & +0 & +0 & (−1)^b0 & +1 & (−1)^b \infty & NaN & NaN & NaN \\
\hline
\end{array}
\]

We distinguished in this table the cases of finite (positive or negative) integer exponents, as \((-1)^b\) is defined in that case. One peculiarity of this operation is that NaN\(^0 = 1\)\(^{\text{NaN}} = 1\), because this relation is obeyed for any number, even ±\(\infty\).

We crammed most of the tests into a single function to save csnames. First treat the case \( b = 0 \): \( a^0 = 1 \) for any \( a \), even \( \text{nan} \). Then test the sign of \( a \).
If it is positive, and \( a \) is a normal number, call \( \_\text{fp\_pow\_normal\_o:ww} \) followed by the two \( \text{fp\ a\ and\ b} \). For \( a = +0 \) or \( +\infty \), call \( \_\text{fp\_pow\_zero\_or\_inf\_o:ww} \) instead, to return either \(+0\) or \(+\infty\) as appropriate.

If \( a \) is a \text{n}an, then skip to the next semicolon (which happens to be conveniently the end of \( b \)) and return \text{n}an.

Finally, if \( a \) is negative, compute \( a^b \) (\( \_\text{fp\_pow\_normal\_o:ww} \) which ignores the sign of its first operand), and keep an extra copy of \( a \) and \( b \) (the second brace group, containing \{ \( b\ a\} \), is inserted between \( a \) and \( b \)). Then do some tests to find the final sign of the result if it exists.

\[
\text{cs\_new:cpn \{ \_\text{fp\_l\_ow\_char:N \_\_o:ww} \}}
\]
\[
\_\_\_\text{fp\ \_\_fp\_\_chk:w \#1\#2\#3; \_\_\_\_fp\ \_\_fp\_\_chk:w \#4\#5\#6;}
\]
\[
\text{\{}\}
\text{\text{if\ meaning:w 0 \#4}}
\]
\[
\_\_\_\text{fp\_case\_return\_o:Nw \c\_one\_fp}
\]
\[
\text{\fi:}\]
\[
\text{\text{if\ case:w \#2 \exp\_stop\_f:}}
\]
\[
\text{\exp\_after:wN \use\_i:nn}
\]
\[
\text{\or:}\]
\[
\_\_\_\text{fp\_case\_return\_o:Nw \c\_nan\_fp}
\]
\[
\text{\else:}\]
\[
\text{\exp\_after:wN \_\_\_\text{fp\_pow\_neg:ww}}
\]
\[
\text{\exp:w \exp\_end\_continue\_f:w \exp\_after:wN \use\:nn}
\]
\[
\text{\fi:}\]
\[
\text{\{}\}
\text{\text{if\ meaning:w 1 \#1}}
\]
\[
\text{\exp\_after:wN \_\_\_\text{fp\_pow\_normal\_o:ww}}
\]
\[
\text{\else:}\]
\[
\text{\exp\_after:wN \_\_\_\text{fp\_pow\_zero\_or\_inf:ww}}
\]
\[
\text{\fi:}\]
\[
\_\_\_\text{fp\ \_\_fp\_\_chk:w \#1\#2\#3;}
\]
\[
\text{\{}\}
\text{\text{if\ meaning:w \#4}}
\]
\[
\_\_\_\text{fp\_case\_return\_same\_o:w}
\]
\[
\text{\fi:}\]
\[
\text{\text{if\ meaning:w \#4}}
\]
\[
\_\_\_\text{fp\_case\_return\_o:Nw \c\_zero\_fp}
\]
\[
\text{\fi:}\]

\_\_\_\text{fp\_pow\_zero\_or\_inf:ww}

Raising \(-0\) or \(-\infty\) to \text{n}an yields \text{n}an. For other powers, the result is \(+0\) if \( 0 \) is raised to a positive power or \( \infty \) to a negative power, and \( +\infty \) otherwise. Thus, if the type of \( a \) and the sign of \( b \) coincide, the result is \( 0 \), since those conveniently take the same possible values, \( 0 \) and \( 2 \). Otherwise, either \( a = \pm\infty \) and \( b > 0 \) and the result is \( +\infty \), or \( a = \pm0 \) with \( b < 0 \) and we have a division by zero unless \( b = -\infty \).
\_fp\_pow\_zero\_or\_inf:ww

We have in front of us $a$, and $b \neq 0$, we know that $a$ is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1, unless $a = -1$ and $b$ is nan. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of $b$:

0 Impossible, we already filtered $b = \pm 0$.

1 Call \_fp\_pow\_npos:ww.

2 Return $+\infty$ or $+0$ depending on the sign of $b$ and whether the exponent of $a$ is positive or not.

3 Return $b$.
\_\_fp case return \_\_o:ww
\fi:
\_\_fp \_\_fp_chk:w 1 #1 (#2) #3 ;
\_\_fp \_\_fp_chk:w #4 #5 }

(End definition for \_\_fp_pow_normal_o:ww.)

\_\_fp_pow_npos_o:Nww We now know that $a \neq \pm 1$ is a normal number, and $b$ is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^p)^y 10^p = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of $z$ up to the 16-th position. Since the exponential of $10^5$ is infinite, we only need at most 21 digits, hence the fixed point result of \_\_fp_pow_o:w is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If $z$ is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

\cs_new:Npn \_\_fp_pow_npos_o:Nww #1 \_\_fp_chk:w 1#2#3
{ \exp_after:wN \_\_fp_sanitize:Nw
\exp_after:wN 0
\int_value:w
\if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
\exp_after:wN \_\_fp_pow_npos_aux:NNnww
\exp_after:wN +
\exp_after:wN \_\_fp_fixed_to_float_o:wN
\else:
\exp_after:wN \_\_fp_pow_npos_aux:NNnww
\exp_after:wN -
\exp_after:wN \_\_fp_fixed_inv_to_float_o:wN
\fi:
{#3}}

(End definition for \_\_fp_pow_npos_o:Nww.)

\_\_fp_pow_npos_aux:NNnww The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of $x$, followed by $b$. Compute $-\ln(x)$.

\cs_new:Npn \_\_fp_pow_npos_aux:NNnww #1#2#3#4#5; \_\_fp_chk:w 1#6#7#8;
{ \exp_after:wN \_\_fp_int_eval:w
\_\_fp_ln_significand:NNNNnnnN #4#5
\_\_fp_pow_exponent:wnN {#3}
\_\_fp_fixed_mul:wwn #8 {0000}{0000} ;
\_\_fp_pow_B:wwN #7;
#1 #2 0 \_\_fp_fixed_to_float_o:wN
}

\cs_new:Npn \_\_fp_pow_exponent:wnN #1; #2
{ \if_int_compare:w #2 > 0 \exp_stop_f:
\exp_after:wN \_\_fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
\exp_after:wN +
\else:
\exp_after:wN \_\_fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
\exp_after:wN -
}

880
\cs_new:Npn \__fp_pow_exponent:Nnnnnnwn \#1; \#3#4#5#6#7#8; 
\{ %^^A todo: use that in ln. 
\exp_after:wN \__fp_fixed_mul_after:wnn 
\int_value:w \__fp_int_eval:w \c__fp_leading_shift_int 
\exp_after:wN \__fp_pack:NNNNNw 
\int_value:w \__fp_int_eval:w \c__fp_middle_shift_int 
\#1#2*23025 - \#1 \#3 
\exp_after:wN \__fp_pack:NNNNNw 
\int_value:w \__fp_int_eval:w \c__fp_middle_shift_int 
\#1#2*8509 - \#1 \#4 
\exp_after:wN \__fp_pack:NNNNNw 
\int_value:w \__fp_int_eval:w \c__fp_middle_shift_int 
\#1#2*2994 - \#1 \#5 
\exp_after:wN \__fp_pack:NNNNNw 
\int_value:w \__fp_int_eval:w \c__fp_middle_shift_int 
\#1#2*0456 - \#1 \#6 
\exp_after:wN \__fp_pack:NNNNNw 
\int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int 
\#1#2*8401 - \#1 \#7 
\#1( \#2*7991 - \#8 ) / 1 0000 ; ; 
\}

\cs_new:Npn \__fp_pow_B:wwN \#1#2#3#4#5#6; \#7; 
\{ 
\if_int_compare:w \#7 < 0 \exp_stop_f: 
\exp_after:wN \__fp_pow_C_neg:w \int_value:w - 
\else: 
\if_int_compare:w \#7 < 22 \exp_stop_f: 
\exp_after:wN \__fp_pow_C_pos:w \int_value:w 
\else: 
\exp_after:wN \__fp_pow_C_overflow:w \int_value:w 
\fi: 
\fi: 
\exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w 
\prg_replicate:nn {\#1} {0} 
\}

\cs_new:Npn \__fp_pow_C_pos:w \#1; \#2; \#3 
\{ 
\if_meaning:w 0 \#1 
\int_value:w \__fp_int_eval:w 10 0000 + \#1 \__fp_int_eval_end: 
\#2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^^A todo: how many 0? 
\}

\cs_new:Npn \__fp_pow_C_overflow:w \int_value:w 
\exp_after:wN \__fp_pow_C_overflow:w \int_value:w 
\#1; \#2; \#3 
\{ 
+ 2 * \c__fp_max_exponent_int 
\exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl 
\}

\cs_new:Npn \__fp_pow_C_overflow:w \int_value:w 
\exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w 
\exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w 
\prg_replicate:nn {\#1} {0} 
\}

\cs_new:Npn \__fp_pow_C_neg:w \int_value:w 
\exp_after:wN \__fp_pow_C_neg:w \int_value:w 
\#1; 
\}

\cs_new:Npn \__fp_pow_C_pos:w \int_value:w 
\exp_after:wN \__fp_pow_C_pos_loop:wn \int_value:w 
\#1; 
\}

\cs_new:Npn \__fp_pow_C_pos_loop:wn \int_value:w 
\exp_after:wN \__fp_pow_C_pos_loop:wn \int_value:w 
\#1; 
\}

\if_meaning:w 0 \#1 

881
This function is followed by three floating point numbers: $a^b$, $a \in [-\infty, -0]$, and $b$. If $b$ is an even integer (case $-1$), $a^b = a^{\bar{b}}$. If $b$ is an odd integer (case $0$), $a^b = -a^{\bar{b}}$, obtained by a call to $\__\_\_fp\_pow\_neg\_aux:$WNN. Otherwise, the sign is undefined. This is invalid, unless $a^b$ turns out to be $+0$ or $\text{nan}$, in which case we return that as $a^b$. In particular, since the underflow detection occurs before $\__\_\_fp\_pow\_neg:$WWW is called, $(-0.1)^{(12345.67)}$ gives $+0$ rather than complaining that the sign is not defined.

This function expects a floating point number, and determines its “parity”. It should be used after $\__\_\_fp\_pow\_neg\_aux:$WNN and $\__\_\_fp\_pow\_neg\_aux:$WNN.

882
even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and ±∞ are even
(because very large finite floating points are even), while \texttt{nan} is a non-integer. The sign
of normal numbers is irrelevant to parity. After \texttt{\__fp\_decimate:nNnnnn} the argument
#1 of \texttt{\__fp\_pow\_neg\_case\_aux:Nnnw} is a rounding digit, 0 if and only if the number
was an integer, and #3 is the 8 least significant digits of that integer.

\begin{verbatim}
c__fp\_prec\_int \cs_new:Npn \__fp\_pow\_neg\_case:w \s__fp \__fp\_chk:w #1#2#3; 
\kern\c__fp\_prec\_int \exp_stop_f:
\kern\c__fp\_prec\_int \exp_stop_f:
\end{verbatim}

33.4 Factorial

\texttt{\c__fp\_fact\_max\_arg\_int} The maximum integer whose factorial fits in the exponent range is 3248, as 3249! ∼ 10^{10000.8}

\begin{verbatim}
c__fp\_fact\_max\_arg\_int \int_const:Nn \c__fp\_fact\_max\_arg\_int \{ 3248 \}
\end{verbatim}

(End definition for \texttt{\__fp\_pow\_neg\_case:w}, \texttt{\__fp\_pow\_neg\_case\_aux:nNnnn}, and \texttt{\__fp\_pow\_neg\_case\_aux:Nnnw})

First detect ±0 and +∞ and \texttt{nan}. Then note that factorial of anything with a negative
sign (except −0) is undefined. Then call \texttt{\__fp\_small\_int:wTF} to get an integer as the
argument, and start a loop. This is not the most efficient way of computing the factorial,
but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```latex
\cs_new:Npn \__fp_fact_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
\begin{verbatim}
\if_case:w #2 \exp_stop_f:
    \__fp_case_return_o:Nw \c_one_fp
\or:
    \or:
    \exp_after:wN \__fp_case_return_same_o:w
\or:
    \__fp_case_use:nw \__fp_invalid_operation_o:fw \{ fact \}
\or:
    \__fp_fact_pos_o:w
\s__fp \__fp_chk:w #2 #3 #4 ;
\end{verbatim}
\end{verbatim}
\textit{\small (End definition for \texttt{\__fp_fact_o:w}.)}
```

\texttt{\__fp_fact_pos_o:w} and \texttt{\__fp_fact_int_o:w}

Then check the input is an integer, and call \texttt{\__fp_fact_int_o:n} with that int as an argument. If it’s too big the factorial overflows. Otherwise call \texttt{\__fp_sanitizer:Nw} with a positive sign marker 0 and an integer expression that will mop up any exponent in the calculation.

```latex
\cs_new:Npn \__fp_fact_pos_o:w #1;
\begin{verbatim}
\__fp_small_int:wTF #1;
\end{verbatim}
\textit{\small (End definition for \texttt{\__fp_fact_pos_o:w} and \texttt{\__fp_fact_int_o:w}.)}
```

\texttt{\__fp_fact_loop_o:w}

The loop receives an integer \#1 whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number \#2 in the form \langle exponent \rangle,\langle mantissa \rangle. The loop goes in steps of two because we compute \#1*\#1-1 as an integer expression (it must fit since \#1 is at most 3248), then multiply with the
result so far. We don’t need to fill in most of the mantissa with zeros because \_fp_ep_mul:wwwn first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than \(10^8\). This limit is chosen because the normalization step cannot deal with larger integers.

\[
\cs_new:Npn \_fp_fact_loop_o:w #1 . #2 \\
\{ \\
\quad \text{if_int_compare:w #1 < 12 \exp_stop_f:} \\
\quad \_fp_fact_small_o:w #1 \\
\quad \fi: \\
\quad \exp_after:wN \_fp_ep_mul:wwwn \\
\quad \exp_after:wN 4 \exp_after:wN , \\
\quad \exp_after:wN \{ \text{int_value:w} \_fp_int_eval:w \#1 * (#1 - 1) \} \\
\quad \{ \} \{ \} \{ \} \{ \} ; \\
\quad \#2 ; \\
\quad \{ \\
\quad \exp_after:wN \_fp_fact_loop_o:w \\
\quad \text{int_value:w} \_fp_int_eval:w \#1 - 2 . \\
\quad \}
\]

\[
\cs_new:Npn \_fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4 \\
\{ \\
\quad \text{if:} \\
\quad \exp_after:wN \_fp_ep_mul:wwwn \\
\quad \exp_after:wN 4 \exp_after:wN , \\
\quad \exp_after:wN \{ \\
\quad \text{int_value:w} \\
\quad \text{if_case:w} \#1 \exp_stop_f: \\
\quad \text{1 \ or: 1 \ or: 2 \ or: 6 \ or: 24 \ or: 120 \ or: 720 \ or: 5040 \ or: 40320 \ or: 362880 \ or: 3628800 \ or: 39916800} \\
\quad \text{fi:} \\
\quad \{ \} \{ \} \{ \} \{ \} \{ \} ; \\
\quad \#3 ; \\
\quad \_fp_ep_to_float_o:wwN 0 \\
\quad \}
\]

(End definition for \_fp_fact_loop_o:w)

\[
\langle/\text{initex} \langle/\text{package}\rangle
\]

34 \textit{l3fp-trig} Implementation

\[
\langle/\text{initex} \langle/\text{package}\rangle
\]

\texttt{@@=fp}

Unary functions.

\[
\langle/\text{initex} \langle/\text{package}\rangle
\]

\texttt{\_fp_parse_word_acos:N \_fp_parse_word_acosd:N \_fp_parse_word_acsc:N \_fp_parse_word_acscd:N \_fp_parse_word_asec:N \_fp_parse_word_asecd:N \_fp_parse_word_asin:N \_fp_parse_word_asind:N \_fp_parse_word_cos:N \_fp_parse_word_cosd:N \_fp_parse_word_cot:N \_fp_parse_word_cotd:N \_fp_parse_word_csc:N \_fp_parse_word_cscd:N \_fp_parse_word_sec:N \_fp_parse_word_secd:N \_fp_parse_word_sind:N \_fp_parse_word_sind:N}

\[\text{\texttt{\_fp_parse_word_sin:N}}\]

\[\text{\texttt{\_fp_parse_word_tan:N}}\]

\[\text{\texttt{\_fp_parse_word_tand:N}}\]

\texttt{\_fp_parse_word_acos:N \_fp_parse_word_acosd:N \_fp_parse_word_acsc:N \_fp_parse_word_acscd:N \_fp_parse_word_asec:N \_fp_parse_word_asecd:N \_fp_parse_word_asin:N \_fp_parse_word_asind:N \_fp_parse_word_cos:N \_fp_parse_word_cosd:N \_fp_parse_word_cot:N \_fp_parse_word_cotd:N \_fp_parse_word_csc:N \_fp_parse_word_cscd:N \_fp_parse_word_sec:N \_fp_parse_word_secd:N \_fp_parse_word_sind:N \_fp_parse_word_sind:N}

\[\text{\texttt{\_fp_parse_word_sin:N}}\]

\[\text{\texttt{\_fp_parse_word_tan:N}}\]

\[\text{\texttt{\_fp_parse_word_tand:N}}\]
\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N
Those functions may receive a variable number of arguments.

- Filter out special cases ($\pm 0$, $\pm \infty$, and NaN).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90]$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\left\lfloor \frac{x}{\pi/4} \right\rfloor \mod 8$ (in degrees, the same formula with $\pi/4 \to 45$), the sign, and the function to compute.

34.1.1 Filtering special cases

\__fp_sin_o:w
This function, and its analogs for cos, csc, sec, tan, and cot instead of sin, are followed either by \use_i:nn and a float in radians or by \use_ii:nn and a float in degrees. The sine of $\pm 0$ or NaN is the same float. The sine of $\pm \infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the trig function to perform argument reduction and if necessary convert the reduced argument to radians.
Then, \_\_fp_sin_series_o:NNwwww is called to compute the Taylor series: this function receives a sign \#3, an initial octant of 0, and the function \_\_fp_ep_to_float_o:wN which converts the result of the series to a floating point directly rather than taking its inverse, since \( \sin(x) = \#3 \sin|\gamma| \).

The cosine of \( \pm 0 \) is 1. The cosine of \( \pm \infty \) raises an invalid operation exception. The cosine of NaN is itself. Otherwise, the trig function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign \( 0 \) regardless of the sign of \( x \), and with an initial octant of 2, because \( \cos(x) = + \sin(\pi/2 + |\gamma|) \).

The cosecant of \( \pm 0 \) is \( \pm \infty \) with the same sign, with a division by zero exception (see \_\_fp_cot_zero_o:Nfw defined below), which requires the function name. The cosecant of \( \pm \infty \) raises an invalid operation exception. The cosecant of NaN is itself. Otherwise, the trig function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign \#3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because \( \csc(x) = \#3(\sin|\gamma|)^{-1} \).
\_\_fp\_sec\_o:w

The secant of ±0 is 1. The secant of ±∞ raises an invalid operation exception. The secant of NaN is itself. Otherwise, the trig function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because sec(x) = +1/sin(π/2 + |x|).

\cs_new:Npn \__fp\_sec\_o:w #1 \s__fp \__fp\_chk\:w #2#3; @
{
  \if_case:w #2 \exp_stop_f:
    \_\_fp\_case\_return\_same\_o:w
  \or:
    \_\_fp\_case\_use\:nw
      \_\_fp\_trig:NNNNNwn #1 \_\_fp\_sin\_series\_o:NNwwww
      \_\_fp\_ep\_inv\_to\_float\_o:wwN #3 0
  \or:
    \_\_fp\_case\_use\:nw
      \_\_fp\_invalid\_operation\_o:fw \_\_fp\_case\_return\_same\_o:w
  \else:
    \_\_fp\_case\_return\_same\_o:w
  \fi:
  \s__fp \__fp\_chk\:w #2 #3 #4;
}

(End definition for \_\_fp\_sec\_o:w.)

\_\_fp\_tan\_o:w

The tangent of ±0 or NaN is the same floating point number. The tangent of ±∞ raises an invalid operation exception. Once more, the trig function does the argument reduction step and conversion to radians before calling \_\_fp\_tan\_series\_o:NNwwww, with a sign #3 and an initial octant of 1 (this shift is somewhat arbitrary). See \_\_fp\_\_cot\_o:w for an explanation of the 0 argument.

\cs_new:Npn \__fp\_tan\_o:w #1 \s__fp \__fp\_chk\:w #2#3#4; @
{
  \if_case:w #2 \exp_stop_f:
    \_\_fp\_case\_return\_same\_o:w
  \or:
    \_\_fp\_case\_use\:nw
      \_\_fp\_trig:NNNNNwn #1 \_\_fp\_tan\_series\_o:NNwwww
      \_\_fp\_tan\_series\_o:NNwwww 0 2
  \or:
    \_\_fp\_case\_use\:nw
      \_\_fp\_invalid\_operation\_o:fw \_\_fp\_case\_return\_same\_o:w
  \else:
    \_\_fp\_case\_return\_same\_o:w
  \fi:
  \s__fp \__fp\_chk\:w #2 #3 #4;
}

(End definition for \_\_fp\_tan\_o:w.)

888
The cotangent of \( \pm 0 \) is \( \pm \infty \) with the same sign, with a division by zero exception (see \texttt{\__fp_cot_zero_o:Nfw}). The cotangent of \( \pm \infty \) raises an invalid operation exception. The cotangent of NaN is itself. We use \( \cot x = -\tan(\pi/2 + x) \), and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding \texttt{\__fp_tan_series_o:NNwwww} two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

The first argument is \texttt{\use_i:nn} if the operand is in radians and \texttt{\use_ii:nn} if it is in degrees. Arguments \#2 to \#5 control what trigonometric function we compute, and \#6 to \#8 are pieces of a normal floating point number. Call the \texttt{\_series} function \#2, with arguments \#3, either a conversion function (\texttt{\_fp_ep_to_float_o:wN} or \texttt{\_fp_ep_inv_to_float_o:wN}) or a sign 0 or 2 when computing tangent or cotangent; \#4, a sign 0 or 2; the octant, computed in an integer expression starting with \#5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four \texttt{\exp_after:wN} are expanded, the
expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining \exp_after:wN hits \#1, which picks the \texttt{trig} or \texttt{trigd} function in whichever branch of the conditional was taken. The final \exp_after:wN closes the conditional. At the end of the day, a number is large if it is $\geq 1$ in radians or $\geq 10$ in degrees, and small otherwise. All four \texttt{trig/trigd} auxiliaries receive the operand as an extended-precision number.

\begin{verbatim}
\cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
\{
\exp_after:wN #2
\exp_after:wN #3
\exp_after:wN #4
\int_value:w \__fp_int_eval:w #5
\exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
\if_int_compare:w #7 > #1 0 1 \exp_stop_f:
#1 \__fp_trig_large:ww \__fp_trigd_large:ww
\else:
 #1 \__fp_trig_small:ww \__fp_trigd_small:ww
\fi:
\#7,#8{0000}{0000};
\}
\end{verbatim}  
(End definition for $\__fp_trig:NNNNNwm$)

\subsection{Small arguments}

$\__fp_trig_small:ww$ This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

\begin{verbatim}
\cs_new:Npn \__fp_trig_small:ww #1,#2;
{ \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
\end{verbatim}  
(End definition for $\__fp_trig_small:ww$)

$\__fp_trigd_small:ww$ Convert the extended-precision number to radians, then call $\__fp_trig_small:ww$ to massage it in the form appropriate for the $\_\text{series}$ auxiliary.

\begin{verbatim}
\cs_new:Npn \__fp_trigd_small:ww #1,#2;
{ \__fp_ep_mul_raw:wwwwN -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
 \__fp_trig_small:ww }
\end{verbatim}  
(End definition for $\__fp_trigd_small:ww$)

\subsection{Argument reduction in degrees}

$\__fp_trigd_large:ww$ \__fp_trig_large_aux:nnn\wd\c@inputlineno \__fp_trig_large_auxii:ww
\__fp_trig_large_auxiii:www

Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent $\#1$ is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times (\text{blocks}_1) \cdots (\text{blocks}_3) \langle \text{block}_4 \rangle \cdots \langle \text{block}_7 \rangle$, or is equal to
it modulo 360 if the exponent \#1 is very large. The first auxiliary finds \langle block1 \rangle + \langle block2 \rangle (mod 9), a single digit, and prepends it to the 4 digits of \langle block3 \rangle. It also unpacks \langle block4 \rangle and grabs the 4 digits of \langle block5 \rangle. The second auxiliary grabs the \langle block3 \rangle plus any contribution from the first two blocks as \#1, the first digit of \langle block4 \rangle (just after the decimal point in hundreds of degrees) as \#2, and the three other digits as \#3. It finds the quotient and remainder of \#1\#2 modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before \#3 to form a new \langle block3 \rangle. The resulting fixed point number is \( x \in [0,0.9] \). If \( x \geq 0.45 \), we add 1 to the octant and feed 0.9 - \( x \) with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to \_\_fp_trigd_small:ww. Otherwise, we feed it \( x \) with an exponent of 2. The third auxiliary also discards digits which were not packed into the various \langle blocks \rangle. Since the original exponent \#1 is at least 2, those are all 0 and no precision is lost (\#6 and \#7 are four 0 each).

\begin{verbatim}
\cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
\exp_after:wN \__fp_pack_eight:wNNNNNNNN
\exp_after:wN \__fp_pack_eight:wNNNNNNNN
\exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
\exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
\exp_after:wN ;
\prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
#2#3#4#5#6#7 0000 0000 0000 !!
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
\exp_after:wN \__fp_trigd_large_auxii:wNw
\int_value:w \__fp_int_eval:w #1 + #2
- (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end: 
#3;
#4; #5{#6#7#8#9};
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
+ (#1#2 - 4) / 9 * 2 \__fp_trigd_large_auxiii:www
\int_value:w \__fp_int_eval:w \#1 + \#2
- (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: \#3 ;
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
\if_int_compare:w #1 < 4500 \exp_stop_f:
\exp_after:wN \__fp_use_i_until_s:nw
\exp_after:wN \__fp_fixed_continue:wn
\else:
+ 1
\fi:
\__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000};
{ \__fp_trigd_small:ww 2, }
\end{verbatim}

891
34.1.5 Argument reduction in radians
Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of $2\pi$, then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate.

When the argument is very large, say, $10^{100}$, an equally large multiple of $2\pi$ must be subtracted, hence we must work with a very good approximation of $2\pi$ in order to get a sensible remainder modulo $2\pi$.

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part.

From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or $-8$ (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start most with 24 zeros. The last 5 digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $6\cdot\text{ulp}$ in all cases.

This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is $10^8$ plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 (41 groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.
The exponent \#1 is between 1 and 10000. We wish to look up decimals \(10^{\#1-16}/(2\pi)\) starting from the digit \#1 + 1. Since they are stored in batches of 8, compute \([\#1/8]\) and fetch blocks of 8 digits starting there. The numbering of items in \__fp_trig_intarray\ starts at 1, so the block \([\#1/8]+1\) contains the digit we want, at one of the eight positions. Each call to \int_value:w \__kernel_intarray_item:Nn\ expands the next, until being stopped by \__fp_trig_large_auxiii:w\ using \exp_stop_f:; Once all these blocks are unpacked, the \exp_stop_f: and 0 to 7 digits are removed by \use_...n...n. Finally, \__fp_trig_large_auxii:w\ packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the auxv auxiliary is called.

\begin{verbatim}
\cs_new:Npn \__fp_trig_large:ww \#1, \#2\#3\#4\#5\#6;
\{ \exp_after:wN \__fp_trig_large_auxi:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN , \__fp_int_eval:w \#1 , ; \#2\#3\#4\#5\#6; \}
\cs_new:Npn \__fp_trig_large_auxii:w \#1, \#2, \{ \exp_after:wN \exp_after:wN \exp_after:wN \__fp_trig_large_auxiii:w \}
\cs_new:Npn \__fp_trig_large_auxiii:w \}
\end{verbatim}

(End definition for \c__fp_trig_intarray.)
First come the first 64 digits of the fractional part of $10^{#1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the auxvi auxiliary receives the significand as #2#3#4#5 and 16 digits of the fractional part as #6#7#8#9, and computes one step of the usual ladder of pack functions we use for multiplication (see e.g., \_fp_fixed_mul:wwn), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last middle shift by the appropriate trailing shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute.
any more step in the ladder. The last semicolon closes the ladder, and we return control to the auxvii auxiliary.

\[
\text{\texttt{\textbackslash cs\_new\_Npn \_\_fp\_trig\_large\_auxv:ww \#1; \#2; \#3;}}
\]

\[
\text{\texttt{\exp\_after:wN \_\_fp\_use\_i\_until\_s:nw}}
\]

\[
\text{\texttt{\exp\_after:wN \_\_fp\_trig\_large\_auxvii:w}}
\]

\[
\text{\texttt{\int\_value:w \_\_fp\_int\_eval:w \c\_fp\_leading\_shift\_int}}
\]

\[
\text{\texttt{\prg\_replicate:nn \{ 13 \}}}
\]

\[
\text{\texttt{\{ \_\_fp\_trig\_large\_auxvi:wnnnnnnn \}}}
\]

\[
\text{\texttt{+ \c\_fp\_trailing\_shift\_int - \c\_fp\_middle\_shift\_int}}
\]

\[
\text{\texttt{\_\_fp\_use\_i\_until\_s:nw}}
\]

\[
\text{\texttt{; \#3 \#1 ; ;}}
\]

\[
\text{\texttt{\cs\_new\_Npn \_\_fp\_trig\_large\_auxvi:wnnnnnnn \#1; \#2\#3\#4\#5\#6\#7\#8\#9}}
\]

\[
\text{\texttt{\{}}
\]

\[
\text{\texttt{\exp\_after:wN \_\_fp\_trig\_large\_pack:NNNNNw}}
\]

\[
\text{\texttt{\int\_value:w \_\_fp\_int\_eval:w \c\_fp\_middle\_shift\_int}}
\]

\[
\text{\texttt{+ \#2*\#9 + \#3*\#8 + \#4*\#7 + \#5*\#6}}
\]

\[
\text{\texttt{\#1; \{\#2\}\{\#3\}\{\#4\}\{\#5\} \{\#7\}\{\#8\}\{\#9\}}}
\]

\[
\text{\texttt{}}
\]

\[
\text{\texttt{\cs\_new\_Npn \_\_fp\_trig\_large\_pack:NNNNNw \#1\#2\#3\#4\#5\#6;}}
\]

\[
\text{\texttt{\{ + \#1\#2\#3\#4\#5 ; \#6 \}}}
\]

(End definition for \_\_fp\_trig\_large\_auxv:ww, \_\_fp\_trig\_large\_auxvi:wnnnnnnn, and \_\_fp\_trig\_large\_pack:NNNNNw.)

\[
\text{\texttt{\_\_fp\_trig\_large\_auxvii:w}}
\]

\[
\text{\texttt{\_\_fp\_trig\_large\_auxviii:w}}
\]

\[
\text{\texttt{\_\_fp\_trig\_large\_auxix:Nw}}
\]

\[
\text{\texttt{\_\_fp\_trig\_large\_auxx:wnNNNN}}
\]

\[
\text{\texttt{\_\_fp\_trig\_large\_auxxi:w}}
\]

The auxvii auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of \#1\#2\#3/125, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last middle shift is converted to a trailing shift. Any integer part (including negative values which come up when the octant is odd) is discarded by \_\_fp\_use\_i\_until\_s:nw. The resulting fractional part should then be converted to radians by multiplying by 2\pi/8, but first, build an extended precision number by abusing \_\_fp\_ep\_to\_ep\_loop:N with the appropriate trailing markers. Finally, \_\_fp\_trig\_small:ww sets up the argument for the functions which compute the Taylor series.

\[
\text{\texttt{\cs\_new\_Npn \_\_fp\_trig\_large\_auxvii:w \#1\#2\#3}}
\]

\[
\text{\texttt{\{}}
\]

\[
\text{\texttt{\exp\_after:wN \_\_fp\_trig\_large\_auxviii:w}}
\]

\[
\text{\texttt{\int\_value:w \_\_fp\_int\_eval:w \c\_fp\_middle\_shift\_int}}
\]

\[
\text{\texttt{+ \#2*\#9 + \#3*\#8 + \#4*\#7 + \#5*\#6}}
\]

\[
\text{\texttt{\#1; \{\#2\}\{\#3\}\{\#4\}\{\#5\} \{\#7\}\{\#8\}\{\#9\}}}
\]

\[
\text{\texttt{}}
\]

\[
\text{\texttt{\cs\_new\_Npn \_\_fp\_trig\_large\_pack:NNNNNw \#1\#2\#3\#4\#5\#6;}}
\]

\[
\text{\texttt{\{ + \#1\#2\#3\#4\#5 ; \#6 \}}}
\]
\cs_new:Npn \_fp_trig_large_auxix:Nw
\exp_after:wN \_fp_use_i_until_s:nw
\exp_after:wN \_fp_trig_large_auxxi:w
\int_value:w \_fp_int_eval:w \c__fp_leading_shift_int
\prg_replicate:nn { 13 }
\exp_after:wN \_fp_trig_large_auxx:wNNNNN
+ \c__fp_trailing_shift_int - \c__fp_middle_shift_int
;
\cs_new:Npn \_fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
\exp_after:wN \_fp_trig_large_pack:NNNNNw
\int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
#2 8 * #3#4#5#6
#1; #2
\cs_new:Npn \_fp_trig_large_auxxi:w #1;
\exp_after:wN \_fp_ep_mul_raw:wwwwN
\int_value:w 0 \_fp_ep_to_ep_loop:N #1 ; ; !
0,{7853}{9816}{3397}{4483}{0961}{5661};
\_fp_trig_small:ww
\_fp_trig_large_auxx:wnNNNNN #1; #2 #3#4#5#6
\cs_new:Npn \_fp_trig_large_auxxx:wwNN
\_fp_trig_large_pack:NNNNNw
\int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
#2 8 * #3#4#5#6
#1; #2

(End definition for \_fp_trig_large_auxvii:w and others.)

34.1.6 Computing the power series

\_fp_sin_series_o:NNwwww
\_fp_sin_series_aux_o:NNnwww

Here we receive a conversion function \_fp_to_float_o:wwN or \_fp_inv_-to_float_o:wwN, a \langle sign \rangle (0 or 2), a (non-negative) \langle octant \rangle delimited by a dot, a \langle fixed point \rangle number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function \#1;
- the final sign, which depends on the octant \#3 and the sign \#2;
- the octant \#3, which controls the series we use;
- the square \#4 * \#4 of the argument as a fixed point number, computed with \_fp_fixed_mul:wwn;
- the number itself as an extended-precision number.

If the octant is in \{1, 2, 5, 6, \ldots\}, we are near an extremum of the function and we use the series
\[
\cos(x) = 1 - x^2 \left( \frac{1}{2!} - x^2 \left( \frac{1}{4!} - x^2 \left( \cdots \right) \right) \right).
\]
Otherwise, the series
\[
\sin(x) = x \left( 1 - x^2 \left( \frac{1}{3!} - x^2 \left( \frac{1}{5!} - x^2 \left( \cdots \right) \right) \right) \right)
\]
is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and \_fp\_sanitize:Nw checks for overflow and underflow.

\cs_new:Npn \__fp\_sin\_series\_o:NNwwww \#1\#2\#3. \#4; \{ \__fp\_fixed\_mul:wwn \#4; \#4; \exp\_after:wN \__fp\_sin\_series\_aux\_o:NNnww \exp\_after:wN \#1 \int\_value:w \if_int_odd:w \__fp\_int\_eval:w \#3 + 2 \) / 4 \__fp\_int\_eval\_end: \#2 \else: \if_meaning:w \#2 0 \else: 0 \fi: \fi: \{ \#3\} \} \cs_new:Npn \__fp\_sin\_series\_aux\_o:NNnwww \#1\#2\#3 \#4; \#5\#6; \{ \if_int_odd:w \__fp\_int\_eval:w \#3 / 2 \__fp\_int\_eval\_end: \exp\_after:wN \use\_i:nn \else: \exp\_after:wN \use\_ii:nn \fi: \{ \% 1/18! \__fp\_fixed\_mul\_sub\_back:wwn \{0000\}0000\{0000\}0000\{5619\}2070; \#4; \{0000\}0000\{0000\}0477\{9477\}3324; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0000\{0000\}4707\{4559\}7730; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0000\{2087\}6756\{9878\}6810; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0275\{5573\}1922\{3985\}8907; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}2480\{1587\}3015\{8730\}1587; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0013\}8888\{8888\}8888\{8889\}; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0016\}6666\{6666\}6666\{6666\}6666; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0000\{0000\}0000\{0000\}; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{1000\}0000\{0000\}0000\{0000\}0000; \{ \__fp\_fixed\_continue:wn 0, \} \} \% 1/17! \__fp\_fixed\_mul\_sub\_back:wwn \{0000\}0000\{0000\}0028\{1145\}7254; \#4; \{0000\}0000\{0000\}7647\{1637\}3182; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0000\{0160\}5904\{3836\}8216; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0002\{5052\}1083\{8544\}1719; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0000\}0275\{5731\}9223\{9858\}9065; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0001\}9841\{2698\}4126\{9841\}2698; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{0083\}3333\{3333\}3333\{3333\}3333; \__fp\_fixed\_mul\_sub\_back:wwn \#4; \{1666\}6666\{6666\}6666\{6666\}6666; \{ \__fp\_ep\_mul:wwwn 0, \} \#5, \#6; \} \\exp\_after:wN \__fp\_sanitize:Nw \exp\_after:wN \#2 \int\_value:w \__fp\_int\_eval:w \#1
Contrary to \texttt{\_fp_sin_series_o:NWWww} which received a conversion auxiliary as \#1, here, \#1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant \#3 starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $[(\#3 + 1)/2]$ is odd. We also have to take into account the original sign of $x$ to get the sign of the final result; it is straightforward to check that the first \texttt{\_fp_tan_series_o:NWWww} expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$
\tan(x) \approx \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5))))}
$$

The ratio is computed by \texttt{\_fp_ep_div:wwwwn}, then converted to a floating point number. For octants \#3 (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this \texttt{\_fp_int_odd:w} test relies on the fact that the octant is at least 1.

\begin{verbatim}
cs_new:Npn \_fp_tan_series_o:NNwwww #1#2#3. #4; 
\__fp_fixed_mul:wwn #4; #4; 
\exp_after:wN \__fp_tan_series_aux_o:Nwww \int_value:w \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end: \exp_after:wN \reverse_if:N \if_meaning:w #1#2 2 \else: 0 \fi: \if#3 \{\#3\} \} 
\cs_new:Npn \_fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5; 
\__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706}; #3; {0000}{0159}{6080}{0274}{5257}{6472}; \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481}; \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147}; \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982}; \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000}; \{ \__fp_ep_mul:wwwwn 0, } #4,#5; 
\__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706}; #3; {0000}{2343}{7175}{1399}{6151}{7670}; \__fp_fixed_mul_sub_back:wwwn #3;{0019}{2638}{4588}{9232}{8861}{3691}; \__fp_fixed_mul_sub_back:wwwn #3;{0536}{6357}{0691}{4344}{6852}{4252}; \__fp_fixed_mul_sub_back:wwwn #3;{5263}{1578}{9473}{6842}{1052}{6315}; \__fp_fixed_mul_sub_back:wwwn#3;{10000}{0000}{0000}{0000}{0000}{0000};
\end{verbatim}
34.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arccosecant, and arcsecant) are based on a function often denoted \( \text{atan2} \). This function is accessed directly by feeding two arguments to arctangent, and is defined by

\[
\text{atan}(y, x) = \text{atan}(y/x)
\]

for generic \( y \) and \( x \). Its advantages over the conventional arctangent is that it takes values in \([-\pi, \pi]\) rather than \([-\pi/2, \pi/2]\], and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of \( \text{atan} \) as

\[
\begin{align*}
\text{acos } x & = \text{atan} \left( \sqrt{1 - x^2}, x \right) \\
\text{asin } x & = \text{atan} \left( x, \sqrt{1 - x^2} \right) \\
\text{asec } x & = \text{atan} \left( \sqrt{x^2 - 1}, 1 \right) \\
\text{acsc } x & = \text{atan} \left( 1, \sqrt{x^2 - 1} \right) \\
\text{atan } x & = \text{atan} \left( x, 1 \right) \\
\text{acot } x & = \text{atan} \left( 1, x \right).
\end{align*}
\]

Rather than introducing a new function, \( \text{atan2} \), the arctangent function \( \text{atan} \) is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument \( y \) and the second \( x \), because \( \text{atan}(y, x) = \text{atan}(y/x) \) is the angular coordinate of the point \((x, y)\).

As for direct trigonometric functions, the first step in computing \( \text{atan}(y, x) \) is argument reduction. The sign of \( y \) gives that of the result. We distinguish eight regions where the point \((x, |y|)\) can lie, of angular size roughly \( \pi/8 \), characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of \( \pi/4 \) and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume \( y > 0 \): otherwise replace \( y \) by \(-y\) below):

\[
\begin{align*}
0 & 0 < |y| < 0.41421x, \text{ then } \text{atan} \left( \frac{|y|}{x} \right) \text{ is given by a nicely convergent Taylor series;}
1 & 0 < 0.41421x < |y| < x, \text{ then } \text{atan} \left( \frac{|y|}{x} \right) = \frac{\pi}{4} - \text{atan} \left( \frac{x-|y|}{x+|y|} \right).
\end{align*}
\]
2 0 < 0.41421|y| < x < |y|, then \( \frac{|y|}{x} = \frac{\pi}{4} + \tan \left( \frac{-x+|y|}{x+|y|} \right) \);

3 0 < x < 0.41421|y|, then \( \frac{|y|}{x} = \frac{\pi}{2} - \tan \left( \frac{x}{|y|} \right) \);

4 0 < -x < 0.41421|y|, then \( \frac{|y|}{x} = \frac{\pi}{2} + \tan \left( \frac{-x}{|y|} \right) \);

5 0 < 0.41421|y| < -x < |y|, then \( \frac{|y|}{x} = \frac{3\pi}{4} - \tan \left( \frac{x}{|y|} \right) \);

6 0 < -0.41421x < |y| < -x, then \( \frac{|y|}{x} = \frac{3\pi}{4} + \tan \left( \frac{-x}{|y|} \right) \);

7 0 < |y| < -0.41421x, then \( \frac{|y|}{x} = \pi - \tan \left( \frac{|y|}{x} \right) \).

In the following, we denote by \( z \) the ratio among \( \frac{|y|}{x} \), \( \frac{|x|}{y} \), \( \frac{|x|+|y|}{x-y} \), \( \frac{|x|-|y|}{x+y} \) which appears in the right-hand side above.

### 34.2.1 Arctangent and arccotangent

The parsing step manipulates \( \text{atan} \) and \( \text{acot} \) like \( \text{min} \) and \( \text{max} \), reading in an array of operands, but also leaves \use_i:nn or \use_ii:nn depending on whether the result should be given in radians or in degrees. The helper \_fp_parse_function_one_two:nnw checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number \_fp_atan_default:w places \c_one_fp (expanded) after it; otherwise \_fp_atan_default:w is omitted by \_fp_parse_function_one_two:nnw.

\[
\text{\_fp_atan_o:Nw, \_fp_acot_o:Nw, \_fp_atan_default:w}
\]

If either operand is \texttt{nan}, we return it. If both are normal, we call \_fp_atan_normal:o:NNNwNnw. If both are zero or both infinity, we call \_fp_atan_inf:o:NNNw with argument 2, leading to a result among \{±\pi/4, ±3\pi/4\} (in degrees, \{±45, ±135\}). Otherwise, one is much bigger than the other, and we call \_fp_atan_inf:o:NNNw with either an argument of 4, leading to the values ±\pi/2 (in degrees, ±90), or 0, leading to \{±0, ±\pi\} (in degrees, \{±0, ±180\}). Since \( \text{acot}(x,y) = \text{atan}(y,x) \), \_fp_acotii:o:ww simply reverses its two arguments.

\[
\text{\_fp_atanii:o:Nw, \_fp_acotii:o:Nw}
\]
This auxiliary is called whenever one number is \(\pm 0\) or \(\pm \infty\) (and neither is NaN). Then the result only depends on the signs, and its value is a multiple of \(\pi/4\). We use the same auxiliary as for normal numbers, \_\_fp_atan_combine_o:NwwNwwN, with arguments the final sign \#2; the octant \#3; \(\tan z/z = 1\) as a fixed point number; \(z = 0\) as a fixed point number; and \(z = 0\) as an extended-precision number. Given the values we provide, \(\tan z\) is computed to be 0, and the result is \([\#3/2] \cdot \pi/4\) if the sign \#5 of \(x\) is positive, and \([(7 - \#3)/2] \cdot \pi/4\) for negative \(x\), where the divisions are rounded up.

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like \(\tan(x, \sqrt{1 - x^2})\) without intermediate rounding errors.
This receives: the sign \#1 of \( y \), its exponent \#2, its 24 digits \#3 in groups of 4, and similarly for \( x \). We prepare to call \_\_fp_atan\_combine_o:NNwwN which expects the sign \#1, the octant, the ratio \((\text{atan} z)/z = 1 - \cdots\), and the value of \( z \), both as a fixed point number and as an extended-precision floating point number with a mantissa in [0.01, 1). For now, we place \#1 as a first argument, and start an integer expression for the octant. The sign of \( x \) does not affect \( z \), so we simply leave a contribution to the octant: \(<\text{octant}> 7 - <\text{octant}>\) for negative \( x \). Then we order \(|y|\) and \(|x|\) in a non-decreasing order: if \(|y| > |x|\), insert 3− in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by \_\_fp_atan\_div:wwwwN after the operands have been ordered.

\[
\begin{align*}
\text{\_\_fp_atan\_test_o:NwwNwwN} \ &\#1#2,#3; #4#5,#6; \\
&\text{\_\_fp_atan\_combine_o:NNwwN} \ #1 \ #1 \\
&\text{\_\_fp_int_eval:w} \ #4 \ #4 \\
&7 - \text{\_\_fp_int_eval:w} \\
&\text{\_\_fp\_ep\_compare:www} \ #2,#3; #5,#6; > 0 \exp_stop_f: \\
&3 - \text{\_\_fp\_reverse\_args:Nww} \\
&\text{\_\_fp\_atn\_div:wwwwN} \ #2,#3; #5,#6; \\
&\text{\_\_fp\_atn\_auxi:ww} \\
\end{align*}
\]

This receives two positive numbers \( a \) and \( b \) (equal to \(|x|\) and \(|y|\) in some order), each as an exponent and 6 blocks of 4 digits, such that \( 0 < a < b \). If \( 0.41421b < a \), the two numbers are "near", hence the point \((y, x)\) that we started with is closer to the diagonals \(|y| = |x|\) than to the axes \( xy = 0 \). In that case, the octant is 1 (possibly combined with the 7− and 3− inserted earlier) and we wish to compute \( \text{atan} \frac{b}{a} \). Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute \( \text{atan} \frac{a}{b} \). In any case, call \_\_fp_atan\_auxi:ww followed by \( z \), as a comma-delimited exponent and a fixed point number.

\[
\begin{align*}
\text{\_\_fp\_atn\_div:wwwwN} \ #1,#2#3; #4,#5#6; \\
&\text{\_\_fp\_atn\_near:www} \\
&\text{\_\_fp\_atn\_near\_aux:www} \\
\end{align*}
\]
\__fp_atan_auxi:ww
\__fp_atan_auxii:w

Convert \( z \) from a representation as an exponent and a fixed point number in \([0, 0.01)\) to a fixed point number only, then set up the call to \( \__fp_atan_Taylor_loop:www \), followed by the fixed point representation of \( z \) and the old representation.

\__fp_atan_Taylor_loop:www
\__fp_atan_Taylor_break:w

We compute the series of \( (\tan z)/z \). A typical intermediate stage has \( #1 = 2k - 1, \ #2 = \frac{1}{2^{k-1}} - z^2(\frac{1}{2^{k-1}} - z^2(\cdots - z^2(\frac{1}{2^{k-3}}))) \), and \( #3 = z^2 \). To go to the next step \( k \rightarrow k - 1 \), we compute \( \frac{1}{2^{k-1}} \), then subtract from it \( z^2 \) times \( #2 \). The loop stops when \( k = 0 \); then \( #2 \) is \( (\tan z)/z \), and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result \( #2 \) afterwards.
This receives a \langle \text{sign} \rangle, an \langle \text{octant} \rangle, a fixed point value of \( \frac{\text{atan}}{z} \), a fixed point number \( z \), and another representation of \( z \), as an \langle \text{exponent} \rangle and the fixed point number \( 10^{\langle \text{exponent} \rangle}z \), followed by either \text{use\_i:nn} (when working in radians) or \text{use\_ii:nn} (when working in degrees). The function computes the floating point result

\[
\text{(sign)} \left( \frac{\langle \text{octant} \rangle}{2} \pi + (-1)^{\langle \text{octant} \rangle} \frac{\text{atan}}{z} z \right),
\]

multiplied by 180/\pi if working in degrees, and using in any case the most appropriate representation of \( z \). The floating point result is passed to \text{\_fp\_sanitize:Nw}, which checks for overflow or underflow. If the octant is 0, leave the exponent \#5 for \text{\_fp\_sanitize:Nw}, and multiply \#3 = \text{atan} with \#6, the adjusted \( z \). Otherwise, multiply \#3 = \frac{\text{atan}}{z} with \#4 = z, then compute the appropriate multiple of \( \frac{\pi}{4} \) and add or subtract the product \#3 \cdot \#4. In both cases, convert to a floating point with \text{\_fp\_fixed_to_float_o:wN}.

\begin{verbatim}
\cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
\exp_after:wN \__fp_sanitize:Nw
\exp_after:wN #1
\int_value:w \__fp_int_eval:w #2 / 2 ; #2;
\else:
\exp_after:wN \use_ii:nn
\fi:
\exp_after:wN \use_i:nn
\fi:
{ \#5 \__fp_fixed_mul:wN \#3; \#6; }
{ \__fp_fixed_mul:wN \#3; \#4;

\exp_after:wN \__fp_atan_combine_aux:ww
\int_value:w \__fp_int_eval:w \#2 / \#2;
}
{ \#7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
#1
}
\cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
{ \__fp_fixed_mul_short:wwn
\{7853}{9186}{3397}{4483}{0961}{5661};
\{\#1}{0000}{0000};
\{
\if_int_odd:w \#2 \exp_stop_f:
\exp_after:wN \__fp_fixed_sub:wwn
\else:
\exp_after:wN \__fp_fixed_add:wwn
\fi:
\}
}
\end{verbatim}

(End definition for \text{\_fp\_atan\_combine\_o:NwwwwwN} and \text{\_fp\_atan\_combine\_aux:ww}.)
34.2.2 Arcsine and arccosine

\_\_fp_asin_o:w

Again, the first argument provided by 13fp-parse is \use_1:nn if we are to work in radians and \use_ii:nn for degrees. Then comes a floating point number. The arcsine of ±0 or NaN is the same floating point number. The arcsine of ±∞ raises an invalid operation exception. Otherwise, call an auxiliary common with 1\_\_fp_acos_o:w, feeding it information about what function is being performed (for “invalid operation” exceptions).

\cs_new:Npn \_\_fp_asin_o:w #1 \s__fp \_\_fp_chk:w #2#3; @
\{
\if_case:w #2 \exp_stop_f:
\_\_fp_case_return_same_o:w
\or:
\_\_fp_case_use:nw
{ \_\_fp_asin_normal_o:NfwNnnnw #1 { #1 \{ asin \} \{ asind \} } }
\or:
\_\_fp_case_use:nw
{ \_\_fp_invalid_operation_o:fw \{ #1 \{ asin \} \{ asind \} \} }
\else:
\_\_fp_case_return_same_o:w
\fi:
\s__fp \_\_fp_chk:w #2 #3;
\}

(End definition for \_\_fp_asin_o:w.)

\_\_fp_acos_o:w

The arccosine of ±0 is π/2 (in degrees, 90). The arccosine of ±∞ raises an invalid operation exception. The arccosine of NaN is itself. Otherwise, call an auxiliary common with \_\_fp_sin_o:w, informing it that it was called by acos or acosd, and preparing to swap some arguments down the line.

\cs_new:Npn \_\_fp_acos_o:w #1 \s__fp \_\_fp_chk:w #2#3; @
\{
\if_case:w #2 \exp_stop_f:
\_\_fp_case_use:nw
{ \_\_fp_atan_inf_o:NNNw #1 0 4 }
\or:
\_\_fp_case_use:nw
{ \_\_fp_asin_normal_o:NfwNnnnw #1 { #1 \{ acos \} \{ acosd \} }
\_\_fp_reverse_args:Nww }
\or:
\_\_fp_case_use:nw
{ \_\_fp_invalid_operation_o:fw \{ #1 \{ acos \} \{ acosd \} \} }
\else:
\_\_fp_case_return_same_o:w
\fi:
\s__fp \_\_fp_chk:w #2 #3;
\}

(End definition for \_\_fp_acos_o:w.)

\_fp_asin_normal_o:NfwNnnnw

If the exponent #5 is at most 0, the operand lies within (−1, 1) and the operation is permitted: call \_\_fp_asin_auxi_o:NmNww with the appropriate arguments. If the number is exactly ±1 (the test works because we know that #5 ≥ 1, #6#7 ≥ 10000000, #8#9 ≥ 0,
with equality only for \( \pm 1 \)), we also call \( \_\_f p\_a s i n\_a u x i\_o: N N W w \). Otherwise, \( \_\_f p\_u s e\_i: w w \) gets rid of the \( \text{asin} \) auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

\[
\begin{align*}
\text{\texttt{\textbackslash cs\_new\_Npn}} & \quad \text{\texttt{\_\_fp\_asin\_normal\_o:\textbackslash N w\textbackslash N n n n n}} \\
\text{\texttt{\_\_fp\_ep\_to\_fixed:ww n N w}} & \quad \text{\texttt{#1\#2\#3}} \\
\text{\texttt{\_\_fp\_ep\_mul:ww n}} & \quad \text{\texttt{#4\#5\#6\#7\#8\#9}}; \\
\{ & \quad \text{\texttt{\_\_fp\_ep\_to\_ep:ww N}} \\
\text{\_\_fp\_atan\_test\_o:ww N w}} & \quad \text{\texttt{\_\_fp\_reverse\_args:Nww \{#2\}}} \\
\text{\_\_fp\_as\_i\_s\_sqrt:wn} & \quad \text{\texttt{\_\_fp\_ep\_is\_sqrt:wwn}}
\end{align*}
\]

\( \_\_f p\_a s i n\_a u x i\_o: N N W w \) We compute \( x/\sqrt{1-x^2} \). This function is used by \( \text{asin} \) and \( \text{acos} \), but also by \( \text{acsc} \) and \( \text{asec} \) after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate \( 1-x^2 \) as \((1+x)(1-x)\): this behaves better near \( x = 1 \). We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root \( 1/\sqrt{1-x^2} \). Finally, multiply by the (positive) extended-precision float \( |x| \), and feed the (signed) result, and the number \( +1 \), as arguments to the arctangent function. When computing the arccosine, the arguments \( x/\sqrt{1-x^2} \) and \( +1 \) are swapped by \( \#2 \) (\( \_\_f p\_r e\_s\_i\_v\_a\_r\_g\_s:N\_w w \) in that case) before \( \_\_f p\_a t a n\_t e\_s\_t\_o: N w w N w N N \) is evaluated. Note that the arctangent function requires normalized arguments, hence the need for \( \_\_f p\_t o\_e p \) and continue after \( \_\_f p\_m u l \).

\[
\begin{align*}
\text{\texttt{\textbackslash cs\_new\_Npn}} & \quad \_\_f p\_a s i n\_a u x i\_o: N N W w \texttt{#1\#2\#3\#4,\#5} \\
\{ & \quad \_\_f p\_e p\_t o\_f i x d: w w \texttt{#4,\#5} \\
\_\_f p\_a s i n\_i s q r t: w n} & \quad \_\_f p\_e p\_m u l: w w w w n \texttt{#4,\#5} \\
\_\_f p\_e p\_t o\_e p: w w N} & \quad \_\_f p\_f i x e d\_c o\_n t i n u e: w n \\
\{ & \quad \#2 \_\_f p\_a t a n\_t e\_s\_t\_o: N W W N W N N \texttt{#3}\} \\
0 1,\{0000\{0000\{0000\{0000\{0000\{0000\} \#1 \\
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
\}
34.2.3 Arccosecant and arcsecent

\__fp_acsc_o:w

Cases are mostly labelled by 2, except when \#2 is 2: then we use \#3\#2, which is 02 = 2 when the number is \(+\infty\) and 22 when the number is \(-\infty\). The arccosecant of \(0\) raises an invalid operation exception. The arccosecant of \(+\infty\) is \(+0\) with the same sign. The arccosecant of NaN is itself. Otherwise, \__fp_acsc_normal_o:NfwNnw does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

\begin{verbatim}
\cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
    { \__fp_invalid_operation_o:fw { #1 \{ acsc \} \{ acscd \} } }
  \or: \__fp_case_use:nw
    \__fp_acsc_normal_o:NfwNnw #1 \{ acsc \} \{ acscd \}
  \or: \__fp_case_return_o:Nw \c_zero_fp
  \or: \__fp_case_return_same_o:w
  \else: \__fp_case_return_o:Nw \c_minus_zero_fp \fi:
  \__fp_case_return_o:Nw \cs_name_use:Nw \c__fp_acsc_normal_o
\end{verbatim}

\begin{verse}
(End definition for \__fp_acsc_o:w.)
\end{verse}

\__fp_asec_o:w

The arccosecant of \(0\) raises an invalid operation exception. The arccosecant of \(\pm\infty\) is \(\pi/2\) (in degrees, 90). The arccosecant of NaN is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a \__fp_reverse_args:Nww following precisely that appearing in \__fp_acos_o:w.

\begin{verbatim}
\cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  \if_case:w #2 \exp_stop_f:
    \__fp_case_use:nw
    \__fp_invalid_operation_o:fw { #1 \{ asec \} \{ asecd \} }
  \or: \__fp_case_use:nw
    \__fp_acsc_normal_o:NfwNnw #1 \{ asec \} \{ asecd \}
    \__fp_reverse_args:Nww
  \or: \__fp_case_use:nw
    \__fp_acsc_normal_o:NfwNnw #1 \{ asec \} \{ asecd \}
    \__fp_reverse_args:Nww
  \else: \__fp_case_return_o:Nw \c__fp_atan_inf_o:NNNw #1 0 4 \fi:
  \__fp_case_return_o:Nw \cs_name_use:Nw \c__fp_asec_o
\end{verbatim}

\begin{verse}
(End definition for \__fp_asec_o:w.)
\end{verse}

\__fp_acsc_normal_o:NfwNnw

If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to \__fp_asin_auxi_o:NNww (with all the appropriate arguments). This computes what we want thanks to \(\text{acsc}(x) = \text{asin}(1/x)\) and \(\text{asec}(x) = \text{acos}(1/x)\).

\begin{verbatim}
\cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2\#3 \s__fp \__fp_chk:w 1#4#5#6; \int_compare:nNnTF \#5 < 1
\end{verbatim}

910
\[\__fp_acsc_normal_o:Nw\]\(\__fp_acsc_normal_o:Nw\)

\(\__fp_tuple_convert:Nw\)
\(\__fp_tuple_convert_loop:nNw\)
\(\__fp_tuple_convert_end:w\)

The first argument is for instance \(\__fp_to_tl_dispatch:w\), which converts any floating point object to the appropriate representation. We loop through all items, putting 
\(,~\) between all of them and making sure to remove the leading 
\(,~\).

\[
\begin{align*}
\cs_new:Npn \__fp_tuple_convert:Nw \ #1 \ \s__fp_tuple \ \__fp_tuple_chk:w \ #2 ;
\int_case:nnF { \__fp_array_count:n {#2} } &
\begin{cases}
0 \{ \() \} &
1 \{ \__fp_tuple_convert:w \ @ \ { \#1 \ #2 , } \} \\
\__fp_tuple_convert_loop:nNw \ {} \ #1 &\ #2 \ { ? \ \__fp_tuple_convert_end:w } ;
\{ \\use_none:nn \}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\cs_new:Npn \__fp_tuple_convert_loop:nNw \ #1#2#3#4; \ #5 @ \ #6 &
\{ \use_none:n \ #3 \\
\exp_args:Nf \__fp_tuple_convert_loop:nNw \ { \#2 \ #3#4 ; } \ #2 \ #5 \\
@ \ { \#6 \ , \ - \ #1 } \\
\cs_new:Npn \__fp_tuple_convert_end:w \ #1 \ @ \ #2 &
\{ \exp_after:wN \ ( \exp:w \ \exp_end_continue_f:w \ #2 ) \}
\end{align*}
\]

\((\text{End definition for } \__fp_tuple_convert:Nw, \ \__fp_tuple_convert_loop:nNw, \text{ and } \__fp_tuple_convert_end:w)\)
35.2 Trimming trailing zeros

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing zero is reached, the second argument is the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```
\cs_new:Npn \__fp_trim_zeros:w #1 ;
{\__fp_trim_zeros_loop:w #1 ; \__fp_trim_zeros_loop:w 0 ; \__fp_trim_zeros_dot:w . ; \s__fp_stop}
\cs_new:Npn \__fp_trim_zeros_loop:w #1 0 ; #2 { #2 #1 ; #2 }
\cs_new:Npn \__fp_trim_zeros_dot:w #1 . ; { \__fp_trim_zeros_end:w #1 ; }
\cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }
```

(End definition for \__fp_trim_zeros:w and others.)

35.3 Scientific notation

The three public functions evaluate their argument, then pass it to \__fp_to_scientific_dispatch:w.

```
\cs_new:Npn \fp_to_scientific:N #1
{\exp_after:wN \__fp_to_scientific_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_scientific:N { c }
\cs_new:Npn \fp_to_scientific:n
{\exp_after:wN \__fp_to_scientific_dispatch:w \exp:w \exp_end_continue_f:w \__fp_parse:n}
```

(End definition for \fp_to_scientific:N and \fp_to_scientific:n. These functions are documented on page 201.)

We allow tuples.

```
\cs_new:Npn \__fp_to_scientific_dispatch:w \__fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
\cs_new:Npn \__fp_tuple_to_scientific:w
{\__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }
```

(End definition for \__fp_to_scientific_dispatch:w, \__fp_to_scientific_recover:w, and \__fp_tuple_to_scientific:w.)

Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (#2 = 2) start with -; we then only need to care about positive numbers and \texttt{nan}. Then
filter the special cases: ±0 are represented as 0; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; \texttt{\texttt{nan}} is represented as 0 after an “invalid_operation” exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

\begin{verbatim}
\cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
{\if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
\if_case:w #1 \exp_stop_f:
\__fp_case_return:nw { 0.00000000000000e0 }
\or: \exp_after:wN \__fp_to_scientific_normal:wnnnn
\or:
\__fp_case_use:nw
\{ \fp_to_scientific:N \c__fp_overflowing_fp \}
\{ \fp_to_scientific \}
}\or:
\__fp_case_use:nw
\{ \fp_to_scientific_normal:nnw
\{ \fp_to_scientific:N \c_zero_fp \}
\{ \fp_to_scientific \}
\}
\fi:
\s__fp \__fp_chk:w #1 #2
}
\cs_new:Npn \__fp_to_scientific_normal:wnnnnn
\__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
{ \exp_after:wN \__fp_to_scientific_normal:wNw
\exp_after:wN e\int_value:w \__fp_int_eval:w #2 - 1
; #3 #4 #5 #6 ;
}
\cs_new:Npn \__fp_to_scientific_normal:wNw \#1 \#2#3
{ \#2.\#3 \#1 }
\end{verbatim}

(End definition for \__fp_to_scientific:w, \__fp_to_scientific_normal:wnnnn, and \__fp_to_scientific_normal:wNw.)

35.4 Decimal representation

\__fp_to_decimal:N \__fp_to_decimal:c \__fp_to_decimal:n

All three public variants are based on the same \__fp_to_decimal_dispatch:w after evaluating their argument to an internal floating point.

\begin{verbatim}
\cs_new:Npn \fp_to_decimal:N #1
{ \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_decimal:N { c }
\cs_new:Npn \fp_to_decimal:n
{ \exp_after:wN \__fp_to_decimal_dispatch:w
\exp:w \exp_end_continue_f:w \__fp_parse:n
}
\end{verbatim}

913
We allow tuples.

\__fp_to_decimal:w
\__fp_to_decimal_normal:wnnnnn
\__fp_to_decimal_large:Nnnw
\__fp_to_decimal_huge:wnnnn

The structure is similar to \__fp_to_scientific:w. Insert - for negative numbers. Zero gives 0, ±∞ and NaN yield an “invalid operation” exception; note that ±∞ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range \[1, 15\] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with \int_value:w, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0⟨zeros⟩⟨digits⟩, trimmed.
\begin{verbatim}
\int_compare:nNnTF {#2} > 0
  \int_compare:nNnTF {#2} < \c__fp_prec_int
    \__fp_decimate:nNnnn { \c__fp_prec_int - #2 }
    \__fp_to_decimal_large:Nnw
  }
  \exp_after:wN \exp_after:wN \exp_after:wN \__fp_to_decimal_huge:wnnnn
  \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
  \exp_after:wN \__fp_trim_zeros:w \int_value:w
    \if_int_compare:w #2 > 0 \exp_stop_f:
      #2
    \fi:
    \exp_stop_f:
    #3.#4 ;
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }
\end{verbatim}

(End definition for \_\_fp_to_decimal:w and others.)

### 35.5 Token list representation

- **\_\_fp_to_tl:N**
- **\_\_fp_to_tl:c**
- **\_\_fp_to_tl:n**

These three public functions evaluate their argument, then pass it to \_\_fp_to_tl_-
\_\_fp_to_tl_dispatch:w.

\begin{verbatim}
\cs_new:Npn \_\_fp_to_tl_dispatch:w #1
  \__fp_change_func_type:NNN #1 \_\_fp_to_tl:w \_\_fp_to_tl_recover:w #1
\end{verbatim}

(End definition for \_\_fp_to_tl:N and \_\_fp_to_tl:n. These functions are documented on page 202.)

We allow tuples.

\begin{verbatim}
\cs_new:Npn \_\_fp_to_tl_dispatch:w #1
  \__fp_change_func_type:NNN #1 \_\_fp_to_tl:w \_\_fp_to_tl_recover:w #1
\cs_new:Npn \_\_fp_to_tl_recover:w #1 #2 ;
\end{verbatim}

915
A structure similar to \_\_fp_to_scientific_dispatch:w and \_\_fp_to_decimal_-
dispatch:w, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range \([-2,16]\), and otherwise use scientific notation.

```
\cs_new:Npn \_\_fp_to_tl:w \s__fp \__fp_chk:w #1#2 { 
  \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
  \if_case:w #1 \exp_stop_f:
    \__fp_case_return:nw { 0 }
    \or: \exp_after:wN \_\_fp_to_tl_normal:nnnnn
    \or: \__fp_case_return:nw { inf }
    \else: \__fp_case_return:nw { nan }
  \fi:
}
```

```
\cs_new:Npn \_\_fp_to_tl_normal:nnnnn #1 { 
  \int_compare:nTF { -2 <= #1 <= \c__fp_prec_int } { \_\_fp_to_decimal_normal:wnnnn } { \_\_fp_to_tl_scientific:wnnnn }
  \s__fp \__fp_chk:w 1 0 {#1}
}
```

```
\cs_new:Npn \_\_fp_to_tl_scientific:wnnnn \s__fp \__fp_chk:w 1 #1 #2#3#4#5#6 ;
```

```
\int_value:w \__fp_int_eval:w #2 - 1 ; #3 #4 #5 #6 ;
```

```
\cs_new:Npn \_\_fp_to_tl_scientific:wNw #1 ; #2#3 ; #1 }
```

(End definition for \_\_fp_to_tl:w and others.)

### 35.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.
35.7 Convert to dimension or integer

All three public variants are based on the same \__fp_to_dim_dispatch:w after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

\begin{verbatim}
\cs_new:Npn \fp_to_dim:N #1 { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_dim:N { c }
\cs_new:Npn \fp_to_dim:n { \exp_after:wN \__fp_to_dim_dispatch:w \exp:w \exp_end_continue_f:w \__fp_parse:n }
\cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
{ \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w #1 #2 ; }
\cs_new:Npn \__fp_to_dim_recover:w #1 { \__fp_invalid_operation:nnw { 0pt } { fp_to_dim } }
\cs_new:Npn \__fp_to_dim:w #1; { \__fp_to_decimal:w #1 ; pt }
\end{verbatim}

(End definition for \fp_to_dim:N and others. These functions are documented on page 201.)

35.8 Convert from a dimension

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (i.e., expressed in scaled points), then multiplied by $2^{-16} = 917$.
0.0000152587890625 to give a value expressed in points. The auxiliary \texttt{\_\_fp_mul\_npos\_o:Nww} expects the desired (\textit{final sign}) and two floating point operands (of the form \texttt{\_s\_fp\ldots;}) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of \texttt{\_\_fp_from_dim\_test:ww}, and is combined with the exponent $-4$ of $2^{-16}$. There is also a need to expand afterwards: this is performed by \texttt{\_\_fp_mul\_npos\_o:Nww}, and cancelled by \texttt{\prg\_do\_nothing}: here.

\begin{verbatim}
\cs_new:Npn \dim_to_fp:n #1 22411 \exp_after:wN \__fp_from_dim_test:ww #1
22412 \exp_after:wN \if_meaning:w 0 \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
22413 \else:
22414 \exp_after:wN \__fp_from_dim:wNw
22415 \if_meaning:w - \exp_after:wN , \exp_after:wN 2 \int_value:w #1
22416 \else:
22417 \exp_after:wN , \exp_after:wN 0 \int_value:w #1
22418 \fi:
22419 \fi:
22420 \cs_new:Npn \__fp_from_dim:wNNnnnnnn ;
22421 \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
22422 \int_value:w \__fp_int_eval:w #1 - 4
22423 \if_meaning:w - #2
22424 \exp_after:wN \__fp_from_dim:wNNnnnnnn ;
22425 \else:
22426 \exp_after:wN \__fp_from_dim:wNNnnnnnn ;
22427 \fi:
22428 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6#7#8
22429 { \__fp_from_dim:wnnnnwNn #1; #2#300} {0000} ; }
22430 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6#7#8
22431 { \__fp_mul\_npos\_o:Nww \_s\_fp\_\_fp\_check:w 1 \_#7 \#5 \_#1 ; }
22432 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6#7#8
22433 { \__fp_mul\_npos\_o:Nww \_s\_fp\_\_fp\_check:w 1 \_#7 \#5 \_#1 ; }
22434 \prg\_do\_nothing:
22435 \end{verbatim}

(End definition for \texttt{\_\_fp_from_dim\_test:ww} and others. This function is documented on page 175.)

### 35.9 Use and eval

\begin{verbatim}
\fp\_use:N \fp\_use:c \fp\_eval:n
\end{verbatim}

Those public functions are simple copies of the decimal conversions.

\begin{verbatim}
\cs\_new\_eq:NN \fp\_use:N \fp\_to\_decimal:N
\cs\_generate\_variant:Nn \fp\_use:N \{ c \}
\cs\_new\_eq:NN \fp\_eval:n \fp\_to\_decimal:n
\end{verbatim}

(End definition for \texttt{\_\_fp_from_dim\_test:ww} and \_\_fp_eval:n. These functions are documented on page 202.)
\fp_sign:n \fp_abs:n \fp_max:nn \fp_min:nn

Trivial but useful. See the implementation of \fp_add:nn for an explanation of why to use \__fp_parse:n, namely, for better error reporting.

\cs_new:Npn \fp_sign:n \fp_abs:n
\{ \fp_to_decimal:n \{ sign \__fp_parse:n \{#1\} \} \}

(End definition for \fp_sign:n. This function is documented on page 201.)

\cs_new:Npn \fp_max:nn \fp_min:nn
\{ \fp_to_decimal:n \{ max ( \__fp_parse:n \{#1\}, \__fp_parse:n \{#2\} ) \} \}

(End definition for \fp_max:nn and \fp_min:nn. These functions are documented on page 216.)

35.10 Convert an array of floating points to a comma list

Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

\cs_new:Npn \__fp_array_to_clist:n \__fp_array_to_clist_loop:Nw
\{ \tl_if_empty:nF {#1} \}
\{ \use_none:n #1 \}
\{ , ~ \} \fp_to_tl:n \{ #1 #2 ; \}
\__fp_array_to_clist_loop:Nw

The \use_ii:nn function is expanded after \__fp_expand:n is done, and it removes ,~ from the start of the representation.
Those functions may receive a variable number of arguments. We won’t use the argument ?.

```latex
\cs_new:Npn \__fp_parse_word_rand:N { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
\cs_new:Npn \__fp_parse_word_randint:N { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }
```

(End definition for \_\_fp_parse_word_rand:N and \_\_fp_parse_word_randint:N.)

### 36.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the false branch first.

```latex
\sys_if_rand_exist:F
{ \__kernel_msg_new:nnn { kernel } { fp-no-random }
\cs_new:Npn \__fp_rand_o:Nw #1 @ { \__kernel_msg_expandable_error:nnn { kernel } { fp-no-random } { fp-rand } \exp_after:wN \c_nan_fp }
\cs_new_eq:NN \__fp_randint_o:Nw \__fp_rand_o:Nw
\cs_new:Npn \int_rand:nn #1#2 { \__kernel_msg_expandable_error:nnn { kernel } { fp-no-random } \int_eval:n {#1} }
\cs_new:Npn \int_rand:n #1 { \__kernel_msg_expandable_error:nnn { kernel } { fp-no-random } \int_eval:n {#1} 1 }
}
\sys_if_rand_exist:T
{ Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).
}
The primitive random number generator (RNG) is provided as $\texttt{\textbackslash tex\_uniformdeviate:D}$. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo $2^{28}$. When $\texttt{\textbackslash tex\_uniformdeviate:D \langle integer\rangle}$ is called (for brevity denote by $N$ the $\langle integer\rangle$), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and $N$ from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that $\texttt{\textbackslash tex\_uniformdeviate:D}$ only gives a uniform distribution from 0 to $N-1$ if $N$ is a divisor of $2^{28}$, so we will mostly call the RNG with such power of 2 arguments. If $N$ does not divide $2^{28}$, then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value $X$ requires about $2^{50}/N$ random trials. But collective patterns can reduce this to about $2^{50}/N^2$. For instance with $N = 3 \times 2^4$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^4/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{50}/2^{2k} = 9 \times 2^{50}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly $2^{20}$ calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortable close to being so, and it becomes worse as $N$ is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo $2^{28}$, hence the lowest $k$ bits of the random numbers only depend on the lowest $k$ bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N-1$ is thus to scale the raw 28-bit integer, as the engine’s RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument $N$ to get a set of $K$ integers in $[0, N-1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N-1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo $N$ (a more detailed analysis shows that 0 appears with frequency close to 3/4). The resulting set thus has more triplets $(a,b,c)$ than expected obeying $a = b + c$ modulo $N$. Namely it will have of order $(K - 55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per $\texttt{\textbackslash int\_rand:nn}$ we would need to investigate linear relations between the $x_{2n}$ on the one hand and between the $x_{2n+1}$ on the other hand. Such relations will have more complicated coefficients than $\pm 1$, which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument $2^{28}$ or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth’s TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG’s underlying 28-bit integers are uniform.
• Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.

• Quick. The algorithm should be fast in TeX, so no “bit twiddling”, but “digit twiddling” is ok.

• Simple. The behaviour must be documentable precisely.

• Predictable. The number of calls to the RNG should be the same for any \int random, because then the algorithm can be modified later without changing the result of other uses of the RNG.

• Robust. It should work even for \int random \{- \c max int \} \{ \c max int \} where the range is not representable as an integer. In fact, we also provide later a floating-point randint whose range can go all the way up to \(2 \times 10^{36} - 1\) possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by random\((N)\) one call to \texttt{tex_uniformdeviate} with argument \(N\), and by ediv\((p, q)\) the \(\varepsilon\)-\TeX interchangeable division giving \([p/q + 1/2]\). Denote by \(\langle \min \rangle\), \(\langle \max \rangle\) and \(R = \langle \max \rangle - \langle \min \rangle + 1\) the arguments of \texttt{int random} and the number of possible outcomes. Note that \(R \in [1, 2^{32} - 1]\) cannot necessarily be represented as an integer (however, \(R \leq 2^{31}\) can). Our strategy is to get two 28-bit integers \(X\) and \(Y\) from the RNG, split each into 14-bit integers, as \(X = X_1 \times 2^{14} + X_0\) and \(Y = Y_1 \times 2^{14} + Y_0\) then return essentially \(\langle \min \rangle + \left\lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \right\rfloor\). For small \(R\) the \(X_0\) term has a tiny effect so we ignore it and we can compute \(R \times Y/2^{28}\) much more directly by random\((R)\).

• If \(R \leq 2^{17} - 1\) then return \(\text{ediv}(\text{random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \min \rangle\). The shifts by \(2^{13}\) and \(-1\) convert \(\varepsilon\)-\TeX division to truncated division. The bound on \(R\) ensures that the number obtained after the shift is less than \(\c max int\). The non-uniformity is at most of order \(2^{37}/2^{42} = 2^{-5}\).

• Split \(R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0\), where \(R_2 \in [0, 15]\). Compute \(\langle \min \rangle + R_2 X_1 \times 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}(2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14})\) then map a result of \(\langle \max \rangle + 1\) to \(\langle \min \rangle\). Writing each ediv in terms of truncated division with a shift, and using \(\lfloor (p + [r/s]) / q \rfloor = \lfloor (ps + r) / (sq) \rfloor\), what we compute is equal to \(\lfloor (\text{exact} + 2^{-29} + 2^{-15} + 2^{-1}) \rfloor\) with \(\langle \text{exact} \rangle = \langle \min \rangle + R \times 0. X_1 Y_1 X_0 Y_0\). Given we map \(\langle \max \rangle + 1\) to \(\langle \min \rangle\), the shift has no effect on uniformity. The non-uniformity is bounded by \(R/2^{56} < 2^{-24}\). It may be possible to speed up the code by dropping tiny terms such as \(R_0 X_0\), but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields \(\langle \max \rangle + 1\) with \(\langle \max \rangle = 2^{31} - 1\) (note that \(R\) is then arbitrary), we compute the result in two pieces. Compute \(\langle \text{first} \rangle = \langle \min \rangle + R_2 X_1 \times 2^{14}\) if \(R_2 \leq 2\) or \(\langle \min \rangle + 8 X_1 \times 2^{14} + (R_2 - 8) X_1 \times 2^{14}\) if \(R_2 \geq 8\), the expressions being chosen to avoid overflow. Compute \(\langle \text{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\ldots)\), at most \(R_2^{2^{14}} + R_1^{2^{14}} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1\), not at risk of overflowing. We have \(\langle \text{first} \rangle + \langle \text{second} \rangle = \langle \max \rangle + 1 = \langle \min \rangle + R\) if and only if \(\langle \text{second} \rangle = R_1^{2^{14}} + R_0 + R_2^{2^{14}}\) and \(2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2\) (namely \(R_2 = 0\) or \(X_1 = 2^{14} - 1\)). In that case, return \(\langle \min \rangle\), otherwise return \(\langle \text{first} \rangle + \langle \text{second} \rangle\), which is safe because it is at most \(\langle \max \rangle\). Note that the decision of what to return
does not need \textit{first} explicitly so we don’t actually compute it, just put it in an integer expression in which \textit{second} is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.

- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by $R$ and add \textit{min}. This requires some care because \texttt{fp-extended} only supports non-negative numbers.

\begin{verbatim}
c_kernel_randint_max_int\par Constant equal to $2^{17} - 1$, the maximal size of a range that \texttt{int_range:nn} can do with its “simple” algorithm.
\end{verbatim}

\begin{verbatim}
c__kernel_randint_max_int\par 
\int_const:Nn c__kernel_randint_max_int { 131071 }
\end{verbatim}

\begin{verbatim}
c_kernel_randint:n\par Used in an integer expression, \texttt{c_kernel_randint:n} \{$R$\} gives a random number $1 + \left\lfloor \left( R \text{random}(2^{14}) + \text{random}(R)/2^{14} \right) \right\rfloor$ that is in $[1, R]$. Previous code was computing $\left\lfloor p/2^{14} \right\rfloor$ as \texttt{ediv}(\texttt{p} - $2^{13}$, $2^{14}$) but that wrongly gives $-1$ for $p = 0$.
\end{verbatim}

\begin{verbatim}
c__kernel_randint:n\par
\cs_new:Npn \__kernel_randint:n #1
\{ \int_value:w \__fp_int_eval:w 9999 + \__kernel_randint:n { 10000 }
\}
\end{verbatim}

\begin{verbatim}
__fp_rand_myriads:n\par Used as \texttt{__fp_rand_myriads:n} \{$XXX$} with one letter \texttt{X} (specifically) per block of four digit we want; it expands to \texttt{;} followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in $[10000, 19999]$ for the usual reason of preserving leading zeros.
\end{verbatim}

\begin{verbatim}
__fp_rand_myriads:n\par
\cs_new:Npn __fp_rand_myriads:n #1
\{ \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: \}
\cs_new:Npn __fp_rand_myriads_loop:w #1 X
\{ \int_value:w __fp_int_eval:w 9999 + __kernel_randint:n \{ 10000 \}
\__fp_rand_myriads_loop:w
\}
\end{verbatim}
36.2 Random floating point

First we check that \texttt{random} was called without argument. Then get four blocks of four
digits and convert that fixed point number to a floating point number (this correctly sets
the exponent). This has a minor bug: if all of the random numbers are zero then the
result is correctly 0 but it raises the \texttt{underflow} flag; it should not do that.

\begin{verbatim}
\cs_new:Npn \__fp_rand_o:Nw ? #1 @
    { \tl_if_empty:nTF {#1}
        { \exp_after:wN \__fp_rand_o:w
            \exp:w \exp_end_continue_f:w
            \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
        }
        { \__kernel_msg_expandable_error:nnnnn
            \{ kernel \} \{ fp-num-args \} \{ rand() \} \{ 0 \} \{ 0 \}
            \exp_after:wN \c_nan_fp
        }
    }
\cs_new:Npn \__fp_rand_o:w ;
    { \exp_after:wN \__fp_sanitize:Nw
        \exp_after:wN 0
        \int_value:w \__fp_int_eval:w \c_zero_int
        \__fp_fixed_to_float_o:wN }
\end{verbatim}

(End definition for \texttt{\__fp_rand_o:Nw} and \texttt{\__fp_rand_o:w}.)

36.3 Random integer

Enforce that there is one argument (then add first argument 1) or two arguments. Call
\texttt{\__fp_randint_badarg:w} on each; this function inserts 1 \texttt{\exp_stop_f:} to end the
\texttt{\if_case:w} statement if either the argument is not an integer or if its absolute value is
\geq 10^{16}. Also bail out if \texttt{\__fp_compare_back:ww} yields 1, meaning that the bounds are
not in the right order. Otherwise an auxiliary converts each argument times $10^{-16}$ (hence
the shift in exponent) to a 24-digit fixed point number (see \texttt{l3fp-extended}). Then compute
the number of choices, $\langle max \rangle + 1 - \langle min \rangle$. Create a random 24-digit fixed-point number
with \texttt{\__fp_randint_myriads:n}, then use a fused multiply-add instruction to multiply the
number of choices to that random number and add it to $\langle min \rangle$. Then truncate to 16
digits (namely select the integer part of $10^{16}$ times the result) before converting back to a
floating point number (\texttt{\__fp_sanitze:Nw} takes care of zero). To avoid issues with
negative numbers, add 1 to all fixed point numbers (namely $10^{16}$ to the integers they
represent), except of course when it is time to convert back to a float.

\begin{verbatim}
\cs_new:Npn \__fp_randint_o:Nw ?
    { \__fp_parse_function_one_two:nww
        \{ randint \}
        \{ \__fp_randint_default:w \__fp_randint_o:w \}
\end{verbatim}

\begin{verbatim}
\cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
\cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
\end{verbatim}
\begin{verbatim}
\_fp_int:wTF \__fp \_fp_chk:w #1\#2\#3;
  \{  
    \if_meaning:w 1 #1
    \if_int_compare:w
      \_fp_use_i_until_s:nw #3 ; > \c__fp_prec_int
    \exp_stop_f:
    \fi:
    \fi:
  \}
  \{ 1 \exp_stop_f: \}
\cs_new:Npn \__fp_randint_o:w #1; #2; @
  \{  \if_case:w
    \__fp_randint_badarg:w #1;  \__fp_randint_badarg:w #2; \if:w 1 \__fp_compare_back:ww #2; #1; 1 \exp_stop_f: \fi:
      0 \exp_stop_f:
      \__fp_randint_auxi_o:ww #1; #2; \or:
      \__fp_invalid_operation_tl_o:ff
      \{ \_fp_int_to_clist:n \{ #1; #2; \} \}
    \exp:w
    \fi:
    \exp_after:wN \exp_end:
  \}\cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
  \{  \if_case:w
    \__fp_randint_badarg:w #1; \__fp_randint_badarg:w #2;
    \if:w 1 \__fp_compare_back:ww #2; #1; 1 \exp_stop_f: \fi:
      0 \exp_stop_f:
      \__fp_randint_auxii:wn #2 ;\{ \__fp_randint_auxii:wn \__fp_randint_auxiii_o:ww \}
    \exp:w
    \fi:
    \exp_after:wN \exp_end:
\cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1\#2\#3\#4 ;
  \{  \if_meaning:w 0 #1
    \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    \{ \exp_after:wN \_fp_fixed_continue:wn \c__fp_one_fixed_tl \}
  \}
  \exp_after:wN \_fp_ep_to_fixed:ww
  \int_value:w \__fp_int_eval:w
    #3 - \c__fp_prec_int , #4 \{0000\} \{0000\} ;
  \{  \if_meaning:w 0 #2
    \exp_after:wN \use_i:nnn
    \exp_after:wN \_fp_fixed_add_one:wn
    \fi:
    \exp_after:wN \_fp_fixed_sub:wwn \c__fp_one_fixed_tl
  \}
  \_fp_fixed_continue:wn
\end{verbatim}
\cs_new:Npn \__fp_randint_auxiiii_o:ww #1 ; #2 ;
{ 
\__fp_fixed_add:wwn #2 ;
{0000} {0000} {0000} {0001} {0000} {0000} ;
\__fp_fixed_sub:wn #1 ;
{ }
\exp_after:wN \use_i:nn
\exp_after:wN \__fp_fixed_mul_add:wwnn
\exp:w \exp_end_continue_f:w \__fp_rand_myriads:n \{ XXXXX \} ;
}
\__fp_randint_auxiv_o:ww #1 ; #2 ;
\__fp_randint_auxv_o:w #1 ; @
\cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
{ 
\if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
\if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
\if_int_compare:w #3#4 > #8#9 \exp_stop_f:
\__fp_use_i_until_s:nw
\fi:
\__fp_randint_auxv_o:w \{1\}{2\}{3\}{4\}{5}
}
\cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 0
{ 
\exp_after:wN \__fp_sanitize:Nw
\int_value:w 
\if_int_compare:w #1 < 10000 \exp_stop_f:
2
\else:
0
\exp_after:wN \exp_after:wN
\exp_after:wN \__fp_reverse_args:Nww
\fi:
\exp_after:wN \__fp_fixed_sub:wn \c__fp_one_fixed_tl
\{1\} \{2\} \{3\} \{4\} \{0000\} \{0000\} ;
{ 
\exp_after:wN \exp_stop_f:
\int_value:w \__fp_int_eval:w \c__fp_prec_int
\__fp_fixed_to_float_o:wN
\}
0
\exp:w \exp_after:wN \exp_end:
}

(End definition for \__fp_randint_o:Nw and others.)

\int_rand:nn \__fp_randint:ww
Evaluate the argument and filter out the case where the lower bound #1 is more than
the upper bound #2. Then determine whether the range is narrower than \c__kernel_-_randint_max_int; #2-#1 may overflow for very large positive #2 and negative #1. If the
range is narrow, call \__kernel_randint:n \{⟨choices⟩\} where ⟨choices⟩ is the number
of possible outcomes. If the range is wide, use somewhat slower code.

\cs_new:Npn \int_rand:nn #1#2
\{
  \int_eval:n
  \{
    \exp_after:wN \__fp_randint:ww
    \int_value:w \int_eval:n {#1} \exp_after:wN ;
    \int_value:w \int_eval:n {#2} ;
  \}
\}
\cs_new:Npn \__fp_randint:ww #1; #2;
\{
  \if_int_compare:w #1 > #2 \exp_stop_f:
    \__kernel_msg-expandable_error:nnnn
    \{ kernel \} \{ randint-backward-range \} \#1 \#2
  \else:
    \if_int_compare:w \__fp_int_eval:w #2
      \if_int_compare:w #1 > \c_zero_int
        - #1 < \__fp_int_eval:w
      \else:
        \__fp_randint:nn #1 \#2
      \fi:
      \__kernel_randint:n
      \{ \__fp_int_eval:w #2 - #1 + 1 \__fp_int_eval_end: \}
    \else:
      \__kernel_randint:nn \#1 \#2
    \fi:
  \fi:
\}
(End definition for \int_rand:nn and \__fp_randint:ww. This function is documented on page 98.)

\__kernel_randint:nn
\__fp_randint_split_o:Nw
\__fp_randint_split_aux:w
\__fp_randinat_wide_aux:w
\__fp_randinat_wide_auxii:w

Any \( n \in [-2^{31}+1, 2^{31} - 1] \) is uniquely written as \( 2^{14}n_1 + n_2 \) with \( n_1 \in [-2^{17}, 2^{17} - 1] \) and \( n_2 \in [0, 2^{14} - 1] \). Calling \__fp_randint_split_o:Nw \( n \); gives \( n_1; n_2; \) and expands the next token once. We do this for two random numbers and apply \__fp_randint_split_o:Nw twice to fully decompose the range \( R \). One subtlety is that we compute \( R - 2^{31} = (\max) - (\min) - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1] \) rather than \( R \) to avoid overflow.

Then we have \__fp_randint_wide_aux:w \( \langle X_1 \rangle; \langle X_0 \rangle; \langle Y_1 \rangle; \langle Y_0 \rangle; \langle R_2 \rangle; \langle R_1 \rangle; \langle R_0 \rangle; \), and we apply the algorithm described earlier.

\cs_new:Npn \__kernel_randint:nn #1#2
\{
  \exp_after:wN \__fp_randint_wide_aux:w
  \int_value:w
  \exp_after:wN \__fp_randint_split_o:Nw
  \tex_uniformdeviate:D 268435456 ;
  \int_value:w
  \exp_after:wN \__fp_randint_split_o:Nw
  \tex_uniformdeviate:D 268435456 ;
  \int_value:w
  \exp_after:wN \__fp_randint_wide_aux:w
  \int_value:w
\}
927
(End definition for \_\_kernel_randint:nn and others.)
\int_rand:n \__fp_randint:n

Similar to \int_rand:nn, but needs fewer checks.

\cs_new:Npn \int_rand:n #1
\{ \int_eval:n \{ \exp_args:Nf \__fp_randint:n \{ \int_eval:n \{ #1 \} \} \}
\}
\cs_new:Npn \__fp_randint:n #1
\{
\if_int_compare:w #1 < 1 \exp_stop_f:
\__kernel_msg_expandable_error:nnnn
\{ kernel \} \{ randint-backward-range \} \{ 1 \} \{ #1 \}
\__fp_randint:ww #1; 1;
\else:
\if_int_compare:w #1 > \c__kernel_randint_max_int
\__kernel_randint:nn \{ 1 \} \{ #1 \}
\else:
\__kernel_randint:n \{ #1 \}
\fi:
\fi:
\}

(End definition for \int_rand:n and \__fp_randint:n. This function is documented on page 98.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

⟨/initex | package⟩

37 l3fparray implementation

⟨/initex | package⟩

\@@=fp

In analogy to l3intarray it would make sense to have \@@=fparray, but we need direct access to \__fp_parse:n from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

37.1 Allocating arrays

There are somewhat more than \((2^{31}-1)^2\) floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

\g__fp_array_int

Used to generate unique names for the three integer arrays.

\int_new:N \g__fp_array_int

(End definition for \g__fp_array_int.)

\l__fp_array_loop_int

Used to loop in \__fp_array_gzero:N.

\int_new:N \l__fp_array_loop_int

(End definition for \l__fp_array_loop_int.)
\fparray_new:Nn Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely \c_zero_fp, as we want.

\cs_new_protected:Npn \fparray_new:Nn #1#2
\tl_new:N #1
\prg_replicate:nn { 3 }
\int_gincr:N \g__fp_array_int
\exp_args:NNc \tl_gput_right:Nn #1
\exp_last_unbraced:Nfo \__fp_array_new:nNNN
{ \int_eval:n {#2} } #1 #1
\cs_generate_variant:Nn \fparray_new:Nn { c }
\cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
\int_compare:nNnTF {#1} < 0
{ \__kernel_msg_error:nnn { kernel } { negative-array-size } {#1}
\cs_undefine:N #1
\int_gsub:Nn \g__fp_array_int { 3 } }
\intarray_new:Nn #2 {#1}
\intarray_new:Nn #3 {#1}
\intarray_new:Nn #4 {#1}
\exp_last_unbraced:Nfo \__fp_array_new:nNNNN
{ \int_eval:n {#2} } #1 #1
\cs_generate_variant:Nn \fparray_new:Nn { c }
(End definition for \fparray_new:Nn and \__fp_array_new:nNNN. This function is documented on page 219.)

\fparray_count:N Size of any of the intarrays, here we pick the third.
\fparray_count:c
\cs_new:Npn \fparray_count:N #1
\exp_after:wN \use_i:nnn
\exp_after:wN \intarray_count:N #1
\cs_generate_variant:Nn \fparray_count:N { c }
(End definition for \fparray_count:N. This function is documented on page 219.)

37.2 Array items

See the \l3intarray analogue: only names change. The functions \fparray_gset:Nnn and \fparray_item:Nn share bounds checking. The T branch is used if #3 is within bounds of the array #2.

\fparray_gset:Nnn \fparray_item:Nn
\__fp_array_bounds:NnTF \__fp_array_bounds_error:Nn
\cs_new:Npn \fparray_gset:Nnn #1#2#3
\if_int_compare:w 1 > #3 \exp_stop_f:
\__fp_array_bounds_error:Nn \fparray_gset:Nnn #1 #2 (#3)
\else:
\__kernel_msg_error:nnn { kernel } { out-of-bounds } {#3}
\fi:
\intarray_gset:NNn #1 #2 #3
\intarray_item:NN #1 #2
\exp_last_unbraced:Nfo \__fp_array_bounds_error:Nn
{ \int_eval:n {#3} } #1 #2 #3
(End definition for \fparray_gset:Nnn and \fparray_item:Nn. These functions are documented on pages 219.)
\begin{verbatim}
\_fp_array_gset:Nnn
\_fp_array_gset:cnn
\__fp_array_gset:NNNNww
\__fp_array_gset_recover:Nw
\__fp_array_gset_special:nnNNN
\__fp_array_gset_normal:w

\end{verbatim}

Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

\begin{verbatim}
\cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
\ks_after:wN \ks_after:wN \exp_after:wN \__fp_array_gset:NNNNww
\exp_after:wN #1 \int_value:w \int_eval:n {#2} \exp_after:wN ;
\exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
\end{verbatim}

(End definition for \_fp_array_gset:NNnTF and \_fp_array_bounds_error:NNn.)
\s__fp \_fp_chk:w #1 #2
\s__fp \_fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
{ \_kernel_intarray_gset:Nnn #7 {#6} \{#2\}
 \_kernel_intarray_gset:Nnn #8 {#6}
 { [if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 } \_kernel_intarray_gset:Nnn #9 {#6} \{ 1 \use:nn #5 \}
\cs_new_protected:Npn \__fp_array_gset_normal:w \__fp_array_gset_special:nnNNN #1#2#3#4#5
{ \_kernel_intarray_gset:Nnn #3 {#2} {#1}
 \_kernel_intarray_gset:Nnn #4 {#2} {0}
 \_kernel_intarray_gset:Nnn #5 {#2} {0}
}
\cs_new_protected:Npn \fparray_gzero:N \fparray_gzero:c
{ \int_zero:N \l__fp_array_loop_int \prg_replicate:nn { \fparray_count:N #1 } { \int_incr:N \l__fp_array_loop_int \exp_after:wN \__fp_array_gset_special:nnNNN \exp_after:wN 0 \exp_after:wN \l__fp_array_loop_int #1 }
}
\cs_generate_variant:Nn \fparray_gzero:N { c }
\cs_new:Npn \fparray_item:Nn \fparray_item:cn \fparray_item_to_tl:Nn \fparray_item_to_tl:cn \__fp_array_item:NwN \__fp_array_item:NNNnN \__fp_array_item:N \__fp_array_item:w \__fp_array_item_special:w \__fp_array_item_normal:w
\fparray_gzero:N \fparray_gzero:c
(End definition for \fparray_gzero:N and others. This function is documented on page 219.)
\fparray_item:Nn \fparray_item:cn \fparray_item_to_tl:Nn \fparray_item_to_tl:cn
\__fp_array_item:NwN \__fp_array_item:NNNnN \__fp_array_item:N \__fp_array_item:w \__fp_array_item_special:w \__fp_array_item_normal:w
(End definition for \fparray_gzero:N. This function is documented on page 219.)
\begin{verbatim}
\_fp_array_bounds:NNnTF \_kernel_msg_expandable_error:nnfff #1 \{#2
\{ \exp_after:wN \_fp_array_item:NNNnN #1 {#2} #3
\}
\exp_after:wN \_fp_array_item:N
\int_value:w \_kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
\int_value:w \_kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
\int_value:w \_kernel_intarray_item:Nn #1 {#4} ;
\}
\cs_new:Npn \_fp_array_item:NNNnN #1#2#3#4
\{ \exp_after:wN \_fp_array_item:N #1
\if_meaning:w 0 #1 \exp_after:wN \_fp_array_item_special:w \fi:
\_fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
\}
\cs_new:Npn \_fp_array_item:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
\{ #9 \_fp \_fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; \}
\cs_new:Npn \_fp_array_item_special:w #1 ; #2 ; #3 ; #4
\{ \exp_after:wN #4
\exp:w \exp_end_continue_f:w
\if_case:w #3 \exp_stop_f:
\exp_after:wN \_c_zero_fp
\or: \exp_after:wN \_c_nan_fp
\or: \exp_after:wN \_c_minus_zero_fp
\or: \exp_after:wN \_c_inf_fp
\else: \exp_after:wN \_c_minus_inf_fp
\fi:
\}
\cs_new:Npn \_fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
\{ #9 \_fp \_fp_chk:w 1 #1 {#8} {#7} {#2#3#4#5} {#6} ; \}
\end{verbatim}

(End definition for \fparray_item:Nn and others. These functions are documented on page 219.)
The sequence has \texttt{\l__sort_length_int} items and is stored from \texttt{\l__sort_min_int} to \texttt{\l__sort_top_int} − 1. While reading the sequence in memory, we check that \texttt{\l__sort_top_int} remains at most \texttt{\l__sort_max_int}, precomputed by \texttt{\_sort_compute_range}. That bound is such that the merge sort only uses \texttt{\toks} registers less than \texttt{\l__sort_true_max_int}, namely those that have not been allocated for use in other code: the user’s comparison code could alter these.

Merge sort is done in several passes. In each pass, blocks of size \texttt{\l__sort_block_int} are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches $2^k$ in the last pass.

When merging two blocks, \texttt{\l__sort_begin_int} marks the lowest index in the two blocks, and \texttt{\l__sort_end_int} marks the highest index, plus 1.

When merging two blocks (whose end-points are \texttt{\beg} and \texttt{\end}), \texttt{\l__sort_A_int} starts from the high end of the low block, and decreases until reaching \texttt{\beg}. The index \texttt{\l__sort_B_int} starts from the top of the range and marks the register in which a sorted item should be put. Finally, \texttt{\l__sort_C_int} points to the copy of the high block in the interval of registers starting at \texttt{\l__sort_length_int}, upwards. \texttt{\l__sort_C_int} starts from the upper limit of that range.

Internal scan marks.
38.2 Finding available \toks registers

After \_sort\_compute\_range: (defined below) determines that \toks registers between \_sort\_min\_int (included) and \_sort\_true\_max\_int (excluded) have not yet been assigned, \_sort\_shrink\_range: computes \_sort\_max\_int to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $A/2 + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^n + 1$ registers we can sort $A - 2^{n-1}$ items. We first find out a power $2^n$ such that $2^n \leq A \leq 2^n + 1$ by repeatedly halving \_sort\_block\_int, starting at $2^{15}$ or $2^{14}$ namely half the total number of registers, then we use the formulas and set \_sort\_max\_int.

\begin{verbatim}
\cs_new_protected:Npn \_sort\_shrink\_range:
\begin{verbatim}
\int_set:Nn \l__sort_A_int { \l__sort_true_max_int - \l__sort_min_int + 1 }
\int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
\_sort\_shrink\_range_loop:
\int_set:Nn \l__sort_max_int { \int_compare:nNnTF { \l__sort_block_int * 3 / 2 } > \l__sort_A_int {
\l__sort_min_int + ( \l__sort_A_int - 1 ) / 2
+ \l__sort_block_int / 4
- 1 }
{ \l__sort_true_max_int - \l__sort_block_int / 2 }
}
\\end{verbatim}
\cs_new_protected:Npn \_sort\_shrink\_range_loop:
\begin{verbatim}
\if_int_compare:w \l__sort_A_int < \l__sort_block_int \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
\exp_after:wN \_sort\_shrink\_range_loop:
\fi:
\end{verbatim}
\end{verbatim}

(End definition for \_sort\_shrink\_range: and \_sort\_shrink\_range\_loop:.)

\_sort\_compute\_range: First find out what \toks have not yet been assigned. There are many cases. In \LaTeX\, with no package, available \toks range from \count15 + 1 to \c_max_register_int included (this was not altered despite the 2015 changes). When \loctoks is defined, namely in plain \TeX, or when the package etex is loaded in \LaTeX\, redefine \_sort\_compute\_range: to use the range \count265 to \count275 - 1. The elocalloc package also defines \loctoks but uses yet another number for the upper bound, namely \e@alloc@top (minus one). We must check for \loctoks every time a sorting function is called, as etex or elocalloc could be loaded.

In \Con\TeX\ MkIV the range is from \c_syst_last_allocated_toks+1 to \c_max_register_int, and in MkII it is from \lastallocatedtoks+1 to \c_max_register_int. In all these cases, call \_sort\_shrink\_range:. The \LaTeX\ format mode is easiest: no \toks are ever allocated so available \toks range from 0 to \c_max_register_int and we precompute the result of \_sort\_shrink\_range:.

(\package)
\cs_new_protected:Npn \_\_sort_compute_range: 
{ 
\int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 } 
\int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 } 
\_\_sort_shrink_range: 
\if_meaning:w \loctoks \tex_undefined:D \else: 
\if_meaning:w \loctoks \scan_stop: \else: 
\_\_sort_redefine_compute_range: 
\_\_sort_compute_range: 
\fi: 
\fi: 
\cs_new_protected:Npn \_\_sort_redefine_compute_range: 
{ 
\cs_if_exist:cTF { ver@elocalloc.sty } 
{ 
\cs_gset_protected:Npn \_\_sort_compute_range: 
{ 
\int_set:Nn \l__sort_min_int { \tex_count:D 265 } 
\int_set_eq:NN \l__sort_true_max_int \e@alloc@top 
\_\_sort_shrink_range: 
} 
} 
\cs_gset_protected:Npn \_\_sort_compute_range: 
{ 
\int_set:Nn \l__sort_min_int { \tex_count:D 275 } 
\int_set:Nn \l__sort_true_max_int { \tex_count:D 275 } 
\_\_sort_shrink_range: 
} 
\cs_if_exist:NT \loctoks { \_\_sort_redefine_compute_range: } 
\tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks } 
\cs_if_exist:NT \#1 
\cs_gset_protected:Npn \_\_sort_compute_range: 
{ 
\int_set:Nn \l__sort_min_int { \#1 + 1 } 
\int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 } 
\_\_sort_shrink_range: 
} 
\cs_if_exist:NT \c__sort_max_length_int 
\int_const:Nn \c__sort_max_length_int 
{ ( \c_max_register_int + 1 ) * 3 / 4 } 
\cs_new_protected:Npn \_\_sort_compute_range: 
{ 
\int_set:Nn \l__sort_min_int { 0 } 
\int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 } 
\int_set:Nn \l__sort_max_int { \c__sort_max_length_int }
38.3 Protected user commands

Sorting happens in three steps. First store items in \texttt{toks} registers ranging from $\l_{\text{sort min int}}$ to $\l_{\text{sort top int}} - 1$, while checking that the list is not too long. If we reach the maximum length, that’s an error; exit the group. Secondly, sort the array of \texttt{toks} registers, using the user-defined sorting function: \texttt{__sort_level}; calls \texttt{__sort_compare} as needed. Finally, unpack the \texttt{toks} registers (now sorted) into the target \texttt{tl}, or into $\g_{\text{sort internal seq}}$ for seq and clist. This is done by \texttt{__sort_seq:NNNn} and \texttt{__sort_tl:NNn}.

Call the main sorting function then unpack \texttt{toks} registers outside the group into the target token list. The unpacking is done by \texttt{__sort_tl_toks:w}; registers are numbered from $\l_{\text{sort min int}}$ to $\l_{\text{sort top int}} - 1$. For expansion behaviour we need a couple of primitives. The \texttt{tl_gclear:N} reduces memory usage. The \texttt{prg_break_point} is used by \texttt{__sort_main:NNNn} when the list is too long.
Use the same general framework for seq and clist. Apply the general sorting code, then unpack `toks` into `\_sort_internal_seq`. Outside the group copy or convert (for clist) the data to the target variable. The `seq_gclear:N` reduces memory usage. The `prg_break_point:` is used by `\_sort_main:NNNn` when the list is too long.

```
\seq_sort:Nn \seq_gsort:Nn \seq_sort:cn \seq_gsort:cn \clist_sort:Nn \clist_gsort:Nn \__sort_seq:NNNNn
```

(End definition for \_sort:Nn and others. These functions are documented on page 79.)
38.4 Merge sort

\_\_sort\_level: This function is called once blocks of size \l\\_\_sort\_block\_int (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

\__sort\_merge\_blocks: This function is called to merge a pair of blocks, starting at the last value of \l\\_\_sort\_end\_int (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in A, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift end upwards by one more block, but keeping it \leq top. Copy this upper block of \toks registers in registers above length, indexed by C: this is covered by \_\_sort\_copy\_block:. Once this is done we are ready to do the actual merger using \_\_sort\_merge\_blocks\_aux:, after shifting A, B and C so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

(End definition for \_\_sort\_level:)

(End definition for \_\_sort\_merge\_blocks:)

939
We wish to store a copy of the “upper” block of \toks registers, ranging between the initial value of \l__sort_B_int (included) and \l__sort_end_int (excluded) into a new range starting at the initial value of \l__sort_C_int, namely \l__sort_top_int.

\cs_new_protected:Npn \__sort_copy_block: \__sort_copy_block: {
\tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
\int_incr:N \l__sort_C_int
\int_incr:N \l__sort_B_int
\if_int_compare:w \l__sort_B_int = \l__sort_end_int
\use_i:nn
\fi:
\__sort_copy_block:
}

(End definition for \__sort_copy_block:.)

\__sort_merge_blocks_aux:
At this stage, the first block starts at \l__sort_begin_int, and ends at \l__sort_A_int, and the second block starts at \l__sort_top_int and ends at \l__sort_C_int. The result of the merger is stored at positions indexed by \l__sort_A_int, which starts at \l__sort_end_int − 1 and decreases down to \l__sort_begin_int, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either \texttt{swapped} or \texttt{same}. Of course, this means the arguments need to be given in the order they appear originally in the list.

\cs_new_protected:Npn \__sort_merge_blocks_aux: \__sort_merge_blocks_aux: {
\exp_after:wN \__sort_compare:nn \exp_after:wN
\{ \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN \}
\exp_after:wN \{ \tex_the:D \tex_toks:D \l__sort_C_int \}
\prg_do_nothing:
\__sort_return_mark:w
\__sort_return_two_error:
\__sort_return_none_error:
}

(End definition for \__sort_merge_blocks_aux:.)

\sort_return_same:
\sort_return_swapped:
\__sort_return_mark:w
\__sort_return_two_error:
\__sort_return_none_error:

Each comparison should call \texttt{\sort_return_same:} or \texttt{\sort_return_swapped:} exactly once. If neither is called, \texttt{\__sort_return_none_error:} is called, since the return mark removes tokens until \s__sort_mark. If one is called, the return mark auxiliary removes everything except \texttt{\__sort_return_same:w} (or its swapped analogue) followed by \texttt{\__sort_return_none_error:}. Finally if two or more are called, \texttt{\__sort_return_two_error:} ends up before any \texttt{\__sort_return_mark:w}, so that it produces an error.
\cs_new_protected:Npn \sort_return_swapped: #1 \__sort_return_mark:w #2 \s__sort_mark
\end{message}
\end{message}
\end{message}
\end{message}

\iftoks:D \l__sort_B_int \iftoks:D \l__sort_C_int
\int_decr:N \l__sort_B_int
\int_decr:N \l__sort_C_int
\int_compare:w \l__sort_C_int < \l__sort_top_int
\use_i:nn
\fi:
\__sort_merge_blocks_aux:
\end{message}
\end{message}

\iftoks:D \l__sort_A_int \iftoks:D \l__sort_C_int
\int_compare:w \l__sort_C_int < \l__sort_top_int
\use_i:nn
\fi:
\__sort_merge_blocks_aux:
\end{message}
\end{message}

\iftoks:D \l__sort_A_int \iftoks:D \l__sort_C_int
\int_compare:w \l__sort_C_int < \l__sort_top_int
\use_i:nn
\fi:
\__sort_merge_blocks_aux:
\end{message}
\end{message}

(End definition for \sort_return_same: and others. These functions are documented on page 220.)

\__sort_return_same:w
If the comparison function returns same, then the second argument fed to \__sort_compare:nn should remain to the right of the other one. Since we build the merger starting from the right, we copy that \toks register into the allotted range, then shift the pointers \textsf{B} and \textsf{C}, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

\cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
\cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:

(End definition for \__sort_return_same:w.)

\__sort_return_swapped:w
If the comparison function returns swapped, then the next item to add to the merger is the first argument, contents of the \textsf{toks} register \textsf{A}. Then shift the pointers \textsf{A} and \textsf{B} to the left, and go for one more step for the merger, unless the left block was exhausted (\textsf{A} goes below the threshold). In that case, all remaining \textsf{toks} registers in the second block, indexed by \textsf{C}, are copied to the merger by \__sort_merge_blocks_end:.

\cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
\_\_sort\_merge\_blocks\_end:

This function’s task is to copy the \toks registers in the block indexed by \textit{C} to the merger indexed by \textit{B}. The end can equally be detected by checking when \textit{B} reaches the threshold \textit{begin}, or when \textit{C} reaches \textit{top}.

\begin{verbatim}
\cs_new_protected:Npn \_\_sort\_merge\_blocks\_end: {
    \\text_toks:D \l__sort\_B\_int \\text_toks:D \l__sort\_A\_int
    \\int_decr:N \l__sort\_B\_int
    \\int_decr:N \l__sort\_A\_int
    \if_int_compare:w \l__sort\_A\_int < \l__sort\_begin\_int
        \use_i:nn
    \fi:
    \_\_sort\_merge\_blocks\_end:
}\end{verbatim}

(End definition for \_\_sort\_merge\_blocks\_end.)

38.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument \textit{#4} of \texttt{\_\_sort:nnNnn}). The arguments of \texttt{\_\_sort:nnNnn} are 1. items less than \textit{#4}, 2. items greater or equal to \textit{#4}, 3. comparison, 4. pivot, 5. next item to test. If \textit{#5} is the tail of the list, call \texttt{\tl\_sort:nN} on \textit{#1} and on \textit{#2}, placing \textit{#4} in between; \texttt{\use:ff} expands the parts to make \texttt{\tl\_sort:nN} \(f\)-expandable. Otherwise, compare \textit{#4} and \textit{#5} using \textit{#3}. If they are ordered, place \textit{#5} amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

\begin{verbatim}
\cs_new:Npn \tl\_sort:nN #1#2
{\tl_if_blank:nF {#1}
{\_\_sort:nnNnn { } { } #2
    \_\_sort\_recursion\_tail \_\_sort\_recursion\_stop
}}}
\cs_new:Npn \_\_sort:nnNnn #1#2#3#4#5
{\quark_if_recursion_tail_stop_do:nn {#5}
\_\_sort\_merge\_blocks\_aux:
}\end{verbatim}

942
There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, \_\_sort:nnNn is called $O(n \log n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to \_\_sort:nnNn. For this, the list is prepared by changing each ⟨item⟩ of the original token list into ⟨command⟩ {⟨item⟩}, just like sequences are stored. We arrange things such that the ⟨command⟩ is the ⟨conditional⟩ provided by the user: the loop over the ⟨prepared tokens⟩ then looks like

\[\begin{align*}
\text{\_\_sort_loop:wNn ... #6#7} \\
\text{#6 \{⟨pivot⟩\} \{#7\} (loop big) (loop small)} \\
\text{\{extra arguments\}} \\
\text{\_\_sort_loop:wNn ... ⟨prepared tokens⟩} \\
\text{\{end-loop\} \{\} \s__sort_stop}
\end{align*}\]

In this example, which matches the structure of \_\_sort_quick_split_i:nnnnNn and a few other functions below, the \_\_sort_loop:wNn auxiliary normally receives the user’s ⟨conditional⟩ as #6 and an ⟨item⟩ as #7. This is compared to the ⟨pivot⟩ (the argument #5, not shown here), and the ⟨conditional⟩ leaves the ⟨loop big⟩ or ⟨loop small⟩ auxiliary, which both have the same form as \_\_sort_loop:wNn, receiving the next pair ⟨conditional⟩ ⟨⟨item⟩⟩ as #6 and #7. At the end, #6 is the ⟨end-loop⟩ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the true and false branches of the conditional. For this, we introduce two versions of \_\_sort:nnNn, which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

\[\begin{align*}
\text{\_\_sort_i:nnnnNn #1#2#3#4#5#6} \\
\text{#5 \{#4\} \{#6\} \_\_sort_ii:nnnnNn \_\_sort_i:nnnnNn} \\
\text{#6 \{ #2 \{#1\} \} \{#3\} \{#4\}}
\end{align*}\]

\[\begin{align*}
\text{\_\_sort_i:nnnnNn #1#2#3#4#5#6} \\
\text{#5 \{#4\} \{#6\} \_\_sort_ii:nnnnNn \_\_sort_i:nnnnNn} \\
\text{#6 \{ #2 \{ #3 \{#1\} \} \} \{#4\}}
\end{align*}\]

Note that the two functions have the form of \_\_sort_loop:wNn above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs ⟨conditional⟩ ⟨⟨item⟩⟩, so we have to replace #6 above by { #5 #6 }, and
The third speed up is to avoid \use:ff using a continuation-passing style: \_\_sort_quick_split:NnNn expects a list followed by \s__sort_mark \{\langle \text{code} \rangle \}, and expands to \langle \text{code} \rangle \{\text{sorted list} \}. Sorting the two parts of the list around the pivot is done with
\begin{verbatim}
\_\_sort_quick_split:NnNn #2 ... \s__sort_mark
\{
  \_\_sort_quick_split:NnNn #1 ... \s__sort_mark \{\langle \text{code} \rangle \}
  \{\langle \text{pivot} \rangle \}
\}
\end{verbatim}

Items which are larger than the \langle \text{pivot} \rangle are sorted, then placed after code that sorts the smaller items, and after the (braced) \langle \text{pivot} \rangle.

The fourth speed up is avoid the recursive call to \tl_sort:nN with an empty first argument. For this, we introduce functions similar to the \_\_sort_i:nnnnNn of the last example, but aware of whether the list of \langle \text{conditional} \rangle \{\langle \text{item} \rangle \} read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see \_\_sort_quick_split:NnNn and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the \langle \text{end-loop} \rangle function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the \langle \text{end-loop} \rangle function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when \TeX{} encounters
\begin{verbatim}
\use:n \{ \use:n \{ \use:n \{ \ldots \} \ldots \} \ldots \}
\end{verbatim}

the argument of the first \use:n is not completely read by the second \use:n, hence must remain in memory; then the argument of the second \use:n is not completely read when grabbing the argument of the third \use:n, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested \use:n. In practice, this means that we must read everything until a trailing \s__sort_stop once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical \TeX{}’s memory.

The code within the \exp_not:f sorts the list, leaving in most cases a leading \exp_-not:f, which stops the expansion, letting the result be return within \exp_not:n. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list \#1 by inserting the conditional \#2 before each item. The prepare auxiliary receives the conditional as \#1, the prepared token list so far as \#2, the next prepared item as \#3, and the item after that as \#4. The loop ends when \#4 contains \prg_break_point; then the prepare_end auxiliary finds the prepared token list as \#4. The scene is then set up for \_\_sort_quick_split:NnNn, which sorts the prepared list and perform the post action placed after \s__sort_mark, namely removing the trailing \s__sort_stop and \s__sort_stop and leaving \exp_stop_f: to stop f-expansion.
The \texttt{\_sort\_quick\_prepare}, \texttt{\_sort\_quick\_prepare\_end}, \texttt{\_sort\_quick\_split}, and \texttt{\_sort\_quick\_cleanUp} auxiliaries receive a useless first argument, the new item \#2 (that they append to either one of the next two arguments), the list \#3 of items less than the pivot, bigger items \#4, the pivot \#5, a \langle function\rangle \#6, and an item \#7. The \langle function\rangle is the user’s \langle conditional\rangle except at the end of the list where it is \texttt{\_sort\_quick\_end}. The comparison is applied to the \langle pivot\rangle and the \langle item\rangle, and calls the \texttt{\_sort\_quick\_split} auxiliaries if the \langle item\rangle is smaller, and the \texttt{\_sort\_quick\_single\_end} auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument \#2 left for the next call has the form \langle conditional\rangle \{\langle item\rangle\}, so that the lists \#3 and \#4 keep the right form to be fed to the next sorting function. The \texttt{\_sort\_split} auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user’s \langle conditional\rangle rather than an ending function.

\begin{verbatim}
\cs_new:Npn \_sort_quick_split:NnNn { \_sort_quick_only_i:NnnnnNn \_sort_quick_only_ii:NnnnnNn \_sort_quick_split_i:NnnnnNn \_sort_quick_split_ii:NnnnnNn } { \_sort_quick_only_i:NnnnnNn \_sort_quick_single_end:nnnnw { \_sort_quick_split_i:NnnnnNn \_sort_quick_split_ii:NnnnnNn \_sort_quick_only_i:NnnnnNn \_sort_quick_single_end:nnnnw { \_sort_quick_only_ii:NnnnnNn } } } {#2}
\end{verbatim}
The \verb|\_sort_quick_end| appears instead of the user’s conditional, and receives as its arguments the pivot \#1, a fake item \#2, a \texttt{true} and a \texttt{false} branches \#3 and \#4, followed by an ending function \#5 (one of the four auxiliaries here) and another copy \#6 of the fake item. All these are discarded except the function \#5. This function receives lists \#1 and \#2 of items less than or greater than the pivot \#3, then a continuation code \#5 just after \verb|\s__sort_mark|. To avoid a memory problem described earlier, all of the ending functions read \#6 until \verb|\s__sort_stop| and place \#6 back into the input stream. When the lists \#1 and \#2 are empty, the single auxiliary simply places the continuation \#5 before the pivot \{#3}. When \#2 is empty, \#1 is sorted and placed before the pivot \{#3}, taking care to feed the continuation \#5 as a continuation for the function sorting \#1. When \#1 is empty, \#2 is sorted, and the continuation argument is used to place the continuation \#5 and the pivot \{#3\} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.
38.6 Messages

\__sort_error: Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with \__sort_-_level: jumping to the break point. This error recovery won’t work in a group.

\__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or similar commands in their comparison code: the \toks registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no \toks allocator.

(End definition for \__sort_quick_end:nnTNN and others.)
When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function \#1.

\begin{verbatim}
\cs_new_protected:Npn \__sort_too_long_error:NNw \#1#2 \fi:
  \fi:
  \__kernel_msg_error:nnxxx { kernel } { too-large }
  { \token_to_str:N \l__sort_true_max_int - \l__sort_min_int }
  { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
  \#1 \__sort_error:

\__kernel_msg_new:nnnn { kernel } { too-large }
{ \token_to_str:N #2 }
{ \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
{ \int_eval:n { \l__sort_top_int - \l__sort_min_int } }

\#1 \__sort_error:
\end{verbatim}

(End definition for \__sort_too_long_error:NNw.)

\begin{verbatim}
\__kernel_msg_new:nnnn { kernel } { return-none }
{ The-comparison-code-did-not-return. }
{ When-sorting-a-list,-the-code-to-compare-items-#1-and-#2-did-not-call- }
\iow_char:N\sort_return_same: -nor-
\iow_char:N\sort_return_swapped: -.
Exactly-one-of-these-should-be-called.

\__kernel_msg_new:nnnn { kernel } { return-two }
{ The-comparison-code-returned-multiple-times. }
{ When-sorting-a-list,-the-code-to-compare-items-#1-and-#2-called- }
\iow_char:N\sort_return_same: -or-
\iow_char:N\sort_return_swapped: -multiple-times.-
Exactly-one-of-these-should-be-called.
\end{verbatim}

(End definition for \__sort_too_long_error:NNw.)

\section{l3tl-analysis implementation}

\subsection{Internal functions}

The format used to store token lists internally uses the scan mark \s\_tl as a delimiter.

(End definition for \s\_tl.)

\subsection{Internal format}

The task of the l3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code,
character code), as well as reconstruct the token list quickly. This internal format is used in \l3regex where we need to support arbitrary tokens, and it is used in conversion functions in \l3str-convert, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any \langle token \rangle (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find \langle tokens \rangle which both \texttt{o-expand} and \texttt{x-expand} to the given \langle token \rangle. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

\begin{verbatim}
\langle tokens \rangle \texttt{s__tl } \langle \texttt{catcode} \rangle \langle \texttt{char code} \rangle \texttt{s__tl}
\end{verbatim}

The \langle tokens \rangle \texttt{o-} and \texttt{x-expand} to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for \l3str-convert). The \langle catcode \rangle is given as a single hexadecimal digit, 0 for control sequences. The \langle char code \rangle is given as a decimal number, −1 for control sequences.

Using delimited arguments lets us build the \langle tokens \rangle progressively when doing an encoding conversion in \l3str-convert. On the other hand, the delimiter \texttt{s__tl} may not appear unbraced in \langle tokens \rangle. This is not a problem because we are careful to wrap control sequences in braces (as an argument to \exp_not:n) when converting from a general token list to the internal format.

The current rule for converting a \langle token \rangle to a balanced set of \langle tokens \rangle which both \texttt{o-expands} and \texttt{x-expands} to it is the following.

\begin{itemize}
  \item A control sequence \texttt{\cs} becomes \exp_not:n { \texttt{\cs} } \texttt{s__tl} 0 −1 \texttt{s__tl}.
  \item A begin-group character \texttt{\{} becomes \exp_after:wN \texttt{\{ \false: \} \fi: } \texttt{s__tl} \texttt{1 } \langle \texttt{char code} \rangle \texttt{s__tl}.
  \item An end-group character \texttt{\}} becomes \texttt{\if_false: { \fi: } } \texttt{s__tl} \texttt{2 } \langle \texttt{char code} \rangle \texttt{s__tl}.
  \item A character with any other category code becomes \exp_not:n \langle \texttt{character} \rangle \texttt{s__tl} \texttt{\hex \texttt{catcode} } \langle \texttt{char code} \rangle \texttt{s__tl}.
\end{itemize}

The scan mark \texttt{s__tl} is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare \texttt{\int_value:w '#1 \texttt{s__tl} with \texttt{\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark} to extract a character code followed by the delimiter in an \texttt{x-expansion}.

\texttt{(End definition for \texttt{s__tl}.)}

The tokens in the token list are probed with the \TeX{} primitive \texttt{\futurelet}. We use \texttt{l__tl_analysis_token} in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is \texttt{l__tl_analysis_char_token}.

\texttt{\cs_new_eq:NN \texttt{l__tl_analysis_token} \texttt{?}}
\texttt{\cs_new_eq:NN \texttt{l__tl_analysis_char_token} \texttt{?}}
\_tl_analysis_normal_int \ The number of normal (N-type argument) tokens since the last special token.
\int_new:N \_tl_analysis_normal_int

\_tl_analysis_index_int \ During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.
\int_new:N \_tl_analysis_index_int

\_tl_analysis_nesting_int \ Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.
\int_new:N \_tl_analysis_nesting_int

\_tl_analysis_type_int \ When encountering special characters, we record their “type” in this integer.
\int_new:N \_tl_analysis_type_int

\g___tl_analysis_result_tl \ The result of the conversion is stored in this token list, with a succession of items of the form
\begin{verbatim}
  \langle tokens \rangle \langle catcode \rangle \langle char code \rangle \langle s__tl \rangle
\end{verbatim}
\tl_new:N \g___tl_analysis_result_tl

\__tl_analysis_extract_charcode: \ Extracting the character code from the meaning of \_tl_analysis_token. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form \texttt{’\langle char \rangle’}.
\cs_new:Npn \__tl_analysis_extract_charcode: #1 ~ #2 ~ { ’ }
\exp_after:wN \__tl_analysis_extract_charcode_aux:w
\token_to_meaning:N \_tl_analysis_token

\__tl_analysis_cs_space_count:w \ Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.
\cs_new:Npn \__tl_analysis_cs_space_count:NN #1 #2
\exp_after:wN \__tl_analysis_cs_space_count:w
\token_to_str:N #2
950
39.4 Plan of attack

Our goal is to produce a token list of the form roughly

\langle token 1 \rangle \s@__ \langle \text{catcode 1} \rangle \langle \text{char code 1} \rangle \s@__
\langle token 2 \rangle \s__tl \langle \text{catcode 2} \rangle \langle \text{char code 2} \rangle \s__tl
... \langle token N \rangle \s__tl \langle \text{catcode N} \rangle \langle \text{char code N} \rangle \s__tl

Most but not all tokens can be grabbed as an undelimited (N-type) argument by \TeX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various \toks registers. In the second pass, which is done within an x-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the \toks registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its \meaning, and what it looks like for \TeX when it is in scanning mode (e.g., when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with \futurelet, then distinguishing true characters from control sequences set equal to them using the \string representation.

The second pass is a simple exercise in expandable loops.
Everything is done within a group, and all definitions are local. We use \texttt{\_\_tl_analysis:n} to avoid problems in case \texttt{\_\_tl_analysis:n} is used within an alignment and its argument contains alignment tab tokens.

\begin{verbatim}
\cs_new_protected:Npn \_\_tl_analysis:n #1
{ \group_begin: \group_align_safe_begin: \_\_tl_analysis:a:n {#1} \_\_tl_analysis:b:n {#1} \group_align_safe_end: \group_end: }
\end{verbatim}

(End definition for \texttt{\_\_tl_analysis:n}.)

### 39.5 Disabling active characters

Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to \texttt{undefined}. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For \TeX{} and up\TeX{} we skip characters beyond [0, 255] because \texttt{lccode} only allows those values.

\begin{verbatim}
\char_set_catcode_active:N \^^@ \cs_new_protected:Npn \_\_tl_analysis_disable:n #1
{ \if_int_compare:w 256 > #1 \exp_stop_f:
\tex_lccode:D 0 = #1 \exp_stop_f:
\tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
\else:
\fi:
}
\bool_lazy_or:nnT{ \sys_if_engine_ptex_p: }{ \sys_if_engine_uptex_p: }
{ \cs_gset_protected:Npn \_\_tl_analysis_disable:n #1
{ \if_int_compare:w 256 > #1 \exp_stop_f:
\tex_lccode:D 0 = #1 \exp_stop_f:
\tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
\else:
\fi:
}
\end{verbatim}

(End definition for \texttt{\_\_tl_analysis_disable:n}.)

### 39.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a \texttt{toks} register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;

952
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is \futurelet. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the \string primitive to the following token, after possibly changing the escape character to ensure that a control sequence’s string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid \N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, \str_tail:nn \langle \langle token \rangle \rangle is non-empty, because the escape character is printable.

\__tl_analysis_a:n We read tokens one by one using \futurelet. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in \l__tl_analysis_nesting_int. This reaches \texttt{-1} when we read the closing brace.

\__tl_analysis_a_loop:w Read one character and check its type.

\__tl_analysis_a_type:w At this point, \l__tl_analysis_token holds the meaning of the following token. We store in \l__tl_analysis_type_int information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
The values 0, 1, −1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over \l__tl_analysis_token if it matches with one of the character tokens (hence is not a primitive conditional).

\cs_new_protected:Npn \_tl_analysis_a_type:w
\begin{verbatim}
\l__tl_analysis_type_int =
\if_meaning:w \l__tl_analysis_token \c_space_token
0\else:
\if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
1\else:
\if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
-1\else:
2\fi:\fi:
\fi:
\exp_stop_f:
\if_case:w \l__tl_analysis_type_int
\exp_after:wN \_tl_analysis_a_space:w
\or: \exp_after:wN \_tl_analysis_a_bgroup:w
\or: \exp_after:wN \_tl_analysis_a_safe:N
\else: \exp_after:wN \_tl_analysis_a_egroup:w
\fi:
\end{verbatim}

(End definition for \_tl_analysis_a_type:w.)

\_tl_analysis_a_space:w In this branch, the following token’s meaning is a blank space. Apply \string to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as \l__tl_analysis_char_token the first character of the string representation then test it in \_tl_analysis_a_space_test:w. Also, since \_tl_analysis_a_store: expects the special token to be stored in the relevant \toks register, we do that. The extra \exp_not:n is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

\cs_new_protected:Npm \_tl_analysis_a_space:w
\begin{verbatim}
\tex_afterassignment:D \_tl_analysis_a_space_test:w
\exp_after:wN \cs_set_eq:NN
\end{verbatim}
The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need \l__tl_analysis_char_token to be a separate control sequence from \l__tl_analysis_token, to compare them.
\__tl_analysis_a_store: This function is called each time we meet a special token; at this point, the \toks register \l__tl_analysis_index_int holds a token list which expands to the given special token. Also, the value of \l__tl_analysis_type_int indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the \lccode of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of \l__tl_analysis_type_int. The number of normal tokens until here and the type of special token are packed into a \skip register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).
This should be the simplest case: since the upcoming token is safe, we can simply grab it
in a second pass. If the token is a single character (including space), the \if_charcode:w
test yields true; we disable a potentially active character (that could otherwise masquer-ade as the true character in the next pass) and we count one “normal” token. On the
other hand, if the token is a control sequence, we should replace it by its string repre-
sentation for compatibility with other code branches. Instead of slowly looping through
the characters with the main code, we use the knowledge of how the second pass works:
if the control sequence name contains no space, count that token as a number of normal
tokens equal to its string length. If the control sequence contains spaces, they should
be registered as special characters by increasing \l__tl_analysis_index_int (no need
to carefully count character between each space), and all characters after the last space
should be counted in the following sequence of “normal” tokens.

\if_int_compare:w \l__tl_analysis_nesting_int = -1 \exp_stop_f:
\cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
\fi:
\__tl_analysis_a_safe:N
\__tl_analysis_a_cs:ww

(End definition for \__tl_analysis_a_store:.)

\__tl_analysis_a_safe:N
\__tl_analysis_a_cs:ww

39.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is
stored in \skip and \toks registers.
Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

\__tl_analysis_b:n \__tl_analysis_b_loop:w

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character’s string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave \exp_not:n { ⟨ token ⟩ } \s__tl \ in the input stream (after \text{-expansion}). Here, \exp_not:n is used rather than \exp_not:N because #3 could be a macro parameter character or could be \s__tl (which must be hidden behind braces in the result).

\__tl_analysis_b_normals:ww \__tl_analysis_b_normal:wwN

If the normal token we grab is a character, leave ⟨ catcode ⟩ ⟨ charcode ⟩ followed by \s__tl in the input stream, and call \__tl_analysis_b_normals:ww with its first argument decremented.
{
  \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
  \token_to_str:N D \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_catcode_other_token
  \token_to_str:N C \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
  \token_to_str:N B \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_math_toggle_token 3
  \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_alignment_token 4
  \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7
  \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_math_subscript_token 8
  \exp_not:N \else:
  \exp_not:N \if_catcode:w #1 \c_space_token
  \token_to_str:N A \exp_not:N \else:
    6
  \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
  \exp_not:N \int_value:w '#1 \s__tl
  \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
  \exp_not:N \int_value:w \int_eval:w - 1 +
}
\end{definition_for \__tl_analysis_b_char:Nww.}
\__tl_analysis_b_cs:Nww
\__tl_analysis_b_cs_test:ww
If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by \s__tl, and call \__tl_analysis_b_normals:ww with updated arguments.
\cs_new:Npn \__tl_analysis_b_cs:Nww #1
  \cs_new:Npn \__tl_analysis_b_cs:Nww #1
  { 0 -1 \s__tl
    \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
  }
\cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
  \exp_after:wN \__tl_analysis_b_normals:ww
  \int_value:w \int_eval:w
    \if_int_compare:w #1 = 0 \exp_stop_f:
      #3
    \else:
      \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
      \fi:
      \fi:
      \fi:
      \fi:
}
\end{definition_for \__tl_analysis_b_cs:Nww and \__tl_analysis_b_cs_test:ww.)
\__tl_analysis_b_special:w
\__tl_analysis_b_special_char:w
\__tl_analysis_b_special_space:w
Here, #1 is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn’t keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the \toks register: when x-expanding again,
we will get the special token. Then leave the category code in the input stream, followed
by the character code, and call \_tl_analysis_b_loop:w with the next index.

```
\group_begin:
\char_set_catcode_other:N A
\cs_new:Npn \_tl_analysis_b_special:w
\fi: \_tl_analysis_b_normal:wwN 0 ; #1 ;
\fi:
\if_int_compare:w #1 = \l__tl_analysis_index_int
\exp_after:wN \prg_break:
\fi:
\tex_the:D \tex_toks:D #1 \s__tl
\if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
\token_to_str:N A
\or: 1
\or: 1
\else: 2
\fi:
\exp_after:wN \_tl_analysis_b_special_char:wN \int_value:w
\int_eval:n { 1 + #1 } \exp_after:wN ;
\token_to_str:N 
\group_end:
\cs_new:Npn \_tl_analysis_b_special_char:wN #1 ; #2
\fi:
\if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
\exp_after:wN \_tl_analysis_b_special_char:wN \int_value:w
\else:
\exp_after:wN \_tl_analysis_b_special_space:w \int_value:w
\fi:
\cs_new_protected:Npn \tl_analysis_map_inline:nn #1
\__tl_analysis:n {#1}
\cs_new:Npn \_tl_analysis_b_special_space:w #1 ; ~
32 \s__tl
\__tl_analysis_b_loop:w #1 ;
\group_end:
\cs_new:Npn \_tl_analysis_b_special_space:w #1 ;
\_tl_analysis_b_special_char:wN \int_value:w
\tl_analysis_map_inline:nn
\_tl_analysis_map_inline:Nn
\__tl_analysis_map_inline_aux:Nn
\__tl_analysis_map_inline:nnn

(End definition for \_tl_analysis_b_special:w, \_tl_analysis_b_special_char:wN, and \_tl_analysis_b_special_space:w.)

39.8 Mapping through the analysis

First obtain the analysis of the token list into \g__tl_analysis_result_tl. To allow
nested mappings, increase the nesting depth \g__kernel_prg_map_int (shared between
all modules), then define the looping macro, which has a name specific to that nesting
depth. That looping grabs the \tokens, \catcode and \char code; it checks for the end
of the loop with \use_none:n ##2, normally empty, but which becomes \tl_map_break:
at the end; it then performs the user’s code \#2, and loops by calling itself. When the
loop ends, remember to decrease the nesting depth.

```
\cs_new_protected:Npn \tl_analysis_map_inline:nn #1
\__tl_analysis:n {#1}
```
\int_gincr:N \g__kernel_prg_map_int
\exp_args:Nc \__tl_analysis_map_inline_aux:Nn
{ __tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
\cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
{ \exp_args:No \tl_analysis_map_inline:nn #1 }
\cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
{ \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
{ \use_none:n ##2
\__tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}
}
\cs_gset_protected:Npn \__tl_analysis_map_inline_aux:nnn ##1##2##3
{ #2
#1
}
\exp_after:wN #1
\g__tl_analysis_result_tl
\s__tl { ? \tl_map_break: } \s__tl
\prg_break_point:Nn \tl_map_break:
{ \int_gdecr:N \g__kernel_prg_map_int }
\cs_new_protected:Npn \tl_analysis_show:N #1
{ \tl_if_exist:NTF #1
{ \exp_args:No \__tl_analysis:n {#1}
\msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
{ } { \__tl_analysis_show: } { } { }
}
{ \tl_show:N #1 }
}
\cs_new_protected:Npn \tl_analysis_show:n #1
{ \__tl_analysis:n {#1}
\msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
{ } { \__tl_analysis_show: } { } { }
}
\cs_gset_protected:Npn \__tl_analysis_show:wNw
Here, \texttt{#1} o- and x-expands to the token; \texttt{#2} is the category code (one uppercase hexadecimal digit), 0 for control sequences; \texttt{#3} is the character code, which we ignore. In the

39.9 Showing the results
\tl_analysis_show:N
\tl_analysis_show:n
Add to \texttt{\__tl_analysis:n} a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as \texttt{\tl_show:N} by simply calling that function.
\cs_new_protected:Npn \tl_analysis_show:N #1
{ \tl_if_exist:NTF #1
{ \exp_args:No \__tl_analysis:n {#1}
\msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
{ } { \__tl_analysis_show: } { } { }
}
{ \tl_show:N #1 }
}
\cs_new_protected:Npn \tl_analysis_show:n #1
{ \__tl_analysis:n {#1}
\msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
{ } { \__tl_analysis_show: } { } { }
}
\cs_gset_protected:Npn \__tl_analysis_show:wNw
(End definition for \texttt{\tl_analysis_map_inline:n} and others. These functions are documented on page 221.)
cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

\cs_new:Npn \__tl_analysis_show: 
{ 
\exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl \s__tl \{ ? \prg_break: \} \s__tl \prg_break_point:
}
\cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl 
{ 
\use_none:n #2 
\iow_newline: > \use:nn { - } { - }
\if_int_compare:w "#2 = 0 \exp_stop_f:
\exp_after:wN \__tl_analysis_show_cs:n
\else:
\if_int_compare:w "#2 = 13 \exp_stop_f:
\exp_after:wN \exp_after:wN \exp_after:wN \__tl_analysis_show_active:n
\else:
\exp_after:wN \exp_after:wN \exp_after:wN \__tl_analysis_show_normal:n
\fi:
\fi:
{#1}
\__tl_analysis_show_loop:wNw
}

(End definition for \__tl_analysis_show: and \__tl_analysis_show_loop:wNw.)

\__tl_analysis_show_normal:n
Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up \TeX's alignment status.

\cs_new:Npn \__tl_analysis_show_normal:n #1 
{ 
\exp_after:wN \token_to_str:N #1 ~
\exp_after:wN \token_to_meaning:N #1 )
}

(End definition for \__tl_analysis_show_normal:n.)

\__tl_analysis_show_value:N
This expands to the value of \#1 if it has any.

\cs_new:Npn \__tl_analysis_show_value:N #1 
{ 
\token_if_expandable:NF #1 
\token_if_chardef:NTF #1 \prg_break: { } 
\token_if_mathchardef:NTF #1 \prg_break: { } 
\token_if_dim_register:NTF #1 \prg_break: { } 
\token_if_int_register:NTF #1 \prg_break: { } 
\token_if_skip_register:NTF #1 \prg_break: { } 
\token_if_toks_register:NTF #1 \prg_break: { } 
\use_none:nmm 
\prg_break_point: 
\use:n { \exp_after:wN = \tex_the:D #1 }

962
Control sequences and active characters are printed in the same way, making sure not to go beyond the \_\l_iow_line_count_int. In case of an overflow, we replace the last characters by \c__tl_analysis_show_etc_str.

\cs_new:Npn \__tl_analysis_show_cs:n #1
{ \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
\cs_new:Npn \__tl_analysis_show_active:n #1
{ \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }
\cs_new:Npn \__tl_analysis_show_long:nn #1
{ \__tl_analysis_show_long_aux:oofn \token_to_str:N #1 \token_to_meaning:N #1 \__tl_analysis_show_value:N #1 }
\cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
{ \int_compare:nNnTF \str_count:n { #1 ~ ( #4 #2 #3 ) } > \l_iow_line_count_int - 3
  \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 } \c__tl_analysis_show_etc_str
  \c__tl_analysis_show_etc_str }
\cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

(End definition for \__tl_analysis_show_value:N.)

\tl_const:Nx \c__tl_analysis_show_etc_str % (\ETC.)
\__kernel_msg_new:nnn { kernel } { show-tl-analysis }
\tl_if_empty:nF {#1} { #1 ~ }
\tl_if_empty:nTF {#2} { is~empty }
\tl_if_empty:nF {#2} { contains-the-tokens: #2 }

\cs_generate_variant:Nn \__tl_analysis_show_etc_str {n and others.}

\__tl_analysis_show_etc_str

39.10 Messages

When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.
\tl_const:Nx \c__tl_analysis_show_etc_str % (\ETC.)
\__kernel_msg_new:nnn { kernel } { show-tl-analysis }
\tl_if_empty:nF {#1} { #1 ~ }
\tl_if_empty:nF {#2} { is~empty }
\tl_if_empty:nF {#2} { contains-the-tokens: #2 }

(End definition for \c__tl_analysis_show_etc_str.)
40 l3regex implementation

\begin{quote}
\texttt{\makeautorefname{initex}{package}}
\end{quote}

\begin{quote}
\texttt{\makeautorefname{@@=regex}}
\end{quote}

40.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-
references come to mind first), but it is costly, and raises the problem of catastrophic
backtracking. Since \TeX{} is not first and foremost a programming language, complicated
code tends to run slowly, and we must use faster, albeit slightly more restrictive,
techniques, coming from automata theory.

Given a regular expression of \( n \) characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal
  representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton
  (NFA) with \( O(n) \) states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a
time, exploring in parallel every possible path (“active thread”) through the NFA,
  considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- \textit{Group}: index of the capturing group, \(-1\) for non-capturing groups.
- \textit{Position}: each token in the query is labelled by an integer \( \langle position \rangle \), with
  \( \text{min\_pos} - 1 \leq \langle position \rangle \leq \text{max\_pos} \). The lowest and highest positions corre-
  spond to imaginary begin and end markers (with inaccessible category code and
  character code).
- \textit{Query}: the token list to which we apply the regular expression.
- \textit{State}: each state of the NFA is labelled by an integer \( \langle state \rangle \) with \( \text{min\_state} \leq
  \langle state \rangle < \text{max\_state} \).
- \textit{Active thread}: state of the NFA that is reached when reading the query token list for
  the matching. Those threads are ordered according to the greediness of quantifiers.
- \textit{Step}: used when matching, starts at \( 0 \), incremented every time a character is read,
  and is not reset when searching for repeated matches. The integer \texttt{\_\_regex\_step\_int}
  is a unique id for all the steps of the matching algorithm.

We use \texttt{l3intarray} to manipulate arrays of integers (stored into some dimension reg-
isters in scaled points). We also abuse \TeX{}’s \texttt{\toks} registers, by accessing them directly
by number rather than tying them to control sequence using the \texttt{\newtoks} allocation
functions. Specifically, these arrays and \texttt{\toks} are used as follows. When building,
\texttt{\toks\langle state \rangle} holds the tests and actions to perform in the \( \langle state \rangle \) of the NFA. When
matching,

- \texttt{\g__regex\_state\_active\_intarray} holds the last \( \langle step \rangle \) in which each \( \langle state \rangle \) was
  active.
• \texttt{\g_regex_thread_state_intarray} maps each ⟨thread⟩ (with \texttt{min_active} ≤ ⟨thread⟩ < \texttt{max_active}) to the ⟨state⟩ in which the ⟨thread⟩ currently is. The ⟨threads⟩ are ordered starting from the best to the least preferred.

• \texttt{\toks⟨thread⟩} holds the submatch information for the ⟨thread⟩, as the contents of a property list.

• \texttt{\g_regex_charcode_intarray} and \texttt{\g_regex_catcode_intarray} hold the character codes and category codes of tokens at each ⟨position⟩ in the query.

• \texttt{\g_regex_balance_intarray} holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

• \texttt{\toks⟨position⟩} holds ⟨tokens⟩ which o- and x-expand to the ⟨position⟩-th token in the query.

• \texttt{\g_regex_submatch_prev_intarray}, \texttt{\g_regex_submatch_begin_intarray} and \texttt{\g_regex_submatch_end_intarray} hold, for each submatch (as would be extracted by \texttt{\regex_extract_all:nnN}), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice \texttt{max_state}, and the used registers go up to \texttt{l_regex_submatch_int}. They are organized in blocks of \texttt{l_regex_capturing_group_int} entries, each block corresponding to one match with all its submatches stored in consecutive entries.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to \texttt{NFA} states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

\subsection{Helpers}

\texttt{\_regex_int_eval:w} Access the primitive: performance is key here, so we do not use the slower route \texttt{via \int_eval:n}.

\texttt{\_regex_toks_use:w} Unpack a \texttt{\toks} given its number.

\begin{macrocode}
\cs_new_eq:NN \_regex_int_eval:w \tex_numexpr:D
\cs_new:Npn \_regex_standard_escapechar: { \int_set:Nn \tex_escapechar:D { \textbackslash } }

\begin{macro}{\_regex_toks_use:w}
\cs_new:Npn \_regex_toks_use:w { \tex_the:D \tex_toks:D }
\end{macro}
\end{macrocode}

(End definition for \_regex_int_eval:w)

(End definition for \_regex_toks_use:w)
Empty a \texttt{toks} or set it to a value, given its number.

\begin{verbatim}
\cs_new_protected:Nn \__regex_toks_clear:N #1 
\{ \__regex_toks_set:Nn #1 { } \}
\cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
\cs_new_protected:Npn \__regex_toks_set:No #1 
\{ \__regex_toks_set:Nn #1 \exp_after:wN \}
\end{verbatim}

(End definition for \texttt{\__regex_toks_clear:N} and \texttt{\__regex_toks_set:Nn}.)

Copy \texttt{#3} \texttt{toks} registers from \texttt{#2} onwards to \texttt{#1} onwards, like C's memcpy.

\begin{verbatim}
\cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
\prg_replicate:nn {#3}
\{ \tex_toks:D #1 = \tex_toks:D #2
\int_incr:N #1
\int_incr:N #2\}
\end{verbatim}

(End definition for \texttt{\__regex_toks_memcpy:NNn}.)

During the building phase we wish to add x-expanded material to \texttt{toks}, either to the left or to the right. The expansion is done “by hand” for optimization (these operations are used quite a lot). The \texttt{Nn} version of \texttt{\__regex_toks_put_right:Nx} is provided because it is more efficient than x-expanding with \texttt{\exp_not:n}.

\begin{verbatim}
\cs_new_protected:Npn \__regex_toks_put_left:Nx #1#2
\cs_set:Npx \__regex_tmp:w {#2}
\tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN 
\{ \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 \}
\end{verbatim}

\begin{verbatim}
\cs_new_protected:Npn \__regex_toks_put_right:Nx #1#2
\cs_set:Npx \__regex_tmp:w {#2}
\tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN 
\{ \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w \}
\end{verbatim}

\begin{verbatim}
\cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
\tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #2 }\end{verbatim}

(End definition for \texttt{\__regex_toks_put_left:Nx} and \texttt{\__regex_toks_put_right:Nx}.)

Expands to the string representation of the token (known to be a control sequence) at the current position \texttt{l__regex_curr_pos_int}. It should only be used in x-expansion to avoid losing a leading space.

\begin{verbatim}
\cs_new:Npn \__regex_curr_cs_to_str:
\exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N 
\tex_the:D \tex_toks:D l__regex_curr_pos_int\end{verbatim}

(End definition for \texttt{\__regex_curr_cs_to_str:}.)
40.2.1 Constants and variables

\ doctr\newcommand{\__regex_tmp:w}{ }

(End definition for \__regex_tmp:w.)

\tl_new:N \l__regex_internal_a_tl
\tl_new:N \l__regex_internal_b_tl
\tl_new:N \l__regex_internal_a_int
\tl_new:N \l__regex_internal_b_int
\tl_new:N \l__regex_internal_bool
\tl_new:N \l__regex_internal_seq
\tl_new:N \l__regex_internal_int
\tl_new:N \g__regex_internal_tl

(End definition for \l__regex_internal_a_tl and others.)

\tl_new:N \l__regex_build_tl

(End definition for \l__regex_build_tl.)

\tl_const:Nn \c__regex_no_match_regex

(End definition for \c__regex_no_match_regex.)

\intarray_new:Nn \g__regex_charcode_intarray { 65536 }
\intarray_new:Nn \g__regex_catcode_intarray { 65536 }
\intarray_new:Nn \g__regex_balance_intarray { 65536 }

(End definition for \g__regex_charcode_intarray, \g__regex_catcode_intarray, and \g__regex_balance_intarray.)

\int_new:N \l__regex_balance_int

(End definition for \l__regex_balance_int.)

\tl_new:N \l__regex_cs_name_tl

(End definition for \l__regex_cs_name_tl.)
40.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int
\int_const:Nn \c__regex_ascii_min_int { 0 }
\int_const:Nn \c__regex_ascii_max_control_int { 31 }
\int_const:Nn \c__regex_ascii_max_int { 127 }

(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_ascii_max_int.)

\c__regex_ascii_lower_int
\int_const:Nn \c__regex_ascii_lower_int { 'a - 'A }

(End definition for \c__regex_ascii_lower_int.)

40.2.3 Internal auxiliaries

\q__regex_recursion_stop
Internal recursion quarks.
\quark_new:N \q__regex_recursion_stop
(End definition for \q__regex_recursion_stop.)

Functions to gobble up to a quark.
\cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
  #1 \q__regex_recursion_stop { }
\cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__regex_recursion_stop {#1}

(End definition for \__regex_use_none_delimit_by_q_recursion_stop:w and \__regex_use_i_delimit_by_q_recursion_stop:nw.)

\q__regex_nil
Internal quarks.
\quark_new:N \q__regex_nil
(End definition for \q__regex_nil.)

\__regex_quark_if_nil_p:n
\__regex_quark_if_nil:n
TF
Branching quark conditional.
\__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
(End definition for \__regex_quark_if_nil:nTF.)

\__regex_break_point:TF
\__regex_break_true:w
When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

\{test1\} ... \{test_n\}
\__regex_break_point:TF \{\langle true code\rangle\} \{\langle false code\rangle\}

If any of the tests succeeds, it calls \__regex_break_true:w, which cleans up and leaves \langle true code\rangle in the input stream. Otherwise, \__regex_break_point:TF leaves the \langle false code\rangle in the input stream.
\cs_new_protected:Npn \__regex_break_true:w
  #1 \__regex_break_point:TF #2 \#3 \#1 \#2 { \#2 }
\cs_new_protected:Npn \__regex_break_point:TF
  #1 \__regex_break_point:TF #2 \#3 { \#1 \#2 { \#2 }

(End definition for \__regex_break_point:TF and \__regex_break_true:w.)
This function makes showing regular expressions easier, and lets us define □ in terms of △ for instance. There is a subtlety: the end of the query is marked by −2, and thus matches □ and other negated properties; this case is caught by another part of the code.

```
\cs_new_protected:Npn \__regex_item_reverse:n #1
\__regex_break_point:TF { } \__regex_break_true:w
```

(End definition for `\__regex_item_reverse:n`.)

Simple comparisons triggering `\__regex_break_true:w` when true.

```
\cs_new_protected:Npn \__regex_item_caseful_equal:n #1
{\if_int_compare:w #1 = \l__regex_curr_char_int \exp_after:wN \__regex_break_true:w \fi:}
\cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
{\reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w \fi: \fi:}
```

(End definition for `\__regex_item_caseful_equal:n` and `\__regex_item_caseful_range:nn`.)

For caseless matching, we perform the test both on the current_char and on the case_changed_char. Before doing the second set of tests, we make sure that case_changed_char has been computed.

```
\cs_new_protected:Npn \__regex_item_caseless_equal:n #1
{\if_int_compare:w #1 = \l__regex_curr_char_int \exp_after:wN \__regex_break_true:w \fi:}
\cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
{\reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w \fi: \fi:}
\cs_new_protected:Npn \__regex_compute_case_changed_char:
```

969
\reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
\reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
\exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
\fi:
\fi:
\fi:
\}

(End definition for \_regex_item_caseless_equal:n and \_regex_item_caseless_range:nn.)

\_regex_compute_case_changed_char: This function is called when \_\_regex_case_changed_char_int has not yet been computed (or rather, when it is set to the marker value \c_max_int). If the current character code is in the range [65, 90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97, 122], subtract 32.

\cs_new_protected:Npn \__regex_compute_case_changed_char:
\int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
\if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
\if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
\if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
\int_sub:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
\fi:
\fi:
\else:
\if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
\int_add:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
\fi:
\fi:
\fi:
\}

(End definition for \_regex_compute_case_changed_char:.)

\_regex_item_equal:n \_regex_item_range:nn Those must always be defined to expand to a caseful (default) or caseless version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

\cs_new_eq:NN \__regex_item_equal:n ?
\cs_new_eq:NN \__regex_item_range:nn ?

(End definition for \_regex_item_equal:n and \_regex_item_range:nn.)

\_regex_item_catcode:nT \_regex_item_catcode_reverse:nT \_regex_item_catcode: The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

\cs_new_protected:Npn \__regex_item_catcode:
\{"\}
\if_case:w \l__regex_curr_catcode_int
1 \or: 4 \or: 10 \or: 40 \or: 100 \or: 1000 \or: 4000 \or: 10000 \or: 100000 \or: 400000 \or: 1000000 \or: 4000000 \or: 10000000 \else: 1*0
\exp_stop_f:
\fi:
This matches an exact (category)-(character code) pair, or an exact control sequence, more precisely one of several possible control sequences.

This matches a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches. The three \exp_after:wN expand the contents of the \toks\{current position\} (of the form \exp_not:n \{control sequence\}) to \control sequence. We store the cs name before building states for the cs, as those states may overlap with toks registers storing the user’s input.
Character property tests

\_\_regex_prop_d: Character property tests for \d, \W, etc. These character properties are not affected by the (?!i) option. The characters recognized by each one are as follows: \d=[0-9], \w=[0-9A-Z_a-z], \s=[/uni2423^^I^^J^^L^^M], \h=[/uni2423^^I], \v=[\^^J-\^^M], and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

\_\_regex_prop_h:
\_\_regex_prop_s:
\_\_regex_prop_v:
\_\_regex_prop_w:
\_\_regex_prop_N:
\_\_regex_posix_alnum: POSIX properties. No surprise.
\_\_regex_posix_alpha: \cs_new_protected:Npn \_\_regex_posix_alnum: {
\_\_regex_posix_digit: \_\_regex_posix_alpha: }
\_\_regex_posix_ascii: \cs_new_protected:Npn \_\_regex_posix_ascii: {
\_\_regex_item_caseful_range:nn \c__regex_ascii_min_int \c__regex_ascii_max_int
\_\_regex_item_caseful_equal:n \c__regex_ascii_max_int}
\_\_regex_posix_blank: \cs_new_eq:NN \_\_regex_posix_blank: \_\_regex_prop_h:
\_\_regex_posix_digit: \cs_new_protected:Npn \_\_regex_posix_digit: 
\_\_regex_posix_graph: \cs_new_protected:Npn \_\_regex_posix_graph: 
\_\_regex_posix_print: \cs_new_protected:Npn \_\_regex_posix_print: 
\_\_regex_posix_punct: \cs_new_protected:Npn \_\_regex_posix_punct: 
\_\_regex_posix_space: \cs_new_eq:NN \_\_regex_posix_space: \_\_regex_posix_upper:
\_\_regex_posix_upper: \cs_new_protected:Npn \_\_regex_posix_upper: 
\_\_regex_posix_word: \cs_new_protected:Npn \_\_regex_posix_word: 
\_\_regex_posix_xdigit: \cs_new_protected:Npn \_\_regex_posix_xdigit: {
\_\_regex_digit: \_\_regex_item_caseful_range:nn { 'A' } { 'F' }
\_\_regex_item_caseful_range:nn { 'a' } { 'f' }
}
(End definition for \_\_regex_posix_alnum: and others.)

40.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting \textbackslash n to the character 10, \textit{etc}. In this pass, we also convert any special
character (*, ?, {, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: \_\_regex_escape_use:nnnn (\inline 1) (\inline 2) (\inline 3) {\{\token list\}}

The \token list is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function \inline 1, and escaped characters are fed to the function \inline 2 within an x-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences \a, \e, \f, \n, \r, \t and \x are recognized, and those are replaced by the corresponding character, then fed to \inline 3. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an x-expanding assignment.

(End definition for \_\_regex_escape_use:nnnn.)

\_\_regex_escape_loop:N

\_\_regex_escape_loop:N reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

(End definition for \_\_regex_escape_loop:N.)
Those functions are never called before being given a new meaning, so their definitions here don’t matter.

The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

When \x is encountered, \__regex_escape_x_test:N is responsible for grabbing some hexadecimal digits, and feeding the result to \__regex_escape_x_end:w. If the number is too big interrupt the assignment and produce an error, otherwise call \__regex_escape_raw:N on the corresponding character token.
\__regex_escape_x:test:N
\__regex_escape_x_testii:N

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either \__regex_escape_x_loop:N or \__regex_escape_x:N.

\cs_new:Npn \__regex_escape_x_test:N #1
\begin{verbatim}
\str_if_eq:nnTF {#1} { break } { ; }
{ \__regex_hexadecimal_use:NTF #1
\exp_after:wN \__regex_escape_loop:N }
\else:
\exp_after:wN \__regex_escape_x_testii:N
\fi:
\end{verbatim}

\end{definition}

\__regex_escape_x:test:N
\__regex_escape_x_testii:N

This looks for the second digit in the unbraced case.

\cs_new:Npn \__regex_escape_x:N #1
\begin{verbatim}
\str_if_eq:nnTF {#1} { break } { ; }
{ \__regex_hexadecimal_use:NTF #1
\exp_after:wN \__regex_escape_loop:N }
\else:
\__regex_escape_x_loop:N \__regex_escape_loop_error:
\fi:
\end{verbatim}

\end{definition}

\__regex_escape_x:loop:N
\__regex_escape_loop_error:

Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

\cs_new:Npn \__regex_escape_x_loop:N #1
\{
  \if_eq:nTF {#1} { break }
  \}
\{
  \__regex_escape_x_loop_error:n { } {#1} }
\{
  \__regex_hexadecimal_use:NTF #1
  \{
  \__regex_escape_x_loop:N }
  \{
    \token_if_eq_charcode:NNTF \c_space_token #1
    \{
      \__regex_escape_x_loop:N }
    \{
      \__regex_escape_loop:N \c_right_brace_str #1
      \{
      \__regex_escape_x_loop_error:n {#1} }
      \}
    \}
  \}
  \}
\}
\cs_new:Npn \__regex_escape_x_loop_error:n #1
  \__kernel_msg_expandable_error:nnn { kernel } { x-missing-rbrace } {#1}
\__regex_escape_loop:N #1
\}
\}
\__regex_hexadecimal_use:NTF
\__regex_char_if_alphanumeric:NTF
\__regex_char_if_special:NTF
\TEX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.
\prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
\{ 
  \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
  #1 \prg_return_true:
  \else:
  \if_case:w
    \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
    A
  \or: B
  \or: C
  \or: D
  \or: E
  \or: F
  \else:
  \prg_return_false:
  \exp_after:wN \use_none:n
  \fi:
  \fi:
\}
\}
(End definition for \__regex_escape_x_loop:N and \__regex_escape_x_loop_error:n.)
(End definition for \__regex_hexadecimal_use:NTF.)
These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as "raw" characters. Namely,
• alphanumerics are “raw” if they are not escaped, and may have a special meaning when escaped;
• non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
• characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

\begin{verbatim}
\prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
\end{verbatim}

(End definition for \__regex_char_if_alphanumeric:NTF and \__regex_char_if_special:NTF.)
40.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- \_regex_class:NnnnN \{boolean\} \{tests\} \{min\} \{more\} \{lazyness\}
- \_regex_group:nnnN \{branches\} \{min\} \{more\} \{lazyness\}, also \_regex_group_no_capture:nnnN and \_regex_group_resetting:nnnN with the same syntax.
- \_regex_branch:n \{contents\}
- \_regex_command_K:
  - \_regex_assertion:Nn \{boolean\} \{assertion test\}
  - \_regex_b_test:
  - \_regex_group_level_int

We make sure to open the same number of groups as we close.

\int_new:N \l__regex_group_level_int
(End definition for \l__regex_group_level_int.)

\l__regex_mode_int
\c__regex_cs_in_class_mode_int
\c__regex_cs_mode_int
\c__regex_outer_mode_int
\c__regex_catcode_mode_int
\c__regex_class_mode_int
\c__regex_catcode_in_class_mode_int

While compiling, ten modes are recognized, labelled \(-63, -23, -6, -2, 0, 2, 3, 6, 23, 63\). See section 40.3.3. We only define some of these as constants.

\int_new:N \l__regex_mode_int
\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\int_const:Nn \c__regex_cs_mode_int { -2 }
\int_const:Nn \c__regex_outer_mode_int { 0 }
\int_const:Nn \c__regex_catcode_mode_int { 2 }
\int_const:Nn \c__regex_class_mode_int { 3 }
\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }

Tests can be the following:

- \_regex_item_caseful_equal:n \{char code\}
- \_regex_item_caseless_equal:n \{char code\}
- \_regex_item_caseful_range:nn \{min\} \{max\}
- \_regex_item_caseless_range:nn \{min\} \{max\}
- \_regex_item_catcode:nT \{catcode bitmap\} \{tests\}
- \_regex_item_catcode_reverse:nT \{catcode bitmap\} \{tests\}
- \_regex_item_reverse:n \{tests\}
- \_regex_item_exact:nn \{catcode\} \{char code\}
- \_regex_item_exact_cs:n \{csnames\}

more precisely given as \{csname\} \scan_stop: \{csname\} \scan_stop: \{csname\} and so on in a brace group.

- \_regex_item_cs:n \{compiled regex\}

40.3.1 Variables used when compiling

We make sure to open the same number of groups as we close.

\int_new:N \l__regex_group_level_int
(End definition for \l__regex_group_level_int.)

While compiling, ten modes are recognized, labelled \(-63, -23, -6, -2, 0, 2, 3, 6, 23, 63\). See section 40.3.3. We only define some of these as constants.
We wish to allow constructions such as \c[^BE] ( \cL[a-z] . .), where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: \l__regex_catcodes_int and \l__regex_default_catcodes_int are bitmaps, sums of $4^c$, for all allowed catcodes $c$. The latter is local to each capturing group, and we reset \l__regex_catcodes_int to that value after each character or class, changing it only when encountering a \c escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare \c[^BE] and \c[BE].

\begin{verbatim}
2429 \int_new:N \l__regex_catcodes_int
2429 \int_new:N \l__regex_default_catcodes_int
2429 \bool_new:N \l__regex_catcodes_bool
\end{verbatim}

Constants: $4^c$ for each category, and the sum of all powers of 4.

\begin{verbatim}
2429 \int_const:Nn \c__regex_catcode_C_int { "1 }
2429 \int_const:Nn \c__regex_catcode_B_int { "4 }
2429 \int_const:Nn \c__regex_catcode_E_int { "10 }
2429 \int_const:Nn \c__regex_catcode_M_int { "40 }
2429 \int_const:Nn \c__regex_catcode_T_int { "100 }
2429 \int_const:Nn \c__regex_catcode_P_int { "1000 }
2429 \int_const:Nn \c__regex_catcode_U_int { "4000 }
2429 \int_const:Nn \c__regex_catcode_D_int { "10000 }
2429 \int_const:Nn \c__regex_catcode_S_int { "100000 }
2429 \int_const:Nn \c__regex_catcode_L_int { "400000 }
2429 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
2429 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
2429 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
\end{verbatim}

\begin{verbatim}
2430 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
\end{verbatim}

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

\begin{verbatim}
2431 \seq_new:N \l__regex_show_prefix_seq
\end{verbatim}

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

\begin{verbatim}
2432 \int_new:N \l__regex_show_lines_int
\end{verbatim}
40.3.2 Generic helpers used when compiling

Used to compare pairs of things like \texttt{\_\_regex_compile_special:N} together. It’s often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behavior of \texttt{\_\_regex_compile_special:N} is very useful as that means we can use \texttt{\textbackslash c_left_brace_str} and the like.

\begin{verbatim}
\prg_new_conditional:Npnnn \_\_regex_two_if_eq:NNNN \#1\#2\#3\#4 { TF }
{ \if_meaning:w \#1 \#3 \if:w \#2 \#4 \prg_return_true: \else: \prg_return_false: \fi: \else: \prg_return_false: \fi: }
\end{verbatim}

(End definition for \texttt{\_\_regex_two_if_eq:NNNNTF}.)

\begin{verbatim}
\cs_new_protected:Npn \_\_regex_get_digits:NTFw \#1\#2\#3\#4\#5 { \if_meaning:w \__regex_compile_raw:N \#1 \if_int_compare:w 1 < 1 \#2 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi: \else: \prg_return_false: \fi: }
\end{verbatim}

(End definition for \texttt{\_\_regex_get_digits:NTFw}.)

If followed by some raw digits, collect them one by one in the integer variable \#1, and take the \texttt{true} branch. Otherwise, take the \texttt{false} branch.

\begin{verbatim}
\cs_new:Npn \_\_regex_get_digits_loop:nw \#1\#2\#3 { \if_meaning:w \__regex_if_raw_digit:NNTF \#2 \#3 { \#3 \_\_regex_get_digits_loop:nw \#1 } \scan_stop: \#1 \#2 \#3 }
\end{verbatim}

(End definition for \texttt{\_\_regex_get_digits:NTFw} and \texttt{\_\_regex_get_digits_loop:w}.)

\begin{verbatim}
\prg_new_conditional:Npn \_\_regex_if_raw_digit:NN \#1\#2 { \if_meaning:w \_\_regex_compile_raw:N \#1 \if_int_compare:w 1 < 1 \#2 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi: \else: \prg_return_false: \fi: }
\end{verbatim}

(End definition for \texttt{\_\_regex_if_raw_digit:NNTF}.)

Test used when grabbing digits for the \{m,n\} quantifier. It only accepts non-escaped digits.
40.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

-6 \[\c{...}\] control sequence in a class,
-2 \c{...} control sequence,
0 ... outer,
2 \c... catcode test,
6 \[\c...\] catcode test in a class,
-63 \[\c{[...]}\] class inside mode −6,
-23 \c{[...]} class inside mode −2,
3 [...] class inside mode 0,
23 \c{[...]} class inside mode 2,
63 \[\c{[...]}\] class inside mode 6.

This list is exhaustive, because \c escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as \( m \rightarrow (m - 15)/13 \), truncated.

- Grouping, assertion, and anchors are allowed in non-positive even modes (0, −2, −6), and do not change the mode. Otherwise, they trigger an error.

- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from \( m \) to \( (m - 15)/13 \), truncated; also, ranges are recognized.

- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from −2 to 0 or −6 to 3, with error recovery for odd modes.

- Properties (such as the \d character class) can appear in any mode.

\__regex_if_in_class:TF\__ Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

\cs_new:Npn \__regex_if_in_class:TF
{\if_int_odd:w \l__regex_mode_int
\exp_after:wN \use_i:nn
\else:
\exp_after:wN \use_ii:nn
\fi:}

982
Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
cs_new:Npn \__regex_if_in_cs:TF
\{
  \if_int_odd:w \l__regex_mode_int
  \exp_after:wN \use_i:nn
  \else:
  \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
    \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
  \fi:
  \fi:
\}
```

Assertions are only allowed in modes 0, −2, and −6, i.e., even, non-positive modes.

```
cs_new:Npn \__regex_if_in_class_or_catcode:TF
\{
  \if_int_odd:w \l__regex_mode_int
  \exp_after:wN \use_i:nn
  \else:
  \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
    \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
  \else:
    \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
  \fi:
  \fi:
\}
```

This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
cs_new:Npn \__regex_if_within_catcode:TF
\{
  \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
  \else:
    \exp_after:wN \use_i:nn
  \fi:
\}
```

The \c escape sequence is only allowed in modes 0 and 3, i.e., not within any other \c escape sequence.

```
cs_new_protected:Npn \__regex_chk_c_allowed:T
\{
  \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
```
\__regex_mode_quit_c:  This function changes the mode as it is needed just after a catcode test.

\cs_new_protected:Npn \__regex_mode_quit_c:
{
  \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
  \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
  \else:
  \if_int_compare:w \l__regex_mode_int =
    \c__regex_catcode_in_class_mode_int
    \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
  \fi:
  \fi:
}

(End definition for \__regex_mode_quit_c.)

40.3.4 Framework
\__regex_compile:w \__regex_compile_end:  Used when compiling a user regex or a regex for the \c{...} escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building \l__regex_internal_regex.

\cs_new_protected:Npn \__regex_compile:w
{
  \group_begin:
  \tl_build_begin:N \l__regex_build_tl
  \int_zero:N \l__regex_group_level_int
  \int_set_eq:NN \l__regex_default_catcodes_int
    \c__regex_all_catcodes_int
  \c__regex_all_catcodes_int
  \l__regex_catcodes_int
  \int_set_eq:NN \l__regex_default_catcodes_int
    \l__regex_catcodes_int
  \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
  \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
  \tl_build_put_right:Nn \l__regex_build_tl
    { \__regex_branch:n { \if_false: } \fi: }
  \group_end:
}

\cs_new_protected:Npn \__regex_compile_end:
{
  \__regex_if_in_class:TF
  { \__kernel_msg_error:nn { kernel } { missing-rbrack } }
  \use:c { \__regex_compile_end: }
}

984
\prg_do_nothing: \prg_do_nothing:
\if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
  \__kernel_msg_error:nnx { kernel } { missing-rparen }
  \int_use:N \l__regex_group_level_int 
\prg_replicate:nn
  \l__regex_group_level_int 
\{
  \tl_build_put_right:Nn \l__regex_build_tl
  \if_false: { \fi: }
  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
  \tl_build_end:N \l__regex_build_tl
  \exp_args:NNNo
  \group_end:
  \tl_build_put_right:Nn \l__regex_build_tl \l__regex_build_tl
  \exp_args:NNNx
  \group_end:
  \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
\fi:
\tl_build_put_right:Nn \l__regex_build_tl \l__regex_build_tl \{ \if_false: { \fi: } \}
\tl_build_end:N \l__regex_build_tl
\exp_args:NNNo
\group_end:
\tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
\}
(End definition for \__regex_compile:w and \__regex_compile_end:.)
\__regex_compile:n

The compilation is done between \__regex_compile:w and \__regex_compile_end:, starting in mode 0. Then \__regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since \__regex_compile_end: does that. However, catch the case of a trailing \cL construction.
\cs_new_protected:Npn \__regex_compile:n #1
\__regex_compile:w
\__regex_standard_escapechar:
\int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
\__regex_escape_use:nnnn
{ \__regex_char_if_special:NTF ##1
  \__regex_compile_special:N \__regex_compile_raw:N ##1
}{
  \__regex_char_if_alphanumeric:NTF ##1
  \__regex_compile_escaped:N \__regex_compile_raw:N ##1
}{ \__regex_compile_raw:N ##1 }
{ #1 }
\prg_do_nothing: \prg_do_nothing:
\__regex_compile_escaped:N
\__regex_compile_special:N

If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

\__regex_compile_one:n

This is used after finding one “test”, such as \d, or a raw character. If that followed a catcode test (e.g., \cL), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add \__regex_class:NnnnnN and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.
This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```
cs_new_protected:Npn \__regex_compile_abort_tokens:n \#1
{\use:x
 \exp_args:No \tl_map_function:nN \{ \tl_to_str:n \{\#1\} \}
 \__regex_compile_raw:N}
```

(End definition for \__regex_compile_abort_tokens:n.)

**40.3.5 Quantifiers**

\__regex_compile_quantifier:w This looks ahead and finds any quantifier (special character equal to either of ?+*{).}

```
cs_new_protected:Npm \__regex_compile_quantifier:w \#1\#2
{\token_if_eq_meaning:NNTF \#1 \__regex_compile_special:N
 \cs_if_exist_use:cF { \__regex_compile_quantifier_\#2:w }
 { \__regex_compile_quantifier_none: \#1 \#2 }
}
```

(End definition for \__regex_compile_quantifier:w.)

\__regex_compile_quantifier_abort:xNN Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```
cs_new_protected:Npm \__regex_compile_quantifier_abort:xNN \#1\#2\#3
{\__regex Compile_quantifier_abort:\#1\#2 \#3
 \__kernel_msg_warning:nnxx \{ kernel \} \{ invalid-quantifier \} \{\#1\} \{\#3\}
 \__regex_compile_abort_tokens:x \{\#1\}
}
```

(End definition for \__regex_compile_quantifier_abort:xNN.)
Once the “main” quantifier (\(?\), \(*\), or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending \texttt{\__regex_classNnnnN} and friends), the start-point of the range, its end-point, and a boolean, \texttt{true} for lazy and \texttt{false} for greedy operators.

\begin{verbatim}
\cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
\begin{verbatim}
\__regex_compile_quantifier_?:w \__regex_compile_quantifier_:*:w \__regex_compile_quantifier_+:w
\end{verbatim}
\end{verbatim}

For each “basic” quantifier, ?, \(*\), +, feed the correct arguments to \texttt{\__regex_compile_quantifier_lazyness:nnNN}, \texttt{−1} means that there is no upper bound on the number of repetitions.

\begin{verbatim}
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w \__regex_compile_quantifier_braced_auxii:w \__regex_compile_quantifier_braced_auxiii:w
\end{verbatim}

Three possible syntaxes: \{\langle\texttt{int}\rangle\}, \{\langle\texttt{int}\rangle,\}, or \{\langle\texttt{int}\rangle,\langle\texttt{int}\rangle\}. Any other syntax causes us to abort and put whatever we collected back in the input stream, as raw characters, including the opening brace. Grab a number into \texttt{\l__regex_internal_a_int}. If the number is followed by a right brace, the range is \([a, a]\). If followed by a comma, grab one more number, and call the \_\_\_\texttt{ii} or \_\_\_\texttt{iii} auxiliary. Those auxiliaries check for a closing brace, leading to the range \([a, \infty]\) or \([a, b]\), encoded as \{\texttt{a}\}{\texttt{−}1}\texttt{\_l\_regex_compile_quantifier_lazyness:nnNN}{\texttt{b}}{\texttt{−}a}.

\begin{verbatim}
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2\__regex_compile_quantifier_braced_auxii:w
\end{verbatim}

988
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiv:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxv:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxvi:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxvii:w #1#2
\cs_new_protected:Npn \__regex_compile_quantifier_braced Aux:w #1#2

(End definition for \__regex_compile_quantifier_braced Aux and others.)
40.3.6 Raw characters

Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```
\cs_new_protected:Npn \__regex_compile_raw_error:N #1
\__kernel_msg_error:nnx { kernel } { bad-escape } {#1}
\__regex_compile_raw:N #1
```

(End definition for \__regex_compile_raw_error:N.)

If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```
\cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
\__regex_if_in_class:TF
\__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N -
\__regex_compile_range:Nw #1
\__regex_compile_one:n \__regex_item_equal:n \int_value:w '#1 #2 #3
\__regex_compile_one:n \__regex_item_equal:n \int_value:w '#1 #2 #3
```

(End definition for \__regex_compile_raw:N.)

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```
\prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
\if_meaning:w \__regex_compile_raw:N #1
\prg_return_true:
\else:
\if_meaning:w \__regex_compile_special:N #1
\if_charcode:w ] #2
\prg_return_false:
\else:
\prg_return_true:
\fi:
\else:
\prg_return_false:
\fi:
```

\__regex_compile_range:Nw
\__regex_if_end_range:NNTF
40.3.7 Character properties

\__regex_compile_.:  In a class, the dot has no special meaning. Outside, insert \__regex_prop_.:, which matches any character or control sequence, and refuses −2 (end-marker).


The constants \__regex_prop_d:, etc. hold a list of tests which match the corresponding character class, and jump to the \__regex_break_point:TF marker. As for a normal character, we check for quantifiers.
40.3.8 Anchoring and simple assertions

In modes where assertions are allowed, anchor to the start of the query, the start of the
match, or the end of the query, depending on the integer \#1. In other modes, \#2 treats
the character as raw, with an error for escaped letters ($ is valid in a class, but \A is
definitely a mistake on the user’s part).

\CsSetProtectedNpn __regex_compile_/d: \{ __regex_compile_/d: \}
\CsSetProtectedCpx { __regex_compile_/\#: } \{ __regex_compile_/\#: \}
\CsSetProtectedCpx { __regex_compile_/#2: } \{ __regex_compile_/#2: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_compile_/A: \}
\CsSetProtectedCpx { __regex_compile_/G: } \{ __regex_compile_/G: \}
\CsSetProtectedCpx { __regex_compile_/Z: } \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/z: } \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/A: \{ __regex_compile_/A: \}
\CsSetProtectedNpn __regex_compile_/G: \{ __regex_compile_/G: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedNpn __regex_compile_/z: \{ __regex_compile_/z: \}
\CsSetProtectedNpn __regex_compile_/N: \{ __regex_compile_/N: \}
\CsSetProtectedNpn __regex_compile_/Z: \{ __regex_compile_/Z: \}
\CsSetProtectedCpx { __regex_compile_/A: } \{ __regex_comp...
Contrarily to ^ and $, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

\_regex\_compile\_/b:\n\_regex\_compile\_/B:\n
\cs\_new\_protected:cpn { \_regex\_compile\_/b: \}
\{\n\_regex\_if\_in\_class\_or\_catcode:TF
\{\_regex\_compile\_raw\_error:N b \}
\{\tl\_build\_put\_right:Nn \l\_regex\_build\_tl
\{\_regex\_assertion:Nn \c\_true\_bool \{ \_regex\_b\_test: \} \}
\}
\cs\_new\_protected:cpn { \_regex\_compile\_/B: \}
\{\n\_regex\_if\_in\_class\_or\_catcode:TF
\{\_regex\_compile\_raw\_error:N B \}
\{\tl\_build\_put\_right:Nn \l\_regex\_build\_tl
\{\_regex\_assertion:Nn \c\_false\_bool \{ \_regex\_b\_test: \} \}
\}

(End definition for \_regex\_compile\_/b: and \_regex\_compile\_/B:.)

40.3.9 Character classes

\_regex\_compile\_\[\]:
Outside a class, right brackets have no meaning. In a class, change the mode \((m \to (m - 15)/13, \text{truncated})\) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of \([\ldots\cL[\ldots\ldots\ldots]\].\) quantifiers.

\cs\_new\_protected:cpn { \_regex\_compile\_\[\]: \}
\{\n\_regex\_if\_in\_class:TF
\{\_regex\_compile\_raw\_error:N \}
\{\tl\_build\_put\_right:Nn \l\_regex\_build\_tl
\{\_regex\_assertion:Nn \c\_false\_bool \{ \_regex\_b\_test: \} \}
\}\}

(End definition for \_regex\_compile\_\[\:].)

\_regex\_compile\_\[\]:
In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following \(\c\langle\text{category}\rangle\), we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes \(0, -2\) and \(-6\)) just parse the class. The mode is updated later.

\cs\_new\_protected:cpn { \_regex\_compile\_\[\]: \}

993
\_\_regex\_compile\_class:TF
{ \_\_regex\_compile\_class\_posix\_test:w }
{ \_\_regex\_if\_within\_catcode:TF
{ \_\_regex\_compile\_class\_catcode:w
\exp\_after:wN \_\_regex\_compile\_class\_catcode:w
\int\_use:N \l\_\_regex\_catcodes\_int ;
}
{ \_\_regex\_compile\_class\_normal:w }
}

(End definition for \_\_regex\_compile\_[:]

\_\_regex\_compile\_class\_normal:w
In the “normal” case, we insert \_\_regex\_class:NnnnN (\textit{boolean}) in the compiled code. The (\textit{boolean}) is true for positive classes, and false for negative classes, characterized by a leading \textasciitilde. The auxiliary \_\_regex\_compile\_class:TFNN also checks for a leading ] which has a special meaning.
\cs_new_protected:Npn \_\_regex\_compile\_class\_normal:w
{ \_\_regex\_compile\_class:TFNN
\_\_regex\_compile\_class\_normal:w
}

(End definition for \_\_regex\_compile\_class\_normal:w)

\_\_regex\_compile\_class\_catcode:w
This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting \_\_regex\_item\_catcode:nT or the \textit{reverse} variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.
\cs_new_protected:Npn \_\_regex\_compile\_class\_catcode:w #1;
{ \_\_regex\_compile\_class\_catcode:w
\exp\_after:wN \_\_regex\_compile\_class\_catcode:w
\if_int_compare:w \l\_\_regex\_mode\_int = \c\_\_regex\_catcode\_mode\_int
\tl\_build\_put\_right:Nn \_\_regex\_build\_tl
{ \_\_regex\_class:NnnnN \c\_true\_bool }
\fi:
\fi:
\int\_set\_eq:NN \l\_\_regex\_catcodes\_int \l\_\_regex\_default\_catcodes\_int
\_\_regex\_compile\_class\_TFNN
\_\_regex\_item\_catcode:nT \#1 }
{ \_\_regex\_item\_catcode\_reverse:nT \#1 }
}

(End definition for \_\_regex\_compile\_class\_catcode:w)

\_\_regex\_compile\_class:TFNN
\_\_regex\_compile\_class:NN
If the first character is \textasciitilde, then the class is negative (use \#2), otherwise it is positive (use \#1). If the next character is a right bracket, then it should be changed to a raw one.
\cs_new_protected:Npn \_\_regex\_compile\_class:TFNN \#1#2#3#4
{ \l\_\_regex\_mode\_int = \int\_value:w \l\_\_regex\_mode\_int 3 \exp\_stop\_f:
\_\_regex\_two\_if\_eq:NNNN#1#2#3#4 \_\_regex\_compile\_special:N \textasciitilde
}
Here we check for a syntax such as [:alpha:]. We also detect [= and [, which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that’s missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra \_\_regex_item_reverse:n for negative classes.

```latex
\cs_new_protected:Npn \_\_regex_compile_class_posix_test:w #1#2
\cs_new_protected:Npn \_\_regex_compile_class_posix:x #1 #2
\cs_new_protected:Npn \_\_regex_compile_class_posix_end:w
\cs_new_protected:Npn \_\_regex_compile_class_posix_loop:w
\cs_new_protected:Npn \_\_regex_compile_class_posix_loop_end:w
\cs_new_protected:Npn \_\_regex_compile_class_posix_loop:NNNw
\cs_new_protected:Npn \_\_regex_compile_class_posix_loop_end:NNNw
```

(End definition for \_\_regex_compile_class:TFNN and \_\_regex_compile_class:NN.)
40.3.10 Groups and alternations

The contents of a regex group are turned into compiled code in \_\_regex_build_\_tl, which ends up with items of the form \_\_regex_branch:n \{\langle concatenation\rangle\}. This construction is done using \_\_build... functions within a \TeX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument \#1 is \_\_regex_group:nnnN or a variant thereof. A small subtlety to support \cL(abc) as a shorthand for (\cLa\cLb\cLc): exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.
\__regex_compile_(: \ In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch [a\cL(bcd)e]. Otherwise check for a ?, denoting special groups, and run the code for the corresponding special group.

\__regex_compile_(: }
{ \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
{ \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f: \\
\tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
\tl_build_end:N \l__regex_build_tl \\
\exp_args:NNN\exp_after:wN \__regex_compile_quantifier:w
\else: \\
\__kernel_msg_warning:nn { kernel } { extra-rparen }
\exp_args:NNN\exp_after:wN \__regex_compile_raw:N \exp_after:wN )
\fi: }

(End definition for \__regexCompile_groupBegin:n and \__regexCompile_groupEnd::)
\_regex_compile_raw:N \ ? \#3 \#4
\}
\}
\_regex_compile_group_begin:N \_regex_group:nnnN
\#1 \#2 \#3 \#4
\}

(End definition for \_regex_compile_(:)

\_regex_compile_|: In a class, the pipe is not special. Otherwise, end the current branch and open another one.
\cs_new_protected:cpn { \_regex_compile_|: }
\{
\_regex_if_in_class:TF { \_regex_compile_raw:N | }
\{
\tl_build_put_right:Nn \l__regex_build_tl
\{ \if_false: { \fi: } \_regex_branch:n \{ \if_false: \fi: \}
\}
\}

(End definition for \_regex_compile_|:

\_regex_compile_): Within a class, parentheses are not special. Outside, close a group.
\cs_new_protected:cpn { \_regex_compile_;: }
\{
\_regex_if_in_class:TF { \_regex_compile_raw:N ) }
\{
\_regex_compile_group_end: 
\}

(End definition for \_regex_compile_;:

\_regex_compile_special_group_::w \_regex_compile_special_group_|:w Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.
\cs_new_protected:cpn { \_regex_compile_special_group_::w }
\{
\_regex_compile_group_begin:N \_regex_group_no_capture:nnnN
\cs_new_protected:cpn { \_regex_compile_special_group_|:w }
\{
\_regex_compile_group_begin:N \_regex_group_resetting:nnnN

(End definition for \_regex_compile_special_group_::w and \_regex_compile_special_group_|:w)

\_regex_compile_special_group_i:w \_regex_compile_special_group_-:w The match can be made case-insensitive by setting the option with (?!); the original behaviour is restored by (?-i). This is the only supported option.
\cs_new_protected:Npn \_regex_compile_special_group_i:w #1#2
\{
\_regex_two_if_eq:NNNNTF #1 #2 \_regex_compile_special:N )
\{
\cs_set:Npn \_regex_item_equal:n
\{ \_regex_item_caseless_equal:n 
\cs_set:Npn \_regex_item_range:nn
\{ \_regex_item_caseless_range:nn 
\}
\}
\_kernel_msg_warning:nxx \{ \kernel \} \{ unknown-option \} \{ (?i \#2 \}

998
\__regex_compile_/c: \\_regex_compile_c_test:NN

The \c escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

\cs_new_protected:cpn { \__regex_compile_/c: } { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
\cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
  { \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N { \__regex_compile_c_C:NN } { \cs_if_exist_use:cF { \__regex_compile_c_#2:w } } }

\_kernel_msg_warning:nnx { kernel } { unknown-option } { (?-#2#4 } \\_regex_compile_raw:N ( \\_regex_compile_special_group_i:w and \_regex_compile_special_group_-:w)

40.3.11 Catcodes and csnames
If \cC is not followed by . or (..) then complain because that construction cannot match anything, except in cases like \cC\{\c{...}\}, where it has no effect.

When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ~ which negates the list of category codes.
The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don’t escape the scope of \c{...}.

Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{\{} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use \_regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:
\flag_new:n { __regex_cs }
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
{
  \__regex_if_in_cs:TF
  { \__regex_compile_end_cs: }
  { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
}
\cs_new_protected:Npn \__regex_compile_end_cs:
{ \__regex_compile_end:
  \flag_clear:n { __regex_cs }
  \tl_set:Nx \l__regex_internal_a_tl
  { \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
    \q__regex_nil \q__regex_nil \q__regex_recursion_stop
  }
  \exp_args:Nx \__regex_compile_one:n
  { \flag_if_raised:nTF { __regex_cs }
    { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
    { \__regex_item_exact_cs:n { \tl_tail:N \l__regex_internal_a_tl }
    }
  }
}
\cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
{ \bool_lazy_all:nTF
  { \cs_if_eq:p:NN #1 \__regex_class:NnnnN }
  { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
  { \int_compare_p:nNn {\tl_count:n {#3}} = {2} }
  { \int_compare_p:nNn {#5} = {0} }
}
{ \__regex_quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
{ \__regex_use_none_delimit_by_q_recursion_stop:w
}
}
\cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
{ \bool_lazy_all:nTF
  { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
  {\tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
  { \int_compare_p:nNn {\tl_count:n {#3}} = {2} }
  { \int_compare_p:nNn {#5} = {0} }
}
{ \prg_replicate:nn #4
  { \char_generate:nn { \use_ii:nn #3 } (12) }
\__regex_compile_cs_aux:NNnnnN

\__regex_quark_if_nil:NF #1

\flag_raise_if_clear:n { \__regex_cs }

\__regex_use_i_delimit_by_q_recursion_stop:nw

\__regex_use_none_delimit_by_q_recursion_stop:w

(End definition for \__regex_compile_; and others.)

\__regex_compile_u_loop:NN
\__regex_compile_u:NN

40.3.12 Raw token lists with \u

The \u escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of \u within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

\cs_new_protected:cpn { \__regex_compile_u: } #1#2

\__regex_if_in_class_or_catcode:TF

\__regex_compile_raw_error:N u #1 #2

\__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_left_brace_str

\tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:

\__regex_compile_u_loop:NN

\__kernel_msg_error:nn { kernel } { u-missing-lbrace }

\__regex_compile_raw:N u #1 #2

\exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2

\if_false: { \fi: } \__regex_compile_u_end: }

\exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2

\if_false: { \fi: } \__regex_compile_u_end: }

\__kernel_msg_error:nnx { kernel } { u-missing-rbrace } {#2}

\__regex_compile_u_end:
Once we have extracted the variable's name, we store the contents of that variable in \l__regex_internal_a_tl. The behaviour of \u then depends on whether we are within a \c{...} escape (in this case, the variable is turned to a string), or not.

\cs_new_protected:Npn \__regex_compile_u_end: \l__regex_internal_a_tl \g__regex_internal_tl \exp_args:No \__kernel_str_to_other_fast:n { \l__regex_internal_a_tl } \tl_build_put_right:Nx \l__regex_build_tl \tl_map_function:NN \g__regex_internal_tl \__regex_compile_u_in_cs_aux:n \__regex_class:NnnnN \c_true_bool \__regex_item_caseful_equal:n { \int_value:w '#1 } \c_false_bool \tl_analysis_map_inline:Nn \l__regex_internal_a_tl \cs_new:Npn \__regex_compile_u_in_cs_aux:n #1 \__regex_class:NnnnN \c_true_bool \__regex_item_caseful_equal:n { \int_value:w '#1 } \c_false_bool

(End definition for \__regex_compile_u_end:)

\__regex_compile_u_in_cs: When \u appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

\cs_new_protected:Npn \__regex_compile_u_in_cs: \l__regex_internal_a_tl \exp_args:No \__kernel_str_to_other_fast:n \l__regex_internal_a_tl \tl_build_put_right:Nx \l__regex_build_tl \tl_map_function:NN \g__regex_internal_tl \__regex_compile_u_in_cs_aux:n

(End definition for \__regex_compile_u_in_cs:)

\__regex_compile_u_not_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l__regex_internal_a_tl. If a given \{token\} is a control sequence, then insert a string comparison test, otherwise, \__regex_item_exact:nn which compares catcode and character code.

\cs_new_protected:Npn \__regex_compile_u_not_cs: \tl_analysis_map_inline:Nn \l__regex_internal_a_tl

1004
40.3.13 Other

\_\_regex\_compile/\K: 

The \K control sequence is currently the only “command”, which performs some action, 
rather than matching something. It is allowed in the same contexts as \b. At the 
compilation stage, we leave it as a single control sequence, defined later.

```latex
\cs_new_protected:cpn { \_\_regex\_compile/\K: }
\int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
{ \tl_build_put_right:Nn \l__regex_build_tl \{ \_\_regex\_command/\K: \} }
{ \_\_regex\_compile\_raw\_error:N K }
```

(End definition for \_\_regex\_compile/\K:)

40.3.14 Showing regexes

\_\_regex\_show:N 

Within a group and within \tl_build\_begin:N ... \tl_build\_end:N we redefine all 
the function that can appear in a compiled regex, then run the regex. The result stored 
in \_\_regex\_internal\_a\_tl is then meant to be shown.

```latex
\cs_new_protected:Npn \_\_regex\_show:N \_\_regex\_show:N #1
\group\begin:
\tl_build\_begin:N \l__regex\_build\_tl
\cs_set\_protected:Npn \_\_regex\_branch:n
\seq\_pop\_right:NN \_\_regex\_show\_prefix\_seq
\_\_regex\_internal\_a\_tl
\_\_regex\_show\_one:n \{ +\_branch \}
\seq\_put\_right:No \_\_regex\_show\_prefix\_seq
\_\_regex\_internal\_a\_tl
\use:n
\cs_set\_protected:Npn \_\_regex\_group:nnnN { \_\_regex\_show\_group\_aux:nnnN { } }
\cs_set\_protected:Npn \_\_regex\_group\_no\_capture:nnnN { \_\_regex\_show\_group\_aux:nnnN \{ \_\_no\_capture \} }
```
Every part of the final message goes through this function, which adds one line to the output, with the appropriate prefix.
\__regex_show_push:n Enter and exit levels of nesting. The \texttt{scope} function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

\cs_new_protected:Npn \__regex_show_push:n #1
\{ \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } \}

\__regex_show_pop: We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd \texttt{\use_ii:nn} avoids printing a spurious +-branch for the first branch.

\cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
\{ \__regex_show_one:n { ,-group~begin #1 } \__regex_show_push:n { | } \use_ii:nn #2 \__regex_show_pop: \__regex_show_one:n { '~-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 } \}

\__regex_show_class:NnnnN I’m entirely unhappy about this function: I couldn’t find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write \texttt{Match} or \texttt{Don’t match} on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That’s clunky, but not too expensive, since it’s only one test.

\cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
\{ \group_begin: \tl_build_begin:N \l__regex_build_tl \int_zero:N \l__regex_show_lines_int \int_zero:N \l__regex_show_lines_int \__regex_show_push:n {~} \}

(End definition for \__regex_show_one:n.)
\int_compare:nTF { \l__regex_show_lines_int = 0 } {
\group_end:
\__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
\group_end:
\bool_if:nTF
{ #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
{ #2
\tl_build_put_right:Nn \l__regex_build_tl
{ \__regex_msg_repeated:mmN {#3} {#4} #5 }
\group_end:
\tl_build_end:N \l__regex_build_tl
\exp_args:NNNo
\group_end:
\tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
\__regex_show_one:n
{ \bool_if:NTF #1 { Match } { Don’t-match }
\__regex_msg_repeated:mmN {#3} {#4} #5 }
\group_end:
\tl_build_put_right:Nx \l__regex_build_tl
{ \exp_not:o \l__regex_internal_a_tl }
\group_end:
}
(End definition for \__regex_show_class:nnnN.)
\__regex_show_anchor_to_str:N
The argument is an integer telling us where the anchor is. We convert that to the relevant info.
\cs_new:Npn \__regex_show_anchor_to_str:N #1
{ anchor-at-
\str_case:nnN { #1 }
{ \l__regex_min_pos_int } { start-(\iow_char:N\A) }
{ \l__regex_start_pos_int } { start-of-match-(\iow_char:N\G) }
{ \l__regex_max_pos_int } { end-(\iow_char:N\Z) }
{ <error:-‘#1’-not-recognized> }
}
(End definition for \__regex_show_anchor_to_str:N.)
\__regex_show_item_catcode:NnT
Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.
\cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
{ \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
\seq_set_filter:NNN \l__regex_internal_seq \l__regex_internal_seq
1008
Building

Variables used while building

The last state that was allocated is \l__regex_max_state_int – 1, so that \l__regex_max_state_int always points to a free state. The min_state variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in \c{...} constructions.

Alternatives are implemented by branching from a left state into the various choices, then merging those into a right state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

\l__regex_capturing_group_int is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering resetting groups.

(End definition for \l__regex_capturing_group_int.)
40.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a \toks. The operations which can appear in the \toks are:

- \__regex_action_start_wildcard: inserted at the start of the regular expression to make it unanchored.
- \__regex_action_success: marks the exit state of the NFA.
- \__regex_action_cost:{(shift)} is a transition from the current \langle state \rangle to \langle state \rangle + \langle shift \rangle, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- \__regex_action_free:{(shift)}, and \__regex_action_free_group:{(shift)} are free transitions, which immediately perform the actions for the state \langle state \rangle + \langle shift \rangle of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the group variant must be used for transitions back to the start of a group.
- \__regex_action_submatch:{(key)} where the \langle key \rangle is a group number followed by < or > for the beginning or end of group. This causes the current position in the query to be stored as the \langle key \rangle submatch boundary.

We strive to preserve the following properties while building:

- The current capturing group is capturing_group – 1, and if a group opened now it would be labelled capturing_group.
- The last allocated state is max_state – 1, so max_state is a free state.
- The left_state points to a state to the left of the current group or of the last class.
- The right_state points to a newly created, empty state, with some transitions leading to it.
- The left/right sequences hold a list of the corresponding end-points of nested groups.

The \n-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of capturing_group). Finally, if the match reaches the last state, it is successful.
\_\_regex\_build\_new\_state:
\_\_regex\_toks\_put\_right:Nn \l__regex\_left\_state\_int
{ \_\_regex\_action\_start\_wildcard: }
\_\_regex\_group:mmNN \{#1\} \{ 0 \} \c\_false\_bool
\_\_regex\_toks\_put\_right:Nn \l__regex\_right\_state\_int
{ \_\_regex\_action\_success: }
}

(End definition for \_\_regex\_build:n and \_\_regex\_build:N.)

\_\_regex\_build\_for\_cs:n

The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- \g__regex\_state\_active\_intarray from \l__regex\_min\_state\_int to \l__regex\_max\_state\_int − 1;
- \g__regex\_thread\_state\_intarray from \l__regex\_min\_active\_int to \l__regex\_max\_active\_int − 1.

In fact, some data is stored in \toks registers (local) in the same ranges so these ranges mustn’t overlap. This is done by setting \l__regex\_min\_active\_int to \l__regex\_max\_state\_int after building the NFA. Here, in this nested call to the matching code, we need the new versions of these ranges to involve completely new entries of the intarray variables, so we begin by setting (the new) \l__regex\_min\_state\_int to (the old) \l__regex\_max\_active\_int to use higher entries.

When using a regex to match a cs, we don’t insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate left and right states in their sequence.

\cs\_new\_protected:Npn \_\_regex\_build\_for\_cs:n \#1
{ \int\_set\_eq:NN \l__regex\_min\_state\_int \l__regex\_max\_active\_int
  \int\_set\_eq:NN \l__regex\_max\_state\_int \l__regex\_min\_state\_int
  \_\_regex\_build\_new\_state:
  \_\_regex\_build\_new\_state:
  \_\_regex\_push\_lr\_states:
  #1
  \_\_regex\_pop\_lr\_states:
  \_\_regex\_toks\_put\_right:Nn \l__regex\_right\_state\_int
  { \if_int\_compare:w \l__regex\_curr\_pos\_int = \l__regex\_max\_pos\_int
    \exp\_after:wN \_\_regex\_action\_success:
    \fi:
  }
}

(End definition for \_\_regex\_build\_for\_cs:n.)

40.4.3 Helpers for building an nfa

\_\_regex\_push\_lr\_states:
\_\_regex\_pop\_lr\_states:

When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from \TeX’s grouping.

\cs\_new\_protected:Npn \_\_regex\_push\_lr\_states:
{
\seq_push:No \l__regex_left_state_seq
{ \int_use:N \l__regex_left_state_int }
\seq_push:No \l__regex_right_state_seq
{ \int_use:N \l__regex_right_state_int }
}
\cs_new_protected:Npn \__regex_pop_lr_states:
{ \seq_pop:NN \l__regex_left_state_seq \l__regex_internal_a_tl
\int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
\seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
\int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
}
(End definition for \__regex_push_lr_states: and \__regex_pop_lr_states:.)

\__regex_build_transition_left:NNN
\__regex_build_transition_right:nNn
Add a transition from #2 to #3 using the function #1. The left function is used for higher priority transitions, and the right function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.
\cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
{ \__regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
{ \__regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
(End definition for \__regex_build_transition_left:NNN and \__regex_build_transition_right:nNn.)

\__regex_build_new_state:
Add a new empty state to the NFA. Then update the left, right, and max states, so that the right state is the new empty state, and the left state points to the previously “current” state.
\cs_new_protected:Npn \__regex_build_new_state:
{ \__regex_toks_clear:N \l__regex_max_state_int
\int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
\int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
\int_incr:N \l__regex_max_state_int
}
(End definition for \__regex_build_new_state:)

\__regex_build_transitions_lazyness:NNNNN
This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.
\cs_new_protected:Npn \__regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
{ \__regex_build_new_state:
\__regex_toks_put_right:Nx \l__regex_left_state_int
{ \if_meaning:w \c_true_bool #1
  #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
  #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
  \else:
  #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
  #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
\fi:
}
40.4.4 Building classes

The arguments are: \langle boolean \rangle \{ \{ tests \} \} \{ \{ min \} \} \{ \{ more \} \} \{ lazyness \}. First store the tests with a trailing \__regex_action_cost:n, in the true branch of \__regex_break_point:TF for positive classes, or the false branch for negative classes. The integer \langle more \rangle is 0 for fixed repetitions, −1 for unbounded repetitions, and \langle max \rangle − \langle min \rangle for a range of repetitions.

\__regex_class:NnnnN

\__regex_tests_action_cost:n

This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for \#1 = 0 repetitions: nothing is built.

\__regex_class_repeat:n

This implements unbounded repetitions of a single class (e.g. the \* and + quantifiers). If the minimum number \#1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call \__regex_class_repeat:n for the code to match \#1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean \#2.
We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from max_state.

\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
{ \if_int_compare:w #3 = 0 \exp_stop_f: \__regex_build_new_state: \l__regex_internal_a_int \l__regex_max_state_int + #2 - 1 \prg_replicate:nn { #2 } { \__regex_build_transitions_lazyness:NNNNN \__regex_action_free:n \l__regex_left_state_int \__regex_tests_action_cost:n \l__regex_left_state_int } \__regex_action_free:n \l__regex_right_state_int \__regex_build_transitions_lazyness:NNNNN \__regex_action_free:n \l__regex_internal_a_int \fi: }

(End definition for \__regex_group_aux:nnnnN.)

40.4.5 Building groups

\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
{ \if_int_compare:w #3 = 0 \exp_stop_f: \__regex_build_new_state: \l__regex_internal_a_int \assert_int:n { \l__regex_max_state_int = \l__regex_right_state_int + 1 } \prg_replicate:nn { #2 } { \__regex_build_transitions_lazyness:NNNNN \__regex_action_free:n \l__regex_left_state_int \__regex_tests_action_cost:n \l__regex_left_state_int } \__regex_action_free:n \l__regex_right_state_int \__regex_build_transitions_lazyness:NNNNN \__regex_action_free:n \l__regex_internal_a_int \fi: }

(End definition for \__regex_group_aux:nnnnN.)
\_\_regex\_left\_state\_int \_\_regex\_right\_state\_int

\fi:
\_\_regex\_build\_new\_state:
\_\_regex\_push\_lr\_states:
#2
\_\_regex\_pop\_lr\_states:
\if_case:w - #4 \exp\_stop\_f:
  \_\_regex\_group\_repeat:nn {#1} {#3}
\or: \_\_regex\_group\_repeat:nnN {#1} {#3} #5
\else: \_\_regex\_group\_repeat:nnnN {#1} {#3} {#4} #5
\fi:
}

(End definition for \_\_regex\_group\_aux:nnnN.)
\_\_regex\_group:nnnN \_\_regex\_group\_no\_capture:nnnN

Hand to \_\_regex\_group\_aux:nnnnN the label of that group (expanded), and the group itself, with some extra commands to perform.
\cs\_new\_protected:Npn \_\_regex\_group:nnnN #1
  \exp\_args:No \_\_regex\_group\_aux:nnnnN
  { \int\_use:N \_\_regex\_capturing\_group\_int }
  \int\_incr:N \_\_regex\_capturing\_group\_int
  #1

\cs\_new\_protected:Npn \_\_regex\_group\_no\_capture:nnnN
  \{ \_\_regex\_group\_aux:nnnnN { -1 } \}

(End definition for \_\_regex\_group:nnnN and \_\_regex\_group\_no\_capture:nnnN.)
\_\_regex\_group\_resetting:nnnN \_\_regex\_group\_resetting\_loop:nnnN

Again, hand the label −1 to \_\_regex\_group\_aux:nnnnN, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form \_\_regex\_branch:n { ⟨branch⟩}. 
\cs\_new\_protected:Npn \_\_regex\_group\_resetting\_loop:nnnN
  \{ \_\_regex\_group\_aux:nnnnN { -1 } \}

\cs\_new\_protected:Npn \_\_regex\_group\_resetting_loop:nnnN
  \{ \_\_regex\_group\_aux:nnnnN \{ -1 \} \}

\use\_none:nn #3 { \int\_set:Nn \_\_regex\_capturing\_group\_int {#1} }
\int\_set:Nn \_\_regex\_capturing\_group\_int {#2}
\int\_max:nn {#1} { \_\_regex\_capturing\_group\_int }
Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

\cs_new_protected:Npn \__regex_branch:n #1
\__regex_build_new_state:
\seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
\int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
\__regex_build_transition_right:nNn \__regex_action_free:n
\l__regex_left_state_int \l__regex_right_state_int #1
\seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
\__regex_build_transition_right:nNn \__regex_action_free:n
\l__regex_right_state_int \l__regex_internal_a_tl
\__regex_build_new_state:
\if_int_compare:w #2 = 0 \exp_stop_f:
\int_set:Nn \l__regex_max_state_int \l__regex_left_state_int
\__regex_build_new_state:
\else:
\__regex_group_repeat_aux:n {#2}
\__regex_group_submatches:nNN {#1}
\l__regex_internal_a_int \l__regex_right_state_int
\__regex_build_new_state:
\fi:
\}

\cs_new_protected:Npn \__regex_group_repeat:nn #1#2
\if_int_compare:w #2 = 0 \exp_stop_f:
\int_set:Nn \l__regex_max_state_int \l__regex_left_state_int
\__regex_build_new_state:
\else:
\__regex_group_repeat_aux:n {#2}
\__regex_group_submatches:nNN {#1}
\l__regex_internal_a_int \l__regex_right_state_int
\__regex_build_new_state:
\fi:
\}

\cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
\if_int_compare:w #1 > - 1 \exp_stop_f:
\__regex_toks_put_left:Nx #2 { \__regex_action_submatch:n { #1 < } }
\__regex_toks_put_left:Nx #3 { \__regex_action_submatch:n { #1 > } }
\fi:
\}

\__regex_group_repeat:nn This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don’t reset the capturing_group label). Otherwise, the auxiliary \__regex_group_repeat_aux:n copies #2 times the \toks for the group, and leaves internal_a pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary.

\__regex_group_submatches:nNN This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of −1.
Here we repeat \texttt{toks} ranging from \texttt{left_state} to \texttt{max_state}, #1 > 0 times. First add a transition so that the copies “chain” properly. Compute the shift \(c\) between the original copy and the last copy we want. Shift the \texttt{right_state} and \texttt{max_state} to their final values. We then want to perform \(c\) copy operations. At the end, \(b\) is equal to the \texttt{max_state}, and \(a\) points to the left of the last copy of the group.

\begin{verbatim}
\cs_new_protected:Npn \__regex_group_repeat_aux:n #1
  {
    \__regex_build_transition_right:nNn \__regex_action_free:n
    \l__regex_right_state_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
    \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
    \if_int_compare:w \int_eval:n {#1} > 1 \exp_stop_f:
      \int_set:Nn \l__regex_internal_c_int {(#1 - 1) * (\l__regex_internal_b_int - \l__regex_internal_a_int )}
      \int_add:Nn \l__regex_right_state_int {\l__regex_internal_c_int}
      \int_add:Nn \l__regex_max_state_int {\l__regex_internal_c_int}
      \__regex_toks_memcpy:NNn \l__regex_internal_b_int \l__regex_internal_a_int \l__regex_internal_c_int
    \fi:
  }
\end{verbatim}

(End definition for \texttt{\__regex_group_repeat_aux:n})

\begin{verbatim}
\cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
  {
    \if_int_compare:w #2 = 0 \exp_stop_f:
      \__regex_group_submatches:nNN {#1}
      \l__regex_left_state_int \l__regex_right_state_int
      \int_set:Nn \l__regex_internal_a_int {\l__regex_left_state_int - 1}
      \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_internal_a_int
      \__regex_build_new_state:
      \if_meaning:w \c_true_bool #3
        \__regex_build_new_state:
      \fi:
  }
\end{verbatim}

(End definition for \texttt{\__regex_group_repeat:nnN})

This function is called to repeat a group at least \(n\) times; the case \(n = 0\) is very different from \(n > 0\). Assume first that \(n = 0\). Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state \(a\) (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from \(a\) to a new state.

Now consider the case \(n > 0\). Repeat the group \(n\) times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from \texttt{\__regex_group_repeat_aux:n}.
\_regex_group_repeat:nnnN  

We wish to repeat the group between #2 and #2 + #3 times, with a lazyness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.
\_regex_build_transition_right:mm \_regex_action_free:n
\l__regex_right_state_int \l__regex_max_state_int
}
\if_int_compare:w #2 = 0 \exp_stop_f:
\int_set:Nn \l__regex_right_state_int
{ \l__regex_left_state_int - 1 }
\else:
\int_sub:Nn \l__regex_right_state_int
{ \l__regex_internal_b_int - \l__regex_internal_a_int }
\fi:
\_regex_build_transition_right:mm \_regex_action_free:n
\l__regex_right_state_int \l__regex_max_state_int
\fi:
\_regex_build_new_state:
\fi:
\_regex_build_new_state:

(End definition for \_regex_group_repeat:mm.)

40.4.6 Others

Usage: \_regex Assertion:Nn \{⟨boolean⟩ \{⟨test⟩⟩}, where the ⟨test⟩ is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The \_regex b test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use \_regex anchor:N, with a position controlled by the integer #1.

\cs_new_protected:Npn \_regex_assertion:Nn \{⟨boolean⟩ \{⟨test⟩⟩\}
{\_regex_build_new_state:
\_regex_toks_put_right:Nx \l__regex_left_state_int
\exp_not:n {\l__regex_last_char_int}
\_regex_break_point:TF
\bool_if:NF #1 { { } }
{ \_regex_action_free:n
\{ \int_eval:n
{ \l__regex_right_state_int - \l__regex_left_state_int }
\}
\bool_if:NT #1 { { } }
}
\cs_new_protected:Npn \_regex_anchor:N \#1
{\if_int_compare:w #1 = \l__regex_curr_pos_int
\exp_after:wN \_regex_break_true:w
\fi:
\cs_new_protected:Npn \_regex_b_test:
{\group_begin:
\int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int

40.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in \g{\_regex_thread_state_intarray}: this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching \( (a?)^* \) against \( a \) is broken, isn’t it? No. The group first matches \( a \), as it should, then repeats; it attempts to match \( a \) again but fails; it skips \( a \), and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) \( a \). What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.
We solve this by distinguishing “normal” free transitions \_\_regex_action_free:n from transitions \_\_regex_action_free_group:n which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

40.5.1 Variables used when matching

The tokens in the query are indexed from \texttt{min_pos} for the first to \texttt{max_pos} – 1 for the last, and their information is stored in several arrays and \texttt{toks} registers with those numbers. We don’t start from 0 because the \texttt{toks} registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the \texttt{current_pos} in the query. The starting point of the current match attempt is \texttt{start_pos}, and \texttt{success_pos}, updated whenever a thread succeeds, is used as the next starting position.

\begin{verbatim}
\l__regex_min_pos_int \l__regex_max_pos_int \l__regex_curr_pos_int \l__regex_start_pos_int \l__regex_success_pos_int
\end{verbatim}

The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token (A-Z \leftrightarrow a-z). This last integer is only computed when necessary, and is otherwise \texttt{\c_max_int}. The \texttt{current_char} variable is also used in various other phases to hold a character code.

\begin{verbatim}
\l__regex_curr_char_int \l__regex_curr_catcode_int \l__regex_last_char_int \l__regex_case_changed_char_int
\end{verbatim}

For every character in the token list, each of the active states is considered in turn. The variable \texttt{\l__regex_curr_state_int} holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

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

The submatches for the thread which is currently active are stored in the \texttt{current_submatches} property list variable. This property list is stored by \_\_regex_action_cost:n into the \texttt{toks} register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to \texttt{\l__regex_success_submatches_prop}: only the last successful thread remains there.

\begin{verbatim}
\l__regex_curr_submatches_prop \l__regex_success_submatches_prop
\end{verbatim}

(End definition for \texttt{\l__regex_min_pos_int} and others.)
This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in \texttt{\g__regex_state_active_intarray} the last step in which each \texttt{⟨state⟩} in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to \texttt{step} when we have started performing the operations of \texttt{toks(⟨state⟩)}, but not finished yet. However, once we finish, we store \texttt{step + 1} in \texttt{\g__regex_state_active_intarray}. This is needed to track submatches properly (see building phase). The \texttt{step} is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

All the currently active threads are kept in order of precedence in \texttt{\g__regex_thread_state_intarray}, and the corresponding submatches in the \texttt{toks}. For our purposes, those serve as an array, indexed from \texttt{min_active} (inclusive) to \texttt{max_active} (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and \texttt{max_active} is reset to \texttt{min_active}, effectively clearing the array.

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See \texttt{\_regex_single_match:} and \texttt{\_regex_multi_match:n}.

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting \texttt{\_regex_fresh_thread_bool} to \texttt{true} for threads which directly come from the start of the regex or from the \texttt{\K} command, and testing that boolean whenever a thread succeeds. The function \texttt{\_regex_if_two_empty_matches:F} is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don’t have two identical empty matches, so the function is \texttt{\use:n}.
The boolean \texttt{\_regex_match_success_bool} is true if the current match attempt was successful, and \texttt{\_regex_success_bool} is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with \texttt{\c{...}}. This is done by saving the global variable into \texttt{\_regex_saved_success_bool}, which is local, hence not affected by the changes due to inner regex functions.

\begin{verbatim}
\bool_new:N \g__regex_success_bool
\bool_new:N \l__regex_saved_success_bool
\bool_new:N \l__regex_match_success_bool
\end{verbatim}

### 40.5.2 Matching: framework

First store the query into \texttt{\toks} registers and arrays (see \texttt{\_regex_query_set:nnn}). Then initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (\texttt{\g__regex_state_active_intarray}), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

\begin{verbatim}
\cs_new_protected:Npn \_regex_match:n #1
\cs_new_protected:Npn \_regex_match_cs:n #1
\cs_new_protected:Npn \_regex_match_init:
\end{verbatim}

1023
\__regex_match_once: This function finds one match, then does some action defined by the every_match token list, which may recursively call \__regex_match_once:. First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous success_pos, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start, and get that token, to set last_chars properly for word boundaries. Then call \__regex_match_loop:, which runs through the query until the end or until a successful match breaks early.
\_\_regex\_single\_match:\nFor a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action \#1, then find the next match.

\cs\_new\_protected:Npn \_\_regex\_single\_match:\n\{\n\tl\_set:Nn \l\_\_regex\_every\_match\_tl\n\{\n\tl\_set:Nn \l\_\_regex\_every\_match\_tl\n\bool\_gset\_eq:NN \g\_\_regex\_success\_bool \l\_\_regex\_match\_success\_bool\n\}\n\}\n\cs\_new\_protected:Npn \_\_regex\_multi\_match:n \#1\n\{\n\tl\_set:Nn \l\_\_regex\_every\_match\_tl\n\{\n\tl\_set:Nn \l\_\_regex\_every\_match\_tl\n\if\_meaning:w \c\_true\_bool \l\_\_regex\_match\_success\_bool\n\bool\_gset\_true:N \g\_\_regex\_success\_bool \#1\n\exp\_after:wN \_\_regex\_match\_once:\n\fi:\n\}\n\}\n
(End definition for \_\_regex\_single\_match: and \_\_regex\_multi\_match:n.)

\_\_regex\_match\_loop:\n\_\_regex\_match\_one\_active:n
At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (\texttt{max\_active}). This results in a sequence of \_\_regex\_use\_state\_and\_submatches:nn \{(\texttt{state}) \{(\texttt{prop})\}\), and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is what \_\_regex\_match\_once: matches. We explain the \texttt{fresh\_thread} business when describing \_\_regex\_action\_wildcard:. 

\cs\_new\_protected:Npn \_\_regex\_match\_loop:\n\{\n\int\_add:Nn \l\_\_regex\_step\_int \{ \texttt{2} \}\n\int\_incr:N \l\_\_regex\_curr\_pos\_int\n\int\_set:eq:NN \l\_\_regex\_last\_char\_int \l\_\_regex\_curr\_char\_int\n\int\_set:eq:NN \l\_\_regex\_case\_changed\_char\_int \c\_max\_int\n\_\_regex\_query\_get:\n\use:x
\{\n\int\_set:eq:NN \l\_\_regex\_max\_active\_int \l\_\_regex\_min\_active\_int\n\int\_step\_function:mm\NN\n\{ \l\_\_regex\_min\_active\_int \}\n\{ \l\_\_regex\_max\_active\_int - \texttt{1} \}\n
1025
\l__regex_match_one_active:n
}

\prg_break_point:
\bool_set_false:N \l__regex_fresh_thread_bool
\if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
\if_int_compare:w \l__regex_curr_pos_int < \l__regex_max_pos_int
\exp_after:wN \exp_after:wN \exp_after:wN \__regex_match_loop:
\fi:
\fi:
\cs_new:Npn \__regex_match_one_active:n #1
{
\__regex_use_state_and_submatches:nn
{ \__kernel_intarray_item:Nn \g__regex_thread_state_intarray {#1} }
{ \__regex_toks_use:w #1 }
}

(End definition for \__regex_match_loop: and \__regex_match_one_active:n.)

\__regex_query_set:nnn The arguments are: tokens that o and x expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a \toks register and some arrays, then update the balance.
\cs_new:Npn \__regex_query_set:nnn #1#2#3
{
\__kernel_intarray_gset:Nnn \g__regex_charcode_intarray
{ \l__regex_curr_pos_int } {#3}
\__kernel_intarray_gset:Nnn \g__regex_catcode_intarray
{ \l__regex_curr_pos_int } {#2}
\__kernel_intarray_gset:Nnn \g__regex_balance_intarray
{ \l__regex_curr_pos_int } { \l__regex_balance_int }
\__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
\int_incr:N \l__regex_curr_pos_int
\if_case:w #2 \exp_stop_f:
\or: \int_incr:N \l__regex_balance_int
\or: \int_decr:N \l__regex_balance_int
\fi:
}

(End definition for \__regex_query_set:nnn.)

\__regex_query_get: Extract the current character and category codes at the current position from the appropriate arrays.
\cs_new:Npn \__regex_query_get: \l__regex_curr_char_int
= \__kernel_intarray_item:Nn \g__regex_charcode_intarray
{ \l__regex_curr_pos_int } \scan_stop:
\l__regex_curr_catcode_int
= \__kernel_intarray_item:Nn \g__regex_catcode_intarray
{ \l__regex_curr_pos_int } \scan_stop:
}

(End definition for \__regex_query_get::)
40.5.3 Using states of the NFA

\_\_regex\_use\_state:
Use the current NFA instruction. The state is initially marked as belonging to the current
step: this allows normal free transition to repeat, but group-repeating transitions won’t.
Once we are done exploring all the branches it spawned, the state is marked as step + 1:
any thread hitting it at that point will be terminated.

\cs\_new\_protected:Npn \_\_regex\_use\_state:
\{ \_\_kernel\_intarray\_gset:Nnn \g__regex\_state\_active\_intarray
\{ \l__regex\_curr\_state\_int \} \{ \l__regex\_step\_int \}
\_\_regex\_toks\_use:w \l\_\_regex\_curr\_state\_int
\_\_kernel\_intarray\_gset:Nnn \g__regex\_state\_active\_intarray
\{ \l__regex\_curr\_state\_int \}
\{ \int\_eval:n \{ \l\_\_regex\_step\_int \+ \+ \} \}
\}
(End definition for \_\_regex\_use\_state:.)

\_\_regex\_use\_state\_and\_submatches:nn
This function is called as one item in the array of active threads after that array has been
unpacked for a new step. Update the current state and current submatches and use
the state if it has not yet been encountered at this step.

\cs\_new\_protected:Npn \_\_regex\_use\_state\_and\_submatches:nn \#1 \#2
\{ \int\_set:Nn \l\_\_regex\_curr\_state\_int \{\#1\}
\if\_int\_compare:w \_\_kernel\_intarray\_item:Nn \g__regex\_state\_active\_intarray
\{ \l\_\_regex\_curr\_state\_int \}
\< \_\_regex\_step\_int
\tl\_set:Nn \l\_\_regex\_curr\_submatches\_prop \{\#2\}
\exp\_after:wN \_\_regex\_use\_state:
\fi:
\scan\_stop:
\}
(End definition for \_\_regex\_use\_state\_and\_submatches:nn.)

40.5.4 Actions when matching

\_\_regex\_action\_start\_wildcard:
For an unanchored match, state 0 has a free transition to the next and a costly one to
itself, to repeat at the next position. To catch repeated identical empty matches, we need
to know if a successful thread corresponds to an empty match. The instruction resetting
\_\_regex\_fresh\_thread\_bool may be skipped by a successful thread, hence we had to
add it to \_\_regex\_match\_loop: too.

\cs\_new\_protected:Npn \_\_regex\_action\_start\_wildcard:
\{ \bool\_set\_true:N \_\_regex\_fresh\_thread\_bool
\_\_regex\_action\_free:n \{1\}
\bool\_set\_false:N \_\_regex\_fresh\_thread\_bool
\_\_regex\_action\_cost:n \{0\}
\}
(End definition for \_\_regex\_action\_start\_wildcard:.)
These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of \l__regex_curr_state_int and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the group version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the "normal" version will revisit a state even within the thread itself.

\cs_new_protected:Npn \__regex_action_free:n
\cs_new_protected:Npn \__regex_action_free_group:n
\cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
{ \use:x
\int_add:Nn \l__regex_curr_state_int {#2}
\exp_not:n
\if_int_compare:w
\__kernel_intarray_item:Nn \g__regex_state_active_intarray { \l__regex_curr_state_int } #1
\exp_after:wN \__regex_use_state:
\fi:
\int_set:Nn \l__regex_curr_state_int
\int_use:N \l__regex_curr_state_int
\tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
\exp_not:o \l__regex_curr_submatches_prop
}

(End definition for \__regex_action_free:n, \__regex_action_free_group:n, and \__regex_action_free_aux:nn.)

\__regex_action_cost:n
A transition which consumes the current character and shifts the state by \#1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

\cs_new_protected:Npn \__regex_action_cost:n \#1
{ \exp_args:Nx \__regex_store_state:n \int_eval:n { \l__regex_curr_state_int + \#1 } }

(End definition for \__regex_action_cost:n.)

\__regex_store_state:n, \__regex_store_submatches:
Put the given state in \g__regex_thread_state_intarray, and increment the length of the array. Also store the current submatch in the appropriate \toks.

\cs_new_protected:Npn \__regex_store_state:n \#1
{ \__regex_store_submatches:
\__kernel_intarray_gset:Nnn \g__regex_thread_state_intarray { \l__regex_max_active_int } \#1
\int_incr:N \l__regex_max_active_int

1028
Some user functions don’t require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with \texttt{\textbackslash prg\_break}; and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.
40.6 Replacement

40.6.1 Variables and helpers used in replacement

The behaviour of closing braces inside a replacement text depends on whether a sequence \c{ or \u{ has been encountered. The number of “open” such sequences that should be closed by } is stored in \l__regex_replacement_csnames_int, and decreased by 1 by each }.

\int_new:N \l__regex_replacement_csnames_int

(End definition for \l__regex_replacement_csnames_int.)

\l__regex_replacement_category_tl \l__regex_replacement_category_seq

This sequence of letters is used to correctly restore categories in nested constructions such as \cL(abc\cD(_d).

\tl_new:N \l__regex_replacement_category_tl
\seq_new:N \l__regex_replacement_category_seq

(End definition for \l__regex_replacement_category_tl and \l__regex_replacement_category_seq.)

\l__regex_balance_tl

This token list holds the replacement text for \__regex_replacement_balance_one_match:n while it is being built incrementally.

\tl_new:N \l__regex_balance_tl

(End definition for \l__regex_balance_tl.)

\__regex_replacement_balance_one_match:n

This expects as an argument the first index of a set of entries in \g__regex_submatch_begin_intarray (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

\cs_new:Npn \__regex_replacement_balance_one_match:n #1
{ - \__regex_submatch_balance:n (#1) }

(End definition for \__regex_replacement_balance_one_match:n.)

\__regex_replacement_do_one_match:n

The input is the same as \__regex_replacement_balance_one_match:n. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

\cs_new:Npn \__regex_replacement_do_one_match:n #1
{ \__regex_query_range:nn
\__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1}
\__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1}
}
This function lets us navigate around the fact that the primitive \exp_not:n requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as \c_parameter_token. Indeed, within an x-expanding assignment, \exp_not:N # behaves as a single #, whereas \exp_not:n {#} behaves as a doubled ##.

(End definition for \__regex_replacement_exp_not:N.)

\__regex_query_range:nn
\__regex_query_range_loop:ww

When it is time to extract submatches from the token list, the various tokens are stored in \toks registers numbered from \__regex_min_pos_int inclusive to \__regex_max_pos_int exclusive. The function \__regex_query_range:nn \{\langle min\rangle\} \{\langle max\rangle\} unpacks registers from the position \langle min\rangle to the position \langle max\rangle − 1 included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

(End definition for \__regex_query_range:nn and \__regex_query_range_loop:ww.)

\__regex_query_submatch:n

Find the start and end positions for a given submatch (of a given match).

(End definition for \__regex_query_submatch:n.)

\__regex_submatch_balance:n

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the \langle max pos\rangle and \langle min pos\rangle. These two positions are found in the corresponding “submatch” arrays.
The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \l__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.
\prg_do_nothing: \prg_do_nothing:
\if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
  \__kernel_msg_error:nnx { kernel } { replacement-missing-rbrace }
  \seq_if_empty:NF \l__regex_replacement_category_seq
  { \__kernel_msg_error:nnx { kernel } { replacement-missing-rparen }
    \seq_clear:N \l__regex_replacement_category_seq }
  \cs_gset:Npx \__regex_replacement_balance_one_match:n ##1
    { + \int_use:N \l__regex_balance_int
      \l__regex_balance_tl
    - \__regex_submatch_balance:n (#1)
  }
\tl_build_end:N \l__regex_build_tl
\exp_args:NNo
\group_end: \__regex_replacement_aux:n \l__regex_build_tl
\ps_new_protected:Np \__regex_replacement_aux:n #1
\ps_new:Np \__regex_replacement_do_one_match:n #1
\__regex_query_range:nn
  \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {##1}
  \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {##1}
#1
}

(End definition for \__regex_replacement:n and \__regex_replacement_aux:n.)

\__regex_replacement_normal:n Most characters are simply sent to the output by \tl_build_put_right:Nn, unless a
particular category code has been requested: then \_\_regex_replacement_c_A:w or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of \l__regex_replacement_category_tl.

\cs_new_protected:Npn \_\_regex_replacement_normal:n #1
\tl_if_empty:NTF \l__regex_replacement_category_tl
\{ % ( 
\token_if_eq_charcode:NNTF #1 )
\seq_pop:NN \l__regex_replacement_category_seq
\l__regex_replacement_category_tl
\}
\use:c
\{ \_\_regex_replacement_c_
\} \l__regex_replacement_category_tl :w
\l__regex_replacement_normal:n {#1}
\}
\}

(End definition for \_\_regex_replacement_normal:n.)

\_\_regex_replacement_escaped:N
As in parsing a regular expression, we use an auxiliary built from #1 if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character. We use \token_to_str:N to give spaces the right category code.

\cs_new_protected:Npn \_\_regex_replacement_escaped:N #1
\cs_if_exist_use:cF { \_\_regex_replacement_#1:w }
\{\}
\{ \if_int_compare:w 1 < #1 \exp_stop_f:
\_\_regex_replacement_put_submatch:n {#1}
\else:
\exp_args:No \_\_regex_replacement_normal:n
\{ \token_to_str:N #1 \}
\fi:
\}

(End definition for \_\_regex_replacement_escaped:N.)

40.6.4 Submatches

\_\_regex_replacement_put_submatch:n
Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a \c{...} or \u{...} construction, it must be taken into account in the brace balance. Later on, ##1 will be replaced by a pointer to the 0-th submatch for a given match.
There is an \exp_not:N here as at the point-of-use of \__regex_balance_tl there is an x-type expansion which is needed to get \#\#1 in correctly.
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
{\if_int_compare:w #1 < \l__regex_capturing_group_int\tl_build_put_right:Nn \l__regex_build_tl\{\int_eval:n \{ #1 + \#\#1 \}\}\fi:}
\tl_put_right:Nn \l__regex_balance_tl\{\exp_not:N \int_eval:n \{ #1 + \#\#1 \}\}
\fi:
\exp_stop_f:
\fi:
\exp_stop_f:
(End definition for \__regex_replacement_put_submatch:n.)
\__regex_replacement_g:w
\__regex_replacement_g_digits:NN
Grab digits for the \g escape sequence in a primitive assignment to the integer \l__-regex_internal_a_int. At the end of the run of digits, check that it ends with a right brace.
\cs_new_protected:Npn \__regex_replacement_g:w #1#2
{\__regex_two_if_eq:NNNNTF #1 #2 \__regex_replacement_normal:n \c_left_brace_str\{\l__regex_internal_a_int = \__regex_replacement_g_digits:NN\}\{\__regex_replacement_error:NNN g #1 \#2\}}
\cs_new:Npn \__regex_replacement_g_digits:NN #1#2
{\token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n\{\if_int_compare:w 1 < 1#2 \exp_stop_f:\#2\exp_after:wN \use_i:nnn\exp_after:wN \__regex_replacement_g_digits:NN\else:\exp_stop_f:\exp_after:wN \__regex_replacement_error:NNN \exp_after:wN \__regex_replacement_g_digits:NN\exp_after:wN g\fi:\}}
\token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n\{\if_int_compare:w 1 < 1 \exp_stop_f:\#2\exp_after:wN \use_i:nnn\exp_after:wN \__regex_replacement_g_digits:NN\else:\exp_stop_f:\exp_after:wN \__regex_replacement_error:NNN \exp_after:wN \__regex_replacement_g_digits:NN\exp_after:wN g\fi:\}}
#1 #2

(End definition for \_\_regex_replacement_g:w and \_\_regex_replacement_g_digits:Nw.)

40.6.5 Csnames in replacement

\_\_regex_replacement_c:w \c may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with \u. Otherwise test whether the category is known; if it is not, complain.

\cs_new_protected:Npn \_\_regex_replacement_c:w #1\#2
  { \token_if_eq_meaning:NNTF #1 \_\_regex_replacement_normal:n
    { \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
      { \_\_regex_replacement_cu_aux:Nw \_\_regex_replacement_exp_not:N }
      { \cs_if_exist:cTF { \_\_regex_replacement_c_#2:w }
        { \_\_regex_replacement_cat:NNN #2 }
        { \_\_regex_replacement_error:NNN c #1\#2 }
      }
    }{ \_\_regex_replacement_error:NNN c #1\#2 }
  }

(End definition for \_\_regex_replacement_c:w.)

\_\_regex_replacement_cu_aux:Nw Start a control sequence with \cs:w, protected from expansion by #1 (either \_\_regex_replacement_exp_not:N or \exp_not:V), or turned to a string by \tl_to_str:V if inside another cuname construction \c or \u. We use \tl_to_str:V rather than \tl_to_str:N to deal with integers and other registers.

\cs_new_protected:Npn \_\_regex_replacement_cu_aux:Nw #1
  { \if_case:w \l__regex_replacement_csnames_int
    \tl_build_put_right:Nn \l__regex_build_tl
    \else:
    \tl_build_put_right:Nn \l__regex_build_tl
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }

(End definition for \_\_regex_replacement_cu_aux:Nw.)

\_\_regex_replacement_u:w Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp_not:V or \tl_to_str:V depending on the current context.

\cs_new_protected:Npn \_\_regex_replacement_u:w #1#2
  { \_\_regex_two_if_eq:NNNNTF #1 #2 \_\_regex_replacement_normal:n \c_left_brace_str
    { \_\_regex_replacement_cu_aux:Nw \exp_not:V }
    { \_\_regex_replacement_error:NNN u #1#2 }
  }

1036
Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

\cs_new_protected:Npn \__regex_replacement_rbrace:N #1
\begin{verbatim}
  \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
  \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
  \int_decr:N \l__regex_replacement_csnames_int
  \else:
  \__regex_replacement_normal:n {#1}
  \fi:
\end{verbatim}

Characters in replacement

Here, \#1 is a letter among \texttt{BEMTPUDSLOA} and \#2\#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside \c{...} or \u{...}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

\begin{verbatim}
\cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
\begin{verbatim}
  \token_if_eq_meaning:NNTF \prg_do_nothing: #3
  { \__kernel_msg_error:nn { kernel } { replacement-catcode-end } }
  { \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
    { \__kernel_msg_error:nnnn { kernel } { replacement-catcode-in-cs } {#1} {#3}
      #2 #3
    }
    { \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (#1)
      \use:c { __regex_replacement_c_#1:w } #2 #3
    }
  }
\end{verbatim}
\end{verbatim}
We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

\group_begin:

\_regex_replacement_char:n\{\}

The only way to produce an arbitrary character–catcode pair is to use the \lowercase or \uppercase primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: \#3 is the character whose character code to reproduce. We could use \char_generate:nn but only for some catcodes (active characters and spaces are not supported).

\cs_new_protected:Npn \_regex_replacement_char:n\{\}
\{
\tex_lccode:D 0 = '\#3 \scan_stop:
\tex_lowercase:D \{ \tl_build_put_right:Nn \l__regex_build_tl {#1} \}
\}

\_regex_replacement_c_A:w

For an active character, expansion must be avoided, twice because we later do two x-expansions, to unpack \toks for the query, and to expand their contents to tokens of the query.

\char_set_catcode_active:N \^^@
\cs_new_protected:Npn \_regex_replacement_c_A:w
\{
\if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
\int_incr:N \l__regex_balance_int
\fi:
\_regex_replacement_char:n\{\}
\exp_not:n \{ \exp_not:N \^^@ \if_false: \fi: \}
\}

\_regex_replacement_c_B:w

An explicit begin-group token increases the balance, unless within a \c{...} or \u{...} construction. Add the desired begin-group character, using the standard \if_false: trick. We eventually x-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The \exp_after:wN is not strictly needed, but is more consistent with \l3tl-analysis.

\char_set_catcode_group_begin:N \^^@
\cs_new_protected:Npn \_regex_replacement_c_B:w
\{
\if_int_compare:w \l__regex_replacement_cnames_int = 0 \exp_stop_f:
\int_incr:N \l__regex_balance_int
\fi:
\_regex_replacement_char:n\{\}
\exp_not:n \{ \exp_after:wN \^^@ \if_false: \fi: \fi: \}
\}

(End definition for \_regex_replacement_c_B:w.)
This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

\cs_new_protected:Npn \_regex_replacement_c_C:w #1#2
\tl_build_put_right:Nn \l__regex_build_tl
\{ \exp_not:N \exp_not:N \exp_not:c {#2} \}

(End definition for \_regex_replacement_c_C:w.)

Subscripts fit the mould: \lowercase the null byte with the correct category.

\char_set_catcode_math_subscript:N \^^@ \cs_new_protected:Npn \_regex_replacement_c_D:w \{ \_regex_replacement_char:nNN { ^^@ } \}

(End definition for \_regex_replacement_c_D:w.)

Similar to the begin-group case, the second x-expansion produces the bare end-group token.

\char_set_catcode_group_end:N \^^@ \cs_new_protected:Npn \_regex_replacement_c_E:w \{ \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f: \int_decr:N \l__regex_balance_int \fi: \_regex_replacement_char:nNN \exp_not:n \{ \if_false: \{ \fi: \^^@ \} \}

(End definition for \_regex_replacement_c_E:w.)

Simply \lowercase a letter null byte to produce an arbitrary letter.

\char_set_catcode_letter:N \^^@ \cs_new_protected:Npn \_regex_replacement_c_L:w \{ \_regex_replacement_char:nNN \{ \^^@ \}

(End definition for \_regex_replacement_c_L:w.)

No surprise here, we lowercase the null math toggle.

\char_set_catcode_math_toggle:N \^^@ \cs_new_protected:Npn \_regex_replacement_c_M:w \{ \_regex_replacement_char:nNN \{ \^^@ \}

(End definition for \_regex_replacement_c_M:w.)

Lowercase an other null byte.

\char_set_catcode_other:N \^^@ \cs_new_protected:Npn \_regex_replacement_c_O:w \{ \_regex_replacement_char:nNN \{ \^^@ \}

(End definition for \_regex_replacement_c_O:w.)
For macro parameters, expansion is a tricky issue. We need to prepare for two \textit{x}-expansions and passing through various macro definitions. Note that we cannot replace one \texttt{\exp_not:n} by doubling the macro parameter characters because this would misbehave if a mischievous user asks for \texttt{c\{cP\#}}, since that macro parameter character would be doubled.

\begin{verbatim}
26373 \char_set_catcode_parameter:N \^^@
26374 \cs_new_protected:Npn \__regex_replacement_c_P:w
26375 { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
26378 }
(End definition for \__regex_replacement_c_P:w.)
\end{verbatim}

Spaces are normalized on input by \LaTeX{} to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

\begin{verbatim}
26379 \cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
26380 { \if_int_compare:w '#2 = 0 \exp_stop_f:
26382 \__kernel_msg_error:nn { kernel } { replacement-null-space }
26383 \fi:
26384 \tex_lccode:D '{ \#2 \scan_stop:
26385 \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {~} }
26386 }
(End definition for \__regex_replacement_c_S:w.)
\end{verbatim}

No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don’t cause trouble in alignment settings.

\begin{verbatim}
26387 \char_set_catcode_alignment:N \^^@
26388 \cs_new_protected:Npn \__regex_replacement_c_T:w
26389 { \__regex_replacement_char:nNN { \^^@ } }
(End definition for \__regex_replacement_c_T:w.)
\end{verbatim}

Simple call to \texttt{\__regex_replacement_char:nNN} which lowercases the math superscript \texttt{\~@}.

\begin{verbatim}
26390 \char_set_catcode_math_superscript:N \^^@
26391 \cs_new_protected:Npn \__regex_replacement_c_U:w
26392 { \__regex_replacement_char:nNN { \~@ } }
(End definition for \__regex_replacement_c_U:w.)
\end{verbatim}

Restore the catcode of the null byte.

\texttt{\group_end:}
40.6.7 An error

Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```latex
\cs_new_protected:Npn \__regex_replacement_error:NNN #1#2#3
{ \__kernel_msg_error:nnx { kernel } { replacement-#1 } {#3} #2 #3 }
```

(End definition for `\__regex_replacement_error:NNN`.)

40.7 User functions

```latex
\regex_new:N
Before being assigned a sensible value, a regex variable matches nothing.

```latex
\cs_new_protected:Npn \regex_new:N #1
{ \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 229.)

```latex
\l_tmpa_regex \l_tmpb_regex \g_tmpa_regex \g_tmpb_regex
The usual scratch space.

```latex
\regex_new:N \l_tmpa_regex \regex_new:N \l_tmpb_regex \regex_new:N \g_tmpa_regex \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 231.)

```latex
\regex_set:Nn \regex_gset:Nn \regex_const:Nn
Compile, then store the result in the user variable with the appropriate assignment function.

```latex
\cs_new_protected:Npn \regex_set:Nn #1#2
{ \__regex_compile:n {#2} \tl_set_eq:NN #1 \l__regex_internal_regex }

```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 229.)

```latex
\regex_show:N \regex_show:n
User functions: the n variant requires compilation first. Then show the variable with some appropriate text. The auxiliary is defined in a different section.

```latex
\cs_new_protected:Npn \regex_show:n #1
{ \__regex_compile:n {#1} \__regex_show:N \l__regex_internal_regex }
```

```latex
\cs_new_protected:Npn \regex_show:N \l__regex_internal_regex
```

1041
\msg_show:nxxxx { LaTeX / kernel } { show-regex }
{ \tl_to_str:n {#1} } { }
{ \l__regex_internal_a_tl } { }
\cs_new_protected:Npn \regex_show:N #1
{ \__kernel_chk_defined:NT #1
  { \__regex_show:N #1
    \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
    { } { \token_to_str:N #1 }
    { \l__regex_internal_a_tl } { }
  }
}
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
{ \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
  \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
  \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
}
(End definition for \regex_show:N and \regex_show:n. These functions are documented on page 229.)

\regex_match:nnTF
\regex_match:NnTF
Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to \prg_return_:true: or false.
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
{ \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
  \__regex_return:
}
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
{ \__regex_if_match:nn { \__regex_build:N #1 } {#2}
  \__regex_return:
}
(End definition for \regex_match:nnTF and \regex_match:NnTF. These functions are documented on page 229.)

\regex_count:nnN
\regex_count:NnN
Again, use an auxiliary whose first argument builds the NFA.
\cs_new_protected:Npn \regex_count:nnN #1
{ \__regex_count:nnN { \__regex_build:n {#1} } }
\cs_new_protected:Npn \regex_count:NnN #1
{ \__regex_count:nnN { \__regex_build:N #1 } }
(End definition for \regex_count:nnN and \regex_count:NnN. These functions are documented on page 230.)

\regex_extract_once:nnN
\regex_extract_once:nNTF
\regex_extract_once:NnNTF
\regex_extract_all:nnN
\regex_extract_all:nNTF
\regex_extract_all:NnNTF
\regex_replace_once:nnN
\regex_replace_once:nNTF
\regex_replace_once:NnNTF
\regex_replace_all:nnN
\regex_replace_all:nNTF
\regex_replace_all:NnNTF
\regex_split:nnN
\regex_split:nNTF
\regex_split:NnNTF
We define here 40 user functions, following a common pattern in terms of :nnN auxiliaries, defined in the coming subsections. The auxiliary is handed \_\_regex_build:n or \_\_regex_build:N with the appropriate regex argument, then all other necessary arguments (replacement text, token list, etc. The conditionals call \_\_regex_return: to return either true or false once matching has been performed.
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
{ \cs_new_protected:Npn #2 #2 #1 { #1 { \_\_regex_build:n {##1} } }
  \cs_new_protected:Npn #3 #3 #1 { #1 { \_\_regex_build:N #1 } }
  \prg_new_protected_conditional:Npnn \regex_match:nn #2 #2 #2 #2 #2 #3 { T , F , TF }
}
1042
40.7.1 Variables and helpers for user functions

\l__regex_match_count_int

The number of matches found so far is stored in \l__regex_match_count_int. This is only used in the \regex_count:nnN functions.

\l__regex_min_submatch_int, \l__regex_submatch_int, and \l__regex_zeroth_submatch_int

The end-points of each submatch are stored in two arrays whose index (submatch) ranges from \l__regex_min_submatch_int (inclusive) to \l__regex_submatch_int (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at zeroth_submatch. The entry \l__regex_zeroth_submatch_int in \g__regex_submatch_prev_intarray holds the position at which that match attempt started: this is used for splitting and replacements.

\g__regex_submatch_begin_intarray, \g__regex_submatch_end_intarray

Hold the place where the match attempt begun and the end-points of each submatch.
\_\_regex\_return: This function triggers either \texttt{prg\_return\_false} or \texttt{prg\_return\_true} as appropriate to whether a match was found or not. It is used by all user conditionals.

\begin{verbatim}
\cs_new_protected:Npn \_\_regex\_return:
{ \if_meaning:w \c_true_bool \g__regex\_success\_bool \prg\_return\_true:
\else:
 \prg\_return\_false:
 \fi:
}
\end{verbatim}

(End definition for \_\_regex\_return:.)

40.7.2 Matching

\_\_regex\_if\_match:nn We don’t track submatches, and stop after a single match. Build the NFA with \#1, and perform the match on the query \#2.

\begin{verbatim}
\cs_new_protected:Npn \_\_regex\_if\_match:nn #1#2
{ \group_begin:
 \_\_regex\_disable\_submatches:
 \_\_regex\_single\_match:
 #1
 \_\_regex\_match:n {#2}
 \group_end:
}
\end{verbatim}

(End definition for \_\_regex\_if\_match:nn.)

\_\_regex\_count:nnN Again, we don’t care about submatches. Instead of aborting after the first “longest match” is found, we search for multiple matches, incrementing \texttt{l\_\_regex\_match\_-count\_int} every time to record the number of matches. Build the NFA and match. At the end, store the result in the user’s variable.

\begin{verbatim}
\cs_new_protected:Npn \_\_regex\_count:nnN #1#2#3
{ \group_begin:
 \_\_regex\_disable\_submatches:
 \int_zero:N \l__regex\_match\_count\_int
 \_\_regex\_multi\_match:n { \int_incr:N \l__regex\_match\_count\_int }
 #1
 \_\_regex\_match:n {#2}
 \exp_args:NNNo
 \group_end:
 \int_set:Nn #3 { \int_use:N \l__regex\_match\_count\_int }
}
\end{verbatim}

(End definition for \_\_regex\_count:nnN.)

40.7.3 Extracting submatches

\_\_regex\_extract\_once:nn \_\_regex\_extract\_all:nnN Match once or multiple times. After each match (or after the only match), extract the submatches using \_\_regex\_extract:. At the end, store the sequence containing all the submatches into the user variable \#3 after closing the group.
\textit{\textbackslash{cs\_new\_protected:Npn}} \texttt{\textbackslash{__regex\_extract\_once:nnN}} \#1\#3
\{
\group\_begin:
\texttt{\textbackslash{\_\_regex\_single\_match:}}
\#1
\texttt{\_\_regex\_match:n} \{\#2\}
\texttt{\_\_regex\_extract:}
\texttt{\_\_regex\_group\_end\_extract\_seq:N} \#3
\}
\textit{\textbackslash{cs\_new\_protected:Npn}} \texttt{\_\_regex\_extract\_all:nnN} \#1\#3
\{
\group\_begin:
\texttt{\_\_regex\_multi\_match:n} \{ \texttt{\_\_regex\_extract:} \}
\#1
\texttt{\_\_regex\_match:n} \{\#2\}
\texttt{\_\_regex\_group\_end\_extract\_seq:N} \#3
\}

(End definition for \texttt{\_\_regex\_extract\_once:nnN} and \texttt{\_\_regex\_extract\_all:nnN}.)

\texttt{\_\_regex\_split:nnN} Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement \texttt{\_\_regex\_submatch\_int}, which controls which matches will be used.

\textit{\textbackslash{cs\_new\_protected:Npn}} \texttt{\_\_regex\_split:nnN} \#1\#3
\{
\group\_begin:
\texttt{\_\_regex\_multi\_match:n}
\{
\if_int\_compare:w
\texttt{\_\_regex\_start\_pos\_int} < \texttt{\_\_regex\_success\_pos\_int}
\texttt{\_\_regex\_extract:}
\texttt{\_\_kernel\_intarray\_gset:Nnn} \texttt{\_\_regex\_submatch\_begin\_intarray}
\{ \texttt{\_\_regex\_zeroth\_submatch\_int} \} \{ 0 \}
\texttt{\_\_kernel\_intarray\_gset:Nnn} \texttt{\_\_regex\_submatch\_end\_intarray}
\texttt{\_\_regex\_zeroth\_submatch\_int} \}
\}
\texttt{\_\_kernel\_intarray\_gset:Nnn} \texttt{\_\_regex\_submatch\_begin\_intarray}
\{ \texttt{\_\_regex\_zeroth\_submatch\_int} \}
\texttt{\_\_regex\_start\_pos\_int}
\}
\fi:
\}

\#1
\texttt{\_\_regex\_match:n} \{\#2\}
\assert_int:n \{ \texttt{\_\_regex\_curr\_pos\_int} = \texttt{\_\_regex\_max\_pos\_int} \}
\texttt{\_\_kernel\_intarray\_gset:Nnn} \texttt{\_\_regex\_submatch\_begin\_intarray}
\{ \texttt{\_\_regex\_submatch\_int} \} \{ 0 \}
\texttt{\_\_kernel\_intarray\_gset:Nnn} \texttt{\_\_regex\_submatch\_end\_intarray}
The end-points of submatches are stored as entries of two arrays from \texttt{\l__regex_min_submatch_int} to \texttt{\l__regex_max_submatch_int} (exclusive). Extract the relevant ranges into \texttt{\l__regex_internal_a_tl}. We detect unbalanced results using the two flags \texttt{\_regex_begin} and \texttt{\_regex_end}, raised whenever we see too many begin-group or end-group tokens in a submatch.

\begin{verbatim}
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
\flag_clear:n { \_regex_begin } \flag_clear:n { \_regex_end } \seq_set_from_function:NnN \l__regex_internal_seq
{ \int_step_function:nnN { \l__regex_min_submatch_int } \l__regex_max_submatch_int - 1 }
\__regex_extract_seq_aux:n \int_compare:nNnF
{ \flag_height:n { \_regex_begin } + \flag_height:n { \_regex_end } } = 0
\__kernel_msg_error:nnxxx { kernel } { result-unbalanced } \{ splitting-or-extracting-submatches \}
\\{ \flag_height:n { \_regex_end } \} \\{ \flag_height:n { \_regex_begin } \}
\seq_set_map:NNn \l__regex_internal_seq \l__regex_internal_seq {##1}
\exp_args:NNNo \group_end:
\tl_set:Nn #1 { \l__regex_internal_seq }
\end{verbatim}

(End definition for \texttt{\_regex_group_end_extract_seq:N}.)

The \texttt{\_regex_extract_seq_aux:n} auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

\begin{verbatim}
\__regex_extract_seq_aux:ww
\end{verbatim}
Our task here is to extract from the property list \( \text{l__regex_success_submatches_prop} \) the list of end-points of submatches, and store them in appropriate array entries, from \( \text{l__regex_zeroth_submatch_int} \) upwards. We begin by emptying those entries. Then for each \( \langle \text{key} \rangle - \langle \text{value} \rangle \) pair in the property list update the appropriate entry. This is somewhat a hack: the \( \langle \text{key} \rangle \) is a non-negative integer followed by < or >, which we use in a comparison to \(-1\). At the end, store the information about the position at which the match attempt started, in \( \text{g__regex_submatch_prev_intarray} \).

\( \text{cs_new:lpn \__regex_extract_seq_aux:n \#1} \)
\( \text{\exp_after:wN \__regex_extract_seq_aux:ww \int_value:w \__regex_submatch_balance:n \{ \#1 \}; \#1;} \)
\( \text{\cs_new:lpn \__regex_extract_seq_aux:ww \#1; \#2;} \)
\( \text{\if_int_compare:w \#1 < 0 \exp_stop_f:} \)
\( \text{\flag_raise:n { __regex_end } \prg_replicate:nn {-\#1} { \exp_not:n { { \if_false: } \fi: } } \fi:} \)
\( \text{\if_int_compare:w \#1 > 0 \exp_stop_f:} \)
\( \text{\flag_raise:n { __regex_begin } \prg_replicate:nn {\#1} { \exp_not:n { \if_false: { \fi: } } } \fi:} \)

(End definition for \( \__regex_extract_seq_aux:n \) and \( \__regex_extract_seq_aux:ww \)).
End definition for \texttt{\_regex_extract:}, \texttt{\_regex_extract_b:wn}, and \texttt{\_regex_extract_e:wn.}

\subsection*{40.7.4 Replacement}

\texttt{\_regex_replace_once:nnN} Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user’s variable after closing the group: this step involves an additional \texttt{x}-expansion, and checks that braces are balanced in the final result.

(End definition for \texttt{\_regex_replace_once:nnN.})
\__regex_replace_all:nnN  Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from \l__regex_min_submatch_int to \l__regex_submatch_int hold information about submatches of every match in order; each match corresponds to \l__regex_capturing_group_int consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

\begin{verbatim}
\cs_new_protected:Npn \__regex_replace_all:nnN #1#2#3
\group_begin:
\__regex_multi_match:n { \__regex_extract: } #1
\__regex_replacement:n {#2}
\exp_args:No \__regex_match:n {#3}
\int_set:Nn \l__regex_balance_int {0}
\int_step_function:nnnN { \l__regex_min_submatch_int } \l__regex_capturing_group_int { \l__regex_submatch_int - 1 }
\__regex_replacement_balance_one_match:n
\tl_set:Nx \l__regex_internal_a_tl {\int_step_function:nnnN { \l__regex_min_submatch_int } \l__regex_capturing_group_int { \l__regex_submatch_int - 1 }
\__regex_replacement_do_one_match:n \__regex_query_range:nn \l__regex_start_pos_int \l__regex_max_pos_int }
\__regex_group_end_replace:N #3
\end{verbatim}

(End definition for \__regex_replace_all:nnN.)

\__regex_group_end_replace:N  If the brace balance is not 0, raise an error. Then set the user’s variable #1 to the x-expansion of \l__regex_internal_a_tl, adding the appropriate braces to produce a balanced result. And end the group.

\begin{verbatim}
\cs_new_protected:Npn \__regex_group_end_replace:N #1
\group_begin:
\if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
\else:
\__kernel_msg_error:nnxxx { kernel } { result-unbalanced } \l__regex_replacing
\int_max:nn { - \l__regex_balance_int } \l__regex_max_pos_int
\__regex_start_pos_int \l__regex_max_pos_int
\l__regex_query_range:nn
\int_max:nn { \l__regex_balance_int } \l__regex_balance_int
\fi:
\use:x
\end{verbatim}

1049
\group_end: \
\tl_set:Nn \exp_not:N #1 
\{ 
  \if_int_compare:w \l__regex_balance_int < 0 \exp_stop_f: 
  \prg_replicate:nn { - \l__regex_balance_int } 
      \{ \if_false: \fi: \} 

  \fi: 
  \l__regex_internal_a_tl 
  \if_int_compare:w \l__regex_balance_int > 0 \exp_stop_f: 
  \prg_replicate:nn { \l__regex_balance_int } 
      \{ \if_false: \fi: \} 

  \fi: 
\}
\}

(End definition for \__regex_group_end_replace:N.)

40.7.5 Storing and showing compiled patterns

40.8 Messages

Messages for the preparsing phase.

\use:x  
\{ 
  \__kernel_msg_new:nnnn { kernel } { trailing-backslash } 
  \{ Trailing-escape-char-\iow_char:N\'~in~regex-or-replacement. \} 
  \__kernel_msg_new:nnnn { kernel } { x-missing-rbrace } 
  \{ Missing-brace-\iow_char:N\}'~in~regex- 
    '...\iow_char:N\x\iow_char:N\{...#1'. 
  \} 
  \__kernel_msg_new:nnnn { kernel } { x-overflow } 
  \{ Character-code-#1-too-large-in- 
    \iow_char:N\x\iow_char:N\{#2\iow_char:N\}-regex. 
  \} 
\} 

Invalid quantifier.

\__kernel_msg_new:nnnn { kernel } { invalid-quantifier } 
\{ Braced-quantifier-\'#1\'-may-not-be-followed-by-\'#2\'. \} 
\{ The-character-\'#2\'-is-invalid-in-the-braced-quantifier-\'#1\'.- 
  The-only-valid-quantifiers-are-\'*\',-\'?,-\'+',-\'{<int>}',- 
  \'{<min>,\'}-and-\'{<min>,<max}>\',-optionally-followed-by-\'?\'. 
\} 

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

\__kernel_msg_new:nnnn { kernel } { missing-rbrack } 
\{ Missing-right-bracket-inserted-in-regular-expression. \} 
\{ LaTeX-was-given-a-regular-expression-where-a-character-class- 
  was-started-with-\'[\',-but-the-matching-\']\'-is-missing. 

1050
Some escaped alphanumerics are not allowed everywhere.

\__kernel_msg_new:nnnn { kernel } { bad-escape }

\__regex_if_in_cs:TF \{ within-a-control-sequence. \}
\__regex_if_in_class:TF \{ in-a-character-class. \}
\__regex_if_in_class:TF \{ following-a-category-test. \}

\__kernel_msg_new:nnnn { kernel } { range-missing-end }
\__kernel_msg_new:nnnn { kernel } { range-backwards }

Range errors.
In ranges of characters '[x-y]' appearing in character classes, the first character code must not be larger than the second. Here, '#1' has character code \int_eval:n { '#1'}, while '#2' has character code \int_eval:n { '#2'}. 

Errors related to \c and \u.

\__kernel_msg_new:nnnn { kernel } { c-bad-mode }
{ Invalid-nested '\iow_char:N\c' escape in regular expression. }
{ The '\iow_char:N\c' escape cannot be used within a control sequence test '\iow_char:N\c{...}' nor another category test. To combine several category tests, use '\iow_char:N\c[...]'. }

\__kernel_msg_new:nnnn { kernel } { c-C-invalid }
{ '\iow_char:N\cC' should be followed by '.', 'or', '!', not '-' . }
{ The '\iow_char:N\cC' construction restricts the next item to be a control sequence or the next group to be made of control sequences. It only makes sense to follow it by '.' or by a group. }

\__kernel_msg_new:nnnn { kernel } { c-lparen-in-class }
{ Catcode test cannot apply to group in character class }
{ Construction such as '\iow_char:N\c(abc)' are not allowed inside a class '[...]' because classes do not match multiple characters at once. }

\__kernel_msg_new:nnnn { kernel } { c-missing-rbrace }
{ Missing-right-brace-inserted-for- '\iow_char:N\c' escape. }
{ LaTeX was given a regular expression \iow_char:N\c{...} -construction was not ended with a closing brace '\iow_char:N\}'.

\__kernel_msg_new:nnnn { kernel } { c-missing-rbrack }
{ Missing-right-bracket-inserted-for- '\iow_char:N\c' escape. }
{ A construction '\iow_char:N\c[...]' -appears in a regular expression, but the closing ']' is not present. }

\__kernel_msg_new:nnnn { kernel } { c-missing-category }
{ Invalid-character '-', '#1' -following- '\iow_char:N\c' escape. }
{ In regular expressions, the '-' '\iow_char:N\c' escape sequence may only be followed by a left brace, a left bracket, or a capital letter representing a character category, namely one of 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. }

\__kernel_msg_new:nnnn { kernel } { c-trailing }
{ Trailing-category-code-escape- '\iow_char:N\c'... }
{ A regular expression ends with '\iow_char:N\c' followed by a letter. It will be ignored. }

1052
\_\_kernel_msg_new:nnnn { kernel } { u-missing-lbrace }
{ Missing-left-brace-following-\textbackslash iow_char:N\textbackslash u'-escape. }

\_\_kernel_msg_new:nnnn { kernel } { u-missing-rbrace }
{ Missing-right-brace-inserted-for-\textbackslash iow_char:N\textbackslash u'-escape. }

LaTeX
\str_if_eq:eeTF { } {#2}
{ reached-the-end-of-the-string- }
{ encountered-an-escaped-alphanumeric-character \textbackslash iow_char:N\textbackslash #2- }
when-parsing-the-argument-of-an-
\textbackslash iow_char:N\textbackslash \{...\}'-escape.

Errors when encountering the POSIX syntax [:...:].
\_\_kernel_msg_new:nnnn { kernel } { posix-unsupported }
{ POSIX-collating-element-\textbackslash [#1 - #1]-not-supported. }

\_\_kernel_msg_new:nnnn { kernel } { posix-unknown }
{ POSIX-class-\textbackslash [:#1:]-unknown. }

\_\_kernel_msg_new:nnnn { kernel } { posix-missing-close }
{ Missing-closing-\textbackslash [:#1:]-for-POSIX-class. }

The-POSIX-syntax-\textbackslash #1'-must-be-followed-by-\textbackslash :\]'-not-\textbackslash #2'.

In various cases, the result of a \texttt{l3regex} operation can leave us with an unbalanced
token list, which we must re-balance by adding begin-group or end-group character
tokens.
\_\_kernel_msg_new:nnnn { kernel } { result-unbalanced }
{ Missing-brace-inserted-when-\textbackslash #1. }

LaTeX-was-asked-to-do-some-regular-expression-operation,-
and-the-resulting-token-list-would-not-have-the-same-number-of-
begin-group-and-end-group-tokens.-Braces-were-inserted:-
\#2-left,-\#3-right.

Error message for unknown options.
\_\_kernel_msg_new:nnnn { kernel } { unknown-option }
{ Unknown-option-\textbackslash #1'-for-regular-expressions. }

The-only-available-option-is-'case-insensitive',-toggled-by-
Errors in the replacement text.

\__kernel_msg_new:nnnn { kernel } { replacement-c }
\{ Misused-`\iow_char:N\c'\command-in-a-replacement-text. \}
\{ In-a-replacement-text,-the-`\iow_char:N\c'\escape-sequence-
can-be-followed-by-one-of-the-letters-'ABCDELMOPSTU'-
or-a-brace-group,-not-by-`#1'. \}

\__kernel_msg_new:nnnn { kernel } { replacement-u }
\{ Misused-`\iow_char:N\u'\command-in-a-replacement-text. \}
\{ In-a-replacement-text,-the-`\iow_char:N\u'\escape-sequence-
must-be-followed-by-a-brace-group-holding-the-name-of-the-
variable-to-use. \}

\__kernel_msg_new:nnnn { kernel } { replacement-g }
\{ Missing-brace-for-the-`\iow_char:N\g'\construction-
in-a-replacement-text. \}
\{ In-the-replacement-text-for-a-regular-expression-search,-
submatches-are-represented-either-as-`\iow_char:N \g{dd..d}',-
or-`\d',-where-`d'\are-single-digits.-Here,-a-brace-is-missing. \}

\__kernel_msg_new:nnnn { kernel } { replacement-catcode-end }
\{ Missing-character-for-the-`\iow_char:N\c<category><character>'-
construction-in-a-replacement-text. \}
\{ In-a-replacement-text,-the-`\iow_char:N\c'\escape-sequence-
can-be-followed-by-one-of-the-letters-'ABCDELMOPSTU'-representing-
the-character-category.-Then,-a-character-must-follow.-LaTeX-
reached-the-end-of-the-replacement-when-looking-for-that. \}

\__kernel_msg_new:nnnn { kernel } { replacement-catcode-escaped }
\{ Escaped-letter-or-digit-after-category-code-in-replacement-text. \}
\{ In-a-replacement-text,-the-`\iow_char:N\c'\escape-sequence-
can-be-followed-by-one-of-the-letters-'ABCDELMOPSTU'-representing-
the-character-category.-Then,-a-character-must-follow,-not-
`\iow_char:N\#2'. \}

1054
\__kernel_msg_new:nnnn \{ kernel \} \{ replacement-catcode-in-cs \}
\{
  Category-code-`\iow\char:N\c#1#3'-ignored-inside-
  `\iow\char:N\c\{\ldots\}'-in-a-replacement-text.
}\{
  In-a-replacement-text,-the-category-codes-of-the-argument-of-
  `\iow\char:N\c\{\ldots\}'-are-ignored-when-building-the-control-
  sequence-name.
}\}
\__kernel_msg_new:nnnn \{ kernel \} \{ replacement-null-space \}
\{ TeX-cannot-build-a-space-token-with-character-code-0. \}
\{ You-asked-for-a-character-token-with-category-space,-
  and-character-code-0,-for-instance-through-
  `\iow\char:N\c\#S\iow\char:N\x00'.-This-specific-case-is-impossible-and-will-be-replaced-
  by-a-normal-space.
}\}
\__kernel_msg_new:nnnn \{ kernel \} \{ replacement-missing-rbrace \}
\{ Missing-right-brace-inserted-in-replacement-text. \}
\{ There- `\int_compare:nTF \{ #1 = 1 \} \{ \var\ \#2 \} \{ \text{were} \} \} - 
  #1-missing-right-`\int_compare:nTF \{ #1 = 1 \} \{ \brace \} \{ \text{braces} \} .
\}
\__kernel_msg_new:nnnn \{ kernel \} \{ replacement-missing-rparen \}
\{ Missing-right-parenthesis-inserted-in-replacement-text. \}
\{ There- `\int_compare:nTF \{ #1 = 1 \} \{ \var\ \#2 \} \{ \text{were} \} \} - 
  #1-missing-right-
  `\int_compare:nTF \{ #1 = 1 \} \{ \parenthesis \} \{ \parentheses \} .
\}
Used when showing a regex.
\__kernel_msg_new:nnnn \{ kernel \} \{ show-regex \}
\{ `-Compiled-regex-
  `\tl_if_empty:nTF \{#1\} \{ \text{variable-} \#2 \} \{ \text{(#1)} \} : 
  \#3
\}
\__regex_msg_repeated:nnN
This is not technically a message, but seems related enough to go there. The arguments are:
  #1 is the minimum number of repetitions; 
  #2 is the number of allowed extra repetitions (-1 for infinite number), and 
  #3 tells us about lazyness.
\cs_new:Npn \__regex_msg_repeated:nnN \#1\#2\#3
\{
  `\str_if_eq:eeF \{ \#1 \#2 \} \{ 1 0 \} 
  \{ - \text{repeated} -
  `\int_case:nNF \{\#2\}
  \{
    \text{-1} \{ \#1-or-more-times,-`\bool_if:nTF \#3 \{ lazy \} \{ greedy \} \}
    \{ 0 \} \{ \#1-times \}
  \}
\}
40.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

```latex
\begin{verbatim}
\cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
{ \__regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
{ \__regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace:nnx #1#2#3
{ \int_compare:nNnF { \int_use:c { g__regex_trace_#1_int } } < {#2}
{ \iow_term:x { Trace:~#3 } }
}
\int_new:N \g__regex_trace_regex_int
\__regex_trace_states:n
\end{verbatim}
```

(End definition for \_regex_trace_push:nnN, \_regex_trace_pop:nnN, and \_regex_trace:nnx.)
41 13box implementation

41.1 Support code

\_\_box\_dim\_eval:w \_\_box\_dim\_eval:n

Evaluating a dimension expression expandably. The only difference with \dim\_eval:n is the lack of \dim\_use:N, to produce an internal dimension rather than expand it into characters.

\cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
\cs_new:Npn \__box_dim_eval:n #1
{ \__box_dim_eval:w #1 \scan_stop: }

(End definition for \_\_box\_dim\_eval:w and \_\_box\_dim\_eval:n.)

41.2 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box\_new:N \box\_new:c

Defining a new \langle box \rangle register: remember that box 255 is not generally available.

\cs_new_protected:Npn \box\_new:N #1
{ \__kernel_chk_if_free_cs:N #1 \cs:w newbox \cs_end: #1 }
\cs_generate_variant:Nn \box\_new:N { c }

\box\_clear:N \box\_clear:c
\box\_gclear:N \box\_gclear:c

Clear a \langle box \rangle register.

\cs_new_protected:Npn \box\_clear:N #1
{ \box\_set_eq:NN #1 \c_empty_box }
\cs_new_protected:Npn \box\_gclear:N #1
{ \box\_gset_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box\_clear:N { c , Nc , cc }
\cs_generate_variant:Nn \box\_gclear:N { c , Nc , cc }

\box\_clear\_new:N \box\_clear\_new:c \box\_gclear\_new:N \box\_gclear\_new:c

Clear or new.

\cs_new_protected:Npn \box\_clear\_new:N #1
{ \if_exist:NTF #1 { \box\_clear\_new:N #1 } { \box\_new:N #1 } }
\cs_new_protected:Npn \box\_gclear\_new:N #1
{ \if_exist:NTF #1 { \box\_gclear\_new:N #1 } { \box\_new:N #1 } }
\cs_generate_variant:Nn \box\_clear\_new:N { c }
\cs_generate_variant:Nn \box\_gclear\_new:N { c }

Assigning the contents of a box to be another box.

\cs_new_protected:Npn \box\_set\_eq:NN #1#2
{ \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box\_set\_eq:cN #1#2
{ \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box\_set\_eq:Nc #1#2
{ \tex_setbox:D \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box\_set\_eq:cc #1#2
{ \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\cs_generate_variant:Nn \box\_set\_eq:NN { c , Nc , cc }
\cs_generate_variant:Nn \box\_set\_eq:cN { c , Nc , cc }
\cs_generate_variant:Nn \box\_set\_eq:Nc { c , Nc , cc }

\box\_set\_eq:NN \
\box\_set\_eq:cN \
\box\_set\_eq:Nc \
\box\_set\_eq:cc

1057
Assigning the contents of a box to be another box, then drops the original box.

\begin{verbatim}
\cs_new_protected:Npn \box_set_eq_drop:NN #1#2
{ \tex_setbox:D #1 \tex_box:D #2 }
\cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
{ \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\end{verbatim}

Copies of the \texttt{cs} functions defined in \texttt{l3basics}.

\begin{verbatim}
\prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c { TF , T , F , p }
\end{verbatim}

41.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a \langle box \rangle register.

\begin{verbatim}
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N { c }
\cs_generate_variant:Nn \box_dp:N { c }
\cs_generate_variant:Nn \box_wd:N { c }
\end{verbatim}

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression \#2 is surrounded by parentheses to catch early termination.

\begin{verbatim}
\cs_new_protected:Npn \box_set_dp:Nn #1#2
{ \tex_setbox:D #1 = \tex_copy:D #1 \box_dp:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_set_dp:Nn { c }
\cs_new_protected:Npn \box_gset_dp:Nn #1#2
{ \box_dp:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_dp:Nn { c }
\cs_new_protected:Npn \box_set_ht:Nn #1#2
{ \tex_setbox:D #1 = \tex_copy:D #1 \box_ht:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_set_ht:Nn { c }
\cs_new_protected:Npn \box_gset_ht:Nn #1#2
{ \box_ht:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_ht:Nn { c }
\cs_new_protected:Npn \box_set_wd:Nn #1#2
{ \tex_setbox:D #1 = \tex_copy:D #1 \box_wd:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_set_wd:Nn { c }
\cs_new_protected:Npn \box_gset_wd:Nn #1#2
{ \box_wd:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_wd:Nn { c }
\end{verbatim}

1058
41.4 Using boxes

Using a ⟨box⟩. These are just \TeX primitives with meaningful names.

\box_use_drop:N \box_use_drop:c \box_use:N \box_use:c

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

\box_move_left:nn \box_move_right:nn \box_move_up:nn \box_move_down:nn

41.5 Box conditionals

The primitives for testing if a ⟨box⟩ is empty/void or which type of box it is.

\if_hbox:N \if_vbox:N \if_box_empty:N

\if_horizontal_p:N \if_horizontal_p:c \if_horizontal:N \if_horizontal:c

\if_vertical_p:N \if_vertical_p:c \if_vertical:N \if_vertical:c

Testing if a ⟨box⟩ is empty/void.

\if_empty_p:N \if_empty_p:c \if_empty:N \if_empty:c

(End definition for \box_new:N and others. These functions are documented on page 235.)

41.6 The last box inserted

Set a box to the previous box.

\box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c
41.7 Constant boxes

\c_empty_box
A box we never use.

\box_new:N \c_empty_box

(End definition for \c_empty_box. This variable is documented on page 237.)

41.8 Scratch boxes

\l_tmpa_box \l_tmpb_box \g_tmpa_box \g_tmpb_box
Scratch boxes.

\box_new:N \l_tmpa_box \box_new:N \l_tmpb_box \box_new:N \g_tmpa_box \box_new:N \g_tmpb_box

(End definition for \l_tmpa_box and others. These variables are documented on page 238.)

41.9 Viewing box contents

\TeX’s \showbox is not really that helpful in many cases, and it is also inconsistent with other \TeX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

\box_show:N \box_show:c \box_show:Nnn \box_show:cnn
Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

\box_log:N \box_log:c \box_log:Nnn \box_log:cnn \__box_log:nNnn
Getting \TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the ε-\TeX extensions are needed.

\box_new:pton \box_show:N \box_show:c \box_show:Nnn \box_show:cnn \__box_log:nNnn

(End definition for \box_show:N, \box_show:Nnn, and \__box_log:nNnn. These functions are documented on page 238.)
The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The \use:n here gives better output appearance. Setting \tracingonline and \errorcontextlines is used to control what appears in the terminal.

27164 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
27165 { \box_if_exist:NTF #2
27166 { \group_begin:
27167 \int_set:Nn \tex_showboxbreadth:D {#3}
27168 \int_set:Nn \tex_showboxdepth:D {#4}
27169 \int_set:Nn \tex_tracingonline:D {#1}
27170 \int_set:Nn \tex_errorcontextlines:D { -1 }
27171 \tex_showbox:D \use:n {#2}
27172 \group_end:
27173 }
27174 { \__kernel_msg_error:nnx { kernel } { variable-not-defined } { \token_to_str:N #2 }
27175 }
27176 }
27177 \cs_generate_variant:Nn \__box_show:NNnn { NNff }
\end{definition} for \__box_show:NNnn.

41.10 Horizontal mode boxes

\hbox:n (The test suite for this command, and others in this file, is m3box002.ltx.)

Put a horizontal box directly into the input stream.

27182 \cs_new_protected:Npn \hbox:n #1
27183 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }
\end{definition} for \hbox:n. This function is documented on page 238.

\hbox_set:Nn \hbox_set:cn \hbox_gset:Nn \hbox_gset:cn
27184 \cs_new_protected:Npn \hbox_set:Nn #1#2
27185 { \tex_setbox:D #1 \tex_hbox:D { \color_group_begin: #2 \color_group_end: } }
27186 \cs_new_protected:Npn \hbox_gset:Nn #1#2
27187 { \tex_global:D \tex_setbox:D #1 \tex_hbox:D { \color_group_begin: #2 \color_group_end: } }
27188 \cs_generate_variant:Nn \hbox_set:Nn { c }
27189 \cs_generate_variant:Nn \hbox_gset:Nn { c }
\end{definition} for \hbox_set:Nn and \hbox_gset:Nn. These functions are documented on page 239.

\hbox_set_to_wd:Nnn \hbox_set_to_wd:cnm \hbox_gset_to_wd:Nnn \hbox_gset_to_wd:cnm
27190 Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

27195 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
27196 { \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
27197 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { Nnn }
Storing material in a horizontal box. This type is useful in environment definitions.

Combining the above ideas.

(End definition for \hbox_set_to_wd:Nn and \hbox_gset_to_wd:Nnn. These functions are documented on page 239.)
Put a horizontal box directly into the input stream.

\hbox_to_wd:nn
\hbox_to_zero:n

\hbox_overlap_left:n
\hbox_overlap_right:n

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

\hbox_unpack:N
\hbox_unpack:c
\hbox_unpack_drop:N
\hbox_unpack_drop:c

Unpacking a box and if requested also clear it.

41.11 Vertical mode boxes

\vbox:n
\vbox_top:n

The following test files are used for this code: m3box003.lvt.

The following test files are used for this code: m3box003.lvt.
Put a vertical box directly into the input stream.

\begin{verbatim}
\cs_new_protected:Npn \vbox_to_ht:nn #1#2
\{ \tex_vbox:D to \__box_dim_eval:n {#1}
\{ \color_group_begin: #2 \par \color_group_end: \}
\}
\cs_new_protected:Npn \vbox_to_zero:n #1
\{ \tex_vbox:D to \c_zero_dim { \color_group_begin: #1 \par \color_group_end: }\}
\end{verbatim}

(End definition for \vbox_to_ht:nn and others. These functions are documented on page \pageref{page:240}.)

Storing material in a vertical box with a natural height.

\begin{verbatim}
\cs_new_protected:Npn \vbox_set:Nn #1#2
\{ \tex_setbox:D #1 \tex_vbox:D { \color_group_begin: #2 \par \color_group_end: }\}
\cs_new_protected:Npn \vbox_gset:Nn #1#2
\{ \tex_global:D \tex_setbox:D #1 \tex_vbox:D { \color_group_begin: #2 \par \color_group_end: }\}
\end{verbatim}

(End definition for \vbox_set:Nn and \vbox_gset:Nn. These functions are documented on page \pageref{page:240}.)

Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

\begin{verbatim}
\cs_new_protected:Npn \vbox_set_top:Nn \vbox_set_top:cn \vbox_gset_top:Nn \vbox_gset_top:cn
\{ \tex_setbox:D #1 \tex_vtop:D { \color_group_begin: #2 \par \color_group_end: }\}
\cs_new_protected:Npn \vbox_gset_top:Nn \vbox_gset_top:cn
\{ \tex_global:D \tex_setbox:D #1 \tex_vtop:D { \color_group_begin: #2 \par \color_group_end: }\}
\end{verbatim}

(End definition for \vbox_set_top:Nn and \vbox_gset_top:Nn. These functions are documented on page \pageref{page:240}.)

Storing material in a vertical box with a specified height.

\begin{verbatim}
\cs_new_protected:Npn \vbox_set_to_ht:Nnn \vbox_set_to_ht:cn \vbox_gset_to_ht:Nnn \vbox_gset_to_ht:cn
\{ \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
\{ \color_group_begin: #3 \par \color_group_end: \}
\}
\end{verbatim}

1064
Storing material in a vertical box. This type is useful in environment definitions.

A combination of the above ideas.
Unpacking a box and if requested also clear it.

\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
\cs_generate_variant:Nn \vbox_unpack:N { c }
\cs_generate_variant:Nn \vbox_unpack_drop:N { c }

(End definition for \vbox_unpack:N and \vbox_unpack_drop:N. These functions are documented on page 241.)

Splitting a vertical box in two.

\cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
{ \tex_setbox:D #1 \tex_vsplit:D #2 to \__box_dim_eval:n {#3} }
\cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }

\cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
{ \tex_global:D \tex_setbox:D #1 \tex_vsplit:D #2 to \__box_dim_eval:n {#3} }
\cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

(End definition for \vbox_set_split_to_ht:NNn and \vbox_gset_split_to_ht:NNn. These functions are documented on page 241.)

41.12 Affine transformations

\fp_new:N \l__box_angle_fp

(End definition for \l__box_angle_fp.)

\fp_new:N \l__box_cos_fp
\fp_new:N \l__box_sin_fp

(End definition for \l__box_cos_fp and \l__box_sin_fp.)

These are the positions of the four edges of a box before manipulation.

\dim_new:N \l__box_top_dim
\dim_new:N \l__box_bottom_dim
\dim_new:N \l__box_left_dim
\dim_new:N \l__box_right_dim

(End definition for \l__box_top_dim and others.)

These are the positions of the four edges of a box after manipulation.

\dim_new:N \l__box_top_new_dim
\dim_new:N \l__box_bottom_new_dim
\dim_new:N \l__box_left_new_dim
\dim_new:N \l__box_right_new_dim

(End definition for \l__box_top_new_dim and others.)

Scratch space, but also needed by some parts of the driver.

\box_new:N \l__box_internal_box
Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer.

```
cs_new_protected:Npn \box_rotate:Nn #1#2
  { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
cs_generate_variant:Nn \box_rotate:Nn { c }
```

```
cs_new_protected:Npn \box_grotate:Nn #1#2
  { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
cs_generate_variant:Nn \box_grotate:Nn { c }
```

```
cs_new_protected:Npn \__box_rotate:NnN #1#2#3
  { #3 #1
    { \fp_set:Nn \l__box_angle_fp {#2}
      \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
      \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
      \__box_rotate:N #1
    }
  }
```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
cs_new_protected:Npn \__box_rotate:N #1
  { \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim }
```

The next step is to work out the $x$ and $y$ coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices $B$, $C$, $D$ and $E$ is illustrated (Figure 1). The vertex $O$ is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point $P$ and angle $\alpha$:

\[
\begin{align*}
  P_x' &= P_x - O_x \\
  P_y' &= P_y - O_y \\
  P_x'' &= (P_x' \cos(\alpha)) - (P_y' \sin(\alpha)) \\
  P_y'' &= (P_x' \sin(\alpha)) + (P_y' \cos(\alpha)) \\
  P_x''' &= P_x'' + O_x + L_x \\
  P_y''' &= P_y'' + O_y
\end{align*}
\]
The “extra” horizontal translation $L_x$ at the end is calculated so that the leftmost point of the resulting box has $x$-coordinate 0. This is desirable as \TeX\ boxes must have the reference point at the left edge of the box. (As $O$ is always $(0,0)$, this part of the calculation is omitted here.)

\begin{verbatim}
\fp_compare:nNnTF { \l__box_sin_fp > \c_zero_fp } { \fp_compare:nNnTF { \l__box_cos_fp > \c_zero_fp } { \__box_rotate_quadrant_one: } { \__box_rotate_quadrant_two: } } { \fp_compare:nNnTF { \l__box_cos_fp < \c_zero_fp } { \__box_rotate_quadrant_three: } { \__box_rotate_quadrant_four: } }
\end{verbatim}

The position of the box edges are now known, but the box at this stage be misplaced relative to the current \TeX\ reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

\begin{verbatim}
\hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
\hbox_set:Nn \l__box_internal_box { \tex_kern:D -\l__box_left_new_dim \hbox:n
\{ \__box_backend_rotate:Nn \l__box_internal_box \l__box_angle_fp \} }
\end{verbatim}

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

\begin{verbatim}
\box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
\box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
\box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim - \l__box_left_new_dim }
\box_use_drop:N \l__box_internal_box
\end{verbatim}

These functions take a general point $(\#1,\#2)$ and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both $x'$ and $y'$ at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

\begin{verbatim}
\cs_new_protected:Npn \__box_rotate_xdir:nnN \#1\#2\#3
\{ \dim_set:Nn \#3
\{ \fp_to_dim:n
\{ \l__box_cos_fp * \dim_to_fp:n \#1
- \l__box_sin_fp * \dim_to_fp:n \#2
\end{verbatim}
Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting $y$-values, whereas the left and right edges need the $x$-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting $y$-values, whereas the left and right edges need the $x$-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting $y$-values, whereas the left and right edges need the $x$-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.
Scaling is potentially different in the two axes.

\fp_new:N \l__box_scale_x_fp
\fp_new:N \l__box_scale_y_fp

(End definition for \l__box_scale_x_fp and \l__box_scale_y_fp.)

Resizing a box starts by working out the various dimensions of the existing box.


The \textit{x}-scaling and resulting box size is easy enough to work out: the dimension is that given as \#2, and the scale is simply the new width divided by the old one.

\fp_set:Nn \l__box_scale_x_fp \dim_to_fp:n \dim_to_fp:n \{ \l__box_right_dim \}

The \textit{y}-scaling needs both the height and the depth of the current box.

\fp_set:Nn \l__box_scale_y_fp \dim_to_fp:n \dim_to_fp:n \{ \l__box_top_dim - \l__box_bottom_dim \}

Hand off to the auxiliary which does the rest of the work.

\cs_new_protected:Nnn \box_resize_set_corners:N \box_resize:N #1
With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the $x$ direction this is relatively easy: just scale the right edge. In the $y$ direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).
When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```latex
\cs_new_protected:Npn \box_scale:Nnn #1#2#3
\{ \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn \}
\cs_generate_variant:Nn \box_scale:Nnn { c }
\cs_new_protected:Npn \box_gscale:Nnn #1#2#3
\{ \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn \}
\cs_generate_variant:Nn \box_gscale:Nnn { c }
\cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
\{ #4 #1
\fp_set:Nn \l__box_scale_x_fp {#2}
\fp_set:Nn \l__box_scale_y_fp {#3}
\__box_scale:N #1
\}
\cs_new_protected:Npn \__box_scale:N #1
\{ \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
\dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
\dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
\dim_zero:N \l__box_left_dim
\dim_set:Nn \l__box_top_new_dim { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
\dim_set:Nn \l__box_bottom_new_dim { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
\dim_set:Nn \l__box_right_new_dim { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
\__box_resize_common:N #1
\}
\cs_new_protected:Np \box_scale:Nnn #1#2#3
\{ \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn \}
\cs_generate_variant:Nn \box_scale:Nnn { c }
```

Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```latex
\cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
\{ \__box_autosize:NnnnN #1 #2 #3 \hbox_set:Nn \}
\cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
\{ \__box_autosize:NnnnN #1 #2 #3 \hbox_gset:Nn \}
\cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nn #1#2#3
\{ \__box_autosize:NnnnN #1 #2 #3 \hbox_set:Nn \}
```

The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

\__box_resize_common:N

\cs_new_protected:Npn \__box_resize_common:N #1
  \hbox_set:Nn \l__box_internal_box
  { \__box_backend_scale:Nnn #1 \l__box_scale_x_fp \l__box_scale_y_fp }

The new height and depth can be applied directly.

\fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
  { \box_set_ht:Nn \l__box_internal_box \l__box_top_new_dim } \box_set_dp:Nn \l__box_internal_box \l__box_bottom_new_dim
\fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
  { \box_set_dp:Nn \l__box_internal_box \l__box_top_new_dim } \box_set_ht:Nn \l__box_internal_box \l__box_bottom_new_dim

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.
42 l3coffins Implementation

42.1 Coffins: data structures and general variables

Scratch variables.
\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl

(the definition for \l__coffin_internal_box, \l__coffin_internal_dim, and \l__coffin_internal_tl)

\c__coffin_corners_prop
The “corners” of a coffin define the real content, as opposed to the \TeX{} bounding box. They all start off in the same place, of course.
\prop_const_from_keyval:Nn \c__coffin_corners_prop
\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl

(End definition for \l__coffin_internal_box, \l__coffin_internal_dim, and \l__coffin_internal_tl)

\c__coffin_poles_prop
Pole positions are given for horizontal, vertical and reference-point based values.
\prop_const_from_keyval:Nn \c__coffin_poles_prop

(End definition for \c__coffin_corners_prop)
\begin{verbatim}
\__coffin_poles_prop
\end{verbatim}

\begin{verbatim}
\__coffin_slope_A_fp \__coffin_slope_B_fp
\end{verbatim}

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

\begin{verbatim}
\__coffin_offset_x_dim \__coffin_offset_y_dim
\end{verbatim}

\begin{verbatim}
\__coffin_pole_a_tl \__coffin_pole_b_tl
\end{verbatim}

\begin{verbatim}
\__coffin_x_dim \__coffin_y_dim \__coffin_x_prime_dim \__coffin_y_prime_dim
\end{verbatim}

## 42.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

\begin{verbatim}
\__coffin_to_value:N
\end{verbatim}
Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

\begingroup
\begin{Verbatim}
\prg_new_conditional:Npnn \coffin_if_exist:N #1 \{ p , T , F , TF \}
\prg_generate_conditional_variant:Nnn \coffin_if_exist:N \{ c \} \{ p , T , F , TF \}
\end{Verbatim}
\endgroup

\begingroup
\begin{Verbatim}
\cs_new_protected:Npn \__coffin_if_exist:NT #1#2
\{ \coffin_if_exist:NTF #1 \{ #2 \} \}
\end{Verbatim}
\endgroup

(End definition for \coffin_if_exist:NT. This function is documented on page 246.)

Clearing coffins means emptying the box and resetting all of the structures.

\begingroup
\begin{Verbatim}
\cs_new_protected:Npn \__coffin_clear:N \#1\#2
\cs_new_protected:Npn \__coffin_gclear:N \{ c \}
\end{Verbatim}
\endgroup

(End definition for \__coffin_clear:N and \__coffin_gclear:N. These functions are documented on page 246.)
Creating a new coffin means making the underlying box and adding the data structures. The `\debug_suspend:` and `\debug_resume:` functions prevent `\prop_gclear_new:c` from writing useless information to the log file.

```latex
\cs_new_protected:Npn \coffin_new:N \#1
\box_new:N \#1
\debug_suspend:
\prop_gclear_new:c { coffin ~ \__coffin_to_value:N \#1 ~ corners }
\prop_gclear_new:c { coffin ~ \__coffin_to_value:N \#1 ~ poles }
\prop_gset_eq:cN { coffin ~ \__coffin_corners_prop }
\prop_gset_eq:cN { coffin ~ \__coffin_poles_prop }
\debug_resume:
```

`\cs_generate_variant:Nn \coffin_new:N { c }`

(End definition for `\coffin_new:N`. This function is documented on page 246.)

Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```latex
\hcoffin_set:Nn \hcoffin_set:cn \hcoffin_gset:Nn \hcoffin_gset:cn
\__coffin_set_vertical:NnnNN
```

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a
whatsit to the start of the vertical box and mess up the location of the T pole (see \TeX\ by Topic for discussion of the \vtop primitive, used to do the measuring).

\begin{verbatim}
\cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
\__coffin_set_vertical:NnnNN #1 {#2} {#3} \vbox_set:Nn \__coffin_update:N
\end{verbatim}

\begin{verbatim}
\cs_generate_variant:Nn \vcoffin_set:Nnn { c }
\cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
\__coffin_set_vertical:NnnNN #1 {#2} {#3} \vbox_gset:Nn \__coffin_gupdate:N
\cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
\cs_new_protected:Npn \__coffin_set_vertical:NnnNN #1#2#3#4#5
\__coffin_if_exist:NT #1
{ #4 #1
\dim_set:Nn \tex_hsize:D {#2}
\dim_set_eq:NN \linewidth \tex_hsize:D
\dim_set_eq:NN \columnwidth \tex_hsize:D
\end{verbatim}

\begin{verbatim}
\hcoffin_set:Nw \hcoffin_set:cw \hcoffin_gset:Nw \hcoffin_gset:cw \hcoffin_set_end: \hcoffin_gset_end:
\end{verbatim}

These are the “begin”/“end” versions of the above: watch the grouping!

\begin{verbatim}
\cs_new_protected:Npn \hcoffin_set:Nw #1
\__coffin_if_exist:NT #1
{ #4 #1
\hbox_set:Nw #1 \color_ensure_current:
\cs_set_protected:Npn \hcoffin_set_end:
{ }
\end{verbatim}

(End definition for \vcoffin_set:Nnn, \vcoffin_gset:Nnn, and \__coffin_set_vertical:NnnNN. These functions are documented on page 247.)
The same for vertical coffins.

\vcoffin_set:Nnw
\vcoffin_set:csw
\vcoffin_gset:Nnw
\vcoffin_gset:csw
\vcoffin_set_vertical:Nnnnnnw
\vcoffin_gset_vertical:Nnnnnnw
\vcoffin_set_end:
\vcoffin_gset_end:

(End definition for \hcoffin_set:Nw and others. These functions are documented on page 247.)
Setting two coffins equal is just a wrapper around other functions.

Special coffins: these cannot be set up earlier as they need \texttt{\coffin_new:N}. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.
The usual scratch space.

\l_tmpa_coffin
\l_tmpb_coffin
\g_tmpa_coffin
\g_tmpb_coffin

The usual scratch space.

42.3 Measuring coffins
\coffin_dp:N
\coffin_dp:c
\coffin_ht:N
\coffin_ht:c
\coffin_wd:N
\coffin_wd:c

Coffins are just boxes when it comes to measurement. However, semantically a separate
set of functions are required.

\cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
\prop_get:cnNF
\__coffin_to_value:N #1 poles
\exp_not:n {#2} \token_to_str:N #1
\tl_set:Nn #3 { { 0pt } { 0pt } { 0pt } { 0pt } }

\__coffin_get_pole:NnN
\__coffin_reset_structure:N
\__coffin_greset_structure:N

Resetting the structure is a simple copy job.

\__coffin_greset_structure:N
\__coffin_get_pole:NnN

A simple wrapper around the recovery of a coffin pole, with some error checking and
recovery built-in.

\__coffin_get_pole:NnN

(End definition for \l_tmpa_coffin and others. These variables are documented on page 256.)

(End definition for \c_empty_coffin, \l_coffin_aligned_coffin, and \l_coffin_aligned_internal_coffin. This variable is documented on page 256.)

(End definition for \l_tmpa_coffin, \l_tmpb_coffin, \g_tmpa_coffin, and \g_tmpb_coffin. These variables are documented on page 256.)

(End definition for \coffin_dp:N, \coffin_ht:N, and \coffin_wd:N. These functions are documented on page 249.)
Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```
\cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
{ \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
\cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
\cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
{ \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
\cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
\cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
{ \__coffin_if_exist:NT #1
  { #4 { coffin - \__coffin_to_value:N #1 - poles }
    {#2}
    { \dim_eval:n {#3} { Opt } { 1000pt } { Opt }
      4pt { Opt } { 0pt } { 1000pt } { Opt }
    }
  }
}
\cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
{ \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
\cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
\cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
{ \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
\cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
\cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
{ \__coffin_if_exist:NT #1
  { #4 { coffin - \__coffin_to_value:N #1 - poles }
    {#2}
    { \dim_eval:n {#3} { Opt } { 1000pt } { Opt }
      4pt { Opt } { 0pt } { 1000pt } { Opt }
    }
  }
}
\cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
{ \prop_put:cnn { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} {#3} }
\cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }
```

(End definition for \__coffin_set_horizontal_pole:Nnn and \__coffin_set_vertical_pole:Nnn and \__coffin_gset_horizontal_pole:Nnn and \__coffin_gset_vertical_pole:Nnn.)
Simple shortcuts.
\cs_new_protected:Npn \__coffin_update:N #1
\{ \__coffin_reset_structure:N #1
\__coffin_update_corners:N #1
\__coffin_update_poles:N #1
\}
\cs_new_protected:Npn \__coffin_gupdate:N #1
\{ \__coffin_greset_structure:N #1
\__coffin_gupdate_corners:N #1
\__coffin_gupdate_poles:N #1
\}
(End definition for \__coffin_update:N and \__coffin_gupdate:N.)
\__coffin_update_corners:N
\__coffin_gupdate_corners:N
\__coffin_update_corners:NN
\__coffin_update_corners:NNN
Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying \TeX box.
\cs_new_protected:Npn \__coffin_update_corners:N #1
\{ \__coffin_update_corners:NN #1 \prop_put:Nnx \}
\cs_new_protected:Npn \__coffin_gupdate_corners:N #1
\{ \__coffin_update_corners:NN #1 \prop_gput:Nnx \}
\cs_new_protected:Npn \__coffin_update_corners:NN #1#2
\{ \exp_args:Nc \__coffin_update_corners:NNN
\{ \__coffin_to_value:N #1 \__coffin_corners \}
#1 #2
\}
\cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
\{ #3 #1
\begin{Verbatim}
#3 #1
\{ tl \}
\{ dim_eval:n \{ box_ht:N #2 \} \}
#3 #1
\{ tr \}
\{ dim_eval:n \{ box_wd:N #2 \} \}
\{ dim_eval:n \{ box_ht:N #2 \} \}
#3 #1
\{ bl \}
\{ dim_eval:n \{ -box_dp:N #2 \} \}
#3 #1
\{ br \}
\{ dim_eval:n \{ box_wd:N #2 \} \}
\{ dim_eval:n \{ -box_dp:N #2 \} \}
\end{Verbatim}
\}
(End definition for \__coffin_update_corners:N and others.)
\__coffin_update_poles:N
\__coffin_gupdate_poles:N
\__coffin_update_poles:NN
\__coffin_update_poles:NNN
This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is
dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

\cs_new_protected:Npn \__coffin_update_poles:N #1
\cs_new_protected:Npn \__coffin_gupdate_poles:N #1
\cs_new_protected:Npn \__coffin_update_poles:NN #1#2
\exp_args:Nc \__coffin_update_poles:NNN
\__coffin_update_poles:NNN

(End definition for \__coffin_update_poles:N and others.)

42.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

1085
The two poles passed here each have four values (as dimensions), \((a, b, c, d)\) and \((a', b', c', d')\). These are arguments 1–4 and 5–8, respectively. In both cases \(a\) and \(b\) are the co-ordinates of a point on the pole and \(c\) and \(d\) define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by \(d/c\) and \(d'/c'\). However, if one of the poles is either horizontal or vertical then one or more of \(c, d, c',\) and \(d'\) are zero and a special case is needed.

The case where the first pole is vertical. So the \(x\)-component of the interaction is at \(a\). There is then a test on the second pole: if it is also vertical then there is an error.

The second pole may still be horizontal, in which case the \(y\)-component of the intersection is \(b'\). If not, \(y = \frac{d'}{c'} (a - a') + b'\) with the \(x\)-component already known to be \(1\).
If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the $x$- and $y$-components interchanged.

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'} (b - b') + a'$$

which is again handled by the same auxiliary.

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the $x$-value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the $y$-value with

$$y = s(x - a) + b$$
\cs_set_protected:Npn \__coffin_calculate_intersection:nnnnnnn #1#2#3#4#5#6
{
  \fp_compare:nNnTF {#1} = {#2}
  { \bool_set_true:N \l__coffin_error_bool }
  {
    \dim_set:Nn \l__coffin_x_dim
    {
      \fp_to_dim:n
      {
        #1 * \dim_to_fp:n {#3}
        - #2 * \dim_to_fp:n {#5}
        + \dim_to_fp:n {#4}
      }
      - \dim_to_fp:n {#6}
    }
    / \dim_to_fp:n {#1 - #2}
  }
  \dim_set:Nn \l__coffin_y_dim
  {
    \fp_to_dim:n
    {
      #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
      + \dim_to_fp:n {#4}
    } \dim_to_fp:n {#6}
  }
}
(End definition for \__coffin_calculate_intersection:nnnnnnn, \__coffin_calculate_intersection:nnnnn, and \__coffin_calculate_intersection:nnnn.)

42.6 Affine transformations

\l__coffin_sin_fp \l__coffin_cos_fp
Used for rotations to get the sine and cosine values.
\fp_new:N \l__coffin_sin_fp
\fp_new:N \l__coffin_cos_fp
(End definition for \l__coffin_sin_fp and \l__coffin_cos_fp.)

\l__coffin_bounding_prop
A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.
\prop_new:N \l__coffin_bounding_prop
(End definition for \l__coffin_bounding_prop.)

\l__coffin_corners_prop \l__coffin_poles_prop
Used to avoid needing to track scope for intermediate steps.
\prop_new:N \l__coffin_corners_prop
\prop_new:N \l__coffin_poles_prop
(End definition for \l__coffin_corners_prop and \l__coffin_poles_prop.)

\l__coffin_bounding_shift_dim
The shift of the bounding box of a coffin from the real content.
\dim_new:N \l__coffin_bounding_shift_dim

1088
These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set \l__coffin_sine_fp and \l__coffin_cosine_fp, which are carried through unchanged for the rest of the procedure.

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.
The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x-direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y-direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

\hbox_set:Nn \l__coffin_internal_box
\tex_kern:D
\dim_eval:n
{ \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
\exp_stop_f:
\box_move_down:nn { \l__coffin_bottom_corner_dim }
{ \box_use:N #1 }

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

\box_set_ht:Nn \l__coffin_internal_box { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
\box_set_dp:Nn \l__coffin_internal_box { 0pt }
\box_set_wd:Nn \l__coffin_internal_box { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
#5 #1 { \box_use_drop:N \l__coffin_internal_box }

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

\prop_map_inline:Nn \l__coffin_corners_prop
{ \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
\prop_map_inline:Nn \l__coffin_poles_prop
{ \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }

Update the coffin data.

#4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
\l__coffin_corners_prop
#4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
\l__coffin_poles_prop
}

(End definition for \coffin_rotate:Nn, \coffin_grotate:Nn, and \__coffin_rotate:NmNHN. These functions are documented on page 248.)

\__coffin_set_bounding:N

The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

\cs_new_protected:Npn \__coffin_set_bounding:N #1
{ \prop_put:Nnx \l__coffin_bounding_prop { tl } { 0pt } { \dim_eval:n { \box_ht:N #1 } } }
\prop_put:Nnx \l__coffin_bounding_prop { tr } { \dim_eval:n { \box_wd:N #1 } }

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

\cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
defines \__coffin_rotate_bounding:nnn
\cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
defines \__coffin_rotate_corner:Nnnn
\cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
defines \__coffin_rotate_pole:Nnnnnn
\cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
defines \__coffin_rotate_vector:nnNN

A rotation function, which needs only an input vector (as dimensions) and an output space. The values \l__coffin_cos_fp and \l__coffin_sin_fp should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

\cs_new_protected:Npn \__coffin_rotate_vector:nnNN

(End definition for \__coffin_set_bounding:N.)

(End definition for \__coffin_rotate_bounding:nnn and \__coffin_rotate_corner:Nnnn.)

(End definition for \__coffin_rotate_pole:Nnnnnn.)

(End definition for \__coffin_rotate_vector:nnNN.)
The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

\__coffin_find_corner_maxima:N
\__coffin_find_corner_maxima_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

\__coffin_find_bounding_shift:
\__coffin_find_bounding_shift_aux:nn

\__coffin_find_bounding_shift:
Shifting the corners and poles of a coffin means subtracting the appropriate values from the $x$- and $y$-components. For the poles, this means that the direction vector is unchanged.

When scaling, the values given have to be turned into absolute values.

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.
The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

Negative $x$-scaling values place the poles in the wrong location: this is corrected here.

(End definition for \_\_coffin_resize\_common:NnnN.)
For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the \texttt{fp} module.

\begin{verbatim}
\cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
\{ \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN \}
\cs_generate_variant:Nn \coffin_scale:Nnn { c }
\cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
\{ \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN \}
\cs_generate_variant:Nn \coffin_gscale:Nnn { c }
\cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
\{ \fp_set:Nn \l__coffin_scale_x_fp {#2} \fp_set:Nn \l__coffin_scale_y_fp {#3} \#4 \#1 \{ \l__coffin_scale_x_fp \} \l__coffin_internal_dim \dim_set:Nn \l__coffin_scaled_total_height_dim \{ \coffin_ht:N \#1 \fp_add:n \coffin_dp:N \#1 \} \dim_set:Nn \l__coffin_scaled_width_dim \fp_sub:n \l__coffin_scaled_total_height_dim \fp_abs:n \l__coffin_scale_y_fp \dim_set:Nn \l__coffin_scaled_width_dim \fp_sub:n \l__coffin_scaled_width_dim \fp_abs:n \l__coffin_scale_x_fp \coffin_wd:N \#1 \__coffin_resize_common:NnnN \#5 \}
\end{verbatim}

(End definition for \texttt{\coffin_scale:Nnn}, \texttt{\coffin_gscale:Nnn}, and \texttt{\coffin_scale:NnnNN}. These functions are documented on page 248.)

\__coffin_scale_vector:nnNN

This function scales a vector from the origin using the pre-set scale factors in \textit{x} and \textit{y}. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

\begin{verbatim}
\cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
\{ \dim_set:Nn \l__coffin_x_dim \fp_to_dim:n \dim_to_fp:n \#1 \l__coffin_scale_x_fp \dim_set:Nn \l__coffin_y_dim \fp_to_dim:n \dim_to_fp:n \#2 \l__coffin_scale_y_fp \}
\end{verbatim}

(End definition for \texttt{\__coffin_scale_vector:nnNN}.)

\__coffin_scale_corner:Nnnn
\__coffin_scale_pole:Nnnn

Scaling both corners and poles is a simple calculation using the preceding vector scaling.
These functions correct for the $x$ displacement that takes place with a negative horizontal scaling.

\begin{verbatim}
\cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
  { \prop_put:Nnx \l__coffin_corners_prop {#2}
      { \dim_eval:n { #3 + \box_wd:N #1 } } {#4} }
\cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
  { \prop_put:Nnx \l__coffin_poles_prop {#2}
      { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
      {#5} {#6} }
\end{verbatim}

(End definition for \_\_coffin_scale_corner:Nnnn and \_\_coffin_scale_pole:Nnnnnn.)

\section{Aligning and typesetting of coffins}

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

\begin{verbatim}
\cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
  { \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} }
\cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
\cs_new_protected:Npn \__coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8
  { \__coffin_align:NnnNnnnnN \l__coffin_aligned_coffin }
\cs_generate_variant:Nn \__coffin_join:NnnNnnnnN { c , Nnnc , cnnc }
\end{verbatim}

Correct the placement of the reference point. If the $x$-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the $x$-offset and the width of the second box. So a second kern may be needed.
The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```latex
\__coffin_reset_structure:N \l__coffin_aligned_coffin
\prop_clear:c
{
  \a@space_tl corners
\}
\__coffin_update_poles:N \l__coffin_aligned_coffin
```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```latex
\dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
{\__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt } \__coffin_offset_poles:Nnn #4 { 0pt } { \l__coffin_offset_y_dim } \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt } \__coffin_offset_corners:Nnn #4 { 0pt } { \l__coffin_offset_y_dim } }
{\__coffin_offset_poles:Nnn #1 { 0pt } { 0pt } \__coffin_offset_poles:Nnn #4 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim } \__coffin_offset_corners:Nnn #1 { 0pt } { \l__coffin_offset_y_dim } \__coffin_offset_corners:Nnn #4 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim } }
\__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin #9 #1 \l__coffin_aligned_coffin
```

(End definition for \coffin_join:NnnNnnnn, \coffin_gjoin:NnnNnnnn, and \__coffin_join:NnnNnnnnN. These functions are documented on page 248.)

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is here also as it is similar but without the structure updates.
The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.
Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.
The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

\cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
\cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
\cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9

(End definition for \__coffin_offset_corners:Nnn and \__coffin_offset_corner:Nnnnn.)
\c__coffin_empty_coffin

An empty-but-horizontal coffin.

\c__coffin_empty_coffin
\tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

(End definition for \c__coffin_empty_coffin.)

\coffin_typeset:Nnnnn
\coffin_typeset:cnmnn

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

\cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
\mode_leave_vertical:
\__coffin_align:NnnnN \c__coffin_empty_coffin { H } { l } #1 #2 #3 #4 #5 \l__coffin_aligned_coffin
\box_use_drop:N \l__coffin_aligned_coffin
}
\cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

(End definition for \coffin_typeset:Nnnnn. This function is documented on page 249.)

42.8 Coffin diagnostics

\l__coffin_display_coffin
\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin

Used for printing coffins with data structures attached.

\l__coffin_display_coffin
\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin

(End definition for \l__coffin_display_coffin, \l__coffin_display_coord_coffin, and \l__coffin_display_pole_coffin.)

\l__coffin_display_handles_prop

This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

\prop_new:N \l__coffin_display_handles_prop
\prop_put:Nnn \l__coffin_display_handles_prop { tl } { { b } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { thc } { { b } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { tr } { { b } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcl } { { vc } { r } { -1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vchc } { { vc } { hc } { 0 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcr } { { vc } { l } { 1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bl } { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bhc } { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { br } { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { TL } { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Thc } { { t } { l } { 1 } { -1 } }

1101
\prop_put:Nnn \l__coffin_display_handles_prop { Tr } \{ \{ \mathit{t} \} \{ \mathit{hc} \} \{ 0 \} \{ -1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Hl } \{ \{ \mathit{vc} \} \{ \mathit{r} \} \{ -1 \} \{ 1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Hhc } \{ \{ \mathit{vc} \} \{ \mathit{hc} \} \{ 0 \} \{ 1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Hr } \{ \{ \mathit{vc} \} \{ 1 \} \{ 1 \} \{ 1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Bl } \{ \{ \mathit{b} \} \{ \mathit{r} \} \{ -1 \} \{ -1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Bhc } \{ \{ \mathit{b} \} \{ \mathit{hc} \} \{ 0 \} \{ -1 \} \} \\
\prop_put:Nnn \l__coffin_display_handles_prop { Br } \{ \{ \mathit{b} \} \{ 1 \} \{ 1 \} \{ -1 \} \}

\l__coffin_display_offset_dim The standard offset for the label from the handle position when displaying handles.
\dim_new:N \l__coffin_display_offset_dim \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

\l__coffin_display_x_dim \l__coffin_display_y_dim As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.
\dim_new:N \l__coffin_display_x_dim \dim_new:N \l__coffin_display_y_dim

\l__coffin_display_poles_prop A property list for printing poles: various things need to be deleted from this to get a “nice” output.
\prop_new:N \l__coffin_display_poles_prop

\l__coffin_display_font_tl Stores the settings used to print coffin data: this keeps things flexible.
\tl_new:N \l__coffin_display_font_tl \tl_set:Nn \l__coffin_display_font_tl { } \% TODO
\tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }

\__coffin_color:n Calls \color, and otherwise does nothing if \color is not defined.
\cs_new_protected:Npn \__coffin_color:n \{ \cs_if_exist:NT \color \{ \color \{ \#1 \} \} \}

(End definition for \l__coffin_display_handles_prop.)

(End definition for \l__coffin_display_offset_dim.)

(End definition for \l__coffin_display_poles_prop.)

(End definition for \l__coffin_display_poles_prop.)

(End definition for \l__coffin_display_font_tl.)

(End definition for \l__coffin_color:n.)
Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.
Printing the poles starts by removing any duplicates, for which the \texttt{H} poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.
This is a dedicated version of `\coffin_attach:Nnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.
\begin{verbatim}
\__coffin_calculate_intersection:Nnn #1 {#2} {#3}
\dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
\dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
\dim_set:Nn \l__coffin_offset_x_dim
  { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
\dim_set:Nn \l__coffin_offset_y_dim
  { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
\hbox_set:Nn \l__coffin_aligned_coffin
  { \box_use:N \l__coffin_display_coffin
    \tex_kern:D -\box_wd:N \l__coffin_display_coffin
    \tex_kern:D \l__coffin_offset_x_dim
    \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
  }
\box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N \l__coffin_display_coffin }
\box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N \l__coffin_display_coffin }
\box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N \l__coffin_display_coffin }
\box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin }
\end{verbatim}

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

\cs_new_protected:Npn \coffin_show_structure:N
    { \__coffin_show_structure:NN \msg_show:nnxxxx }
\cs_generate_variant:Nn \coffin_show_structure:N { c }
\cs_new_protected:Npn \coffin_log_structure:N
    { \__coffin_show_structure:NN \msg_log:nnxxxx }
\cs_generate_variant:Nn \coffin_log_structure:N { c }
\cs_new_protected:Npn \__coffin_show_structure:NN #1#2
    { \__coffin_if_exist:NT #2
      { \token_to_str:N #2
        \iow_newline: >~ ht ~=~ \dim_eval:n { \coffin_ht:N #2 }
        \iow_newline: >~ dp ~=~ \dim_eval:n { \coffin_dp:N #2 }
        \iow_newline: >~ wd ~=~ \dim_eval:n { \coffin_wd:N #2 }
      }
      \prop_map_function:cN
        { coffin ~ \__coffin_to_value:N #2 ~ poles }
        \msg_show_item_unbraced:nn
      }
    }
\cs_new_protected:Npn \coffin_show_structure:NN #1#2
    { \token_to_str:N #2
      \iow_newline: >~ ht ~=~ \dim_eval:n { \coffin_ht:N #2 }
      \iow_newline: >~ dp ~=~ \dim_eval:n { \coffin_dp:N #2 }
      \iow_newline: >~ wd ~=~ \dim_eval:n { \coffin_wd:N #2 }
    }
\end{verbatim}

(End definition for \coffin_display_handles:Nn and others. This function is documented on page 249.)
42.9 Messages

\texttt{\_\_kernel_msg_new:nnnn \{ kernel \} \{ no-pole-intersection \}}

\{ No-intersection-between-coffin-poles. \}

\{ LaTeX-was-asked-to-find-the-intersection-between-two-poles,-
but-they-do-not-have-a-unique-meeting-point:-
the-value-(0pt,-0pt)-will-be-used. \}

\texttt{\_\_kernel_msg_new:nnnn \{ kernel \} \{ unknown-coffin \}}

\{ Unknown-coffin-‘#1’. \}

\texttt{\_\_kernel_msg_new:nnnn \{ kernel \} \{ unknown-coffin-pole \}}

\{ Pole-‘#1’-unknown-for-coffin-‘#2’. \}

\texttt{\_\_kernel_msg_new:nnn \{ kernel \} \{ show-coffin \}}

\{ Size-of-coffin-#1 : #2 \}

\texttt{\_\_color_current_tl}

The color currently active for foreground (text, etc.) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range \([0, 1]\):

- \texttt{gray} \(\langle\text{gray}\rangle\) Grayscale color with the \(\langle\text{gray}\rangle\) value running from 0 (fully black) to 1 (fully white)
- \texttt{cmyk} \(\langle\text{cyan}\rangle\) \(\langle\text{magenta}\rangle\) \(\langle\text{yellow}\rangle\) \(\langle\text{black}\rangle\)
- \texttt{rgb} \(\langle\text{red}\rangle\) \(\langle\text{green}\rangle\) \(\langle\text{blue}\rangle\)

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- \texttt{spot} \(\langle\text{name}\rangle\) \(\langle\text{tint}\rangle\) A pre-defined spot color, where the \(\langle\text{name}\rangle\) should be a pre-defined string color name and the \(\langle\text{tint}\rangle\) should be in the range \([0, 1]\).

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.
**TeXHackers note:** The content of \l__color_current_tl is space-separated as this allows it to be used directly in specials in many common cases. This internal representation is close to that used by the dvips program.

(End definition for \l__color_current_tl.)

\color_group_begin: Grouping for color is the same as using the basic \group_begin: and \group_end: functions. However, for semantic reasons, they are renamed here.

\color_group_end: (End definition for \color_group_begin: and \color_group_end:. These functions are documented on page 251.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

\color_ensure_current: (End definition for \color_ensure_current:. This function is documented on page 251.)

\s__color_stop Internal scan marks.

\s__color_stop: (End definition for \s__color_stop.)

\__color_select:n % \__color_select:V \__color_select:w \__color_select_cmyk:w \__color_select_gray:w \__color_select_rgb:w \__color_select_spot:w Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level experimental material.

\__color_select:n \__color_select:V \__color_select:w \__color_select_cmyk:w \__color_select_gray:w \__color_select_rgb:w \__color_select_spot:w (End definition for \__color_select:n \__color_select:V \__color_select:w \__color_select_cmyk:w \__color_select_gray:w \__color_select_rgb:w \__color_select_spot:w.)

\l__color_current_tl As the setting data is used only for specials, and those are always space-separated, it makes most sense to hold the internal information in that form.

\l__color_current_tl: (End definition for \l__color_current_tl.)
44 l3luatex implementation

44.1 Breaking out to Lua

\__lua_escape:n Copies of primitives.
\__lua_now:n \cs_new_eq:NN \__lua_escape:n \tex_luaescapestring:D
\__lua_now:n \cs_new_eq:NN \__lua_now:n \tex_directlua:D
\__lua_shipout:n \cs_new_eq:NN \__lua_shipout:n \tex_latelua:D

(End definition for \__lua_escape:n, \__lua_now:n, and \__lua_shipout:n.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.
\cs_undefine:N \lua_escape:e
\cs_undefine:N \lua_now:e
\lua_now:n \lua_now:e
\lua_shipout_e:n \cs_new:Npn \lua_now:e #1 { \__lua_now:n {#1} }
\cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }
\cs_new_protected:Npn \lua_shipout_e:n #1 { \__lua_shipout:n {#1} }
\cs_new_protected:Npn \lua_shipout:n #1 { \lua_shipout_e:n { \exp_not:n {#1} } }
\cs_new:Npn \lua_escape:e #1 { \__lua_escape:n {#1} }
\cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }
\sys_if_engine_luatex:F
\clist_map_inline:nn \sys_if_engine_luatex:F
\clist_map_inline:nn { \lua_escape:e, \lua_escape:n }
\clist_map_inline:nn { \lua_shipout:e, \lua_shipout:n }

(End definition for \lua_now:n and others. These functions are documented on page 252.)
44.2 Messages
\__kernel_msg_new:nnnn { kernel } { luatex-required }
{ LuaTeX-engine-not-in-use!-Ignoring-#1. }
}\__kernel_msg_new:nnnn { kernel } { luatex-required }
{ LuaTeX-engine-not-in-use!-Ignoring-#1. }

The feature you are using is only available
with the LuaTeX-engine. LaTeX3 ignored '#1'.

44.3 Lua functions for internal use
(*lua)

Most of the emulation of pdfTeX here is based heavily on Heiko Oberdiek’s pdftex-cmds package.

\l3kernel
Create a table for the kernel’s own use.
\l3kernel = l3kernel or { }

(End definition for l3kernel. This function is documented on page 253.)

Local copies of global tables.

local io = io
local kpse = kpse
local lfs = lfs
local math = math
local md5 = md5
local os = os
local string = string
local tex = tex
local texio = texio
local tonumber = tonumber
local unicode = unicode

Local copies of standard functions.

local abs = math.abs
local byte = string.byte
local floor = math.floor
local format = string.format
local gsub = string.gsub
local lfs_attr = lfs.attributes
local md5_sum = md5.sum
local open = io.open
local os_clock = os.clock
local os_date = os.date
local os_exec = os.execute
local setcatcode = tex.setcatcode
local sprint = tex.sprint
local cprint = tex.cprint
local write = tex.write
local write_nl = texio.write_nl

Newer ConTeXt releases replace the unicode library by utf and since Lua 5.3 we can even use the Lua standard utf8 library.

local utf8_char = (utf8 and utf8.char) or (utf and utf.char) or unicode.utf8.char

1110
Deal with Con\TeXt: doesn’t use kpse library.

local kpse\_find = (resolvers and resolvers\_findfile) or kpse\_find\_file

escapehex

An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdftexcmds but is not currently required here.

local function escapehex(str)
write((gsub(str, ".", function (ch) return format("%02X", byte(ch)) end)))
end

(End definition for escapehex.)

l3kernel.charcat

Creating arbitrary chars using tex.cprint.

local charcat
function charcat(charcode, catcode)
cprint(catcode, utf8\_char(charcode))
end
l3kernel.charcat = charcat

(End definition for l3kernel.charcat. This function is documented on page 253.)

l3kernel.elapsedtime

Simple timing set up: give the result from the system clock in scaled seconds.

local base\_time = 0
local function elapsedtime()
local val = (os\_clock() - base\_time) * 65536 + 0.5
if val > 2147483647 then
  val = 2147483647
end
write(format("%d",floor(val)))
end
l3kernel.elapsedtime = elapsedtime
local function resettimer()
  base\_time = os\_clock()
end
l3kernel.resettimer = resettimer

(End definition for l3kernel.elapsedtime and l3kernel.resettimer. These functions are documented on page 253.)

l3kernel.filedump

Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

local function filedump(name,offset,length)
local file = kpse\_find(name,"tex",true)
if file then
  local length = tonumber(length) or lfs\_attr(file,"size")
  local offset = tonumber(offset) or 0
  local f = open(file,"rb")
  if f then
    if offset > 0 then
      f:seek("set",offset)
    end
    local data = f:read(length)
  escapehex(data)
l3kernel.filemdfivesum Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see pdftexcmds and how it handles strings that have passed through LuaTeX). The file is read in binary mode so that no line ending normalisation occurs.

```lua
local function filemdfivesum(name)
    local file = kpse_find(name, "tex", true)
    if file then
        local f = open(file, "rb")
        if f then
            local data = f:read("*a")
            escapehex(md5_sum(data))
        f:close()
        end
    end
end
l3kernel.filemdfivesum = filemdfivesum
```

(End definition for l3kernel.filemdfivesum. This function is documented on page 253.)

l3kernel.filemoddate See procedure makepdftime in utils.c of pdftex.

```lua
local function filemoddate(name)
    local file = kpse_find(name, "tex", true)
    if file then
        local date = lfs_attr(file, "modification")
        if date then
            local d = os_date("!*t", date)
            if d.sec >= 60 then
                d.sec = 59
            end
            local u = os_date("!*t", date)
            local off = 60 * (d.hour - u.hour) + d.min - u.min
            if d.year ~= u.year then
                if d.year > u.year then
                    off = off + 1440
                elseif d.yday ~= u.yday then
                    if d.yday > u.yday then
                        off = off + 1440
                    end
                end
            end
            local timezone
            if off == 0 then
```
timezone = "Z"
else
  local hours = floor(off / 60)
  local mins = abs(off - hours * 60)
  timezone = format("%+03d", hours)
  .. "":" .. format("%02d", mins) .. ""
end
write("D:
.. format("%04d", d.year)
.. format("%02d", d.month)
.. format("%02d", d.day)
.. format("%02d", d.hour)
.. format("%02d", d.min)
.. format("%02d", d.sec)
.. timezone)
end
end
end
l3kernel.filemoddate = filemoddate

(End definition for l3kernel.filemoddate. This function is documented on page 253.)

l3kernel.filesize A simple disk lookup.
local function filesize(name)
  local file = kpse_find(name, "tex", true)
  if file then
    local size = lfs_attr(file, "size")
    if size then
      write(size)
    end
  end
end
l3kernel.filesize = filesize

(End definition for l3kernel.filesize. This function is documented on page 253.)

l3kernel.strcmp String comparison which gives the same results as pdf\TeX{}’s `\pdfstrcmp`, although the ordering should likely not be relied upon!
local function strcmp(A, B)
  if A == B then
    write("0")
  elseif A < B then
    write("-1")
  else
    write("1")
  end
end
l3kernel.strcmp = strcmp

(End definition for l3kernel.strcmp. This function is documented on page 253.)

l3kernel.shellescape Replicating the pdf\TeX{} log interaction for shell escape.
local function shellescape(cmd)
  local status, msg = os_exec(cmd)
  if status == nil then
write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ")\n")
elseif status == 0 then
write_nl("log","runsystem(" .. cmd .. ")...executed\n")
else
write_nl("log","runsystem(" .. cmd .. ")...failed " .. (msg or ") .. ")\n")
end
end
l3kernel.shellescape = shellescape

(End definition for l3kernel.shellescape. This function is documented on page 253.)

44.4 Generic Lua and font support

A small amount of generic code is used by almost all LuaTEX material so needs to be loaded by the format.

attribute_count_name = "g__alloc_attribute_int"
bytecode_count_name = "g__alloc_bytecode_int"
chunkname_count_name = "g__alloc_chunkname_int"
whatsit_count_name = "g__alloc_whatsit_int"

require("ltluatex")

With the above available the font loader code used by plain TEX and LaTeX when used with LuaTEX can be loaded here. This is thus being treated more-or-less as part of the engine itself.

require("luaotfload-main")
local _void = luaotfload.main()

45 l3unicode implementation

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as \lccode and \uccode values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines. For performance reasons, some of the code here is very low-level: the material is read during loading expl3 in package mode.

\ior_new:N \g__char_data_ior
\bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
{ \group_begin:
Access the primitive but suppress further expansion: active chars are otherwise an issue.
\cs_set:Npn \__char_generate_char:n #1
A fast local implementation for generating characters; the chars may be active, so we prevent further expansion.

\cs_set:Npx \__char_generate:n #1
{ \exp_not:N \tex_unexpanded:D \exp_not:N \exp_after:wN
{ \sys_if_engine_luatex:TF
{ \exp_not:N \tex_directlua:D
{ l3kernel.charcat
( \exp_not:N \tex_number:D #1 ,
\exp_not:N \tex_the:D \tex_catcode:D #1
)
}
}
{ \exp_not:N \tex_Ucharcat:D
#1 ~ \tex_catcode:D #1 ~
}
}
}

Parse the main Unicode data file for two things. First, we want the titlecase exceptions: the one-to-one lowercase mappings it contains are all be covered by the TeX data. Second, we need normalization data: at present, just the canonical \textit{nfd} mappings. Those all yield either one or two codepoints, so the split is relatively easy.

\ior_open:Nn \g__char_data_ior { UnicodeData.txt }
\cs_set_protected:Npn \__char_data_auxi:w #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
{ \tl_if_blank:nF {#6}
{ \tl_if_head_eq_charcode:nNF {#6} < % >
{ \__char_data_auxii:w #1 ; #6 - \q_stop }
}
\__char_data_auxiii:w #1 ;}
\cs_set_protected:Npn \__char_data_auxii:w #1 ; #2 - #3 \q_stop
{ \tl_const:cx
{ c__char_nfd_ \__char_generate_char:n {#1} _tl }
{ \__char_generate:n { "#2 }
\tl_if_blank:nF {#3}
{ \__char_generate:n { "#3 } }
}
}
\cs_set_protected:Npn \__char_data_auxiii:w #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 - \q_stop
{
The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The setup for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For more complex foldings, always store the result, splitting up the two or three code points in the input as required.
For upper- and lowercasing special situations, there is a bit more to do as we also have
title casing to consider, plus we need to stop part-way through the file.
\ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
\cs_set_protected:Npn \__char_data_auxi:w
\#1 \#2 \#3 \#4 \#5 \q_stop
{ \use:n { \__char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
\use:n { \__char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
\str_if_eq:nnF {#3} {#4}
{ \use:n { \__char_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
}
\cs_set_protected:Npn \__char_data_auxii:w
#1 #2 #3 #4 #5 \q_stop
{ \tl_if_empty:nF {#4}
{ \tl_const:cx { c__char_ #2 case_ \__char_generate_char:n {#1} _tl }

{ \__char_generate:n { "#3 }
\__char_generate:n { "#4 }
\tl_if_blank:nF {#5}
{ \__char_generate:n { "#5 } }
}
}
\ior_str_map_inline:Nn \g__char_data_ior
{ \str_if_eq:eeTF {\tl_head:w #1 \c_hash_str \q_stop}
{ \c_hash_str } \q_stop
\str_if_eq:eeTF {#1}
{ \c_hash_str \c_space_tl Conditional-Mappings }
\ior_map_break: }
{ \__char_data_auxi:w #1 \q_stop }
\ior_close:N \g__char_data_ior
\group_end:
}

For the 8-bit engines, the above is skipped but there is still some set up required. As
case changing can only be applied to bytes, and they have to be in the ASCII range, we
define a series of data stores to represent them, and the data are used such that only
these are ever case-changed. We do open and close one file to force allocation of a read:
this keeps all engines in line.
\group_begin:
\cs_set_protected:Npn \__char_tmp:NN #1#2
{\quark_if_recursion_tail_stop:N #2
 \tl_const:cn { c__char_uppercase_ #2 _tl } {#1}
 \tl_const:cn { c__char_lowercase_ #1 _tl } {#2}
 \tl_const:cn { c__char_foldcase_ #1 _tl } {#2}
 \__char_tmp:NN}
\__char_tmp:NN
AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
\ior_open:Nn \g__char_data_ior \{ UnicodeData.txt \}
\ior_close:N \g__char_data_ior
\group_end:

46 \textbf{l3text implementation}

\langle \texttt{/initex | package} \rangle

46.1 Internal auxiliaries

\s__text_stop Internal scan marks.
\scan_new:N \s__text_stop
(End definition for \s__text_stop.)

\q__text_nil Internal quarks.
\quark_new:N \q__text_nil
(End definition for \q__text_nil.)

\__text_quark_if_nil_p:n Branching quark conditional.
\__text_quark_if_nil:nTF
\__kernel_quark_new_conditional:Nn \__text_quark_if_nil:n { TF }
(End definition for \__text_quark_if_nil:nTF.)

\q__text_recursion_tail Internal recursion quarks.
\q__text_recursion_stop
\quark_new:N \q__text_recursion_tail
\quark_new:N \q__text_recursion_stop
(End definition for \q__text_recursion_tail and \q__text_recursion_stop.)
\textbf{functions to gobble up to a quark.}
\cs_new:Npn \__text_use_i_delimit_by_q_recursion_stop:nw
\__text_use_i_delimit_by_q_recursion_stop:nw
#1 #2 \q__text_recursion_stop {#1}
(End definition for \__text_use_i_delimit_by_q_recursion_stop:nw.)

\textbf{functions to query recursion quarks.}
\__kernel_quark_new_test:N \__text_if_recursion_tail_stop:do:N
(End definition for \__text_if_recursion_tail_stop:do:N.)
46.2 Utilities

The idea here is to take a token and ensure that if it’s an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

\group_begin:
\char_set_catcode_active:n { 0 }
\cs_new:Npn \__text_token_to_explicit:N #1
{
\if_catcode:w \exp_not:N #1
\if_catcode:w \scan_stop: \exp_not:N #1
\scan_stop:
\else:
\exp_not:N ^^@
\fi:
\exp_after:wN \__text_token_to_explicit_cs:N
\else:
\exp_after:wN \__text_token_to_explicit_char:N
\fi:
#1
}
\group_end:

For control sequences, we can check for macros versus other cases using \if_meaning:w, then explicitly check for \chardef and \mathchardef.

\cs_new:Npn \__text_token_to_explicit_cs:N #1
{
\exp_after:wN \if_meaning:w \exp_not:N #1 #1
\exp_after:wN \use:nn \exp_after:wN
\__text_token_to_explicit_cs_aux:N
\else:
\exp_after:wN \__text_token_to_explicit_char:N
\fi:
#1
}
\cs_new:Npn \__text_token_to_explicit_cs_aux:N #1
{
\bool_lazy_or:nnTF
{ \token_if_chardef_p:N #1 }
{ \token_if_mathchardef_p:N #1 }
{ \char_generate:nn {#1} }
{ \char_value_catcode:n {#1} }


}

For character tokens, we need to filter out the implicit characters from those that are explicit. That’s done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the \TeX messages are either \langle \text{something} \rangle \ character \langle \text{char} \rangle or the \langle \text{type} \rangle (\text{char}).

\cs_new:Npn \__text_token_to_explicit_char:N #1
{


\if:w
\if_catcode:w \exp_args:No \str_tail:n \token_to_str:N #1 \^ \token_to_str:N #1 \^ \else:
AB \fi:
\exp_after:wN \exp_not:n \else:
\exp_after:wN \_text_token_to_explicit:n \fi:
{#1}
\}
\cs_new:Npn \_text_token_to_explicit:n #1
{\exp_after:wN \_text_token_to_explicit_auxi:w \int_value:w
\if_catcode:w \c_group_begin_token #1 \exp_not:N \c_group_begin_token #1 \1 \else:
\if_catcode:w \c_group_end_token #1 \exp_not:N \c_group_end_token #1 \2 \else:
\if_catcode:w \c_math_toggle_token #1 \exp_not:N \c_math_toggle_token #1 \3 \else:
\if_catcode:w \c_math_subscript_token #1 \exp_not:N \c_math_subscript_token #1 \8 \else:
\if_catcode:w \c_math_superscript_token #1 \exp_not:N \c_math_superscript_token #1 \10 \else:
\if_catcode:w \c_space_token #1 \exp_not:N \c_space_token #1 \10 \else:
\if_catcode:w \c_math_subscript_token #1 \exp_not:N \c_math_subscript_token #1 \8 \else:
\if_catcode:w \c_math_superscript_token #1 \exp_not:N \c_math_superscript_token #1 \10 \else:
4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
\exp_after:wN \token_to_meaning:N \s__text_stop
\}
\cs_new:Npn \_text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
{ \char_generate:nn { \if_int_compare:w #1 < 9 \exp_stop_f:
\exp_after:wN \_text_token_to_explicit_auxii:w \else:
\exp_after:wN \_text_token_to_explicit_auxii:w \fi:
#2 \} \}
\exp_last_unbraced:NNNo \cs_new:Npn \_text_token_to_explicit_auxii:w #1 \{ \tl_to_str:n { character - } \} \{ \'}
\cs_new:Npn \_text_token_to_explicit_auxiii:w #1 - #2 - { \'}
(End definition for \_text_token_to_explicit:N and others.)

\_text_char_catcode:N
An idea from l3char: we need to get the category code of a specific token, not the general case.
\cs_new:Npn \_text_char_catcode:N #1
{ \if_catcode:w \exp_not:N \c_math_toggle_token \exp_not:N \c_math_toggle_token #1
\exp_not:N \c_math_toggle_token \exp_not:N \c_math_toggle_token \exp_not:N \c_math_toggle_token 3
46.3 Configuration variables

Special cases for accents and letter-like symbols, which in some cases will need to be converted further.

(End definition for \text_if_expandable:NTF.)

\text_if_expandable:NTF Test for tokens that make sense to expand here: that is more restrictive than the engine view.

4.6.3 Configuration variables

\l_text_accents_tl \l_text_letterlike_tl Special cases for accents and letter-like symbols, which in some cases will need to be converted further.
\l_text_case_exclude_arg_tl Non-text arguments.
\tl_new:N \l_text_case_exclude_arg_tl
\tl_set:Nn \l_text_case_exclude_arg_tl { \begin \cite \end \label \ref }

(End definition for \l_text_case_exclude_arg_tl. This variable is documented on page 258.)

\l_text_math_arg_tl Math mode as arguments.
\tl_new:N \l_text_math_arg_tl
\tl_set:Nn \l_text_math_arg_tl { \ensuremath }

(End definition for \l_text_math_arg_tl. This variable is documented on page 258.)

\l_text_math_delims_tl Paired math mode delimiters.
\tl_new:N \l_text_math_delims_tl
\tl_set:Nn \l_text_math_delims_tl { $ $ \( \) }

(End definition for \l_text_math_delims_tl. This variable is documented on page 258.)

\l_text_expand_exclude_tl Commands which need not to expand.
\tl_new:N \l_text_expand_exclude_tl
\tl_set:Nn \l_text_expand_exclude_tl { \begin \cite \end \label \ref }

(End definition for \l_text_expand_exclude_tl. This variable is documented on page 258.)

\l__text_math_mode_tl Used to control math mode output: internal as there is a dedicated setter.
\tl_new:N \l__text_math_mode_tl

(End definition for \l__text_math_mode_tl.)
46.4 Expansion to formatted text

Markers for implicit char handling.

Markers for implicit char handling.

After precautions against & tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected and take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.)

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens.
Before we get into the real work, we have to watch out for problematic implicit characters: spaces and grouping tokens. Converting these to explicit characters later would lead to real issues as they are not \(N\)-type. A space is the easy case, so it's dealt with first: just insert the explicit token and continue the loop.

Implicit {/} offer two issues. First, the token could be an implicit brace character: we need to avoid turning that into a brace group, so filter out the cases manually. Then we handle the case where an implicit group is present. That is done in an “open-ended” way: there's the possibility the closing token is hidden somewhere.
The first step in dealing with \textit{N}-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need \texttt{\quark\_if\_recursion\_tail\_stop:N} first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the \textit{N}-type ones against the end marker.
At this stage, either we have a control sequence or a simple character: split and handle.

\begin{macrocode}
\cs_new:Npn \__text_expand_loop:Nw #1
\cs_if_exist:cTF {#1}
  \__text_expand_math_search:NNN #1
\end{macrocode}

% Next we exclude math commands: this is mainly as there \texttt{\textbackslash emph\{might\}} be an
% \texttt{\textbackslash ensuremath}. We also handle accents, which are basically the same issue
% but are kept separate for semantic reasons.
% \begin{macrocode}
\cs_new:Npn \__text_expand_exclude:N #1
\token_if_eq_meaning:NNTF #1 #1
  \__text_expand_math_exclude:NNN #1
\end{macrocode}
Another list of exceptions: these ones take no arguments so are easier to handle.
\texttt{\LaTeX}'s \texttt{\protect} makes life interesting. Where possible, we simply remove it and replace with the "parent" command; of course, the \texttt{\protect} might be explicit, in which case we need to leave it alone if it's required.

\texttt{\texttt{\LaTeX}2e's \texttt{\protect} makes life interesting. Where possible, we simply remove it and replace with the "parent" command; of course, the \texttt{\protect} might be explicit, in which case we need to leave it alone if it's required.}

\texttt{\texttt{\LaTeX}2e's \texttt{\protect} makes life interesting. Where possible, we simply remove it and replace with the "parent" command; of course, the \texttt{\protect} might be explicit, in which case we need to leave it alone if it's required.}
See if there is a dedicated replacement, and if there is, insert it.

\cs_new:Npn \__text_expand_encoding_escape:NN #1 \{{ \__text_expand_replace:N \#1 \}}

\cs_new:Npn \__text_expand_replace:N #1 {\cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } \begin{itemize}
\item \bool_lazy_and:nnTF { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } } { \bool_lazy_or_p:nn { \token_if_cs_p:N #1 } { \token_if_active_p:N #1 } } { \exp_args:Nv \__text_expand_replace:n { l__text_expand_ \token_to_str:N #1 _tl } } { \__text_expand_cs_expand:N #1 } \end{itemize}
\cs_new:Npn \__text_expand_cs_expand:N #1 { \__text_if_expandable:NTF #1 { \token_if_eq_meaning:NNTF #1 \exp_not:n { \__text_expand_noexpand:w } } { \__text_expand_store:n {#1} \__text_expand_loop:w } }
\cs_new:Npn \__text_expand_noexpand:w #1#2 { \__text_expand_noexpand:nn {#1} }
\cs_new:Npn \__text_expand_noexpand:nn #1#2 { #1 \__text_expand_store:n #1 {#2} \__text_expand_loop:w }

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is \exp_not:n, as that must apply to the following balanced text. There might be an \exp_after:wN there, so we check for it.

\cs_new:Npn \__text_expand_cs_expand:N #1 { \__text_if_expandable:NTF #1 { \token_if_eq_meaning:NNTF #1 \exp_not:n { \__text_expand_noexpand:w } } { \__text_expand_store:n {#1} \__text_expand_loop:w } }
\cs_new:Npn \__text_expand_noexpand:w #1#2 { \__text_expand_noexpand:nn {#1} }
\cs_new:Npn \__text_expand_noexpand:nn #1#2 { #1 \__text_expand_store:n #1 {#2} \__text_expand_loop:w }

(End definition for \textexpand:nn and others. This function is documented on page 255.)

\text_declare_expand_equivalent:Nn
\text_declare_expand_equivalent:cn

Create equivalents to allow replacement.
\texttitlecase
\textuppercase
\texttitlecasefirst
\textlowercase
\textuppercase
\texttitlecase
\texttitlecasefirst

As for the expansion code, the business end of case changing is the handling of \N-type tokens. First, we expand the input fully (so the loops here don’t need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be \f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

\cs_set_eq:NN \MakeLowercase \text_lowercase
\MakeLowercase{\enquote*{A} text}
If we use an e-type expansion and wrap each token in \exp_not:n, that would explode: the document command grabs \exp_not:n as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one \exp_not:n, or rather in the kernel version.

As for expansion, collect up the tokens for future use.

The main loop is the standard tl action type.

For a group, we could worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is
rather weakly-defined anyway. So if there is a group, we simply assume that a character
has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid
having too much testing, we use a two-step process here to allow the titlecase functions
to be separate.

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be
done separately from the first real step as otherwise we pick up the wrong delimiter. The
loop here is the same as the expand one, just passing the additional data long. If no
close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

\begin{verbatim}
\cs_new:Npn \__text_change_case_N_type:nnN #1#2#3
\{ \__text_if_recursion_tail_stop_do:Nn #3 \}
\__text_change_case_end:w \}
\__text_change_case_N_type_aux:nnN {#1} {#2} \#3
\}
\cs_new:Npn \__text_change_case_N_type_aux:nnN #1#2#3
\exp_args:NV \__text_change_case_N_type:nnnN \l_text_math_delims_tl {#1} {#2} \#3
\}
\cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
\{ \__text_change_case_math_search:nnNNN {#2} {#3} \#4 \#1 \q__text_recursion_tail \q__text_recursion_tail \q__text_recursion_stop \}
\cs_new:Npn \__text_change_case_math_search:nnNNN #1#2#3#4#5
\{ \__text_if_recursion_tail_stop_do:Nn #4 \}
\__text_change_case_cs_check:nnn {#1} {#2} {#3} \)
\token_if_eq_meaning:NNTF \#3 \#4
\{ \__text_use_i_delimit_by_q_recursion_stop:nw
\__text_change_case_store:n {#3}
\__text_change_case_math_loop:nnNw {#1} {#2} \#5
\}
\}
\__text_change_case_math_search:nnNNN {#1} {#2} \#3 \}
\cs_new:Npn \__text_change_case_math_loop:nnnN #1#2#3#4 \q__text_recursion_stop
\{ \tl_if_head_is_N_type:nTF \#4
\{ \__text_change_case_math_N_type:nnN \}
\}
\tl_if_head_is_group:nTF \#4
\{ \__text_change_case_math_group:nnNn \}
\}
\__text_change_case_math_space:nnNw \#1 \#2 \#3 \#4 \q__text_recursion_stop
\}
\cs_new:Npn \__text_change_case_math_loop:nnnnN #1#2#3#4
\{ \__text_if_recursion_tail_stop_do:Nn \}
\__text_change_case_end:w \}
\__text_change_case_store:n \#4
\__text_change_case_math_loop:nnww {#1} {#2} {#3} \#4
\}
\cs_new:Npn \__text_change_case_math_group:nnnN #1#2#3#4
\end{verbatim}
Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That’s done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.
Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple character mapping.
If the current character is an uppercase sigma, a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters. The one exception is Dutch: see below.

For titlecasing, we need to fully expand the new character to see if it is a letter (or active) But that means looking ahead in the 8-bit case, so we have to grab the required tokens up-front. Life is a lot easier for Unicode TeX's, where we just have one token to worry about. The one wrinkle here is that for look-ahead we'd get into trouble: luckily, only Dutch has that issue.
For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)
\int_compare:nNnTF { '#4 } < { "F0 }
\{ \__text_change_case_char_UTFviii:nnNnn \}
\{ \__text_change_case_char_UTFviii:nnNnnnn \}
\}
\{#1\} \{#2\} \{#3\} \#4
\}
\{ \__text_change_case_char_aux:nnnN \{#1\} \{#2\} \{#3\} \#4 \}
\{ \__text_change_case_char_UTFviii:nnnn \{#1\} \{#2\} \{#3\} \#4 \}
\{ \__text_change_case_char_utfviii:nnnNN \{#1\} \{#2\} \{#3\} \#4 \}
\{ \__text_change_case_char_utfviii:nnnNNNN \{#1\} \{#2\} \{#3\} \#4 \}
\{ \__text_change_case_char_utfviii:nnnNNNNN \{#1\} \{#2\} \{#3\} \#4 \}
\cs_new:Npn \__text_change_case_char_UTFviii:nnnNN #1#2#3#4#5
\{ \__text_change_case_char_UTFviii:nnnNNN \{#1\} \{#2\} \{#3\} \#4 \}
\cs_new:Npn \__text_change_case_char_UTFviii:nnnNNN #1#2#3#4#5#6
\cs_new:Npn \__text_change_case_char_UTFviii:nnnNNNN #1#2#3#4#5#6#7
\cs_new:Npn \__text_change_case_char_UTFviii:nnnNNNNN #1#2#3#4
\cs_if_exist:cTF { c__text_#1_case_ \tl_to_str:n {#4} _tl }
\{ \__text_change_case_store:v
\{ c__text_#1_case_ \tl_to_str:n {#4} _tl \}
\}
\{ \__text_change_case_store:n \{#4\} \}
\use:c \{ \__text_change_case_char_next_ \#2 :nn \} \{\#2\} \{\#3\}
\}
\}
\cs_new:Npn \__text_change_case_char_next_lower:nn #1#2
\{ \__text_change_case_loop:nnw \{#1\} \{#2\} \}
\cs_new_eq:NN \__text_change_case_char_next_upper:nn \__text_change_case_char_next_lower:nn
\cs_new_eq:NN \__text_change_case_char_next_title:nn \__text_change_case_char_next_lower:nn
\cs_new_eq:NN \__text_change_case_char_next_titleonly:nn \__text_change_case_char_next_lower:nn
\cs_new:Npn \__text_change_case_char_next_end:nn #1#2
\{ \__text_change_case_break:w \}
\}}
\cs_new:Npn \__text_change_case_upper_de-alt:nnnN
\cs_new:Npn \__text_change_case_upper_de-alt:nnnNN
\bool_lazy_or:nnTF
\{ \sys_if_engine_luatex_p: \}
\{ \sys_if_engine_xetex_p: \}
\{ \char_generate:nn \{ "1E9E \} \{ \__text_char_catcode:N \#4 \}
\use:c \{ \__text_change_case_char_next_ \#2 :nn \} \{\#2\} \{\#3\}
\}
\}
\cs_new:Npn \__text_change_case_upper_de-alt:nnnNN
\cs_new:Npn \__text_change_case_upper_de-alt:nnnNNN
(End definition for \__text_change_case:nnn and others.)

A simple alternative version for German.
\bool_lazy_or:nTF
\{ \sys_if_engine_luatex_p: \}
\{ \sys_if_engine_xetex_p: \}
\{ \cs_new:cpn \{ \__text_change_case_upper_de-alt:nnnN \} \#1\#2\#3\#4 \}
\int_compare:nNnTF \{ '#4 \} = \{ "00DF \}
\{ \__text_change_case_store:e
\{ \char_generate:nn \{ "1E9E \} \{ \__text_char_catcode:N \#4 \}
\use:c \{ \__text_change_case_char_next_ \#2 :nn \} \{\#2\} \{\#3\}
\}
\}
\cs_new:Npn \__text_change_case_char:nnnN \{#1\} \{#2\} \{#3\} \#4
\endgroup
For Greek uppercasing, we need to know if characters in the Greek range have accents. That means doing a \texttt{nfd} conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found \textit{after} any U+0308 (diaeresis) and are done in two groups to allow for the canonical ordering.
In addition to the Greek accents, we list three cases here where an accent outside the Greek range has a nfd that would make it equivalent. That includes U+0344, which has to insert U+0308.

\begin{verbatim}
\cs_new:Npn \__text_change_case_upper_el:nnN \#1 \#2 \#3
\{\__text_change_case_loop:nnw \#1 \#2 \#3 \q__text_recursion_stop

\token_if_cs:NTF \#3
\{ \__text_change_case_loop:nnw \#1 \#2 \#3 \}
\int_compare:nNnTF { \#3 } = { "0308 }
\{ \int_compare:nNnTF { \#3 } = { "0300 } \int_compare:nNnTF { \#3 } = { "0301 } \int_compare:nNnTF { \#3 } = { "0304 } \int_compare:nNnTF { \#3 } = { "0306 } \int_compare:nNnTF { \#3 } = { "0308 } \int_compare:nNnTF { \#3 } = { "0313 } \int_compare:nNnTF { \#3 } = { "0314 } \int_compare:nNnTF { \#3 } = { "0342 } \int_compare:nNnTF { \#3 } = { "0340 } \int_compare:nNnTF { \#3 } = { "0341 } \int_compare:nNnTF { \#3 } = { "0343 } \}
\__text_change_case_upper_el_loop:nnw \#1 \#2 \}
\int_compare:nNnTF { \#3 } = { "0344 }
\{ \int_compare:nNnTF { \#3 } = { "0345 }
\__text_change_case_loop:nnw \#1 \#2 \#3 \}
\bool_lazy_any:nTF
\{ \int_compare_p:nNn \#3 = { "0300 } \}
\int_compare_p:nNn \#3 = { "0301 } \int_compare_p:nNn \#3 = { "0304 } \int_compare_p:nNn \#3 = { "0306 } \int_compare_p:nNn \#3 = { "0308 } \int_compare_p:nNn \#3 = { "0313 } \int_compare_p:nNn \#3 = { "0314 } \int_compare_p:nNn \#3 = { "0342 } \int_compare_p:nNn \#3 = { "0340 } \int_compare_p:nNn \#3 = { "0341 } \int_compare_p:nNn \#3 = { "0343 } \}
\__text_change_case_upper_el_loop:nnw \#1 \#2 \}
\int_compare:nNnTF { \#3 } = { "0345 }
\{ \int_compare:nNnTF { \#3 } = { "0346 }
\__text_change_case_loop:nnw \#1 \#2 \#3 \}
\}
\prg_new_conditional:Npn \__text_change_case_if_greek:n \#1 \{ \TF \}
\}
\end{verbatim}

1140
Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.
If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J and I-ogonek.

```latex
\cs_new:Npn \__text_change_case_lower_lt_auxi:nnnN #1#2#3#4
\tl_if_blank:nTF {#1}
{ \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnN #2 #3 #4 }

\int_case:nn { #4 }
{ "0049 } { "0069 }
{ "004A } { "006A }
{ "012E } { "012F }
}
{#2} {#3} #4
{\__text_change_case_store:e }
{ \char_generate:nn { "0069 } { \__text_char_catcode:N #4 } }
{ \__text_change_case_lower_lt:nnw {#2} {#3} }
\cs_new:Npn \__text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
{ \tl_if_head_is_N_type:nTF {#3}
{ \__text_change_case_lower_lt:nnN }
{ \__text_change_case_loop:nnw }
{#1} {#2} #3 \q__text_recursion_stop }
cs_new:Npn \__text_change_case_lower_lt:nnN #1#2#3
{ \bool_lazy_and:nnT { ! \token_if_cs_p:N #3 }
{ \bool_lazy_any_p:n }
{ #1#2#3 }
\cs_new:Npn \__text_change_case_lower_lt:nn:nnw #1#2#3 \q__text_recursion_stop
{\__text_change_case_store:e }
{ \char_generate:nn {#1} { \__text_char_catcode:N #3 } }
{ \__text_change_case_lower_lt:nnw {#2} }
\cs_new:Npn \__text_change_case_lower_lt:nn:nnN #1#2#3
{ \int_case:nn { #3 }
{ "0069 } { "006A }
{ "012E } { "012F }
} #1#2#3
\cs_new:Npn \__text_change_case_lower_lt:nn:nnN #1#2#3
{ \tl_if_blank:nTF {#1}
{ \exp_args:Ne \__text_change_case_lower_lt:nn:nnN #2 #3 #4 }

\int_case:nn { #4 }
{ "0049 } { "0069 }
{ "004A } { "006A }
{ "012E } { "012F }
}
{#2} {#3} #4
{\__text_change_case_store:e }
{ \char_generate:nn { "0069 } { \__text_char_catcode:N #4 } }
{ \__text_change_case_lower_lt:nnw {#2} {#3} }
\cs_new:Npn \__text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
{ \tl_if_head_is_N_type:nTF {#3}
{ \__text_change_case_lower_lt:nnN }
{ \__text_change_case_loop:nnw }
{#1} {#2} #3 \q__text_recursion_stop }
\cs_new:Npn \__text_change_case_lower_lt:nnN #1#2#3
{ \bool_lazy_and:nnT { ! \token_if_cs_p:N #3 }
{ \bool_lazy_any_p:n }
{ #1#2#3 }
```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.
The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```latex
\bool_lazy_or:nT
{ \sys_if_engine_luatex_p: }
{ \sys_if_engine_xetex_p: }
{ \cs_new:Npn \__text_change_case_upper_lt:nnnN #1#2#3#4
\exp_args:Ne \__text_change_case_upper_lt_aux:nnnN
\int_case:nn { '#4 }
{ "0069 } { "0049 }
{ "006A } { "004A }
{ "012F } { "012E }
\cs_new:Npn \__text_change_case_upper_lt_aux:nnnN #1#2#3#4
\tl_if_blank:nTF {#1}
{ \__text_change_case_char:nnnN { upper } {#2} {#3} #4 }
\__text_change_case_store:e
{ \char_generate:nn { "0307 } { \__text_char_catcode:N #3 } }
\__text_change_case_loop:nnw {#1}{#2}#3

(End definition for \_text_change_cases_lower_lt:nnnN and others.)
```
For Dutch, there is a single look-ahead test for \textit{ij} when title casing. If the appropriate letters are found, produce \textit{IJ} and gobble the \textit{j/J}.

\begin{verbatim}
\cs_new:Npn \__text_change_case_title_nl:nnnN #1#2#3#4
  { \bool_lazy_or:nnTF
  { \int_compare_p:nNn { '#4 } = { "0049 } }
  { \int_compare_p:nNn { '#4 } = { "0069 } }
  { \__text_change_case_store:e
    { \char_generate:nn { "0049 } { \__text_char_catcode:N #4 } }
  \__text_change_case_title_nl:nnw {#2} {#3}
  }
  \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
\end{verbatim}

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

\begin{verbatim}
\bool_lazy_or:nnTF
{ \int_compare_p:nNn { '#3 } = { "004A } }
{ \int_compare_p:nNn { '#3 } = { "006A } }
{ \__text_change_case_store:e
  { \char_generate:nn { "004A } { \__text_char_catcode:N #3 } }
  \__text_change_case_char_next_ \#1 :nn } {#1} {#2}
}{ \use:c { __text_change_case_char_next_ \#1 :nn } {#1} {#2} #3 }
\end{verbatim}
After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it’s not there, preserve the UTF-8 sequence as-is. With 8bit engines, we cannot completely preserve category codes, so we have to make some assumptions:
output a “normal” i for the dotted case. As the original character here is catcode-13, we have to make a choice about handling of i: generate a “normal” one.

```
\cs_new:Npn \__text_change_case_lower_tr:nnnN #1#2#3#4#5
\{ \int_compare:nNnTF { \#4 } = { "0049 } \{
   \__text_change_case_store:V \c__text_dotless_i_tl
   \__text_change_case_loop:nw {#1} {#3}
\}
\{ \int_compare:nNnTF { \#4 } = { "00C4 } \{
   \__text_change_case_lower_tr:nnnNN {#1} {#2} {#3} \#4 \}
\}
\{ \__text_change_case_char:nnnN {#1} {#2} {#3} \#4\} \}
\cs_new:Npn \__text_change_case_lower_tr:nnnNN #1#2#3#4#5
\{ \int_compare:nNnTF { \#5 } = { "00B0 } \{
   \__text_change_case_store:e
   \{ \char_generate:nn { "0069 } \{ \char_value_catcode:n { "0069 } \}
   \} \__text_change_case_loop:nw {#1} {#3}
\}
\{ \__text_change_case_char:nnnN {#1} {#2} {#3} \#4\} \}
\}
```

(End definition for \__text_change_case_lower_tr:nnn and others.)

\__text_change_case_upper_tr:nnnUppercasing is easier: just one exception with no context.
```
\cs_new:Npx \__text_change_case_upper_tr:nnnN #1#2#3#4
\{ \exp_not:N \int_compare:nNnTF { \#4 } = { "0069 } \{
   \bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: } \{
      \exp_not:N \__text_change_case_store:e
      \{ \char_generate:nn { "0130 } \{ \exp_not:N \__text_char_catcode:N \#4 \}
      \} \__text_change_case_char:nnnN {#1} {#2} {#3} \#4\} \}
\}
\cs_new:Npx \__text_change_case_upper_tr:nnnN #1#2#3#4#5
\{ \exp_not:N \int_compare:nNnTF { \#5 } = { "00B0 } \{
   \__text_change_case_store:e
   \{ \exp_not:N \char_generate:nn { "0130 } \{ \exp_not:N \__text_char_catcode:N \#4 #5 \}
   \} \__text_change_case_char:nnnN {#1} {#2} {#3} \#4\} \}
\)
```

(End definition for \__text_change_case_upper_tr:nnn and others.)
47.2 Case changing data for 8-bit engines

For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```latex
\text{\c__text_dotless_i_tl} \text{\c__text_dotted_I_tl} \text{\c__text_i_ogonek_tl} \text{\c__text_I_ogonek_tl} \text{\c__text_grosses_Eszett_tl}
```

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded.
(there’s no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

\begin{verbatim}
\bool_lazy_or:nnF
 \{ \sys_if_engine_luatex_p: \}
 \{ \sys_if_engine_xetex_p: \}
 \ifnum \cs_set_protected:Npn \__text_loop:nn #1#2
 \quark_if_recursion_tail_stop:n {#1}
 \use:x
 \ifnum \__text_tmp:w
 \char_to_utfviii_bytes:n { "#1 } \char_to_utfviii_bytes:n { "#2 } \__text_loop:nn
 \cs_set_protected:Npn \__text_tmp:nnnn #1#2#3#4#5
 \tl_const:cx
 \c__text_ #1 case_
 \char_generate:nn {#2} { 12 } \char_generate:nn {#3} { 12 } _tl
 \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N \char_generate:nn {#5} { 13 } \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N \char_generate:nn {#6} { 13 } \__text_loop:nn
 \cs_set_protected:Npn \__text_tmp:w #1#2#3#4#5#6#7#8
 \tl_const:cx
 \c__text_lowercase_
 \char_generate:nn {#1} { 12 } \char_generate:nn {#2} { 12 } _tl
 \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N \char_generate:nn {#5} { 13 } \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N \char_generate:nn {#6} { 13 } \__text_loop:nn
 \ifnum \__text_tmp:nnnn { upper } {#5} {#6} {#1} {#2} \__text_tmp:nnnn { title } {#5} {#6} {#1} {#2}
 \__text_loop:nn
 \tl_const:cx
 { 00C0 } { 00E0 }
\end{verbatim}

1148
Add T2 (Cyrillic) as this is doable using a classical \texttt{\MakeUppercase} approach.
Core Greek support: there may need to be a little more work here to deal completely with accents.
The (fixed) look-up mappings for letter-like control sequences.

```latex
\texttt{\textbackslash cs_set_protected:Npn \_\_text\_change\_case\_setup:NN #1#2#3}
\{
  \texttt{\textbackslash group\_begin:}
  \texttt{\textbackslash cs_set_protected:Npn \_\_text\_tmp:w #1#2#3}
  \{
    \texttt{\textbackslash tl\_const:cx}
    \{
      \texttt{c\_\_text\_#3 case}
      \texttt{\char\_generate:nn \#1 \{ 12 \}}
      \texttt{\char\_generate:nn \#2 \{ 12 \}}
      \texttt{\_tl}
    }
    \{\#2\}
  }
  \texttt{\textbackslash use:x}
  \{ \_\_text\_tmp:w \texttt{\textbackslash char\_to\_utf\_viii\_bytes:n \{ \#1 \}} \}
  \texttt{\textbackslash group\_end:}
\}
\texttt{\_\_text\_tmp:w \{ 00DF \} \{ SS \} \{ upper \}}
\texttt{\_\_text\_tmp:w \{ 00DF \} \{ Ss \} \{ title \}}
\texttt{\_\_text\_tmp:w \{ 0131 \} \{ I \} \{ upper \}}
\}
\texttt{\textbackslash group\_end:}
```

The (fixed) look-up mappings for letter-like control sequences.
To deal with possible encoding-specific extensions to \@uclclist, we check at the
dend of the preamble. This will therefore only apply to \LaTeX\ package mode.
\begin{verbatim}
\cs_if_exist:cT { @uclclist }
{
    \AtBeginDocument
    {
        \group_begin:
        \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
        {
            \quark_if_recursion_tail_stop:N #1
            \tl_if_single_token:nT {#2}
            {
                \cs_if_exist:cF
                { c__text_uppercase_ \token_to_str:N #1 _tl } { \#1 }
                \tl_const:cn
                { c__text_uppercase_ \token_to_str:N #1 _tl } { \#2 }
            }
            \cs_if_exist:cF
            { c__text_lowercase_ \token_to_str:N #2 _tl } { \#2 }
            \tl_const:cn
            { c__text_lowercase_ \token_to_str:N #2 _tl } { \#1 }
        }
        \__text_change_case_setup:Nn
        }\exp_after:wN \__text_change_case_setup:Nn \q_recursion_stop
        \group_end:
    }
\end{verbatim}
48  \text-purify implementation

\text_purify:n
\text_purify:n
(End definition for \_text_if_recursion_tail_stop:N.)

As in the other parts of the module, we start off with a standard “action” loop, with
expansion applied up-front.

\cs_new:Npn \text_purify:n #1
\exp_after:wN { \exp:w \exp_args:Ne \__text_purify:n { \text_expand:n {#1} } }
\cs_new:Npn \__text_purify:n #1
\__text_purify_loop:w #1 \q__text_recursion_tail \q__text_recursion_stop
\__text_purify_result:n { }
\cs_new:Npn \__text_purify_store:n #1
{ \__text_purify_store:nw {#1} }
\cs_new:Npn \__text_purify_store:nw #1#2 \__text_purify_result:n #3
{ #2 \__text_purify_result:n { #3 #1 } }
\cs_new:Npn \__text_purify_end:w #1 \__text_purify_result:n #2
{ \exp_end: #2 }

As for expansion, collect up the tokens for future use.

\cs_new:Npn \__text_purify_store:n #1
\__text_purify_store:nw {#1} \__text_purify_result:n { }
\cs_new:Npn \__text_purify_group:n
\__text_purify_math_search:NNN \__text_purify_math_start:NNw
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw \__text_purify_math_cmd:N
\__text_purify_math_cmd:NN \__text_purify_math_replace:N
\__text_purify_math_store:n \__text_purify_math_store:nw
\__text_purify_math_end:w \__text_purify_math_loop:NNw
\__text_purify_math_N_type:NNN \__text_purify_math_group:NNn
\__text_purify_math_space:NNw

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any
groups have to be run inline. Most of the business end is as before in the N-type token
processing.

\cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
{ \tl_if_head_is_N_type:nTF {#1} \tl_if_head_is_group:nTF {#1} }

1154
The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.
Then handle math mode as an argument: same outcomes, different input syntax.

For \(\text{N}\)-type tokens, we first look for a string-context replacement before anything else.
this can therefore cover anything. Assuming we don’t find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for \protect: there’s an assumption that we don’t have \protect \{ \oops \} or similar, but that’s also in the expansion code and seems like a reasonable balance.

\cs_new:Npn \__text_purify_replace:N #1
\bool_lazy_and:nnTF
  \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _tl } \exp_args:Nv
\__text_purify_replace:n { l__text_purify_ \token_to_str:N #1 _tl }
\bool_lazy_or_p:nn
  \token_if_cs_p:N #1 \token_if_active_p:N #1 \token_if_cs:NTF #1
\__text_purify_protect:N #1
\__text_if_recursion_tail_stop_do:Nn #1 \__text_purify_end:w
\exp_after:wN \__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_protect:N #1
\__text_if_recursion_tail_stop_do:Nn #1 \__text_purify_end:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w
\__text_purify_loop:w

(End definition for \text_purify:n and others. This function is documented on page 258.)
Now pre-define a range of standard commands that need dedicated definitions in
purified text. First handle font-related stuff: all of this needs to be disabled.

\tl_map_inline:nn
\{ \fontencoding
  \fontfamily
  \fontshape
}\text_declare_purify_equivalent:Nn #1 { \use_none:n }
\text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
\text_declare_purify_equivalent:Nn \selectfont { }
\text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
\tl_map_inline:nn
\{ \emph
  \text
  \textnormal
  \textrom
  \textsf
  \texttt
  \textbf
  \textmd
  \textit
  \textsl
  \textup
  \textsc
  \textulc
\}
\text_declare_purify_equivalent:Nn \#1 { \use_none:n }
\text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
\text_declare_purify_equivalent:Nn \selectfont { }
\text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
\tl_map_inline:nn
\{ \normalfont
  \rmfamily
  \sffamily
  \ttfamily
  \bfseries
  \mdseries
  \itshape
  \scshape
  \slshape
  \upshape
  \em
  \Huge
  \LARGE
  \large
  \footnotesize
  \tiny
48.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is \SS, which gets converted to two letters. (At some stage an alternative version can presumably be added to babel or similar.)
Accent \texttt{LICR} handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.
First set up the combining accents.

\group_begin:

\cs_set_protected:Npn __text_loop:Nn #1#2
{
  \quark_if_recursion_tail_stop:N #1
  \tl_const:cx { c__text_purify_ \token_to_str:N #1 _tl }
  { __text_tmp:n {#2} }
  __text_loop:Nn
}

\bool_lazy_or:nnTF
{ \sys_if_engine_luatex_p: }
{ \sys_if_engine_xetex_p: }
{ \cs_set:Npn __text_tmp:n #1
  { \char_generate:nn { "#1 } { 13 } }
}
{ \cs_set:Npn __text_tmp:n #1
  { \exp_args:Ne __text_tmp_aux:n
    \char_to_utfviii_bytes:n { "#1 } }
}

\cs_set:Npn __text_tmp:nnnn #1#2#3#4
{
  \exp_after:wN \exp_after:wN \exp_after:wN
  \exp_not:N \char_generate:nn {#1} { 13 }
  \exp_after:wN \exp_after:wN \exp_after:wN
  \exp_not:N \char_generate:nn {#2} { 13 }
  \exp_after:wN \exp_after:wN \exp_after:wN
  \exp_not:N \char_generate:nn {#3} { 13 }
  \exp_after:wN \exp_after:wN \exp_after:wN
  \exp_not:N \char_generate:nn {#4} { 13 }
}

\group_end:
Now we handle the pre-composed accents: the list here is taken from puenc.def. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the utf-8 character is used, it will have the real accent character too.
49 \textbf{l3legacy Implementation}

\texttt{\legacy_if:nTF} A friendly wrapper.

\begin{verbatim}
\prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF } { \exp_args:Nc \if_meaning:w { if#1 } \iftrue \prg_return_true: \else: \prg_return_false: \fi: }
\end{verbatim}

(End definition for \_\texttt{text\_purify\_accent:Nn}.)

50 \textbf{l3candidates Implementation}

50.1 \textbf{Additions to l3box}

\begin{verbatim}
\box_clip:N \box_clip:c \box_gclip:N \box_gclip:c
\end{verbatim}

A wrapper around the driver-dependent code.
Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. \texttt{\box_move_down:nn} is used in both cases so the resulting box always contains a \texttt{\lower} primitive. The internal box is used here as it allows safe use of \texttt{\box_set_dp:Nn}.

Same thing, this time from the top of the box.
The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

\cs_new_protected:Npn \box_set_viewport:Nnnnn \#1\#2\#3\#4\#5
  { \__box_set_viewport:NnnnnN \#1 {\#2} {\#3} {\#4} {\#5} \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
\cs_new_protected:Npn \box_gset_viewport:Nnnnn \#1\#2\#3\#4\#5
  { \__box_set_viewport:NnnnnN \#1 {\#2} {\#3} {\#4} {\#5} \box_gset_eq:NN }
\cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
\cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6
  {
    \hbox_set:Nn \l__box_internal_box
    {
      \tex_kern:D - \__box_dim_eval:n {#2}
      \box_use:N #1
      \tex_kern:D \__box_dim_eval:n { #4 - \box_wd:N #1 }
    }
    \dim_compare:nNnTF {#3} < \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
        {
          \box_move_down:nn \c_zero_dim
          \{ \box_use_drop:N \l__box_internal_box \}
        }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
      {
        \hbox_set:Nn \l__box_internal_box
        \{ \box_move_down:nn \c_zero_dim \}
        \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
      }
  }
\cs_new_protected:Npn \box_set_viewport:Nnnnn \#1\#2\#3\#4\#5
  { \__box_set_viewport:NnnnnN \#1 {\#2} {\#3} {\#4} {\#5} \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
\cs_new_protected:Npn \box_gset_viewport:Nnnnn \#1\#2\#3\#4\#5
  { \__box_set_viewport:NnnnnN \#1 {\#2} {\#3} {\#4} {\#5} \box_gset_eq:NN }
\cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
\dim_compare:nNnT \#3 > \c_zero_dim
{ -(\#3) }
\}
\}
\hbox_set:Nn \l__box_internal_box
{ \box_move_up:nn { - \__box_dim_eval:n \#5 } { \box_use_drop:N \l__box_internal_box } }
\box_set_ht:Nn \l__box_internal_box \c_zero_dim
\}
\box_set_viewport:Nnnnn \box_gset_viewport:Nnnnn, and \__box_viewport:NnnnnN. These functions are documented on page 261.

50.2 Additions to l3flag
\flag_raise_if_clear:n It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable in the future.
\cs_new:Npn \flag_raise_if_clear:n \#1
{ \ifcs_exist:w flag~\#1~0 \cs_end: \else: \cs:w flag~\#1 \cs_end: 0 ; \fi: }

(End definition for \flag_raise_if_clear:n. This function is documented on page 262.)

50.3 Additions to l3msg
\msg_expandable_error:nnnnn Pass to an auxiliary the message to display and the module name
\cs_new:Npn \msg_expandable_error:nnnnn \msg_expandable_error:nnfff \msg_expandable_error:nnnn \msg_expandable_error:nnnnf \msg_expandable_error:nnf \msg_expandable_error:nn \__msg_expandable_error_module:nn
{ \exp_args:Ne \__msg_expandable_error_module:nn
{ \exp_args:Nc \exp_args:Noooo
{ \c__msg_text_prefix_tl \#1 / \#2 } \{ \tl_to_str:n \#3 \} \{ \tl_to_str:n \#4 \} \{ \tl_to_str:n \#5 \} \{ \tl_to_str:n \#6 \} \}
\}

(End definition for \msg_expandable_error:nnnnn. This function is documented on page 263.)
\cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
    { \msg_expandable_error:nnnnn { #1 } { #2 } { #3 } { #4 } { #5 } { } }
\cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
    { \msg_expandable_error:nnnnn { #1 } { #2 } { #3 } { #4 } { } { } }
\cs_new:Npn \msg_expandable_error:nnnn #1#2#3
    { \msg_expandable_error:nnnnn { #1 } { #2 } { #3 } { } { } { } }
\cs_new:Npn \msg_expandable_error:nnn #1#2#3
    { \msg_expandable_error:nnnnn { #1 } { #2 } { } { } { } { } }
\cs_new:Npn \msg_expandable_error:nn #1#2
    { \msg_expandable_error:nnnnn { #1 } { #2 } { } { } { } { } }
\cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnffff }
\cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnfff }
\cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
\cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }
\cs_new:Npn \__msg_expandable_error_module:nn #1#2
    { \exp_after:wN \exp_after:wN \exp_after:wN \__msg_use_none_delimit_by_s_stop:w
      \use:n { \::error ! ~ #2 : ~ #1 } \s__msg_stop }
\msg_show_eval:Nn
\msg_log_eval:Nn
\__msg_show_eval:nnN
\int_show:n
\tl_show:n
\__msg_use_none_delimit_by_s_stop:w
\cs_new_protected:Npn \msg_show_eval:Nn #1#2
    { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
\cs_new_protected:Npn \msg_log_eval:Nn #1#2
    { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
\cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3
    { #3 { #2 = #1 } }
\cs_new_protected:Npx \msg_show_item:n #1
    { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
\cs_new_protected:Npx \msg_show_item_unbraced:n #1
    { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
\cs_new_protected:Npx \msg_show_item:nn #1#2
    { \iow_newline: > \use:nn { ~ } { ~ } \exp_not:N \tl_to_str:n { {#1} } \use:nn { ~ } { ~ } \exp_not:N \tl_to_str:n { {#2} } }
\cs_new_protected:Npx \msg_show_item_unbraced:nn #1#2
    { \iow_newline: > \use:nn { - } { - } \exp_not:N \tl_to_str:n { {#1} } \use:nn { - } { - } \exp_not:N \tl_to_str:n { {#2} } }
(End definition for \msg_expandable_error:nnnnn and others. These functions are documented on page 263.)

A short-hand used for \int_show:n and similar functions that passes to \tl_show:n the result of applying #1 (a function such as \int_eval:n) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by \iow_wrap:nnnN. This is only important for expressions involving the \currentgrouplevel or \currentgrouptype. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

(End definition for \msg_show_eval:Nn, \msg_log_eval:Nn, and \__msg_show_eval:nnN. These functions are documented on page 263.)

Each item in the variable is formatted using one of the following functions. We cannot use \ and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.
50.4 Additions to l3prg

\bool_set_inverse:N
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c

Set to false or true locally or globally.

\cs_new_protected:Npn \bool_set_inverse:N #1
{ \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
\cs_generate_variant:Nn \bool_set_inverse:N { c }
\cs_new_protected:Npn \bool_gset_inverse:N #1
{ \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
\cs_generate_variant:Nn \bool_gset_inverse:N { c }

\s__bool_mark
\s__bool_stop

Internal scan marks.

\cs_new:N \s__bool_mark
\cs_new:N \s__bool_stop

\bool_case_true:nTF
\bool_case_true:nT #1#2
\bool_case_true:nF #1
\bool_case_true:n #1
\bool_case_false:nTF
\bool_case_false:nT #1#2
\bool_case_false:nF #1
\bool_case_false:n #1
\bool_case_end:nw

For boolean cases the overall idea is the same as for \tl_case:nn(TF) as described in l3tl.
50.5 Additions to l3prop

Functions to gobble up to a scan mark.

\prop_rand_key_value:N\prop_rand_key_value:c\__prop_rand_item:w

Contrarily to clist, seq and tl, there is no function to get an item of a prop given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because \int_rand:nn is always within bounds. The initial \int_-value:w is stopped by the first \s__prop in #1.

\cs_new:Npn \prop_rand_key_value:N #1
\cs_generate_variant:Nn \prop_rand_key_value:N { c }
\cs_new:Npn \__prop_rand_item:w #1 \s__prop \__prop_pair:wn #2 \s__prop #3

(End definition for \prop_rand_key_value:N and \__prop_rand_item:w. This function is documented on page 265.)

50.6 Additions to l3seq

The idea is to first expand both sequences, adding the usual \{ ? \prg_break: \} \} to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items
in both sequences are \s__seq \s__seq_item:n. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

\cs_new:Npn \seq_mapthread_function:NNN #1#2#3
{ \exp_after:wN \__seq_mapthread_function:wNN #2 \s__seq_stop #1 #3 }
\cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \s__seq_stop #2#3
{ \exp_after:wN \__seq_mapthread_function:wNw #2 \s__seq_stop #3 #1 { ? \prg_break: } { } \prg_break_point: }
\cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \s__seq_stop #2
{ \__seq_mapthread_function:Nnnwnn #2 #1 { #3 } { #6 } \s__seq_stop }
\cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
{ \use_none:n #2 \use_none:n #5 #1 { #3 } { #6 } \__seq_mapthread_function:Nnnvnn #1 #4 \s__seq_stop }
\cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }

(End definition for \seq_mapthread_function:NNN and others. This function is documented on page 265.)

\seq_set_filter:NNn \seq_gset_filter:NNn \__seq_set_filter:NNNn
Similar to \seq_map_inline:Nn, without a \prg_break_point: because the user’s code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The \__seq_wrap_item:n function inserts the relevant \s__seq_item:n without expansion in the input stream, hence in the x-expanding assignment.

\cs_new_protected:Npn \seq_set_filter:NNn \seq_gset_filter:NNn
{ \__seq_set_filter:NNNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gset_filter:NNn
{ \__seq_set_filter:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
{ \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } } #1 #2 { #3 } \__seq_pop_item_def: }
\cs_set_map:NNn \seq_gset_map:NNn \__seq_set_map:NNNn
Very similar to \seq_set_filter:NNn. We could actually merge the two within a single function, but it would have weird semantics.

\cs_new_protected:Npn \seq_set_map:NNn \seq_gset_map:NNn
{ \__seq_set_map:NNNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gset_map:NNn
{ \__seq_set_map:NNNn \tl_gset:Nx }

1173
(End definition for \seq_set_map:NNn, \seq_gset_map:NNn, and \__seq_set_map:NNNn. These functions are documented on page 265.)

\seq_set_from_inline_x:Nnn \seq_gset_from_inline_x:Nnn \__seq_set_from_inline_x:NNnn
Set \__seq_item:n then map it using the loop code.

(End definition for \seq_set_from_inline_x:Nnn, \seq_gset_from_inline_x:Nnn, and \__seq_set_from_inline_x:NNnn. These functions are documented on page 266.)

\seq_set_from_function:NnN \seq_gset_from_function:NnN
Reuse \seq_set_from_inline_x:Nnn.

(End definition for \seq_set_from_function:NnN and \seq_gset_from_function:NnN. These functions are documented on page 266.)

\seq_indexed_map_function:NN \seq_indexed_map_inline:Nn
Similar to \seq_map_function:NN but we keep track of the item index as a \ASCII{-}delimited argument of \__seq_indexed_map:Nw.

(End definition for \seq_indexed_map_function:NN and \seq_indexed_map_inline:Nn. These functions are documented on page 266.)
50.7 Additions to l3sys

\c_sys_engine_version_str

Various different engines, various different ways to extract the data!

\c_sys_engine_version_str
\st_const:Nx \c_sys_engine_version_str
{ \str_case:on \c_sys_engine_str
{ pdftex
{ \fp_eval:n \round{\int_use:N \tex_pdftexversion:D / 100 , 2}
, \tex_pdftexrevision:D
}
ptex
{ \cs_if_exist:NT \tex_ptexversion:D
{ p
\int_use:N \tex_ptexversion:D
, \int_use:N \tex_ptexminorversion:D
\tex_ptexrevision:D
\int_use:N \tex_epTeXversion:D
}
}
}
\st_case:on \c_sys_engine_str
{ luatex
{ \fp_eval:n \round{\int_use:N \tex_luatemversion:D / 100 , 2}
, \tex_luatexrevision:D
}
uplox
{ \cs_if_exist:NT \tex_ptexversion:D
}
\ior_shell_open:Nn \__ior_shell_open:nN

Actually much easier than either the standard open or input versions! When calling \__kernel_ior_open:Nn the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by \__kernel_file_name_quote:n.

```
\cs_new_protected:Npn \ior_shell_open:Nn #1#2
\sys_if_shell:TF
\exp_args:No \__ior_shell_open:nN { \tl_to_str:n {#2} } #1
\__kernel_msg_error:nn { kernel } { pipe-failed } }
{ \__kernel_ior_open:Nn #2 { |#1 } }
\__kernel_msg_new:nnnn { kernel } { pipe-failed }
\tl_if_in:nnTF {#1} { " }
\__kernel_msg_error:nnx { kernel } { quote-in-shell } {#1}
{ \__kernel_ior_open:Nn #2 { |#1 } }
\__kernel_msg_new:nnnn { kernel } { pipe-failed }
\{ Cannot-run-piped-system-commands. \}
\LaTeX-tried-to-call-a-system-process-but-this-was-not-possible.\Try-the-"--shell-escape"-(or-"--enable-pipes")-option.
```

(End definition for \c_sys_engine_version_str. This variable is documented on page 267.)

50.8 Additions to l3file

\ior_shell_open:Nn and \__ior_shell_open:nN. This function is documented on page 262.)
50.8.1 Building a token list

Between \tl_build_begin:N \tl_build_gbegin:N, the \tl_build_gbegin:N has the structure
\exp_end: ... \exp_end: \__tl_build_last:NNn \assignment \next tl \left \right

where \right is not braced. The “data” it represents is \left followed by the “data” of \next tl followed by \right. The \next tl is a token list variable whose name is that of \tl_build_end:N \tl_build_gend:N followed by \'. There are between 0 and 4 \exp_end: to keep track of when \left and \right should be put into the \next tl. The \assignment is \cs_set_nopar:Npx if the variable is local, and \cs_gset_nopar:Npx if it is global.

First construct the \next tl: using a prime here conflicts with the usual expl3 convention but we need a name that can be derived from \#1 without any external data such as a counter. Empty that \next tl and setup the structure. The local and global versions only differ by a single function \cs_(g)set_nopar:Npx used for all assignments: this is important because only that function is stored in the \tl_build_begin:N and \next tl for subsequent assignments. In principle \__tl_build_begin:NNN could use \tl_(g)clear_new:N to empty \#1 and make sure it is defined, but logging the definition does not seem useful so we just do \#3 \#1 {} to clear it locally or globally as appropriate.

\cs_new_protected:Npn \tl_build_begin:N #1
  { \__tl_build_begin:NN \cs_set_nopar:Npx #1 }
\cs_new_protected:Npn \tl_build_gbegin:N #1
  { \__tl_build_begin:NN \cs_gset_nopar:Npx #1 }
\cs_new_protected:Npn \__tl_build_begin:NN #1#2
  { \exp_args:Nc \__tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
\cs_new_protected:Npn \__tl_build_begin:NNN #1#2#3
  { \#3 \#1 { } \#3 \#2 { \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: } \exp_not:n { \__tl_build_last:NNn \#3 \#1 { } } } }

(End definition for \tl_build_begin:N and others. These functions are documented on page 268.)

\cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
\cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N

(End definition for \tl_build_clear:N and \tl_build_gclear:N. These functions are documented on page 268.)

\__tl_build_last:NNn \__tl_build_put:nn \__tl_build_put:nw

Similar to \tl_put_right:Nn, but apply \exp:w to \#1. Most of the time this just removes one \exp:w. When there are none left, \__tl_build_last:NNn is expanded instead. It resets the definition of the \tl_build_end:N by ending the \exp_not:n and the definition early.
Then it makes sure the \langle next tl \rangle (its argument \#1) is set-up and starts a new definition. Then \_\_tl_build_put:nn and \_\_tl_build_put:nn place the \langle left \rangle part of the original \langle tl var \rangle as appropriate for the definition of the \langle next tl \rangle (the \langle right \rangle part is left in the right place without ever becoming a macro argument). We use \texttt{\exp_after:wN} rather than some \texttt{\exp_args:No} to avoid reading arguments that are likely very long token lists. We use \texttt{\cs_(g)set_nopar:Np} rather than \texttt{\tl_(g)set:Nx} partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an \texttt{x}\texttt{-}\texttt{expansion} of the second argument.

\begin{verbatim}
\cs_new_protected:Npn \tl_build_put_right:Nn #1#2
\cs_set_nopar:Npx #1 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
\cs_new_protected:Npn \tl_build_put_right:Nx #1#2
\cs_set_nopar:Npx #1 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
\cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
\cs_gset_nopar:Npx #1 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
\cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
\cs_gset_nopar:Npx #1 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
\cs_new_protected:Npn \__tl_build_last:NNn #1#2
\if_false: { { \fi:
\exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
\__tl_build_last:NNn #1 #2 { } }
\exp_after:wN \exp_not:n \exp_after:wN { \exp:w \if_false: } } \fi:
\exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
\cs_new_protected:Npn \__tl_build_put_left:NNn \_\_tl_build_last:NNN \#1#2
\cs_set_nopar:Npx \_\_tl_build_last:NNN \#1 \#2
\if_meaning:w \c_empty_tl #2 \_\_tl_build_begin:NN \#1 \#2 \fi:
\cs_gset_nopar:Npx \_\_tl_build_last:NNN \#1 \#2
\__tl_build_put_left_right:Nn #1 #2 #3 #4 #5
\exp_after:wN \exp_not:n \exp_after:wN { \exp:w \if_false: } } \fi:
\exp_after:wN \_\_tl_build_put:nn \exp_after:wN {#2}
\cs_new_protected:Npn \_\_tl_build_put:nn \_\_tl_build_put:nn \#1#2 \{ \_\_tl_build_put:nn \#2 \} \#1
\cs_new_protected:Npn \_\_tl_build_put:nn \_\_tl_build_put:nn \#1#2 \_\_tl_build_last:NNN \#3#4#5
\{ \#2 \_\_tl_build_last:NNN \#3 \#4 \{ \#1 \#5 \} \}
\end{verbatim}

(End definition for \_\_tl_build_put_right:Nn and others. These functions are documented on page 269.)
The idea is to expand the \texttt{⟨tl var⟩} then the \texttt{⟨next tl⟩} and so on, all within an \texttt{x}-expanding assignment, and wrap as appropriate in \texttt{\exp_not:n}. The various \texttt{⟨left⟩} parts are left in the assignment as we go, which enables us to expand the \texttt{⟨next tl⟩} at the right place. The various \texttt{⟨right⟩} parts are eventually picked up in one last \texttt{\exp_not:n}, with a brace trick to wrap all the \texttt{⟨right⟩} parts together.

Get the data then clear the \texttt{⟨next tl⟩} recursively until finding an empty one. It is perhaps wasteful to repeatedly use \texttt{\cs_to_sr:N}. The local/global scope is checked by \texttt{\tl_set:Nx} or \texttt{\tl_gset:Nx}.

(End definition for \texttt{\tl_build_put_left:Nn}, \texttt{\tl_build_gput_left:Nn}, and \texttt{\__tl_build_put_left:NNn}. These functions are documented on page 269.)

(End definition for \texttt{\tl_build_get:NN} and others. This function is documented on page 269.)
50.8.2 Other additions to \texttt{l3tl}

For the braced version \texttt{\_\_tl\_range\_braced:w} sets up \texttt{\_\_tl\_range\_collect\_braced:w} which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up \texttt{\_\_tl\_range\_collect:nn} with a first argument of the form \{ \{collected\}\ \{tokens\}\}, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the \{collected\} tokens.

\begin{verbatim}
\cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
\cs_generate_variant:Nn \tl_range_braced:Nnn { c }
\cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
\cs_new:Npn \tl_range_unbraced:Nnn { \exp_args:No \tl_range_unbraced:nnn }
\cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
\cs_new:Npn \tl_range_unbraced:nnn { \__tl_range:Nnnn \__tl_range_unbraced:w }
\cs_new:Npn \__tl_range_braced:w #1 ; #2
\{ \__tl_range_collect_braced:w #1 ; { } #2 \}
\cs_new:Npn \__tl_range_unbraced:w #1 ; #2
\{ \__tl_range_collect_unbraced:w #1 ; { } #2 \}
\cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
\{ \if_int_compare:w #1 > 1 \exp_stop_f:
\exp_after:wN \__tl_range_collect_braced:w
\int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
\fi:
\{ #2 #3 \}
\}
\cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
\{ \if_int_compare:w #1 > 1 \exp_stop_f:
\exp_after:wN \__tl_range_collect_unbraced:w
\int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
\fi:
\{ #2 #3 \}
\}
\end{verbatim}

(End definition for \texttt{\_\_tl\_range\_braced:Nnn} and others. These functions are documented on page 268.)
50.9 Additions to l3token

\c_catcode_active_space_tl

While \char_generate:nn can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive \tex_lowercase:D trick.

\begin{verbatim}
\group_begin:
  \char_set_catcode_active:N *
  \char_set_lccode:nn { ‘* } { ‘\ }
  \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
\group_end:
\end{verbatim}

(End definition for \c_catcode_active_space_tl. This variable is documented on page 269.)

1181
\section{\texttt{l3deprecation} implementation}

\subsection{Helpers and variables}

\texttt{\_\_deprecation\_grace\_period\_bool}

This is set to \texttt{true} when the deprecated command that is being defined is in its grace period, meaning between the time it becomes an error by default and the time 6 months later where even \texttt{undo-recent-deprecations} stops restoring it.

\texttt{\_\_deprecation\_mark}

Internal scan marks.

\texttt{\_\_deprecation\_stop}

(End definition for \texttt{\_\_deprecation\_mark} and \texttt{\_\_deprecation\_stop})

\texttt{\_\_deprecation\_date\_compare\_nNnTF}

Expects \#1 and \#3 to be dates in the format YYYY-MM-DD (but accepts YYYY or YYYY-MM too, filling in zeros for the missing data). Compares them using \#2 (one of \texttt{<, =, >}).

(End definition for \texttt{\_\_deprecation\_date\_compare\_nNnTF})
\cs_new:Npn \__deprecation_date_compare_aux:w
\#1 - \#2 - \#3 - \#4 \s__deprecation_mark \#5 \#6 - \#7 - \#8 - \#9 \s__deprecation_stop
{
\int_compare:nNnTF {\#1} = {\#6}
{
\int_compare:nNnTF {\#2} = {\#7}
{
\int_compare:nNnTF {\#3} \#5 \{\#8\}
\int_compare:nNnTF {\#2} \#5 \{\#7\}
}
\int_compare:nNnTF {\#1} \#5 \{\#6\}
}
}

(End definition for \__deprecation_date_compare:nNnTF and \__deprecation_date_compare_aux:w.)

\bool_new:N \g__kernel_deprecation_undo_recent_bool
(End definition for \g__kernel_deprecation_undo_recent_bool.)

\__deprecation_not_yet_deprecated:nTF
\__deprecation_minus_six_months:w

Receives a deprecation \langle date\rangle and runs the true (false) branch if the expl3 date is earlier (later) than \langle date\rangle. If undo-recent-deprecations is used we subtract 6 months to the expl3 date (equivalently add 6 months to the \langle date\rangle). In addition, if the expl3 date is between \langle date\rangle and \langle date\rangle plus 6 months, \l__deprecation_grace_period_bool is set to true, otherwise false.

\cs_new_protected:Npn \__deprecation_not_yet_deprecated:nTF #1
{
\bool_set_false:N \l__deprecation_grace_period_bool
\exp_args:No \__deprecation_date_compare:nNnTF { \ExplLoaderFileDate } < {\#1}
{ \use_i:nn }
\exp_args:Nf \__deprecation_date_compare:nNnTF
{ \exp_after:wN \__deprecation_minus_six_months:w \ExplLoaderFileDate -0-0- \s__deprecation_stop
\int_compare:nNnTF {\#2} > 6
{ \#1 - \int_eval:n \{ \#2 - 6 \} - \#3 }
\int_eval:n \{ \#1 - 1 \} - \int_eval:n \{ \#2 + 6 \} - \#3
}
}

\cs_new:Npn \__deprecation_minus_six_months:w #1 - #2 - #3 - #4 \s__deprecation_stop
{\int_compare:nNnTF {\#2} > 6
{ #1 - \int_eval:n \{ \#2 - 6 \} - \#3 }
{ \int_eval:n \{ \#1 - 1 \} - \int_eval:n \{ \#2 + 6 \} - \#3
}
}

(End definition for \__deprecation_not_yet_deprecated:nTF and \__deprecation_minus_six_months:w.)
51.2 Patching definitions to deprecate

\__kernel_patch_deprecation:nnNNpn \{\langle\text{date}\rangle\} \{\langle\text{replacement}\rangle\} \{\langle\text{definition}\rangle\} \{\langle\text{parameters}\rangle\} \{\langle\text{code}\rangle\}

defines the \langle\text{function}\rangle to produce a warning and run its \langle\text{code}\rangle, or to produce an error and not run any \langle\text{code}\rangle, depending on the expl3 date.

- If the expl3 date is less than the \langle\text{date}\rangle (plus 6 months in case undo-recent-deprecations is used) then we define the \langle\text{function}\rangle to produce a warning and run its \langle\text{code}\rangle. The warning is actually suppressed in two cases:
  - if neither undo-recent-deprecations nor enable-debug are in effect we may be in an end-user’s document so it is suppressed;
  - if the command is expandable then we cannot produce a warning.
- Otherwise, we define the \langle\text{function}\rangle to produce an error.

In both cases we additionally make \debug_on:n \{\text{deprecation}\} turn the \langle\text{function}\rangle into an \outer error, and \debug_off:n \{\text{deprecation}\} restore whatever the behaviour was without \debug_on:n \{\text{deprecation}\}.

In later sections we use the \l3doc key deprecated with a date equal to that \langle\text{date}\rangle plus 6 months, so that \l3doc will complain if we forget to remove the stale \langle\text{parameters}\rangle and \{\langle\text{code}\rangle\}.

In the explanations below, \langle\text{definition}\rangle \langle\text{function}\rangle \langle\text{parameters}\rangle \{\langle\text{code}\rangle\} or assignments that only differ in the scope of the \langle\text{definition}\rangle will be called “the standard definition”.

\cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn \#1\#2\#3\#4\#5\#6
\{ \__deprecation_patch_aux:nnNNnn \{\#1\} \{\#2\} \#3 \#4 \{\#5\} \}
\cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn \#1\#2\#3\#4\#5\#6
\{ \__kernel_deprecation_code:nn
\tex_let:D \#4 \scan_stop:
\__kernel_deprecation_error:Nnn \#4 \{\#2\} \{\#1\} \}
\{ \tex_let:D \#4 \scan_stop: \} \__deprecation_not_yet_deprecated:nTF \{\#1\}
\{ \bool_if:nTF

(The parameter text is grabbed using #5#.) The arguments of \__kernel_deprecation_code:nn are run upon \debug_on:n \{\text{deprecation}\} and \debug_off:n \{\text{deprecation}\}, respectively. In both scenarios we the \langle\text{function}\rangle may be \outer so we undefine it with \tex_let:D before redefining it, with \__kernel_deprecation_error:Nnn or with some code added shortly.

Then check the date (taking into account undo-recent-deprecations) to see if the command should be deprecated right away (false branch of \__deprecation_not_yet_deprecated:nTF), in which case \__deprecation_not_yet_deprecated:nnn makes \langle\text{function}\rangle into an error (not \outer), ignoring its \langle\text{parameters}\rangle and \langle\text{code}\rangle completely.

Otherwise distinguish cases where we should give a warning from those where we shouldn’t: warnings can only happen for protected commands, and we only want them if either undo-recent-deprecations or enable-debug is in force, not for standard users.

\cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn \#1\#2\#3\#4\#5\#6
\{ \__deprecation_patch_aux:nnNNnn \{\#1\} \{\#2\} \#3 \#4 \{\#5\} \}
\cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn \#1\#2\#3\#4\#5\#6
\{ \__kernel_deprecation_code:nn
\tex_let:D \#4 \scan_stop:
\__kernel_deprecation_error:Nnn \#4 \{\#2\} \{\#1\} \}
\{ \tex_let:D \#4 \scan_stop: \} \__deprecation_not_yet_deprecated:nTF \{\#1\}
\{ \bool_if:nTF

1184
In case we want a warning, the (`function`) is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the (`function`) should have, with arguments, and call that definition. The `x-type` expansion and \exp_-\not:\n avoid needing to double the `#`, which we could not do anyways. We then deal with the code for \debug_off:\n {deprecation}: presumably someone doing that does not need the warning so we simply do the standard definition.

```latex
\cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
{ \cs_gset_protected:Npx #3
{ \__kernel_if_debug:TF
{ \exp_not:N \__kernel_msg_warning:nnxxx
{ kernel } { deprecated-command } #1
{ \token_to_str:N #3 }
{ \tl_to_str:n {#2} }
}
}
\exp_not:N \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #3 #4 {#5} }
\exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
}
```

In case we want neither warning nor error, the (`function`) is given its standard definition. Here #1 is `\cs_new:NN` or `\cs_new_protected:NN` and #2 is (`function`) `{\langle parameters\rangle}`, so #1#2 performs the assignment. For \debug_off:\n {deprecation} we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```latex
\cs_new_protected:NN \__deprecation_patch_aux:Nn #1 #2
{ \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #1 #2 } }
```

Finally, if we want an error we reuse the same `\__deprecation_patch_aux:Nn` as the previous case. Indeed, we want \debug_off:\n {deprecation} to make the (`function`) into an error, just like it is by default. The error is expandable or not, and the last argument of the error message is empty or is `grace` to denote the case where we are in the 6 month grace period, in which case the error message is more detailed.
\cs_new_protected:Npn \__deprecation_just_error:nnNN #1#2#3#4
\exp_args:NNx \__deprecation_patch_aux:Nn #3
\exp_not:N #4
\cs_if_eq:NNTF #3 \cs_gset_protected:Npn
\exp_not:N \__kernel_msg_error:nnnnnn
{ \exp_not:N \__kernel_msg_expandable_error:nnnnnn }
{ \kernel } { deprecated-command }
{\#1}
{ \token_to_str:N #4 }
{ \tl_to_str:n {#2} }
{ \bool_if:NT \l__deprecation_grace_period_bool { grace } }
\exp_not:N \__kernel_msg_error:nnxxx
{ \kernel } { deprecated-command }
{ \tl_to_str:n {#3} }
{ \tl_to_str:n {#2} }
\exp_not:N \__kernel_msg_error:nnnnnn
\tl_if_blank:nF {#3} { Use \tl_trim_spaces:n {#3} -not- }
\str_if_eq:nnT {#4} { grace }
\c_space_tl
For 6 months after that date one can restore a deprecated command by loading the expl3 package with the option `undo-recent-deprecations'.
\end_definition_for \_\_kernel_patch_deprecation:nnNpN and others.
\__kernel_deprecation_error:Nnn
The \outer definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.
\exp_not:N \__kernel_msg_error:nnnnnn
\tl_if_blank:nF {#3} { Use \tl_trim_spaces:n {#3} -not- }
\str_if_eq:nnT {#4} { grace }
\c_space_tl
For 6 months after that date one can restore a deprecated command by loading the expl3 package with the option `undo-recent-deprecations'.
\end_definition_for \_\_kernel_deprecation_error:Nnn.
\__kernel_msg_new:nnn { kernel } { deprecated-command }
{\#1}
{ \tl_to_str:n {#3} }
\end_definition_for \_\_kernel_deprecation_error:Nnn.
\__deprecation_old_protected:Nnn \__deprecation_old:Nnn
Short-hands for old commands whose definition does not matter anymore, i.e., commands past the grace period.

51.3 Removed functions
\cs_new_protected:Npn \_deprecation_old_protected:Nnn #1#2#3
\{
\__kernel_patch_deprecation:nnNNpn {#3} {#2}
\cs_gset_protected:Npn #1 { }
\}
\cs_new_protected:Npn \_deprecation_old:Nnn #1#2#3
\{
\__kernel_patch_deprecation:nnNNpn {#3} {#2}
\cs_gset:Npn #1 { }
\}
\__deprecation_old:Nnn \box_resize:Nnn
\{
\box_resize_to_wd_and_ht_plus_dp:Nnn
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \box_use_clear:N
\{
\box_use_drop:N
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \c_job_name_tl
\{
\c_sys_jobname_str
\} \{ 2017-01-01 \}
\__deprecation_old:Nnn \c_minus_one
\{
-1
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \dim_case:nnn
\{
\dim_case:nnF
\} \{ 2015-07-14 \}
\__deprecation_old:Nnn \file_add_path:nN
\{
\file_get_full_name:nN
\} \{ 2019-01-01 \}
\__deprecation_old_protected:Nnn \file_if_exist_input:nT
\{
\file_if_exist:nT and \file_input:n
\} \{ 2018-03-05 \}
\__deprecation_old_protected:Nnn \file_if_exist_input:nTF
\{
\file_if_exist:nT and \file_input:n
\} \{ 2018-03-05 \}
\__deprecation_old:Nnn \file_list:
\{
\file_log_list:
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \file_path_include:n
\{
\seq_put_right:Nn \l_file_search_path_seq
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \file_path_remove:n
\{
\seq_remove_all:Nn \l_file_search_path_seq
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \g_file_current_name_tl
\{
\g_file_curr_name_str
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \int_case:nnn
\{
\int_case:nnF
\} \{ 2015-07-14 \}
\__deprecation_old:Nnn \int_from_binary:n
\{
\int_from_bin:n
\} \{ 2016-01-05 \}
\__deprecation_old:Nnn \int_from_hexadecimal:n
\{
\int_from_hex:n
\} \{ 2016-01-05 \}
\__deprecation_old:Nnn \int_from_octal:n
\{
\int_from_oct:n
\} \{ 2016-01-05 \}
\__deprecation_old:Nnn \int_to_binary:n
\{
\int_to_bin:n
\} \{ 2016-01-05 \}
\__deprecation_old:Nnn \int_to_hexadecimal:n
\{
\int_to_hex:n
\} \{ 2016-01-05 \}
\__deprecation_old:Nnn \int_to_octal:n
\{
\int_to_oct:n
\} \{ 2016-01-05 \}
\__deprecation_old_protected:Nnn \ior_get_str:NN
\{
\ior_str_get:NN
\} \{ 2018-03-05 \}
\__deprecation_old:Nnn \ior_list_streams:
\{
\ior_show_list:
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \ior_logStreams:
\{
\ior_log_list:
\} \{ 2019-01-01 \}
\__deprecation_old:Nnn \iow_list_streams: \{ \iow_show_list: \} { 2019-01-01 }
\__deprecation_old:Nnn \iow_log_streams: \{ \iow_log_list: \} { 2019-01-01 }
\__deprecation_old:Nnn \luatex_if_engine_p: \{ \sys_if_engine_luatex_p: \} { 2017-01-01 }
\__deprecation_old:Nnn \luatex_if_engine:F \{ \sys_if_engine_luatex:F \} { 2017-01-01 }
\__deprecation_old:Nnn \luatex_if_engine:T \{ \sys_if_engine_luatex:T \} { 2017-01-01 }
\__deprecation_old:Nnn \luatex_if_engine:TF \{ \sys_if_engine_luatex:TF \} { 2017-01-01 }
\__deprecation_old:Nnn \pdftex_if_engine_p: \{ \sys_if_engine_pdftex_p: \} { 2017-01-01 }
\__deprecation_old:Nnn \pdftex_if_engine:F \{ \sys_if_engine_pdftex:F \} { 2017-01-01 }
\__deprecation_old:Nnn \pdftex_if_engine:T \{ \sys_if_engine_pdftex:T \} { 2017-01-01 }
\__deprecation_old:Nnn \pdftex_if_engine:TF \{ \sys_if_engine_pdftex:TF \} { 2017-01-01 }
\__deprecation_old:Nnn \prop_get:cn \{ \prop_item:cn \} { 2016-01-05 }
\__deprecation_old:Nnn \prop_get:Nn \{ \prop_item:Nn \} { 2016-01-05 }
\__deprecation_old:Nnn \quark_if_recursion_tail_break:N \{ \prop_item:cn \} { 2016-01-05 }
\__deprecation_old:Nnn \quark_if_recursion_tail_break:n \{ \prop_item:Nn \} { 2016-01-05 }
\__deprecation_old:Nnn \scan_align_safe_stop: \{ protected\texttt{-commands} \} { 2017-01-01 }
\__deprecation_old:Nnn \sort_ordered: \{ \sort_return_same: \} { 2019-01-01 }
\__deprecation_old:Nnn \sort_reversed: \{ \sort_return_swapped: \} { 2019-01-01 }
\__deprecation_old:Nnn \str_case:nnn \{ \str_case:nnF \} { 2015-07-14 }
\__deprecation_old:Nnn \str_case:onn \{ \str_case:onF \} { 2015-07-14 }
\__deprecation_old:Nnn \str_case_x:nnn \{ \str_case_e:nnF \} { 2015-07-14 }
\__deprecation_old:Nnn \tl_case:cnn \{ \tl_case:cnF \} { 2015-07-14 }
\__deprecation_old:Nnn \tl_case:Nnn \{ \tl_case:NNf \} { 2015-07-14 }
\__deprecation_old_protected:Nnn \tl_to_lowercase:n \{ \tex_lowercase:D \} { 2018-03-05 }
\__deprecation_old_protected:Nnn \tl_to_lowercase:n \{ \tex_uppercase:D \} { 2018-03-05 }
\__deprecation_old:Nnn \token_new:Nn \{ \sys_if_engine_xetex_p: \} { 2017-01-01 }
\__deprecation_old:Nnn \xetex_if_engine_p: \{ \sys_if_engine_xetex_p: \} { 2017-01-01 }
\__deprecation_old:Nnn \xetex_if_engine:F \{ \sys_if_engine_xetex:F \} { 2017-01-01 }
51.4 Deprecated primitives

We renamed all primitives to \texttt{\textbackslash ...} so all others are deprecated. In \texttt{l3names}, \texttt{\_kernel_primitives} is defined to contain \texttt{\_kernel_primitive:NN \texttt{\textbackslash beginL \textbackslash etex_\beginL}} and so on, one for each deprecated primitive. We apply \texttt{\exp_not:N} to the second argument of \texttt{\_kernel_primitive:NN} because it may be outer (both when doing and undoing deprecation actually), then \texttt{\_deprecation_primitive:NN} uses \texttt{\textbackslash tex_\let:D} to change the meaning of this potentially outer token. Then, either turn it into an error or make it equal to the primitive \texttt{#1}. To be more precise, \texttt{#1} may not be defined, so try a \texttt{\textbackslash tex_\ldots:} command as well.

\begin{verbatim}
\cs_new_protected:Npn \_deprecation_primitive:NN \#1\#2 { }
\exp_last_unbraced:NNNNo
\cs_new:Npn \_deprecation_primitive:w \#1 { \token_to_str:N _ } { }
\__kernel_deprecation_code:nn
{ \cs_set_protected:Npn \_kernel_primitive:NN \#1
{ \exp_after:wN \_deprecation_primitive:NN \exp_after:wN \#1 \exp_not:N }
\cs_set_protected:Npn \_deprecation_primitive:NN \#1\#2
{ \tex_let:D \#2 \scan_stop:
\exp_args:NNx \_kernel_deprecation_error:NNn \#2
{ \iow_char:N \\ \cs_if_exist:NTF \#1
{ \cs_to_str:N \#1 }
{ \tex_
\exp_last_unbraced:Nf
\_deprecation_primitive:w \{ \cs_to_str:N \#2 }
\}
{ 2020-01-01 }
\}
\_kernel_primitives:
}
{ \cs_set_protected:Npn \_kernel_primitive:NN \#1
{ \exp_after:wN \_deprecation_primitive:NN \exp_after:wN \#1 \exp_not:N }
\}
\end{verbatim}

(End definition for \_deprecation_old_protected:NNn and \_deprecation_old:Nnn.)
51.5 Loading the patches

When loaded first, the patches are simply read here. Here the deprecation code is loaded with the lower-level \__kernel..., macro because we don’t want it to flip the \g__sys_deprecation_bool boolean, so that the deprecation code can be re-loaded later when using undo-recent-deprecations).

51.6 Deprecated l3box functions

(End definition for \box_eq_clear:NN and \box_gset_eq_clear:N.)
51.7 Deprecated l3int functions

\langle@@=int\rangle
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand
\__int_deprecated_constants:nn
\cs_set_protected:Npn \__int_deprecated_constants:Nn #1#2
{ \cs_if_free:NT #1 { \int_const:Nn #1 {#2} } }
\cs_set_protected:Npn \__int_deprecated_constants:nn { 2021-01-01 } { \__int_deprecated_constants:nn { 2020-01-01 } }
\__int_deprecated_constants:nn \__kernel_deprecation_code:nn
{
  \__int_deprecated_constants:nn
  \__kernel_deprecation_error:Nnn \exp_not:N
  \{  \text{2020-01-01} \}
}

\__int_deprecated_constants:nn
{
  \exp_after:wN \use:nnn
  \exp_after:wN \__int_constdef:Nw \exp_not:N
}
\exp_stop_f:
\}

(End definition for \texttt{\_\_zero} and others.)
\__int_value:w Made public.
\cs_gset_eq:NN \__int_value:w \int_value:w
(End definition for \__int_value:w.)

51.8 Deprecated \texttt{l3luatex} functions
\langle \texttt{@@=lua} \rangle
\lua_now_x:n \lua_escape_x:n \lua_shipout_x:n
\__kernel_patch_deprecation:nnNNpn \text{2020-01-01} \text{\texttt{\_\_lua_now:e}}
\cs_gset_protected:Npn \lua_now_x:n \#1 \text{\texttt{\_\_lua_now:n (#1)}}
\text{\texttt{\_\_lua_escape:n (#1)}}
\text{\texttt{\_\_lua_shipout_e:n}}
\cs_gset_protected:Npn \lua_shipout_x:n \#1 \text{\texttt{\_\_lua_shipout:n (#1)}}
(End definition for \texttt{\_\_lua_now:n}, \texttt{\_\_lua_escape:n}, and \texttt{\_\_lua_shipout_x:n}.)

51.9 Deprecated \texttt{l3msg} functions
\langle \texttt{@@=msg} \rangle
\msg_log:n \msg_term:n
\__kernel_patch_deprecation:nnNNpn \text{2020-01-01} \text{\texttt{\_\_iow_log:n}}
\cs_gset_protected:Npn \msg_log:n \#1
\{
  \text{\_\_iow_log:n { ................................................. }}
  \text{\_\_iow_wrap:n \text{\texttt{\_\_iow:n (. - \#1) (. - )}}} \text{\_\_iow_log:n}
  \text{\_\_iow_log:n{ ................................................. }}
  \}
\__kernel_patch_deprecation:nnNNpn \text{2020-01-01} \text{\texttt{\_\_iow_term:n}}
\cs_gset_protected:Npn \msg_term:n \#1
\{
  \text{\_\_iow_term:n { .......................................................... }}
  \text{\_\_iow_wrap:n \text{\texttt{\_\_iow:n (* - \#1) (* - )}}} \text{\_\_iow_term:n}
  \}

1192
(End definition for \msg_log:n and \msg_term:n)

\msg_interrupt:nnn

\_kernel_patch_deprecation:nnn \{ 2020-01-01 \} \{ [Defined-error-message] \}
\cs_gset_protected:Npn \msg_interrupt:nnn #1#2#3
\{\tl_if_empty:nTF {#3}
{\__msg_old_interrupt_wrap:nn { \c__msg_no_info_text_tl }
{#1 \\\ #2 \\\ \c__msg_continue_text_tl }
}
{\__msg_old_interrupt_wrap:nn { \ #3 }
{#1 \\\ #2 \\\ \c__msg_help_text_tl }
}
\}
\cs_gset_protected:Npn \__msg_old_interrupt_wrap:nn #1#2
\{\iow_wrap:nnnN {#1} { | ~ } { } \__msg_old_interrupt_more_text:n
\iow_wrap:nnnN {#2} { ! ~ } { } \__msg_old_interrupt_text:n
\}
\cs_gset_protected:Npn \__msg_old_interrupt_more_text:n #1
\{
\exp_args:Nx \tex_errhelp:D
{\#1 \iow_newline:
|.............................................}
\group_begin:
\char_set_lccode:nn {'\{} {'\ }
\char_set_lccode:nn {'\}} {'\ }
\char_set_lccode:nn {'\&} {'\!}
\char_set_catcode_active:N \&
\tex_lowercase:D
{\group_end:
\cs_gset_protected:Npn \__msg_old_interrupt_text:n #1
\{
\iow_term:x
\{
\iow_newline:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
\iow_newline:
!
\}
\__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
\{
\__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
\}
51.10 Deprecated l3prg functions

Made public, but used by a few third-parties. It’s not possible to perfectly support a mixture of \__prg_map_break:Nn and \prg_map_break:Nn because they use different delimiters. The following code only breaks if someone tries to break from two “old-style” \__prg_map_break:Nn ... \__prg_break_point:Nn mappings in one go. Basically, the \__prg_map_break:Nn converts a single \__prg_break_point:Nn to \prg_break_point:Nn, and that delimiter had better be the right one. Then we call \prg_map_break:Nn which may end up breaking intermediate looks in the (unbraced) argument #1. It is essential to define the break_point functions before the corresponding break functions: otherwise \debug_on:n {deprecation} \debug_off:n {deprecation} would break when trying to restore the definitions because they would involve deprecated commands whose definition has not yet been restored.

51.11 Deprecated l3str functions
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_lowercase:f }
\cs_gset:Npn \str_lower_case:f { \str_lowercase:f }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:nn }
\cs_gset:Npn \str_upper_case:nn { \str_uppercase:nn }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:f }
\cs_gset:Npn \str_upper_case:f { \str_uppercase:f }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:n }
\cs_gset:Npn \str_fold_case:n { \str_foldcase:n }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:V }
\cs_gset:Npn \str_fold_case:V { \str_foldcase:V }

(End definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n.)

\str_case_x:nn \str_case_x:nnTF \str_if_eq_x_p:nn \str_if_eq_x:nnTF
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_case_e:nn }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_case_e:nnT }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_case_e:nnF }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_case_e:nnTF }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:p:nn }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:nnT }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:nnF }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:nnTF }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:eeT }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:eeF }
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \str_if_eq_x:eeTF }

(End definition for \str_case_x:nnTF and \str_if_eq_x:nnTF.)

51.11.1 Deprecated \tl functions

\tl_set_from_file:Nnn \tl_set_from_file:cnn \tl_gset_from_file:Nnn \tl_gset_from_file:cnn
\__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
\cs_gset_protected:Npn \tl_set_from_file:Nnn \tl_gset_from_file:Nnn \tl_gset_from_file:x:Nnn
\tl_set_from_file:cnn \tl_gset_from_file:x:cnn
\__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
\cs_gaest_protected:Npn \tl_set_from_file:Nnn \tl_gset_from_file:Nnn \tl_gset_from_file:x:Nnn
\cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
\__kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
\cs_gget_protected:Npn \tl_gset_from_file:Nnn \tl_gset_from_file:x:Nnn \tl_gset_from_file:x:cnn

(End definition for \tl_set_from_file:Nnn, \tl_gset_from_file:Nnn, and \tl_gset_from_file:x:Nnn.)
\file_get:nnN {#3} {#2} \l__tl_internal_a_tl
\scan_stop:
\tl_set:Nx \l__tl_internal_a_tl { \l__tl_internal_a_tl }
\exp_args:NNNo \group_end:
\tl_set:Nn #1 \l__tl_internal_a_tl
\cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
\cs_gset_protected:Npn \tl_gset_from_file_x:Nnn #1#2#3
{
\group_begin:
\file_get:nnN {#3} {#2} \l__tl_internal_a_tl
\scan_stop:
\tl_gset:Nx #1 { \l__tl_internal_a_tl }
\group_end:
}\cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }

(End definition for \tl_set_from_file:Nn and others.)

\tl_lower_case:n
\tl_lower_case:nn
\tl_upper_case:n
\tl_upper_case:nn
\tl_mixed_case:n
\tl_mixed_case:nn
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:n }
\cs_gset:Npn \tl_lower_case:n #1
{ \text_lowercase:n {#1} }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:nn }
\cs_gset:Npn \tl_lower_case:nn #1#2
{ \text_lowercase:nn {#1} {#2} }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:n }
\cs_gset:Npn \tl_upper_case:n #1
{ \text_uppercase:n {#1} }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:nn }
\cs_gset:Npn \tl_upper_case:nn #1#2
{ \text_uppercase:nn {#1} {#2} }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:n }
\cs_gset:Npn \tl_mixed_case:n #1
{ \text_titlecase:n {#1} }
\__kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:nn }
\cs_gset:Npn \tl_mixed_case:nn #1#2
{ \text_titlecase:nn {#1} {#2} }

(End definition for \tl_lower_case:n and others.)

\tl_show_analysis:N
\tl_show_analysis:n
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_lowercase:n }
\cs_gset_protected:Npn \tl_show_analysis:N \tl_show_analysis:N
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_lowercase:nn }
\cs_gset_protected:Npn \tl_show_analysis:nn \tl_show_analysis:nn
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_uppercase:n }
\cs_gset_protected:Npn \tl_show_analysis:n \tl_show_analysis:n
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_uppercase:nn }
\cs_gset_protected:Npn \tl_show_analysis:nn \tl_show_analysis:nn
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_titlecase:n }
\cs_gset_protected:Npn \tl_show_analysis:n \tl_show_analysis:n
\__kernel_patch_deprecation:nnNNpn { 2020-01-01 } { \text_titlecase:nn }
\cs_gset_protected:Npn \tl_show_analysis:nn \tl_show_analysis:nn

(End definition for \tl_show_analysis:N and \tl_show_analysis:n.)

51.12 Deprecated l3tl-analysis functions

Simple renames.
51.13 Deprecated \texttt{l3token} functions

\begin{verbatim}
\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \token_get_prefix_spec:N { \cs_prefix_spec:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \token_get_arg_spec:N { \cs_argument_spec:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \token_get_replacement_spec:N { \cs_replacement_spec:N }
\end{verbatim}

(End definition for \texttt{token_get_prefix_spec:N}, \texttt{token_get_arg_spec:N}, and \texttt{token_get_replacement_spec:N}.)

\begin{verbatim}
\char_lower_case:N
\char_upper_case:N
\char_mixed_case:N
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:N
\char_str_fold_case:N
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_lower_case:N { \char_lowercase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_upper_case:N { \char_uppercase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_mixed_case:N { \char_titlecase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_fold_case:N { \char_foldcase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_str_lower_case:N { \char_str_lowercase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_str_upper_case:N { \char_str_uppercase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_str_mixed_case:N { \char_str_titlecase:N }
\end{verbatim}

\begin{verbatim}
\cs_gset:Npn \char_str_fold_case:N { \char_str_foldcase:N }
\end{verbatim}

(End definition for \texttt{char_lower_case:N} and others.)

51.14 Deprecated \texttt{l3file} functions

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

\begin{verbatim}
\cs_gset_protected:Npn \c_term_ior { -1 \scan_stop: }
\end{verbatim}

(End definition for \texttt{c_term_ior}.)
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

! 211
\! 32638
" 10901, 10904, 29521, 31379, 31403, 31409, 31413, 31419, 31423, 31429, 31435, 31442, 31443, 31449, 31453, 31455, 31564
\# 6, 5907, 6214, 10901, 13460, 31248
\$ . 5906, 6212, 10901, 10904, 24754, 31248
\% 5908, 6222, 10901, 13462, 31248
& 5899, 6213, 10901, 10904, 12235, 32638, 32639
&& . 210
\' 29521, 31373, 31400, 31407, 31411, 31416, 31421, 31424, 31426, 31433, 31438, 31439, 31446, 31451, 31454, 31462, 31463, 31510, 31511, 31518, 31519, 31530, 31531, 31536, 31537, 31565, 31566, 31588, 31589, 31592, 31593, 31594, 31595
( . 13996, 14216, 14318, 29543
) . 29543
* . 5592, 5615, 11295, 11297, 11301, 11309
** . 211
+ . 210, 211
, . 15214, 31260
- . 210, 211
- . 286
\( . 29521, 31378, 31466, 31467, 31476, 31477, 31486, 31487, 31504, 31516, 31517, 31567, 31568, 31598, 31599, 31602, 31603
/ . 211
/ . 285
: . 5905
:: 37, 348, 374, 2926, 2927, 2928, 2929, 2930, 2931, 2932, 2934, 2938, 2939, 2942, 2945, 2952, 2955, 2958, 2964, 3051, 3120, 3122, 3127, 3134, 3138, 3141, 3146, 3161, 3202, 3203, 3204, 3205, 3216
::N . 37, 2930, 3051, 3205
::V . 37, 2958
::\_unbraced . 37, 3119
::c . 37, 2932
::e . 37, 2936, 3051
::\_unbraced . 37, 3119, 3161
::error . 263, 31770
::f . 37, 2945, 3204
::f_unbraced . 37, 3119
::n . 37, 385, 2929, 3202, 3205
::o . 37, 2934, 3203
::o_unbraced . 37, 3119, 3204, 3205
::p . 37, 348, 2931
::v . 37, 2958
::v_unbraced . 37, 3119
::x . 37, 2952
::x_unbraced . 37, 3119, 3216
\_< . 210
\= . 15215
\& . 29521, 31376, 31456, 31457, 31472, 31473, 31495, 31496, 31497, 31524, 31525, 31550, 31551, 31604, 31605
\> . 210
? . 210
? commands: . 210
?: . 210
" . 2865, 5901, 6209, 6449, 6450, 6453, 6775, 6778, 6779, 6780, 6781, 6786, 6792, 6797, 6804, 6961, 6964, 6965, 6966, 6968, 6974, 6979, 6984, 7135, 7142, 10901, 12138, 12156, 12158, 12163, 12164, 12188, 12198, 12205, 12220, 12664, 12672, 12679, 12691, 12692, 12717, 12718, 12725, 12746, 12748, 12749, 12781, 12794, 12795, 12808, 12875, 12921, 12937, 12941, 12946, 12953, 13465, 14519, 14525, 14532, 16288, 16300, 16306, 17103, 17106, 17107, 17108, 17115, 17118, 17119, 23341, 23344, 23345, 23370, 23371, 23378, 23379, 23826, 25295, 25408, 25409, 25410, 25431, 26730, 26734, 26739, 26773, 26782, 26786, 26791, 26811, 26813, 26814, 26816, 26819, 26821, 26828, 26832, 26835, 26839, 26841, 26845, 26847, 26853, 26855, 26859, 26861, 26865, 26870, 26872, 26914, 26916, 26921, 26923, 26929, 26934, 26935, 26939, 26943, 26953, 26956, 26960, 26961, 26965, 26973, 27030, 29863, 31246, 32021, 32475, 32613, 32614, 32617, 32618


Index

\{ 4, 5902, 6210, 6454, 10901, 13459, 24080, 26734, 26739, 26786, 26835, 26872, 26961, 26965, 29552, 29553, 29554, 31248, 32636

\} 5, 5903, 6211, 6454, 10901, 13461, 26733, 26739, 26836, 26872, 26961, 26965, 29552, 29553, 29554, 31248, 32637

\^ 7, 10, 104, 195, 196, 197, 198, 2553, 3246, 5904, 6215, 6568, 6569, 6902, 6903, 7084, 7085, 7086, 10901, 10904, 10906, 10912, 10964, 12242, 13377, 13413, 20986, 23425, 23498, 23501, 24026, 24031, 24032, 24033, 24034, 24037, 24048, 24085, 24139, 24141, 24143, 24145, 24147, 24149, 24753, 26335, 26338, 26352, 26355, 26364, 26367, 26370, 26373, 26387, 26390, 29521, 31374, 31401, 31408, 31412, 31417, 31422, 31427, 31434, 31440, 31441, 31447, 31452, 31464, 31465, 31482, 31483, 31490, 31491, 31505, 31506, 31507, 31538, 31539, 31560, 31561, 31562, 31563, 32652

^ 211

\^ 4453, 5909, 6217, 10901, 10904, 13463, 24070, 24074, 24080, 29521, 31252, 31375, 31402, 31414, 31418, 31428, 31444, 31448, 31492, 31493, 31494, 31548, 31549

\u 284, 2457, 5592, 5615, 6216, 6454, 6778, 6779, 6780, 10901, 11287, 12719, 12738, 12845, 12994, 13466, 23542, 24025, 24030, 24074, 24084, 24259, 26384, 29276, 29550, 29551, 31259, 32161, 32506, 32636, 32637

A 5593, 5616

\AA 29525, 30956, 31297

\aa 29525, 30956, 31307

\above 287

\abovedisplayshortskip 288

\abovedisplayskip 289

\abovewithdelims 290

abs 211

\accent 291

acos 213

acosd 213

acot 214

acotd 214

acsc 213

acscc 213

acscd 213

\adjdemi erits 292

\adjustspacing 1010, 1672

\advance 169, 185, 293

\AE 29526, 30957, 31298, 31592

\ae 29526, 30957, 31308, 31593

\afterassignment 294

\aftergroup 295

\alignmark 885, 1764

\aligntab 886, 1765

\asec 213

\asecd 213

\asin 213

\asind 213

assert commands:

\assert_int:n 25571, 26548

atan 214

atand 214

\AtBeginDocument 667, 14462, 30974

\atop 296

\atopwithdelims 297

\attribute 887, 1766

\attributedef 888, 1767

\automaticdiscretionary 889, 1768

\automatichyphenmode 891, 1770

\automatichyphenpenalty 892, 1772

\autoscaling 1290, 2044

\autoscaling 1210, 2045

B 29521, 31386

\badness 298

\baselineskip 299

\batchmode 300

\begin 222, 226, 19070, 23822,
23824, 29539, 29547, 29729, 31244

begin internal commands:

\_\_\_\_neighbor 26471

\begin_angle 894, 1774

\begin{group} 13

20, 38, 42, 48, 67, 143, 163, 274, 301

\begin{in} 609, 1189, 1479

\begin{in} 610, 1480

\belowdisplayshortskip 302

\belowdisplayskip 303

\bfseries 31225

\binoppenalty 304

\bodydir 895, 1863

\bodydirection 896
Index

\bool commands:
\bool_case_false:N ...... 264, 31804
\bool_case_false:nTF ...... 264, 31804, 31814, 31816
\bool_case_true:n ...... 264, 31804
\bool_case_true:nTF ...... 264, 31804, 31806, 31808
\bool_const:Nn ........... 107, 9573
\bool_do_until:Nn ........ 110, 9770
\bool_do_until:nn ........ 111, 9776
\bool_do_while:Nn ........ 110, 9770
\bool_do_while:nn ........ 111, 9776
\bool_gset:N .......... 185, 15650
\bool_gset:Nn ........... 107, 9595
\bool_gset_eq:NN .......... 107, 9591, 24016, 25884
\bool_gset_false:N ........ 107, 6093, 6102, 9579, 25843, 31800
\bool_gset_inverse:N .......... 185, 15658
\bool_gset_inverse:nn ........ 24, 31796
\bool_gset_true:N .......... 107, 6083, 9579, 9965, 9971, 25894, 31800
\bool_if:NTF ............ 107, 238, 2734, 6097, 6106, 9612, 9765, 9767, 9771, 9773, 9963, 9969, 13680, 13687, 15397, 15616, 15625, 15816, 15818, 15820, 15866, 15868, 15870, 15908, 15910, 15912, 15928, 15930, 15932, 15973, 16001, 16020, 16022, 16027, 16034, 16101, 16132, 16142, 16170, 24887, 24896, 25299, 25377, 25395, 25424, 25521, 25741, 25749, 27003, 27008, 28137, 28458, 30175, 31797, 31800, 31822, 32246, 32319
\bool_if:nTF ............. 107, 109, 111, 111, 111, 111, 593, 9626, 9644, 9715, 9722, 9741, 9748, 9757, 9778, 9787, 9791, 9800, 9869, 25380, 31827, 31833, 31887, 32269
\bool_if_exist:NTF .......... 108, 9640, 15415, 15431
\bool_if_exist:p:N .......... 108, 9640
\bool_if_p:N ........... 107, 9012
\bool_if_p:n ............ 109, 537, 9576, 9598, 9603, 9644, 9652, 9722, 9748, 9754, 9758
\bool_lazy_all:nTF ........... 109, 109, 109, 9702, 25164
\bool_lazy_all:p:n ........ 109, 9702
\bool_lazy_and:nTF ........ 109, 109, 109, 9719, 10063, 13259, 13793, 29842, 30463, 30516, 30544, 31148
\bool_lazy_and:p:nn ........ 109, 109, 9719
\bool_lazy_any:nTF ........... 109, 110, 6225, 6249, 6271, 6438, 9728, 14113, 29507, 29614, 30340
\bool_lazy_any:p:n ........... 109, 110, 9728, 13798, 30466
\bool_lazy_or:nTF ........... 110, 110, 9745, 11280, 13988, 14022, 14070, 14206, 23431, 29214, 29415, 29631, 29639, 30127, 30141, 30177, 30182, 30217, 30265, 30301, 30395, 30402, 30480, 30525, 30558, 30589, 30636, 30659, 30693, 31261, 31346
\bool_lazy_or:p:nn ........... 110, 9745, 29845, 30547, 31151
\bool_log:N ........ 107, 9627
\bool_log:n ........ 108, 9621
\bool_new:N ........... 106, 5948, 9571, 9636, 9637, 9638, 9639, 9959, 9960, 13408, 15296, 15297, 15304, 15305, 15309, 15415, 15431, 23867, 24294, 25800, 25801, 25803, 25804, 25805, 27738, 29885, 32216, 32232
\bool_not:p:n ........... 110, 9754, 13796
\bool_set:N ........ 185, 15659
\bool_set:Nn ........... 107, 530, 9595
\bool_set_eq:NN ........ 107, 9521, 24010, 26037
\bool_set_false:N ........... 107, 253, 9579, 13514, 13656, 13664, 13672, 13682, 13689, 15336, 15810, 15811, 15812, 15862, 15863, 15867, 15903, 15911, 15913, 15922, 15923, 15933, 15954, 16008, 24861, 25066, 25779, 25856, 25870, 25916, 25978, 28133, 28843, 31797, 32235
\bool_set_inverse:N .......... 185, 15658
\bool_set_inverse:NN ....... 264, 31796
\bool_set_true:N ........... 107, 267, 9579, 13642, 15331, 15817, 15819, 15821, 15861, 15869, 15871, 15904, 15905, 15909, 15924, 15929, 15931, 15949, 16015, 24866, 25070, 25773, 25976, 26036, 28151, 28173, 28204, 29886, 31797, 32245
\bool_show:N ........ 107, 9627
\bool_show:n ........ 107, 9621
\bool_until_do:Nn .......... 110, 9704
\bool_until_do:nn .......... 111, 9776
\bool_while_do:N ........ 110, 9704
\bool_while_do:nn .......... 111, 9776
\bool_xor:nTF ........... 110, 9755
\bool_xor:p:nn ........... 110, 9755
\c_false_bool .. 22, 106, 334, 367, 531, 533, 534, 534, 535, 2311, 2363, 2364, 2395, 2414, 2419, 2451.
Index

\_clist_if_recursion_tail:\_stop:n \ldots 10211, 10228, 10634, 10665
\_clist_if_wrap:nTF \ldots 552, 10233, 10258, 10299, 10460, 10541
\_clist_if_wrap:w \ldots 552, 10233
\_clist_internal_clist \ldots 555, 10204, 10338, 10339, 10351, 10352, 10535, 10536, 10537, 10612, 10613, 10629, 10630
\_clist_internal_remove_clist \ldots 10436, 10444, 10447, 10448, 10450
\_clist_internal_remove_seq \ldots 10436, 10462, 10463, 10464
\_clist_item:nNN \ldots 10702, 10735
\_clist_item:n:nn \ldots 10733
\_clist_item:nn \ldots 10733
\_clist_item_l_loop:nnw \ldots 10702
\_clist_item_loop:nnw \ldots 10733
\_clist_item_n:nn \ldots 10733
\_clist_item_n:nnnN \ldots 10733
\_clist_map_function:Nw \ldots 562, 10556, 10603
\_clist_map_function:Nn \ldots 562, 10556, 10582
\_clist_map_unbrace:Nw \ldots 562, 10556, 10582
\_clist_map_variable:NNN \ldots 10616
\_clist_pop:NNN \ldots 10369
\_clist_pop:NN \ldots 10369
\_clist_pop:NNNN \ldots 556, 10369, 10414
\_clist_pop:TF:NNN \ldots 10369
\_clist_put_left:NNNN \ldots 10332
\_clist_put_right:NNNN \ldots 10345
\_clist_random:n \ldots 10761
\_clist_random:nw \ldots 10761
\_clist_random_all \ldots 10454
\_clist_random_all:NNNN \ldots 10454
\_clist_random_all:w \ldots 559, 10454
\_clist_reverse_duplicates:NN \ldots 10438
\_clist_reverse:wwNN \ldots 560, 10499
\_clist_reverse_end:ww \ldots 560, 10499
\_clist_sanitize:n \ldots 10220, 10262, 10327, 10329
\_clist_sanitize:Nn \ldots 552, 10220
\_clist_set_from_seq:n \ldots 10280
\_clist_set_from_seq:NNNN \ldots 10280
\_clist_show:NN \ldots 10774
\_clist_show:NNn \ldots 10788
\_clist_tmp:w \ldots 559, 10213, 10467, 10489, 10543, 10552, 10556, 10558
\_clist_trim_next:w \ldots 552, 563, 10214, 10223, 10231, 10585, 10593
\_clist_use:nnn \ldots 10669
\_clist_use:nnnnn \ldots 565, 10669
\_clist_use:wnn \ldots 10669
\_clist_use_i_delimit_by_s:\_stop:nw \ldots 10207, 10729

\_clist_use_i_delimit_by_s:\_stop:w \ldots 559, 10207, 10470, 10715, 10720
\_clist_wrap_item:w 552, 10229, 10257
\closein \ldots 313
\closeout \ldots 314
\clubpenalties \ldots 612, 1482
\clubpenalty \ldots 315
cm \ldots 216
code commands:
\_code:n \ldots 185, 15676
coffin commands:
\_coffin_attach:NnnNNnn \ldots 248, 1165, 28558
\_coffin_clear:N \ldots 248, 27769
\_coffin_display_handles:NN \ldots 248, 28810
\_coffin_dp:N \ldots 249, 27900, 28422, 28461, 28934
\_coffin_gattach:NnnNNnnn \ldots 248, 28558
\_coffin_gclear:N \ldots 248, 27769
\_coffin_gjoin:NnnNNnnn \ldots 248, 28507
\_coffin_gresize:Nnn \ldots 248, 28401
\_coffin_grotate:Nnn \ldots 248, 28420
\_coffin_gscale:NNn \ldots 248, 28419
\_coffin_gset_eq:NN \ldots 246, 27900, 28516, 28567
\_coffin_gset_horizontal_pole:NNn \ldots 247, 28000
\_coffin_gset_vertical_pole:NNn \ldots 247, 28000
\_coffin_ht:N \ldots 249, 27900, 28422, 28461, 28933
\_coffin_if_exist:nTF \ldots 246, 27748, 27762
\_coffin_if_exist:p:N \ldots 246, 27748
\_coffin_join:NnnNNnnn \ldots 248, 28507
\_coffin_log_structure:N \ldots 250, 28920
\_coffin_mark_handle:NNnn \ldots 248, 27855
\_coffin_new:N \ldots 246, 1081, 27787, 27963, 27964, 27965, 27966, 27967, 27968, 27969, 28691, 28701, 28702, 28703
\_coffin_resize:Nnnn \ldots 248, 28401
\_coffin_rotate:Nn \ldots 248, 28240
\_coffin_scale:NNn \ldots 248, 28449
\_coffin_scale:NNNN \ldots 28449
\_coffin_set_eq:NN \ldots 246, 27900, 28510, 28561, 28589, 28617, 28831
\_coffin_set_horizontal_pole:NNn \ldots 247, 28000
\_coffin_set_vertical_pole:NNn \ldots 247, 28000
\_coffin_show_structure:N \ldots 248, 28500
\_coffin_typeset:NNnn \ldots 249, 250, 28920
Index
\cs_gset:Npn . . . . . . . 10, 12, 2176,
2595, 2609, 2611, 4154, 4155, 4192,
4234, 8518, 10990, 12128, 12130,
12168, 13911, 13992, 14092, 14099,
14154, 14176, 15695, 15697, 17528,
30223, 32355, 32588, 32590, 32675,
32677, 32679, 32682, 32684, 32686,
32688, 32690, 32692, 32694, 32696,
32698, 32700, 32702, 32704, 32706,
32708, 32710, 32712, 32749, 32752,
32755, 32758, 32761, 32764, 32771,
32773, 32775, 32777, 32779, 32781,
32783, 32785, 32787, 32789, 32791
\cs_gset:Npx . . . . . . . . . . . . . . 12,
2176, 2596, 2609, 2612, 8523, 26147
\cs_gset_eq:NN . . . . . . . . . . . . . . .
. 15, 2627, 2644, 2652, 4263, 4291,
6175, 6179, 8078, 8297, 8528, 8533,
9584, 9586, 10136, 10919, 11750,
12041, 12049, 13125, 13341, 32585
\cs_gset_nopar:Nn . . . . 14, 2715, 2779
\cs_gset_nopar:Npn . . . . . . . . . . . .
. . . . 12, 2176, 2593, 2601, 2605, 3962
\cs_gset_nopar:Npx . . . . . . . . . . . .
. 12, 1177, 2176, 2594, 2601, 2606,
4248, 4269, 4274, 4320, 4322, 4324,
4340, 4342, 4344, 4346, 4364, 4366,
4368, 4370, 32028, 32054, 32059, 32086
\cs_gset_protected:Nn
14, 2715, 2779
.cs_gset_protected:Np . . . 185, 15686
\cs_gset_protected:Npn . . . . . . 12,
2176, 2599, 2621, 2623, 4742, 5541,
9099, 10601, 12044, 13223, 14839,
15699, 15701, 19127, 23009, 23017,
23030, 23435, 23703, 23708, 31923,
32271, 32293, 32302, 32312, 32350,
32516, 32523, 32526, 32531, 32535,
32539, 32592, 32595, 32602, 32609,
32621, 32626, 32643, 32715, 32719,
32728, 32739, 32767, 32769, 32793
\cs_gset_protected:Npx . . . . . . . . .
. . . . . . . . . . . . . . 12, 2176, 2600,
2621, 2624, 9110, 14846, 19134, 32282
\cs_gset_protected_nopar:Nn . . . .
. . . . . . . . . . . . . . . . . 14, 2715, 2779
\cs_gset_protected_nopar:Npn . . .
. . . . . . . . 12, 2176, 2597, 2615, 2617
\cs_gset_protected_nopar:Npx . . .
. . . . . . . . 12, 2176, 2598, 2615, 2618
\cs_if_eq:NNTF . . . . . . . . . . . . 22,
1185, 2811, 2818, 2819, 2822, 2823,
2826, 2827, 10178, 12178, 15598,
17292, 17302, 17328, 17330, 17332,
17533, 25149, 29764, 29787, 29828,

1212
29831, 30083, 31137, 32302, 32312
\cs_if_eq_p:NN
22, 2811, 25166, 32271
\cs_if_exist . . . . . . . . . . . . . . . . 233
\cs_if_exist:N . . . . . . . . 22, 4300,
4301, 8176, 8178, 8839, 8841, 9640,
9642, 10322, 10324, 11977, 11979,
14605, 14607, 14950, 14952, 15046,
15048, 18861, 18862, 27071, 27073
\cs_if_exist:NTF . . . . . . . . 16, 22,
313, 365, 443, 585, 616, 2479, 2536,
2538, 2540, 2542, 2544, 2546, 2548,
2550, 2831, 2936, 3006, 3042, 3132,
3156, 3224, 3255, 3506, 3772, 5400,
6146, 6155, 6159, 6173, 8281, 8814,
8815, 8816, 8817, 9493, 9494, 9540,
9885, 9886, 9887, 9889, 9893, 10061,
10176, 10987, 11146, 11446, 11465,
12099, 12928, 13049, 13053, 13081,
13144, 13282, 13286, 13737, 13931,
14034, 14311, 14460, 14564, 15342,
15451, 15542, 15994, 16036, 16044,
16056, 16068, 16075, 16086, 16104,
16118, 16193, 16237, 16245, 16390,
17526, 17701, 18788, 23007, 23025,
23028, 24883, 26261, 27750, 27752,
28754, 29803, 29819, 30105, 30113,
30246, 30972, 30982, 30989, 31325,
31961, 31980, 32476, 32498, 32512
\cs_if_exist_p:N . . . . . . . 22, 313,
2479, 9901, 11281, 11282, 13989,
14023, 14071, 14207, 29843, 31149
\cs_if_exist_use:N . . . . . . . . . . . .
. . . . . . . . . 16, 336, 2535, 12484,
12502, 13457, 16070, 16090, 25331
\cs_if_exist_use:NTF . . . . . . . . . .
. 16, 2535, 2537, 2539, 2545, 2547,
3802, 3871, 4107, 10151, 15547,
16974, 17659, 17661, 24117, 24124,
24488, 24493, 24530, 24951, 25037,
26196, 30125, 30137, 30200, 30202
\cs_if_free:NTF . . . . 22, 104, 616,
2507, 2576, 3714, 3741, 12474, 32567
\cs_if_free_p:N . . . 21, 22, 104, 2507
\cs_log:N . . . . . . . . . . . . 16, 345, 2856
\cs_meaning:N . . . . . . . . . . . . . . . .
15, 322, 2124, 2140, 2148, 2868, 10058
\cs_new:Nn . . . . . . . 12, 105, 2715, 2779
\cs_new:Npn . . . . . . . . . . . . . . . . . .
. . . 10, 11, 14, 104, 104, 372, 943,
1185, 2266, 2283, 2585, 2609, 2613,
2686, 2688, 2690, 2698, 2750, 2816,
2817, 2818, 2819, 2820, 2821, 2822,
2823, 2824, 2825, 2826, 2827, 2872,
2876, 2885, 2894, 2903, 2906, 2915,


Index

31180, 31323, 31733, 31741, 31754,
31756, 31758, 31760, 31766, 31804,
31806, 31808, 31810, 31812, 31814,
31816, 31818, 31820, 31825, 31831,
31837, 31840, 31841, 31851, 31859,
31861, 31867, 31873, 31915, 31930,
32103, 32111, 32131, 32133,
32134, 32137, 32139, 32141, 32143,
32151, 32219, 32221, 32251, 32461
\cs_new:Npx

11, 34, 34, 366,
367, 2585, 2609, 2614, 3565, 10653,
10663, 13430, 13442, 13812, 13820,
13968, 16722, 17560, 18147, 19293,
21302, 23615, 24138, 24140, 24142,
27973, 27974, 27975, 28069, 28970,
28998, 28999, 29000, 29059, 30097,
30192, 30221, 30257, 30259, 30261,
30654, 30656, 30620, 30641, 32450
\cs_new_eq:NN

15, 2715, 2779
\cs_new_nopar:Nn

11, 337, 338, 2585, 2601, 2607
\cs_new_nopar:Np

11, 337, 338, 2585, 2601, 2608
\cs_new_nopar:NN

15, 2715, 2779
\cs_new_protected:Nn

11, 372,
1185, 2270, 2287, 2585, 2621, 2625,
2627, 2628, 2629, 2630, 2631, 2632,
2633, 2634, 2635, 2640, 2641, 2642,
2643, 2645, 2700, 2710, 2712, 2723,
2732, 2829, 2838, 2840, 2842, 2844,
2846, 2854, 2856, 2857, 2859, 2860,
2862, 2917, 2952, 3117, 3146, 3216,
3247, 3539, 3552, 3570, 3574, 3577,
3586, 3610, 3727, 3737, 3746, 3770,
3778, 3790, 3798, 3809, 3811, 3813,
3815, 3817, 3828, 3830, 3843, 3851,
3862, 3881, 3883, 3885, 3887, 3889,
3959, 4061, 4063, 4074, 4081, 4087,
4105, 4114, 4119, 4124, 4129, 4134,
4140, 4145, 4152, 4158, 4167, 4177,
4179, 4181, 4183, 4185, 4190, 4235,
4260, 4266, 4271, 4278, 4280, 4284,
4286, 4290, 4291, 4294, 4296, 4313,
4315, 4317, 4319, 4321, 4323, 4331,
4333, 4335, 4337, 4339, 4341, 4343,
4345, 4355, 4357, 4359, 4361, 4363,
4365, 4367, 4369, 4373, 4393, 4395,
4397, 4409, 4428, 4443, 4461, 4483,
4485, 4487, 4489, 4495, 4517, 4523,
4552, 4554, 4558, 4560, 4633, 4634,
4635, 4739, 4750, 4768, 4774, 4776,
4851, 4853, 4963, 4965, 5244, 5246,
5248, 5253, 5255, 5268, 5339, 5341,
5343, 5345, 5351, 5366, 5379, 5381,
5385, 5387, 5358, 5553, 5562, 5572,
5574, 5924, 6044, 6060, 6076, 6082,
6086, 6088, 6108, 6126, 6134, 6144,
6153, 6207, 6255, 6263, 6265, 6267,
6277, 6283, 6307, 6318, 6324, 6327,
Index

15244, 15498, 15691, 15693, 16155, 16953, 17447, 17524, 18108, 18182, 18196, 18418, 18435, 18470, 18506, 18521, 18538, 20089, 23328, 24714, 24739, 24748, 25278, 25287, 25289, 25291, 25294, 25296, 25303, 25305, 25310, 25312, 25317, 25319, 25321, 25323, 25326, 26024, 26025, 26452, 27871, 27884, 27918, 28201, 29027, 29242, 29252, 29262, 29287, 29302, 29320, 29328, 29360, 30663, 30666, 30697, 30708, 30724, 30922, 30925, 30946, 30977, 31249, 32030, 32116, 32121, 32129, 32148, 32191, 32498, 32499, 32499, 32499, 32499
\cs_set_protected:Npx 11, 240, 2162, 2021, 2022, 12490, 15502, 32509
\cs_set_protected:nopar:Nn ???????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
Index

\currentiftype \currentspacingmode \currentxspacingmode \currentiflevel

\_cs\_generate\_variant\_same:N \_cs\_generate\_variant\_T:-\_form:mmn \_cs\_generate\_variant\_TP:-\_form:mmn \_cs\_get\_function\_name:N \_cs\_get\_function\_signature:N \_cs\_parm\_from\_arg\_count:-\_test:mmTF \_cs\_split\_function\_aux:i:w \_cs\_split\_function\_auxii:w \_cs\_tmp:w \_cs\_to\_str:N \_cs\_to\_str:w \_cs\_use\_i\_delimit\_by\_s\_stop:nw \_cs\_use\_none\_delimit\_by\_q:-\_recursion\_stop:w \_cs\_use\_none\_delimit\_by\_s:-\_stop:w \csc \cscd \csname \csstring \currentcjktoken \currentgrouplevel \currentgroupotype \currentifbranch \currentiflevel

\debug\_commands:\\debug\_off:: \debug\_off:n \debug\_on:: \debug\_on:n \debug\_resume: \debug\_suspend:

debug\_internal\_commands:\g__debug\_deprecation\_off\_tl \g__debug\_deprecation\_on\_tl

def\def: \default\_commands:\.default:n \default\_auxchar \default\_char \default\_not\_char \default\_separator \default\_shortfall

deprecation\_commands:\\_\_dep\_comparison\_same:NnTF \_\_dep\_comparison\_aux:w \_\_dep\_period:-\_bool \_\_dep\_just\_error:mmNN \_\_dep\_month:-\_months:w \_\_dep\_not\_yet\_deprecated:nTF \_\_dep\_old:Nnn \_\_dep\_old\_protected:Nnn \_\_dep\_patch\_aux:Nn \_\_dep\_patch\_aux:mmNNnn

D
\d \day \dd \deadcycles \debug\_commands:\debug\_off:: \debug\_off:n \debug\_on:: \debug\_on:n \debug\_resume: \debug\_suspend:

debug\_internal\_commands:\g__debug\_deprecation\_off\_tl \g__debug\_deprecation\_on\_tl

def\def: \default\_commands:\.default:n \default\_auxchar \default\_char \default\_not\_char \default\_separator \default\_shortfall

deprecation\_commands:\\_\_dep\_comparison\_same:NnTF

1221
Index

\_deprecation\_primitive:NN ........ 1189, 32459
\_deprecation\_primitive:w ... 32459
\_deprecation\_warn\_once:nnNN ... 32257
\detokenize ... 62, 222, 618, 1488
\DH ... 29527, 30958, 31299
\dh ... 29527, 30958, 31309
dim commands:
\dim\_abs:n ....... 169, 679, 14639
\dim\_add:Nn ....... 169, 14621
\dim\_case:nn ....... 172, 14719
\dim\_case:nnnn ....... 32365
\dim\_case:nnnTF .......
...... 172, 14719, 14724, 14729, 32366
\dim\_compare:nnnTF ....
...... 170, 171, 172, 173, 202,
14673, 14743, 14779, 14877, 14796,
14802, 14814, 14817, 14828, 14847,
28150, 28155, 28169, 28172, 28177,
28255, 28250, 28254, 28269, 282681,
31643, 31660, 31694, 31708, 31718
\dim\_compare:nTF ....
...... 170, 171, 173, 173,
179, 14679, 14751, 14759, 14768, 14774
\dim\_compare:p:n ....... 171, 14679
\dim\_compare:p:nnn ....... 170, 14674
\dim\_const:Nn ........ 168,
672, 682, 14588, 14918, 14919, 16322
\dim\_do\_until:nn ....... 170, 171, 172, 173, 202,
14591, 14722, 14727, 14732, 14737,
14832, 14860, 14913, 14917, 27857,
27927, 28013, 28031, 28068, 28072,
28073, 28077, 28081, 28082, 28099,
28104, 28110, 28117, 28124, 28267,
28291, 28294, 28295, 28302, 28384,
28385, 28392, 28393, 28496, 28503,
28562, 28563, 28532, 28534, 28935
\dim\_gadd:NN ........ 169, 14621
.\_dim\_gset:N ........ 186, 15710
.\_dim\_gset:NN ........ 169, 672, 14699
.\_dim\_gset_eq:NN ..... 169, 14615
.\_dim\_gsub:NN ........ 169, 14621
.\_dim\_gzero:N ....... 168, 14594, 14602
.\_dim\_gzero\_new:N ....... 168, 14599
.\_dim\_if\_exist:NNTF ....
...... 168, 14600, 14602, 14605
.\_dim\_if\_exist:p:N ...... 168, 14605
.\_dim\_log:N ........ 176, 14914
.\_dim\_log:n ........ 176, 14914
.\_dim\_max:nn ........ 169, 14639, 28363, 28367
.\_dim\_min:nn .......... 169, 14639, 28361, 28365, 28378
.\_dim\_new:N .......... 168, 168, 14580, 14590, 14600,
14602, 14920, 14921, 14922, 14923,
27361, 27362, 27363, 27364, 27365,
27366, 27367, 27368, 27715, 27739,
27740, 27743, 27744, 27745, 27746,
28235, 28236, 28237, 28238, 28239,
28399, 28400, 28741, 28743, 28744
.\_dim\_ratio:nn ....... 170, 680, 14670, 14907
.\_dim\_set:N ........ 186, 15710
.\_dim\_set:NN ........ 169, 14609,
27388, 27389, 27390, 27422, 27433,
27517, 27518, 27519, 27534, 27632,
27633, 27634, 27636, 27638, 27640,
27844, 27913, 28149, 28153, 28171,
28175, 28206, 28220, 28297, 28332,
28340, 28351, 28352, 28353, 28354,
28360, 28362, 28364, 28366, 28371,
28377, 28460, 28462, 28464, 28472,
28474, 28528, 28560, 28604, 28606,
28608, 28626, 28627, 28742, 28847,
28848, 28899, 28900, 28901, 28903
.\_dim\_set\_eq:NN ........ 169, 14615, 27846, 27847, 27915, 27916
.\_dim\_show:N ........ 175, 14910
.\_dim\_show:n ........ 176, 681, 14912
.\_dim\_sign:n ........ 174, 14862
.\_dim\_step\_function:nnnN ....
...... 173, 678, 14805, 14587
.\_dim\_step\_inline:nnnn .... 173, 14835
.\_dim\_step\_variable:nnnN .... 174, 14835
.\_dim\_sub:NN ........ 169, 14621
.\_dim\_to\_decimal:n ........ 174, 14883, 14899, 14904
.\_dim\_to\_decimal\_in_bp:n ....
...... 175, 14898
.\_dim\_to\_decimal\_in_sp:n .... 175,
175, 770, 14900, 17587, 17624, 18221
.\_dim\_to\_decimal\_in\_unit:nn 175, 14902
.\_dim\_to\_fp:n ......... 175, 770, 789, 14910,
22415, 27426, 27427, 27437, 27438,
27506, 27509, 27510, 27535, 27550,
27551, 27570, 27571, 27589, 27606,
27609, 27610, 28160, 28161, 28162,
28182, 28183, 28184, 28194, 28195,
28211, 28212, 28213, 28224, 28224,
28225, 28336, 28337, 28344, 28345,
28418, 28421, 28422, 28473, 28475
.\_dim\_until:do:nn ....... 173, 14749
.\_dim\_until:do:nnnN ....... 172, 14777
\endinput .......... 137, 349 \etex .............. 1516
\endL .............. 621, 1492 \etex_output:D ....... 1517
\endlinechar ....... 221, 234, 350 \etex_mutable:D ...... 1518
\endR .............. 622, 1493 \etex_numexpr:D ...... 1519
\enquote ........... 19063 \etex_pagediscards:D ... 1520
\ensuremath ....... 351 \etex_pargen:D .......... 1521
\errorcontextlines 354 \etexparindent:D ....... 1522
\errorstopmode .... 355 \etexparshape:D ....... 1523
\errormessage ...... 117, 129, 353 \etexparshapelength:D 1524
\errhelp .......... 29072 \etexparshapeindent:D .... 1525
\errmessage ...... 11043 \etexparselength:D .... 1526
\errmsg ........... 109, 128, 352 \etexprotected:D .... 1527
\etex .............. 625, 1494 \etex_readline:D .... 1528
\ethyphenpenalty . 364 \etex_redline:D .......... 1529
\etex ................ 363 \etex_right:D .......... 1530
\etex ................ 362 \etex_rightline:D ..... 1531
\etex ................ 361 \etex_rightlinepenalties:D 1532
\etex ................ 360 \etex_rightmark:D ...... 1533
\etex ................ 359 \etex_rightmargin:D ..... 1534
\etex ................ 358 \etex_rightmarks:D ..... 1535
\etex ................ 357 \etex_rule:D .......... 1536
\etex ................ 356 \etex_rulelength:D ..... 1537
\etex ................ 355 \etex_rulemode:D ..... 1538
\etex ................ 354 \etex_rulepenalty:D .... 1539
\etex ................ 353 \etex_rulenode:D ....... 1540
\etex ................ 352 \etex_runchars:D ..... 1541
\etex ................ 351 \etex_runchar:D ...... 1542
\etex ................ 350 \etex_runcharskip:D ... 1543
\etex ................ 349 \etex_runcharpenalty:D 1544
\etex ................ 348 \etex_runcharpenalty:D 1545
\etex ................ 347 \etex_runcharpenalties:D 1546
\etex ................ 346 \etex_runcharpenalties:D 1547
\etex ................ 345 \etex_runcharskip:D ... 1548
\etex ................ 344 \etex_runcharskip:D ... 1549
\etex ................ 343 \etex_runcharskip:D ... 1550
\etex ................ 342 \etex_runcharskip:D ... 1551
\etex ................ 341 \etex_runcharskip:D ... 1552
\etex ................ 340 \etex_runcharskip:D ... 1553
790, 1177, 2122, 2258, 2260, 2948, 2960, 2961, 2966, 2967, 2972, 2977, 2979, 2982, 2991, 2993, 2996, 2997, 2998, 3001, 3003, 3005, 3009, 3014, 3019, 3022, 3027, 3032, 3033, 3034, 3038, 3039, 3040, 3046, 3047, 3054, 3055, 3056, 3060, 3061, 3062, 3066, 3067, 3068, 3072, 3073, 3074, 3078, 3079, 3080, 3084, 3085, 3086, 3087, 3091, 3092, 3093, 3094, 3098, 3099, 3100, 3105, 3106, 3107, 3108, 3112, 3113, 3114, 3115, 3121, 3124, 3125, 3129, 3130, 3134, 3144, 3151, 3153, 3155, 3159, 3163, 3165, 3168, 3169, 3174, 3175, 3179, 3182, 3183, 3187, 3190, 3191, 3192, 3197, 3198, 3199, 3207, 3210, 3211, 3212, 3213, 3218, 3220, 3222, 3223, 3224, 3234, 3237, 3242, 3267, 3268, 3269, 3280, 3281, 3293, 3294, 3295, 3300, 3308, 3309, 3310, 3311, 3312, 3313, 3336, 3337, 3338, 3339, 3349, 3340, 3342, 3424, 3429, 3430, 3432, 3443, 3445, 3449, 3450, 3451, 3454, 3456, 3459, 3463, 3464, 3465, 3470, 3471, 3478, 3482, 3488, 3490, 3494, 3495, 3498, 3499, 3509, 3556, 3560, 3582, 3589, 3609, 3752, 3754, 3781, 3782, 3784, 3821, 3823, 3840, 3858, 3869, 3975, 3981, 3983, 4007, 4058, 4059, 4138, 4172, 4213, 4221, 4232, 4390, 4412, 4413, 4414, 4415, 4474, 4475, 4476, 4537, 4538, 4539, 4544, 4586, 4597, 4598, 4623, 4669, 4671, 4789, 4918, 4945, 4976, 4981, 4982, 4983, 4985, 4999, 5009, 5026, 5050, 5060, 5063, 5080, 5081, 5091, 5096, 5097, 5112, 5113, 5114, 5172, 5174, 5175, 5176, 5181, 5182, 5184, 5262, 5263, 5520, 5521, 5533, 5588, 5611, 5645, 5646, 5657, 5658, 5666, 5674, 5676, 5683, 5688, 5706, 5707, 5708, 5720, 5721, 5748, 5750, 5756, 5762, 5776, 5796, 5807, 5823, 5831, 5839, 5846, 5853, 5865, 5953, 5967, 5986, 5995, 6016, 6017, 6022, 6023, 6048, 6049, 6064, 6065, 6113, 6118, 6183, 6356, 6365, 6411, 6527, 6563, 6655, 6580, 6586, 6750, 6752, 6817, 6836, 6837, 6851, 6852, 6879, 6880, 6995, 7017, 7030, 7031, 7058, 7153, 7174, 7183, 8171, 8173, 8185, 8193, 8337, 8371, 8383, 8396, 8397, 8398, 8420, 8421, 8466, 8501, 8502, 8503, 8563, 8574, 8575, 8601, 8602, 8605, 8701, 8715, 8720, 8723, 8724, 8731, 8732,
Index

F

fact ............................... 211
false ............................... 216
\\fam ............................... 366
\fi ................................. 17, 35, 41, 51, 64,
65, 66, 91, 94, 96, 97, 98, 101, 102,
131, 140, 153, 154, 171, 187, 205, 367
fi commands:
\fi: .................................. 23, 100, 100,
101, 105, 112, 112, 163, 182, 245,
245, 245, 247, 248, 288, 330, 333, 334,
359, 391, 396, 398, 429, 430, 433,
512, 521, 541, 575, 655, 741, 766,
782, 813, 949, 2100, 2147, 2315,
2323, 2331, 2339, 2359, 2364, 2377,
2385, 2393, 2495, 248, 2423, 2430,
2457, 2462, 2488, 2489, 2497, 2503,
2516, 2517, 2525, 2531, 2651, 2672,
2682, 2696, 2753, 2814, 2911, 2921,
2975, 2978, 2985, 2986, 3252, 3259,
3269, 3282, 3295, 3309, 3303, 3304,
3305, 3306, 3314, 3323, 3339, 3403,
3431, 3437, 3446, 3451, 3457, 3460,
3464, 3472, 3482, 3491, 3495, 3499,
3567, 3583, 3590, 3599, 3613, 3614,
3619, 3620, 3621, 3639, 3640, 3641,
3642, 3643, 3644, 3645, 3646, 3647,
3655, 3674, 3676, 3706, 3707, 3708,
3755, 3785, 3857, 3868, 3878, 3976,
3984, 4008, 4022, 4030, 4041, 4051,
4071, 4101, 4102, 4174, 4197, 4216,
4219, 4400, 4407, 4543, 4544, 4570,
4580, 4591, 4606, 4614, 4629, 4641,
4645, 4675, 4690, 4919, 4972, 4977,
4986, 4988, 5016, 5037, 5055, 5064,
5073, 5079, 5086, 5091, 5102, 5106,
5114, 5173, 5185, 5234, 5240, 5241,
5426, 5433, 5439, 5534, 5602, 5606,
5607, 5625, 5678, 5691, 5733, 5739,
5740, 5751, 5764, 5765, 5786, 5824,
5850, 5954, 5960, 5968, 5979, 5996,
5998, 6142, 6194, 6195, 6200, 6203,
6204, 6244, 6297, 6376, 6377, 6412,
6413, 6415, 6423, 6610, 6643, 6679,
6680, 6711, 6712, 6732, 6733, 6751,
6834, 6838, 6848, 6858, 6874, 6878,
6881, 6886, 6888, 6931, 6935, 6936,
7027, 7032, 7043, 7049, 7061, 7064,
7066, 7070, 7184, 7198, 7199, 8237,
8240, 8277, 8338, 8355, 8365, 8419,
8424, 8710, 8711, 8720, 8743, 8760,
8761, 8763, 8780, 8781, 8824, 8878,
8886, 8913, 8921, 8927, 8950, 8988,
8996, 9081, 9296, 9329, 9377, 9382,
18934, 18936, 18937, 18943, 18944,
18947, 18974, 18982, 18983, 18991,
18992, 18994, 19166, 19179,
19189, 19190, 19195, 19196, 19197,
19198, 19199, 19200, 19207, 19217,
19224, 19235, 19236, 19247, 19264,
19309, 19310, 19317, 19330, 19345,
19355, 19369, 19399, 19408, 19442,
19464, 19482, 19499, 19516, 19517,
19519, 19520, 19525, 19540, 19573,
19602, 19603, 19604, 19605, 19606,
19619, 19663, 19736, 19803, 19805,
19806, 19816, 19845, 19848, 19849,
19860, 19880, 19943, 19944, 19945,
19957, 19996, 19997, 19998, 19999,
20005, 20008, 20010, 20020, 20038,
20053, 20065, 20072, 20319, 20323,
20325, 20329, 20336, 20337, 20347,
20348, 20351, 20443, 20513, 20524,
20536, 20548, 20565, 20586, 20602,
20609, 20670, 20727, 20737, 20739,
20749, 20767, 20768, 20800, 20803,
20812, 20814, 20816, 20830, 20844,
20868, 20952, 20960, 20991, 20999,
21005, 21016, 21019, 21022, 21031,
21041, 21043, 21049, 21056, 21059,
21068, 21076, 21097, 21130, 21131,
21158, 21160, 21178, 21179, 21198,
21209, 21218, 21221, 21232, 21235,
21238, 21256, 21266, 21277, 21279,
21288, 21335, 21350, 21365, 21380,
21395, 21410, 21413, 21415, 21432,
21477, 21784, 21820, 21821, 21831,
21872, 21873, 21897, 21924, 21925,
21928, 21930, 21931, 21936, 21948,
21967, 21972, 21980, 21983, 22015,
22025, 22026, 22036, 22058, 22073,
22091, 22099, 22102, 22130, 22138,
22154, 22226, 22244, 22280, 22298,
22331, 22354, 22360, 24333, 24343,
22565, 22566, 22575, 22582, 22587,
22597, 22607, 22632, 22635, 22648,
22680, 22688, 22689, 22717, 22739,
22740, 22741, 22747, 22749, 22769,
22770, 22822, 22823, 22864, 22872,
22925, 22931, 22944, 22990, 23002,
23003, 23058, 23089, 23131, 23142,
23151, 23160, 23215, 23225, 23235,
23349, 23351, 23405, 23409, 23413,
23440, 23451, 23469, 23470, 23471,
23478, 23494, 23500, 23503, 23511,
23521, 23536, 23544, 23552, 23563,
23579, 23599, 23612, 23634, 23652,
23660, 23662, 23665, 23672, 23677,
\file_get:nnN ................. 164, 13917, 14014, 14083, 
14100, 14107, 14168, 14266, 14267 
\file_get:nnNTF ............... 164, 13868, 32714, 32716, 32718, 
32722, 32727, 32731, 32738, 32742 
\file_get:nnN ... 164, 13868, 13870 
\file_get_full_name:NN ........... 164, 315, 14005, 32368 
\file_get_full_name:nNTF ... 164, 
13070, 13875, 14005, 14007, 14019, 
14020, 14324, 14330, 14335, 14347 
\file_get_hex_dump:NN ... 165, 
14182 
\file_get_hex_dump:nnNN ... 165, 14228 
\file_get_hex_dump:nnNTF ...... 165, 14228, 14230 
\file_get_hex_dump:nnNTF ...... 165, 14182, 14183 
\file_get_mdfive_hash:NN ......... 165, 14184, 14192 
\file_get_mdfive_hash:n\file_get:
size:NN ................. 14182 
\file_get_mdfive_hash:n\file_get:
size:nNTF .................. 14182 
\file_get_mdfive_hash:n\file_get:
size:nNTF 165, 14185 
\file_get_size:nNTF .... 166, 14187 
\file_get_size:nNTF .... 166, 14182 
\file_get_timestamp:nNTF .......... 166, 14182 
\file_get_timestamp:nNTF .......... 166, 14182, 14189 
\file_hex_dump:n .... 165, 165, 14104 
\file_hex_dump:nn .......... 165, 165, 14104, 14237 
\file_if_exist:nNTF ........... 164, 164, 166, 6161, 14322, 
14554, 14556, 14560, 32370, 32372 
\file_if_exist_input:n ... 167, 14328 
\file_if_exist_input:nTF ......... 167, 14328, 32369, 32371 
\file_input:n ...... 166, 167, 167, 
167, 6165, 14345, 32370, 32372, 32517 
\file_input_stop: .... 167, 14339 
\file_list: 32373 
\file_log_list: ... 167, 14440, 32374 
\file_mdfive_hash:n ... 165, 165, 14074 
\file_parse_full_name:NNNN ...... 165, 14048, 14377, 14400 
\file_path_include:n ... 167, 32375 
\file_path_remove:n ........ 32377 
\L_file_search_path_seq ...... 164, 164, 165, 165, 166, 
166, 13757, 13928, 14031, 32376, 32378 
\file_show_list: ... 167, 14440 
\file_size:n ........ 166, 166, 14074 
\file_timestamp:n ........ 166, 166, 14074 
file internal commands: 
\__file_base_name_tl ........... 13752, 14028, 14066 
\__file_compare_timestamp:nnN ... 14262 
\__file_const:nn ........ 14568 
\__file_details:nn ........ 14074 
\__file_details_aux:nn . 14074, 14121 
\__file_dir_str ............ 13754, 14049, 14378, 14379 
\__file_ext_check:n ...... .. 13939, 13950, 13957 
\__file_ext_check:nn . 13972, 13977 
\__file_ext_check:nw .... 13963, 13968 
\__file_ext_check:nw .... 13958, 13959, 13966 
\__file_ext_str .......... 13754, 14049, 14050, 14378, 14381 
\__file_full_name:n ........ 13917 
\__file_full_name_aux:n .... 13917 
\__file_full_name_aux:nn .... 13917 
\__file_full_name_tl ........ 13752, 13875, 13878, 
14040, 14042, 14048, 14053, 14055, 
14058, 14065, 14067, 14324, 14330, 
14331, 14335, 14336, 14347, 14348 
\__file_get_auxx:nN .... 13868 
\__file_get_details:nnN .... 14182 
\__file_get_do:Nv .... 13868 
\__file_get_full_name_search:nN . 14005 
\__file_hex_dump:n .... 14101 
\__file_hex_dump:auxx:nN .... 14104 
\__file_hex_dump:auxix:nNnnn ... 14104 
\__file_hex_dump:auxixi:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_hex_dump:auxivx:nNnnn ... 14104 
\__file_if_recursion_tail-
break:NN .................. 13766 
\__file_if_recursion_tail_stop:N 
\__file_if_recursion_tail_stop-
do:NN .................. 13766 
\__file_if_recursion_tail_stop-
do:NN .................. 13766, 13792 
\__file_input:n . 14331, 14336, 14345 
\__file_input_pop: ........ 14345 
\__file_input_pop:nn .... 14345 
\__file_input_push:n .... 14345
Index

1240

\_fp_atan_Taylor_loop:www

\_fp_atan_test_o:Nww

\_fp_atanii_o:Nww

\_fp_basics_pack_high:NNNNww

\_fp_basics_pack_high_carry:w

\_fp_basics_pack_low:NNNNww

\_fp_basics_pack_weird_high:NNNNNNNNw

\_fp_basics_pack_weird_low:NNNNw

\c__fp_big_leading_shift_int

\c__fp_big_middle_shift_int

\c__fp_big_trailing_shift_int

\c__fp_big_leading_shift_int

\c__fp_big_middle_shift_int

\c__fp_big_trailing_shift_int

\c__fp_block_int

\_fp_case_return:o:Nw

\_fp_case_return:i_o:ww

\_fp_case_return_ii_o:ww

\_fp_case_return_o:Nw

\_fp_case_return_o:Nww

\_fp_case_return_same_o:w

\_fp_case_use:nw

\_fp_change_func_type:NNN

\_fp_change_func_type_aux:w

\_fp_change_func_type_chk:NNN

\_fp_chk:w

\_fp_compare:wNNww

\_fp_compare_aux:wn

\_fp_compare_back:ww

\_fp_compare:wNNww

\_fp_compare_aux:wn

\_fp_compare_back:ww

\_fp_compare:wNNww

\_fp_compare_aux:wn

\_fp_compare_back:ww

\_fp_compare:wNNww

\_fp_compare_aux:wn

\_fp_compare_back:ww

\_fp_compare:wNNww

\_fp_compare_aux:wn

\_fp_compare_back:ww
Index

\_fp_sqrt_npos o:w ... 19850, 19853
\_fp_sqrt o:w ......... 19292, 19843
\_fp_step:NNnnn ............ 19123
\_fp_step:NNnnnn ............ 19053
\_fp_step:wwN ............ 19053
\_fp_step_fp:wwN ............ 19053
\_fp_str_if_eq:nn ...... 16933,
18043, 18057, 18345, 18389, 21037
\_fp_sub_back_far_o:NNnnnnN ...
.................... 823, 19463, 19509
\_fp_sub_back_near_after:o:NNNNw ...
.......................... 19469, 19547
\_fp_sub_back_near_o:nnnnnnnnN .
.......................... 822, 19459, 19469
\_fp_sub_back_near_pack:NNnnnnnN ...
.......................... 19469, 19549
\_fp_sub_back_not_far_o:wwNNN ...
.......................... 19524, 19544
\_fp_sub_back_quite_far_ii:o:NN ...
.......................... 19522, 19528
\_fp_sub_back_quite_far_o:o:ww ...
.......................... 19522, 19528
\_fp_sub_back_shift:NNnn ...
.......................... 822, 19498, 19485
\_fp_sub_back_shift_ii:ww .. 19485
\_fp_sub_back_shift_iii:NNnnNNn ...
.......................... 19485
\_fp_sub_back_shift_iv:nnnnN ..
\_fp_sub_back_shift_o:ii_i:o:nnnnN ...
.......................... 19456
\_fp_sub_back_shift_o:very_far_i:o:nnNNN ...
.......................... 19523, 19556
\_fp_sub_eq:o:NNnnw ...... 19434
\_fp_sub_npos_i:o:NNnnw ........ 821, 19439, 19448, 19452
\_fp_sub_npos_ii:o:NNnnw .... 19434
\_fp_sub_npos:o:NwNwNw ........ .... 821, 19354, 19434
\_fp_sub_npos_i:o:NNnnw ........ 821, 19354, 19434
\_fp_sub_npos_ii:o:NNnnw .... 19434
\_fp_sub_npos:o:NwNwNw ........ .... 821, 19354, 19434
\_fp_tan:o:w ............. 21383
\_fp_tan_series_aux_o:NNnnw ..
21864
\_fp_tan_series:o:NNnnn ....... 888, 889, 21390, 21405, 21864
\_fp_ternary:NNnn .. 805, 18557, 19237
\_fp_ternary_auxi:NNnn ...... 805, 815, 19237
\_fp_ternary_auxii:NNnn ..
805, 815, 18559, 19237
\_fp_tanp:w ............. 740, 795,
16838, 16848, 16849, 16850, 16851,
16852, 16853, 16854, 16855, 16856,
16857, 16858, 16859, 16860, 16861,
16862, 16863, 16953, 16955, 17447,
17459, 17460, 17461, 17462, 17463,
17464, 17465, 17524, 17546, 18108,
18125, 18126, 18182, 18187, 18188,
18189, 18190, 18191, 18192, 18196,
18204, 18205, 18206, 18207, 18208,
18209, 18210, 18211, 18212, 18213,
18214, 18418, 18434, 18435, 18458,
18470, 18488, 18490, 18492, 18494,
18496, 18498, 18500, 18502, 18506,
18520, 18521, 18536, 18537, 18538,
18556, 18558, 20089, 20103, 20104
\_fp_to_decimal:w ............ 22268, 22278, 22395, 22412, 22899
\_fp_to_decimal_dispatch:w 913,
916, 917, 19116, 22258, 22262, 22265
\_fp_to_decimal_huge:NNnnn .. 22287
\_fp_to_decimal_large:NNnnn .. 22278
\_fp_to_decimal_normal:NNnnnn ..
.......................... 22278, 22366
\_fp_to_decimal_recover:w ... 22265
\_fp_to_dim:w ................ 22380
\_fp_to_dim_dispatch:w .. 917, 22380
\_fp_to_dim_recover:w ...... 22380
\_fp_to_int:w ................ 917, 918, 22405, 22410
\_fp_to_int_dispatch:w .... 22396
\_fp_to_int_recover:w .... 22396
\_fp_to_sciebtific:w .......... 914, 22214, 22224
\_fp_to_scienceic dispatched:w ...
.................. 912, 916, 22204, 22208, 22211
\_fp_to_scintific_normal:NNnnnn ..
.......................... 22244
\_fp_to_scintific_normal:o:NN ...
22224
\_fp_to_scintific_recover:o:NN ...
22211
\_fp_to_tl:o:w ... 22344, 22352, 22007
\_fp_to_tl_dispatch:w .... 911, 915, 22336, 22340, 22343, 22476
\_fp_to_tl_normal:o:NNnnn .... 22352
\_fp_to_tl_recover:o:NNnnn ...
22343
\_fp_to_tl_scintific:o:NNnnn ...
22352
\_fp_to_tl_scintific:o:NNnnn ...
22352
\_fp_trailing_shift_int ......
.......................... 16783, 20121,
20143, 20216, 21117, 21756, 21793
\_fp_trap_division_by_zero_set:
set:N .................. 17023
\_fp_trap_division_by_zero_set-
error: .................. 17023
\_fp_trap_division_by_zero_set-
flag: .................. 17023
\_fp_trap_division_by_zero_set-
none: .................. 17023
\_fp_trap_invalid_operation_set:
N .................. 16989
\_fp_trap_invalid_operation_set_error: 16989
\_fp\_use\_none\_stop\_f:\:n \:........
\_fp\_use\_none\_until\_s:\:w
\_fp\_use\_s:\:n
\_fp\_use\_s:\:nn
\_fp\_zero_fp:\:N \:16006, 17064, 17412
\_fp\_\_o:ww \:805, 19210
\_fp\_\_tuple_o:ww \:19210
\_fp\_ \:19213, 19220, 19229, 19230
fparray commands:
\fparray\_count:\:N \:........ 219,
\fparray\_gset:\:Nn ... \:219, 930, 22830
\fparray\_gzero:\:N \:........ 219, 22881
\fparray\_item:\:Nn ... \:219, 930, 22894
\fparray\_item\_to\_tl:\:Nn ... \:219, 22894
\fparray\_new:\:Nn \:........ 219, 22778
\futurelet \:........ 374
\G
\gdef \:........ 375
\GetId\_info \:........ 7, 14473
\gleaders \:928, 1803
\global \:167,
\global\_ \:168, 182, 183, 184, 195, 196, 197,
\global\_ \:198, 199, 200, 201, 202, 203, 272, 376
\global\_\_\_ \:377
\glue\_expr \:631, 1502
\glueshrink \:632, 1503
\glueshrink\_order \:633, 1504
\gluestretch \:634, 1505
\gluestretch\_order \:635, 1506
\glueto\_mu \:636, 1507
group commands:
\group\_align\_safe\_begin/end: \:541, 952
\group\_align\_safe\_begin: \:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\:
\halign \:378
\hangafter \:379
\hangindent \:380
\hbadness \:381
\hbox \:382
\hbox\_n \:238, 27182, 27407, 27703, 28760, 28815
\verb \:9, 495, 5108, 2129, 2858, 2861,
20064, 20114, 20119, 20121, 20128, 20131, 20134, 20137, 20140, 20143, 20152, 20164, 20172, 20174, 20184, 20186, 20193, 20202, 20204, 20207, 20210, 20213, 20216, 20229, 20231, 20239, 20241, 20249, 20251, 20261, 20264, 20267, 20274, 20289, 20307, 20310, 20366, 20380, 20382, 20388, 20401, 20403, 20405, 20429, 20445, 20452, 20453, 20497, 20499, 20500, 20501, 20542, 20544, 20581, 20588, 20595, 20616, 20618, 20620, 20622, 20635, 20639, 20640, 20641, 20642, 20643, 20648, 20653, 20655, 20661, 20678, 20679, 20680, 20681, 20682, 20683, 20688, 20690, 20692, 20694, 20696, 20701, 20703, 20705, 20707, 20709, 20711, 20733, 20741, 20757, 20762, 20766, 20825, 20874, 20942, 20951, 20959, 20970, 20972, 20975, 20978, 21067, 21103, 21105, 21108, 21111, 21114, 21117, 21124, 21127, 21129, 21133, 21155, 21157, 21189, 21259, 21269, 21274, 21284, 21426, 21458, 21467, 21699, 21700, 21711, 21714, 21717, 21720, 21723, 21726, 21729, 21732, 21735, 21753, 21763, 21772, 21790, 21799, 21806, 21816, 21860, 21869, 21904, 21947, 21964, 22020, 22031, 22042, 22252, 22288, 22375, 22420, 22428, 22432, 22524, 22547, 22601, 22641, 22653, 22664, 22665, 22695, 22698, 22701, 22703, 22705, 22712, 22715, 22723, 22728, 22733, 22836, 22898, 22906, 22919, 22920, 22921, 22931, 23088, 23402, 23414, 23593, 23635, 23637, 23647, 23655, 23674, 23676, 23684, 24153, 24167, 24169, 24176, 24178, 24196, 24197, 24182, 25246, 25261, 26061, 26062, 26073, 26285, 26266, 26268, 29309, 29441, 31846, 31856, 31934, 31946, 32147, 32155, 32585
\int_while_do:nn \hspace{1em} 94, 8998
\int_zero:N \hspace{1em} 90, 90, 6337, 6383, 8295, 8825, 8830, 8856, 8858, 8860, 8862, 8848, 8488, 16473
\c\_nine \hspace{1em} 23919, 23933, 25905, 27147, 27153
\c\_zero \hspace{1em} 32539
\c\_one \hspace{1em} 8856, 8858, 8860, 8862, 9488, 16473
\int\_zero:nn \hspace{1em} 22547, 22676, 22740
\c\_twelve \hspace{1em} 16333, 16460, 22547, 22676, 22740
\c\_one:nn \hspace{1em} 8713
\c\_case:nn \hspace{1em} 8952
\c\_case_end:nn \hspace{1em} 8952
\c\_compare:nnN \hspace{1em} 518, 8891
\c\_compare:N \hspace{1em} 141, 1191, 512, 518, 8891
\c\_compare:w \hspace{1em} 512, 8891
\c\_compare:=:nn \hspace{1em} 8916
\c\_compare>:nn \hspace{1em} 8916
\c\_compare<:nn \hspace{1em} 8916
\c\_compare<=:nn \hspace{1em} 8916
\c\_compare>=:nn \hspace{1em} 8916
\c\_compare_end=:nn \hspace{1em} 518, 8891
\c\_compare_error: \hspace{1em} 511, 512, 8876, 8894, 8896
\c\_case:nn \hspace{1em} 511, 512, 513, 8876, 8916
\c\_constinit:N \hspace{1em} 1191, 8792, 32581
\c\_preinitial_constants:N \hspace{1em} 32565, 32570
\c\_deprecate_constants:N \hspace{1em} 8732, 8733, 8747, 8749, 8750, 8767.
ior commands:
\ior_close:N 156, 157, 157, 157, 13093, 13117, 14044,
14057, 29285, 29318, 29355, 29372
\ior_get:NN 157, 158, 159, 159, 159, 159, 159, 159, 13158, 13159
\ior_get:NTF 158, 13158, 13159
\ior_get_str:NN 32395
\ior_get_term:NN 262, 13192
\ior_if_eof:N 635
\ior_if_eof:NTF 160, 13142, 13164,
13184, 13224, 13243, 14054, 14068
\ior_if_eof:N 160, 13142
\ior_list_streams: 32397
\ior_log_list: 157, 13129, 32400
\ior_log_streams: 32399

\ior_map_break: 159, 13207, 13225,
13232, 13244, 13250, 29280, 29351
\ior_map_break:n 160, 13207
\ior_map_inline:Nn 159, 159, 13211
\ior_map_variable:Nnn 159, 13247, 29277
\ior_new:N 156, 13060, 13062, 13063, 14073, 29213
\ior_open:N 156, 665, 13064, 29241, 29286, 29319, 29371
\ior_open:NNTF 157, 13065, 13068
\ior_shell_open:N 262, 32003
\ior_shell_open:NN 262
\ior_str_get:NN 157, 13129, 32398
\ior_str_get:NTF 158, 13171, 13172
\ior_str_get_term:NN 262, 13192
\ior_str_map_inline:Nn 159, 159, 13211, 29312, 29342
\ior_str_map_variable:Nnn 159, 13237
\ct_term_iор: 32792
\g_tmpb_iор 163, 13062
\g_tmpb_iор 163, 13062

ior internal commands:
\l__ior_file_name_tl 13067, 13070, 13072
\l__ior_get:NN 13158, 13193, 13212
\l__ior_get_term:NNN 13192
\l__ior_internal_tl 13092, 13230, 13234
\l__ior_list:N 13129
\l__ior_map_inline:Nn 13237
\l__ior_map_inline:NNN 13211
\l__ior_map_inline_loop:NNN 13211
\l__ior_map_variable:Nn 13237
\l__ior_map_variable_loop:Nnn 13237
\l__ior_new:N 631, 13078, 13101
\l__ior_new_aux:N 13083, 13087
\l__ior_open_stream:Nn 13091
\l__ior_shell_open:nN 32003
\l__ior_shell_open:NN 13171, 13195, 13214
\l__ior_stream_tl 13043, 13094, 13102, 13110
\g__ior_streams_prop 13044, 13111, 13122, 13136
\g__ior_streams_seq 13038, 13094, 13123, 13124
\c_term_iор 13037
\c__ior_term_iор 13060, 13119, 13125, 13146, 13202
\c__ior_term_noprompt_iор 13191, 13201
Index


\_kernel_primitives: .................................................. 301, 1189, 1475, 2095, 32486, 32501
\_kernel_quark_new:conditional:Nn .................................. 318, 4001, 4385, 10804, 13763, 15015, 15316, 23891, 29380
\_kernel_quark_new_text:N .................................................. 317, 380, 381, 383, 385, 4001, 4384, 5286, 5287, 6094, 8695, 9111, 10211, 10212, 11745, 13766, 13767, 15319, 20385, 31008
\_kernel_randint:n .......................................................... 318, 318, 318, 726, 926, 926, 16535, 22513, 22225, 22683, 22768
\_kernel_randint:n .................................................. 318, 726, 16531, 22687, 22691, 22766
\_kernel_randint_max_int .................................................. 926, 2099, 16528, 22512, 22681, 22765
\_kernel_register_log:N .................................................. 318, 2838, 9484, 14914, 14915, 15010, 15011, 15080, 15081
\_kernel_register_show:N .................................................. 318, 318, 417, 2838, 9480, 14910, 15006, 15076
\_kernel_register_show_aux:NN ............................................. 2838
\_kernel_register_show_aux:NN ............................................. 2838
\_kernel_show:N .......................................................... 2856
\_kernel_str_to_other:n .................................................. 319, 425, 427, 431, 5586, 5638, 5699
\_kernel_str_to_other_fast:n ............................................. 319, 5547, 5567, 5609, 6112, 13397, 14393, 24105, 25234
\_kernel_str_to_other_fast_:loop:w ...................................... 5609
\_kernel_sys_configuration_-load:n .................................... 9915, 9964, 9970, 32516, 32518
\_kernel_tl_to_str:w .................................................. 319, 401, 2126, 4059, 4958, 4670, 4789, 4982, 5418, 5522, 14259

keys commands:
\l_keys_choice_int .................................................. 185, 187, 189, 189, 191, 15290, 15483, 15486, 15491, 15492
\l_keys_choice_tl .......................................................... 185, 187, 189, 191, 15290, 15490
\keys_define:nn .......................................................... 184, 12793, 15320
\keys_if_choice_exist:nnn ............................................. 194, 16242
\keys_if_choice_exist_p:nn ............................................. 194, 16242
\keys_if_exist:nnn .......................................................... 193, 719, 16235, 16259
\keys_if_exist_p:nn .................................................. 193, 16235
\l_keys_key_str .................................................. 191, 15293, 15424, 15440, 15969, 15971, 16057, 16061, 16087, 16090, 16091, 16131, 16188
\l_keys_key_tl .................................................. 15293, 15971
\keys_log:nn .......................................................... 194, 16250
\l_keys_path_str .................................................. 191, 15298, 15348, 15368, 15377, 15380, 15387, 15391, 15406, 15417, 15419, 15421, 15433, 15435, 15437, 15452, 15455, 15459, 15467, 15469, 15470, 15473, 15488, 15502, 15512, 15517, 15527, 15531, 15538, 15543, 15547, 15552, 15559, 15566, 15567, 15582, 15593, 15599, 15603, 15619, 15628, 15636, 15677, 15692, 15970, 15994, 15997, 16036, 16040, 16045, 16054, 16068, 16070, 16071, 16076, 16084, 16112, 16141, 16164, 16176, 16185
\l_keys_path_tl .................................................. 15298, 15380, 15459
\keys_set:nn .......................................................... 183, 187, 191, 192, 15293, 15554, 15559, 15586
\keys_set_filter:nn .......................................................... 193, 15876
\keys_set_filter:nnn .................................................. 193, 15876
\keys_set_filter:nnnn .................................................. 193, 15876
\keys_set_groups:nn .................................................. 193, 15876
\keys_set_knownn:nn .................................................. 192, 15835
\keys_set_knoynn:nn .................................................. 192, 15835
\keys_set_knoynn:nn .................................................. 192, 15835
\keys_show:nn .................................................. 194, 16250
\l_keys_value_tl .................................................. 191, 15308, 15619, 16039, 16043, 16049, 16060, 16072, 16093, 16108, 16121, 16133, 16143, 16171

keys internal commands:
\keys_bool_set:NN .......................................................... 15413, 15651, 15653, 15655, 15657
\keys_bool_set_inverse:NN ............................................. 15429, 15659, 15661, 15663, 15665
\keys_check_groups .................................................. 15998, 16006
\keys_choice_find .................................................. 15446, 16182
\keys_choice_find:nn .................................................. 16182
\keys_choice_make .................................................. 15416, 15432, 15445, 15477, 15667
\_keyval_split_active_auxi:w \quad \ldots \quad 15118, 15124, 15157, 15202
\_keyval_split_active_auxii:w \quad \ldots \quad 689, 689, 15124
\_keyval_split_active_auxiii:w \quad \ldots \quad 689, 15124
\_keyval_split_active_auxiv:w \quad \ldots \quad 690, 15124
\_keyval_split_active_auxv:w \quad 15124
\_keyval_split_other:w \quad 15109, 15126, 15147, 15159, 15168
\_keyval_split_other_auxi:w \quad 690, 15160, 15164, 15179
\_keyval_split_other_auxii:w \quad 15164
\_keyval_split_other_auxiii:w \quad \ldots \quad 691, 15164
\_keyval_tmp:w \quad 15244, 15280
\_keyval@RequestParam:w \quad 692, 15098, 15216
\_keyval_trim:n \quad 15134, 15155, 15165, 15177, 15183, 15243
\_keyval_trim:NN \quad 15134, 15155, 15165, 15177, 15183, 15243
\_keyval_trim:aux:w \quad 15243
\_keyval_trim:auxii:w \quad 15243
\_keyval_trim:auxiii:w \quad 15243
\kuten \quad 1240, 1274, 2067, 2089
L
\L \quad 29530, 30961, 31302
\l \quad 29530, 30961, 31314
13kernel \quad 253, 29042
13kernel.charcat \quad 253, 29076
13kernel.elapsedtime \quad 253, 29081
13kernel.filedump \quad 253, 29094
13kernel.filedirectory \quad 253, 29111
13kernel.filedirectorysum \quad 253, 29123
13kernel.filesize \quad 253, 29168
13kernel.resettimer \quad 253, 29081
13kernel.shellescape \quad 253, 29188
13kernel.strcmp \quad 253, 29178
\label \quad 29539, 29547, 31250
\language \quad 424
\LARGE \quad 31233
\Large \quad 31234
\large \quad 31237
\lastallocatedtoks \quad 23026
\lastbox \quad 425
\lastkern \quad 426
\lastlinefit \quad 642, 1513
\lastnamedcs \quad 933, 1805
\lastmodechar \quad 1241
\lastmodesubtype \quad 1242
\lastmodetype \quad 643, 1514
\lastpenalty \quad 427
\lastsavedboxresourceindex \quad 1018, 1681
\lastsavedimageresourceindex \quad 1020, 1683
\lastsavedimageresourcelpages \quad 1022, 1685
\lastskip \quad 428
\lastxpos \quad 1024, 1687
\lastypenalty \quad 1025, 1688
\latelua \quad 934, 1506
\lateluafunction \quad 935
LaTeX3 error commands:
\ LaTeX3_error \quad 628
\lccode \quad 167, 182, 195, 197, 199, 201, 203, 429
\leaders \quad 430
\left \quad 431
left commands:
\#L \quad 71, 981, 13814, 24181, 24566, 24570, 24590, 24603, 24627, 25110, 25191, 26223, 26258, 26282, 29635
\leftghost \quad 936, 1565
\leftmargin \quad 432
\leftmargin \quad 791, 1660
\leftskip \quad 433
legacy commands:
\ legacy_if_nTF \quad 259, 31612
\ legacy_if_p:n \quad 259, 31612
\ let \quad 434
\ let \quad 2, 40, 272, 273, 435
\ letcharcode \quad 937, 1807
\ letterspacefont \quad 792, 1661
\ limits \quad 436
\ LineBreak \quad 74, 75, 76, 77, 78, 79, 80, 81, 106, 113, 114, 115, 123, 125
\ linedir \quad 938, 1866
\ linedirection \quad 939
\ linenumber \quad 437
\ lineskip \quad 438
\ lineskipmargin \quad 439
\ linewidth \quad 27846, 27915
\ ln \quad 21092, 21095
\ ln \quad 211
\ localbrokenpenalty \quad 940, 1867
\ localinterlinenumber \quad 941, 1868
\ localleftbox \quad 946, 1870
\ localrightbox \quad 947, 1871
\ loccount \quad 13053, 13286
\ loctoks \quad 22998, 22999, 23025
\ logb \quad 212
\ long \quad 275, 440, 11475, 11479
\ longtext \quad 70, 111, 135
\ looseness \quad 441
\ lower \quad 442
\ overcase \quad 443
\ upcase \quad 793, 1662
lua commands:
\luaescape:n 252, 5402, 5417, 10022, 10035, 13914, 14095, 14160, 14179, 14258, 14307, 29001, 29003, 32589
\luaescape:x:n 32587
\lua_nov:n 252, 5403, 5406, 10021, 10977, 13913, 14094, 14156, 14178, 14247, 14306, 29002, 29003, 32587
\lua_nov:x:n 32587
\lua_shipout:n 252, 29003
\lua_shipout_e:n 252, 10034, 29003, 32591
\lua_shipout:x:n 32587
lua internal commands:
\luabytecode 942
\luabytecodecall 943
\luacopyinputnodes 944
\luadef 945
\luaescapestring 948, 1808
\luafunction 949, 1809
\luafunctioncall 950
lua commands:
\luatex_alignmark:D 1764
\luatex_aligntab:D 1765
\luatex_attribute:D 1766
\luatex_automaticdiscretionary:D 1767
\luatex_automatichyphenmode:D 1771
\luatex_automatichyphenpenalty:D 1773
\luatex_begincsname:D 1774
\luatex_bodydir:D 1863
\luatex_boxdir:D 1864
\luatex_breakafterdirmode:D 1775
\luatex_catcodelatable:D 1776
\luatex_clearmarks:D 1777
\luatex_crampeddisplaystyle:D 1779
\luatex_crampedscriptstyle:D 1781
\luatex_crampedscriptstyle:D 1782
\luatex_cube:D 1783
\luatex_directlua:D 1784
\luatex_dvixextension:D 1785
\luatex_dvifeedback:D 1786
\luatex_dvivariable:D 1787
\luatex_etoksapp:D 1788
\luatex_etokspre:D 1789
\luatex_expanded:D 1792
\luatex_explicitdiscretionary:D 1794
\luatex_explitchyphenpenalty:D 1791
\luatex_firstvalidlanguage:D 1795
\luatex_fontid:D 1796
\luatex_formatname:D 1797
\luatex_gleaders:D 1803
\luatex_hjcode:D 1798
\luatex_hyphenationbounds:D 1800
\luatex_hyphenationmin:D 1801
\luatex_hyphenpenaltymode:D 1802
\luatex_if_engine:TF 32407, 32409, 32411
\luatex_if_engine:p: 32405
\luatex_initcodelist:D 1804
\luatex_localbreakpenalty:D 1806
\luatex_leftghost:D 1805
\luatex_leftmargin:D 1806
\luatex_linefill:D 1807
\luatex_linend:D 1808
\luatex_linedir:D 1809
\luatex_localleftbox:D 1810
\luatex_localrightbox:D 1811
\luatex_localinterlinepenalty:D 1812
\luatex_localbreakpenalty:D 1813
\luatex_mathdir:D 1814
\luatex_mathdisplayskipmode:D 1815
\luatex_mathdisplayskipmode:D 1816
\luatex_mathdisplayskipmode:D 1817
\luatex_mathdisplayskipmode:D 1818
\luatex_mathdisplayskipmode:D 1819
\luatex_mathdisplayskipmode:D 1820
\luatex_mathdisplayskipmode:D 1821
\luatex_mathdisplayskipmode:D 1822
\luatex_mathdisplayskipmode:D 1823
\luatex_mathdisplayskipmode:D 1824
\luatex_mathdisplayskipmode:D 1825
\luatex_mathdisplayskipmode:D 1826
\luatex_mathdisplayskipmode:D 1827
\luatex_mathdisplayskipmode:D 1828
\luatex_mathdisplayskipmode:D 1829
\luatex_mathdisplayskipmode:D 1830
\luatex_mathdisplayskipmode:D 1831
\luatex_mathdisplayskipmode:D 1832
\luatex_mathdisplayskipmode:D 1833
\luatex_mathdisplayskipmode:D 1834
\luatex_mathdisplayskipmode:D 1835
\luatex_mathdisplayskipmode:D 1836
\luatex_mathdisplayskipmode:D 1837
\texttt{\textbackslash\texttt{mathopen}} \hfill 455
\texttt{\textbackslash\texttt{mathoption}} \hfill 960, 1818
\texttt{\textbackslash\texttt{mathord}} \hfill 456
\texttt{\textbackslash\texttt{mathpenaltiesmode}} \hfill 961, 1819
\texttt{\textbackslash\texttt{mathpunct}} \hfill 457
\texttt{\textbackslash\texttt{mathrel}} \hfill 458
\texttt{\textbackslash\texttt{mathrulesfam}} \hfill 962, 1820
\texttt{\textbackslash\texttt{mathscriptboxmode}} \hfill 964, 1822
\texttt{\textbackslash\texttt{mathscriptmode}} \hfill 963, 1821
\texttt{\textbackslash\texttt{mathstyle}} \hfill 966, 1823
\texttt{\textbackslash\texttt{mathsurround}} \hfill 459
\texttt{\textbackslash\texttt{mathsurroundmode}} \hfill 967, 1824
\texttt{\textbackslash\texttt{mathsurroundskip}} \hfill 968, 1825
\texttt{\textbackslash\texttt{max}} \hfill 212
\texttt{\textbackslash\texttt{maxdeadcycles}} \hfill 460
\texttt{\textbackslash\texttt{maxdepth}} \hfill 461
\texttt{\textbackslash\texttt{mdfive}} \hfill 880, 1608
\texttt{\textbackslash\texttt{mdseries}} \hfill 31226
\texttt{\textbackslash\texttt{meaning}} \hfill 462
\texttt{\textbackslash\texttt{medmuskip}} \hfill 463
\texttt{\textbackslash\texttt{message}} \hfill 464
\texttt{\textbackslash\texttt{MessageBreak}} \hfill 123
\texttt{\textbackslash\texttt{meta}} \hfill 187, 15746
\texttt{\textbackslash\texttt{meta:n}} \hfill 187, 15748
\texttt{\textbackslash\texttt{middle}} \hfill 645, 1516
\texttt{\textbackslash\texttt{min}} \hfill 212
\texttt{\textbackslash\texttt{minus}} \hfill 216
\texttt{\textbackslash\texttt{c\_minus\_inf\_fp}} \hfill 206, 215, 16588, 19617, 19701, 20034, 20571, 21418, 22943
\texttt{\textbackslash\texttt{c\_minus\_zero\_fp}} \hfill 205, 16588, 19613, 22137, 22941
\texttt{\textbackslash\texttt{mkern}} \hfill 465
\texttt{\texttt{mm}} \hfill 216
\texttt{\texttt{mode}} \hfill 216
\texttt{\texttt{mode\_if\_horizontal:TF}} \hfill 111, 9853
\texttt{\texttt{mode\_if\_horizontal:p}} \hfill 111, 9853
\texttt{\texttt{mode\_if\_inner:TF}} \hfill 111, 9855
\texttt{\texttt{mode\_if\_inner:p}} \hfill 111, 9855
\texttt{\texttt{mode\_if\_math:TF}} \hfill 111, 9857
\texttt{\texttt{mode\_if\_math:p}} \hfill 112, 9857
\texttt{\texttt{mode\_if\_vertical:TF}} \hfill 112, 9851
\texttt{\texttt{mode\_if\_vertical:p}} \hfill 112, 9851
\texttt{\texttt{mode\_leave\_vertical:}} \hfill 24, 2917, 28095
\texttt{\texttt{\textbackslash\texttt{month}}}} \hfill 466, 1417, 10081
\texttt{\texttt{\textbackslash\texttt{moveleft}}}} \hfill 467
\texttt{\texttt{\textbackslash\texttt{moveright}}}} \hfill 468
\texttt{\texttt{msg}} \hfill \texttt{\textbackslash\texttt{msg\_critical:nn}} \hfill 153, 167, 12371
\texttt{\texttt{msg\_critical:nnn}} \hfill 153, 12371
\texttt{\texttt{msg\_critical:nnnn}} \hfill 153, 12371
\texttt{\texttt{msg\_critical:nnnnn}} \hfill 153, 12371
\texttt{\texttt{msg\_critical:nnnnnn}} \hfill 153, 12371
\texttt{\texttt{msg\_critical:nnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnnnnnn}} \hfill 153, 12379
\texttt{\texttt{msg\_critical:nnnnnnnnnnnnnn}} \hfill 153, 12379
\msg_module_name:n .............. 152, 12177, 12285, 12308, 12389, 12411
\g_msg_module_name_prop .......... 152, 152, 12293, 12310, 12311
\msg_module_type:n .............. 151, 152, 152, 12284, 12297
\g_msg_module_type_prop .......... 152, 152, 12293, 12299, 12300
\msg_new:nnn ................... 150, 12110, 12609
\msg_new:nnnn .................. 150, 607, 12110, 12607
\msg_none:nn ................... 154, 12436
\msg_none:nnn .................. 154, 12436
\msg_none:nnnn ................ 154, 12436
\msg_redirect_class:nn ........... 155, 12558
\msg_redirect_module:nn ......... 155, 12558
\msg_redirect_name:nnn .......... 155, 12549
\msg_see_documentation_text:n .. 152, 12308
\msg_set:nnn ................... 150, 12110, 12613
\msg_set:nnnn .................. 150, 12110, 12611
\msg_show:nn ................... 263, 12437
\msg_show:nnn .................. 263, 12437
\msg_show:nnnn ................ 263, 12437
\msg_show:nnnnnn ............... 263, 264, 504, 568, 606, 8663, 10774, 10788, 12072, 12437, 13129, 13345, 14440, 16251, 16475, 23724, 23732, 26424, 26433, 28921
\msg_show_eval:Nn ............... 263, 9483, 9622, 14913, 15009, 15079, 18843, 31772
\msg_show_item:n .................. 263, 264, 8673, 10784, 10793, 31777
\msg_show_item:nn ............... 264, 606, 12082, 31777
\msg_show_item_unbraced:a:nn .. 264, 31777
\msg_show_item_unbraced:nn .... 264, 633, 13137, 13353, 16261, 28940, 31777
\msg_term:n ..................... 32594
\msg_warning:nn ................ 153, 12386
\msg_warning:nnn ............... 153, 12386
\msg_warning:nnnn ............. 153, 12386
\msg_warning:nnnnnn ........... 153, 12386
\msg_warning_text:n ............ 151, 12296, 12388
msg internal commands:
\__msg_chk_free:nn ............ 12102, 12112
\__msg_chk_if_free:nn ........ 12102
\__msg_class_chk_exist:nF ... 12472, 12487, 12554, 12564, 12569
\l__msg_class_loop_seq ......... 620, 12481, 12573, 12581, 12591, 12592, 12595, 12597
\__msg_class_new:nnn ........ 616, 621, 12319, 12358, 12371, 12379, 12386, 12408, 12430, 12436, 12437
\l__msg_class_tl ............. 617, 620, 12477, 12494, 12507, 12528, 12532, 12535, 12543, 12582, 12584, 12586, 12600
\c__msg_coding_error_text_tl .. 12135, 12663, 12671, 12697, 12715, 12724, 12731, 12745, 12754, 12776, 12785, 12792, 12801, 12807, 12814, 12824, 12836, 12851, 12858, 12866, 12874
\c__msg_continue_text_tl .... 12135, 12184, 32614
\c__msg_critical_text_tl .... 12135, 12376
\l__msg_current_class_tl ...... 619, 12477, 12489, 12527, 12532, 12535, 12543, 12572, 12586
\__msg_error_code:nnnnnnn .... 12656
\__msg_expandable_error:n .. 629, 12070, 12999
\__msg_expandable_error:w 628, 12079
\__msg_expandable_error_module:nn 31741
\__msg_fatal_code:nnnnnnn ...... 12652
\__msg_fatal_exit: ............. 12358
\c__msg_fatal_text_tl .. 12135, 12363
\c__msg_help_text_tl ........... 12135, 12194, 32618
\l__msg_hierarchy_seq ........ 618, 619, 12480, 12510, 12520, 12525
\l__msg_internal_tl ............ 12089, 12220, 12226, 12369, 12461, 12467
\__msg_interrupt:n .......... 12221, 12230
\__msg_interrupt:Nnn .......... 12174
\__msg_interrupt:NnnN ........ 12174, 12360, 12373, 12381
\__msg_interrupt_more_text:n ... 610, 12203
\__msg_interrupt_text:n ...... 12203
\__msg_interrupt_wrap:nnn ...... 12182, 12192, 12203
\__msg_kernel_class_new:nN ... 622, 12614, 12652, 12656, 12657, 12658
\__msg_kernel_class_new_aux:nN 12614
\c__msg_more_text_prefix_tl .. 12095, 12121, 12130, 12179, 12196
\l__msg_name_str .............. 12090, 12177, 12210, 12214, 12389, 12397, 12401, 12411, 12419, 12423
\c__msg_no_info_text_tl
\c__msg_no_more_text:n
\c__msg_old_interrupt_more_text:n
\c__msg_old_interrupt_text:n
\c__msg_old_interrupt_wrap:n
\c__msg_on_line_text_tl
\c__msg_redirect:n
\c__msg_redirect_loop_chk:n
\c__msg_redirect_loop_list:n
\c__msg_redirect_prop
\c__msg_return_text_tl
\c__msg_text_prefix_tl
\c__msg_text:n
\c__msg_text:nn
\c__msg_text_str
\c__msg_use:nnnnn
\c__msg_use_code:nnnnn
\c__msg_use_hierarchy:nwqW
\c__msg_use_none_delimit_by_s-stop:w
\c__msg_use_redirect_module:n
\c__msg_use_redirect_name:n
\mskip
\muexpr
\multichoice commands:
\multichoice:nn
\multichoicecommands:
\multiply
\muskip
\muskip commands:
\c_max_muskip
\muskip_add:Nn
\muskip_const:NN
\muskip_eval:n
\muskip_gadd:Nn
\muskip_gset:Nn
\muskip_gset_eq:NN
\muskip_gsub:Nn
\muskip_gzero:N
\muskip_gzero_new:N
\muskip_if_exist:NTF
\muskip_if_exist_p:N
\muskip_log:N
\muskip_log:n
\muskip_new:N
\ag_tmpa_muskip
\ag_tmpb_muskip
\ag_tmpq_muskip
\ag_tpm_muskip
\c_zero_muskip
\muskipdef
\mutoglue
\nd
\newbox
\newcount
\newdimen
\newlinechar
\linebreak
\next
\Ng
\nbin
\noalign
\noautoscaling
\noautoxspacing
\noboundary
\nobreakspace
\noexpand
\l_prop_internal_tl ............ 602, 11736, 11739, 11943, 11949, 11950, 11966, 11973
\l_prop_item_Nn:nnw ............ 600
\l_prop_item_Nn:nnwn ........... 11894
\l_prop_map_function:nnw ....... 12023
\l_prop_map_tokens:nn ......... 12054
\l_prop_pair:wn ............... 594, 594, 594, 599, 603, 604, 605, 605, 11736, 11737, 11842, 11845, 11897, 11900, 11945, 11968, 11991, 11995, 12029, 12032, 12042, 12044, 12049, 12058, 12061, 31851
\l_prop_put:NNnn ................ 11939
\l_prop_put_if_new:NNnn ........ 11960
\l_prop_rand_item:w ............ 31841
\l_prop_show:NN ............... 12072, 12074, 12076
\l_prop_split:NhTF ............. 595, 602, 602, 603, 11837, 11850, 11856, 11866, 11874, 11883, 11919, 11929, 11948, 11971, 12014
\l_prop_split_aux:NhTF ........ 11837
\l_prop_split_aux:w ........... 599, 11837
\l_prop_use_i_delimit_by_s_-
stop:nn .................. 31840, 31854
\protect ........................ 1128, 13470, 17531, 20900, 20921, 20923, 31172
\protected ........................ 207, 209, 211, 236, 654, 1525, 11477, 11479
\protrudechars ................. 1030, 1093
\ProvidesExplClass ................ 7
\ProvidesExplFile ............. 7, 32504, 32521
\ProvidesExplFileAux ........ 32507, 32509
\ProvidesExplPackage ......... 7
\ProvidesFile .............. 32512, 32513
\pt .......................... 216
\ptex commands:
\ptex_autospacing:D ........... 2044
\ptex_autospacing:D ........... 2045
\ptex_dtou:D .................. 2046
\ptex_epTeXversion:D ........ 2048
\ptex_euc:D .................. 2049
\ptex_ifdbx:D ................ 2050
\ptex_ifddir:D ................ 2051
\ptex_ifmdir:D ................ 2052
\ptex_iftbox:D ................ 2053
\ptex_iftbox:D ................ 2054
\ptex_iftdir:D ................ 2055
\ptex_ifydir:D ................ 2056
\ptex_inhibitglue:D ........... 2057
\ptex_inhibitxspcode:D .......... 2058
\ptex_imputencoding:D .......... 2047
\ptex_jcharwidowpenalty:D ...... 2059
\ptex_jfam:D .................. 2060
\ptex_jfont:D .................. 2061
\ptex_jis:D ................... 2062
\ptex_kanjiskip:D ............. 2063
\ptex_kansuji:D ............... 2064
\ptex_kansuji char:D ........... 2065
\ptex_katcode:D .............. 2066
\ptex_kuten:D .................. 2067
\ptex_noautospacing:D .......... 2068
\ptex_noautospacing:D .......... 2069
\ptex_postbreakpenalty:D ...... 2070
\ptex_prebreakpenalty:D ...... 2071
\ptex_ptexminorversion:D ...... 2072
\ptex_ptexversion:D ........... 2073
\ptex_revision:D .............. 2074
\ptex_showmode:D .............. 2075
\ptex_sjis:D .................. 2076
\ptex_tate:D .................. 2077
\ptex_tbselineshift:D .......... 2078
\ptex_tfont:D .................. 2079
\ptex_zkanjiskip:D ............ 2080
\ptex_xspcode:D .............. 2081
\ptex_ybaselineshift:D ........ 2082
\ptex_yoko:D .................. 2083
\ptexminorversion ............ 1248, 2072
\ptexrevision ................ 1249, 2073
\ptexversion ................. 1250, 2074
\pxdimen ...................... 1631, 1694
\quark commands:
\q_mark ........................ 39, 2949, 2964
\q_nil ........................ 21, 39, 39, 39, 39, 39, 58, 325, 377, 380, 381, 2219, 2222, 2264, 4018, 4037, 4043, 4058, 4059, 4065, 4089, 4093
\q_no_value 38, 39, 39, 39, 77, 77, 77, 77, 77, 77, 83, 83, 83, 116, 125, 144, 144, 145, 158, 158, 164, 164, 165, 165, 166, 166, 166, 377, 379, 496, 497, 556, 600, 600, 3964, 4026, 4047, 4053, 8352, 8360, 8372, 8398, 9978, 10361, 10376, 11868, 11879, 11888, 13159, 13172, 13871, 14008, 14040, 14183, 14185, 14187, 14199, 14231
\quark_if_nil:n ............... 380
\quark_if_nil:NTF .............. 39, 4016
\quark_if_nil:NTF .............. 39, 378, 381, 4034
\quark_if_nil:p:N ............. 39, 4016
\quark_if_nil:p:N ............. 39, 4034
\quark_if_no_value:NTF ........ 39, 4016
\quark_if_no_value:NTF ........ 39, 4016
\quark_if_no_value:p:n ...... 14042, 28782, 28786, 28865, 28869
\quark_if_no_value:p:n ...... 39, 4016
\quark_if_no_value:p:n ...... 39, 4016
\quark_if_no_value:p:n ...... 39, 4034
\q__char_no_value
\q__bool_recursion_stop
\q__clist_recursion_tail_stop
\q__clist_recursion_tail
\q__cs_recursion_stop

\s__deprecation_mark
\s__file_stop
\s__dim_mark
\s__deprecation_stop
\s__color_stop
\s__clist_stop

\s__clist_mark
\s__fp

\a__clist_mark
\a__clist_stop

\a__csnil
\a__cs_stop

\a__deprecation_mark

\a__deprecation_stop

\a__fp

\a__file_stop

\a__file_mark
\a__file_stop

\a__fp

\a__clist_mark

\q__clist_recursion_tail_break:N

\q__clist_recursion_tail_break:n

\q__clist_recursion_tail_stop:N

\q__clist_recursion_tail_stop:n

\q__clist_recursion_tail_stop:do:n

\q__clist_recursion_tail_stop:do:nn

\q__clist_recursion_tail_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\q__clist_recursion_stop

\a__clist_mark

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop

\a__clist_stop
Index

16575, 16578, 16579, 16580, 16581, 16583, 16584, 16585, 16586, 16587, 16686, 22964, 22965, 23385, 28978, 29378, 31802, 31803, 32217, 32218

\secd ................................. 213
\selectfont ............................ 31200

seq commands:
\c_empty_seq ........... 85, 488, 8074
\c_empty_seq ........... 8078, 8082, 8085, 8273, 8351, 8359
\l_foo_seq ............... 227
\seq_clear:N ............. 75, 8007
\seq_clear:N ............. 75, 8087
\seq_concat:N .......... 76, 8170, 14452
\seq_concat:N .......... 76, 8170, 14452
\seq_count:N ............ 77, 82, 84, 196, 8288, 8478, 8492, 8570, 8598, 26144
\seq_elw .................................. 487
\seq_elw .................................. 487
\seq_gclear:N ......... 75, 928, 8081, 8091, 8305, 23121
\seq_gclear_new:N ........ 75, 8087
\seq_gconcat:N ........ 76, 8170, 14452
\seq_get:NN ............. 83, 8651, 25620, 25625
\seq_get:NTF ........... 83, 8657
\seq_get_right:NN ........ 77, 8307, 8651, 8652, 8657, 8658
\seq_get_right:NTF ... 78, 8437
\seq_get_right:NTF ... 78, 8337
\seq_gpop:NN ........... 83, 8651, 14488
\seq_gpop:NTF ........ 84, 8657, 13094, 13311
\seq_gpop_left:NN ...... 77, 8328, 8655, 8656, 8661, 8662
\seq_gpop_left:NTF ...... 78, 8445
\seq_gpop_right:NN ...... 77, 8410
\seq_gpop_right:NTF .... 79, 8445
\seq_gpush:NN ........... 25, 84, 8631, 13124, 13340, 14371
\seq_gput_left:NN ........ 76, 8180, 8641, 8642, 8643, 8644, 8645, 8646, 8647, 8648, 8649, 8650
\seq_gput_right:NN ...... 76, 8201, 13726, 13733, 13749, 14354, 14359
\seq_gremove_all:NN ... 79, 8227, 13274
\seq_gremove_duplicates:NN .... 79, 8211
\seq_greverse:N .......... 79, 8253
\seq_gset_eq:NN ........ 75, 8085, 8093, 8214, 8285, 23005
\seq_gset_eq:NN ........ 265, 31881
\seq_gset_filter:NN .... 75, 8101
\seq_gset_from_clist:NN ...... 75, 8101
\seq_gset_from_clist:NN ...... 75, 8101
\seq_gset_from_function:NN .... 266, 31911
\seq_gset_from_inline:x:NNn .... 266, 8300, 23113, 31901, 31914
\seq_gset_map:NN ........ 265, 31891

scan internal commands:
\g__scan_marks_tl .................................. 396, 4234, 4237, 4243, 4248, 4250
\scantextokens .................. 995, 1848
\scantokens .................... 658, 1529
\scriptbaselineskipfactor .... 1252
\scriptfont ........................... 526
\scriptscriptbaselineskipfactor ... 1254
\scriptscriptfont .................. 527
\scriptscriptscriptfont ........ 528
\scriptsizem ..................... 31239
\scriptspace ..................... 529
\scriptstyle .......................... 530
\scrollmode ................. 531
\scshape ......................... 31228
\sec .................................. 213
Index

\seq_gset_split:Nnn .......................... 76, 8133, 13040, 13268
\seq_gshuffle:N ......................... 80, 8281
\seq_gsort:N .......................... 79, 8271, 23091
\seq_if_empty:NTF .......................... 80, 8271, 8491, 10286, 26141
\seq_if_empty_p:N ..................... 80, 8271
\seq_if_exist:NTF .......................... 76, 8088, 8091, 8176, 8596
\seq_if_exist_p:N ..................... 76, 8176
\seq_if_in:NTF .......................... 80, 84, 85, 8220, 8228, 13123, 13339
\seq_if_in:N .......................... 561
\seq_indexed_map_function:NN .......... 266, 31915
\seq_indexed_map_inline:Nn .............. 266, 31915
\seq_item:Nn .......................... 86, 8663
\seq_map_break: .......................... 81, 265, 267, 8495, 8506, 8541, 8549, 8566, 16016, 31918, 31927
\seq_map_break:n .......................... 82, 500, 8495, 12530, 12544, 13048, 14032, 23092, 23095
\seq_map_function:NN .................. 4, 86, 81, 264, 1174, 8499, 8673, 10292, 12595, 14455, 25343, 25422
\seq_map_inline:Nn ...................... 80, 80, 81, 85, 1173, 8218, 8337, 12525, 14031, 16009, 23092, 23095
\seq_map_tokens:Nn ..................... 80, 81, 8544, 13928
\seq_map_variable:NN .................... 81, 8558
\seq_mapthread_function:NNNN ....... 265, 31859
\seq_new:N .......................... 4, 75, 75, 8075, 8088, 8091, 8210, 8283, 8677, 8678, 8679, 8680, 10437, 10899, 10902, 12480, 12481, 13038, 13265, 13717, 13744, 13757, 13759, 15306, 22951, 23868, 24309, 25440, 25441, 26047
\seq_pop:NN .......................... 83, 9651, 25486, 25488, 26181
\seq_pop:NNTF .......................... 84, 8657
\seq_pop_left:NN ......................... 77, 8278, 8653, 8654, 8659, 8660
\seq_pop_left:NNTF ..................... 78, 8445
\seq_pop_right:NN ......................... 77, 8410, 25280, 25350
\seq_pop_right:NNTF ..................... 79, 8445
\seq_push:NN ........................ 85, 8651, 8638, 25479, 25481, 26309
\seq_put_left:Nn .......................... 76, 8180, 8631, 8632, 8633, 8634, 8635, 8636, 8637, 8638, 8639, 8640, 12520
\seq_put_right:Nn ...................... 76, 84, 85, 8201, 8221, 12581, 25283, 25348, 32376
\seq_rand_item:N ......................... 78, 8489
\seq_remove_all:NN .......................... 76, 79, 84, 85, 8271, 10463, 32378
\seq_remove_duplicates:NN .......................... 79, 84, 85, 8211, 14453
\seq_reverse:N .......................... 79, 493, 8253
\seq_set_eq:NN .......................... 75, 85, 8082, 8093, 8212, 8284, 23092
\seq_set_filter:NNNN .................. 265, 1173, 25417, 31881
\seq_set_from_clist:NN 75, 8101, 10462
\seq_set_from_clist:NN .............. 75, 118, 489, 8101, 14448, 14464, 15943
\seq_set_from_function:NNN .......... 266, 26569, 31911
\seq_set_from_inline_x:NNn ............. 266, 1174, 31901, 31912
\seq_set_map:NNN ...................... 265, 25430, 25657, 31891
\seq_set_split:NNN .............. 76, 8133, 10900, 10903, 25416, 25429
\seq_show:N .......................... 86, 616, 8663
\seq_shuffle:N ......................... 80, 8281
\seq_sort:N .......................... 79, 220, 8271, 23091
\seq_use:N ......................... 83, 8594, 25433
\seq_use:NNnn ......................... 82, 8594
\g_tmpa_seq .......................... 86, 8677
\l_tmpa_seq .......................... 86, 8677
\l_tmpb_seq .......................... 86, 8677
\seq internal commands:
\_seq_count:w .......................... 502, 8570
\_seq_count_end:w .......................... 502, 8570
\_seq_get_left:uvw .......................... 8367
\_seq_get_right_end:NNn .......................... 8392
\_seq_get_right_loop:nu .......................... 497, 8392
\_seq_if_in: .......................... 8328
\_seq_indexed_map:NNN .................. 31917, 31925, 31930
\_seq_indexed_map:nNN .................. 31915
\_seq_indexed_map:N .......................... 1174, 31915
\l\_seq_internal_a_int .......... 8295, 8301, 8310, 8312, 8313
\l\_seq_internal_a_tl .............. 490, 8071, 8141, 8145, 8151, 8156, 8158, 8242, 8247, 8332, 8336
\l\_seq_internal_b_int ............. 8311, 8314, 8315
\skip_gsub:Nn ............... 177, 14964
\skip_gzero:N .......... 177, 14940, 14947
\skip_gzero_new:N ....... 177, 14944
\skip_horizontal:N ....... 179, 14998
\skip_horizontal:n ....... 179, 14998
\skip_if_eq:nTF ......... 178, 14976
\skip_if_eq:pn ........ 178, 14976
\skip_if_exist:N ....... 177, 14945, 14947, 14950
\skip_if_exist_p:N ....... 177, 14950
\skip_if_Finite:nTF ..... 178, 14982
\skip_if_Finite:p:n .... 178, 14982
\skip_log:N ........ 179, 15010
\skip_log:n ........ 179, 15010
\skip_new:N ........... 176, 177, 14926, 14936, 14945, 14947, 15016, 15017, 15018, 15019
\skip_set:N ........ 188, 15776
\skip_set:Nn ........ 177, 14954
\skip_set_eq:NN ....... 177, 14960
\skip_show:N ....... 178, 15006
\skip_show:n ........ 178, 684, 15008
\skip_sub:Nn ....... 177, 14964
\skip_use:N ........ 178, 179, 14987, 14994, 14995
\skip_vertical:N ....... 180, 14998
\skip_vertical:n ....... 180, 14998
\skip_zero:N ....... 177, 177, 180, 14940, 14945
\skip_zero_new:N ....... 177, 14944
\g_tampa_skip ........ 179, 15016
\l_tampa_skip ........ 179, 15016
\g_tamp_skip ........ 179, 15016
\l_tamp_skip ........ 179, 15016
\c_zero_skip ........ 179,
672, 14594, 14596, 14940, 14941, 15014

skip internal commands:
\_\_skip_if_Finite:wwNN ........ 14982
\_\_skip_tmp:w ........ 14982, 14992
\skipdef ........ 544
\slshape ........ 31229
\small ........ 31240

sort commands:
\sort_ordered: ........ 32431
\sort_return_same: ........
220, 220, 940, 23174, 32432
\sort_return_swapped: ........
220, 220, 940, 23174, 32434
\sort_reversed: ........ 32433

sort internal commands:
\_\_sort:nnNnn ........ 942, 943
\_\_sort_A_int ........
940, 22961, 22968, 22975, 22978,
22987, 23138, 23143, 23146, 23166,
23198, 23205, 23220, 23222, 23223
\_\_sort_B_int ........
940, 940, 22961, 23143, 23147,
23155, 23157, 23158, 23210, 23211,
23220, 23221, 23230, 23231, 23233
\_\_sort_begin_int ........
934, 940, 22965, 23135, 23223, 23233
\_\_sort_block_int ........
934, 935, 939, 22958, 22970,
22975, 22979, 22982, 22987, 22988,
23065, 23126, 23129, 23136, 23139
\_\_sort_C_int ........
934, 940, 22961, 23144, 23148,
23155, 23156, 23157, 23199, 23206,
23210, 23212, 23213, 23230, 23232
\_\_sort_compare:nn ........
937, 941, 23064, 23165
\_\_sort_compute_range: ........ 934, 935, 935, 22902, 23052
\_\_sort_copy_block: 939, 23145, 23153
\_\_sort_disable_toksdef: 23051, 23330
\_\_sort_disabled_toksdef:n ........ 23330
\_\_sort_end_int: 934, 939, 940, 940,
22959, 23127, 23135, 23136, 23137,
23138, 23139, 23140, 23141, 23158
\_\_sort_error: ........ 23324, 23337, 23356
\_\_sort_i:nnnn ........ 944
g\_\_sort_internal_seq ........
937, 938, 22951, 23113, 23120, 23121
g\_\_sort_internal_tl ........
22951, 23076, 23079, 23080
\_\_sort_length_int ........
934, 934, 22953, 23062, 23126
\_\_sort_level: ........
937, 947, 23066, 23124, 23238
\_\_sort_loop:nnn ........ 943
\_\_sort_main:NNNnn ........
937, 938, 23049, 23075, 23112
\_\_sort_max_int ........
934, 935, 22953, 22972, 23046, 23056
\_\_sort_max_length_int ........
22992
\_\_sort_merge_blocks: ........
23128, 23133, 23237
\_\_sort_merge_blocks_aux: ........
939, 23149, 23163, 23216, 23226, 23326
\_\_sort_merge_blocks_end: ........
941, 23224, 23228
\_\_sort_min_int: 934, 935, 937, 937,
22953, 22969, 22977, 22995, 23011,
23019, 23032, 23044, 23053, 23063,
23077, 23116, 23127, 2354, 23355
\_\_sort_quick_cleanup:nn ........ 23238
\_\_sort_quick_end:nnn ........ 943, 946, 23258, 23298
\_\_sort_quick_only_i:Nnnnn ........ 23263
\sys_if_engine_ptex_p: .............. 114, 9883, 23432
\sys_if_engine_uptex:TF ... 114, 9883
\sys_if_engine_uptex_p: .............. 114, 9883, 23433
\sys_if_engine_xetex:TF .......... 5, 114, 3349, 9883, 9921, 10165, 10970, 32454, 32456, 32458
\sys_if_engine_xetex_p: .............. 114, 9883, 23433
\sys_if_platform_unix:TF ........ 10165, 10970, 32454, 32456, 32458
\sys_if_platform_unix_p: .............. 114, 6228, 6252, 6274, 6441, 9883, 10064, 29214, 30129, 30143, 30179, 30219, 30267, 30303, 30397, 30404, 30482, 30560, 30638, 30661, 30695, 31263, 31348, 32452
\sys_if_platform_dvi:TF ........ 115, 10146
\sys_if_platform_dvi_p: .............. 115, 10146
\sys_if_platform_pdf:TF ........ 115, 9935, 10146, 10168
\sys_if_platform_pdf_p: .............. 115, 10146
\sys_if_rand_exist:TF .............. 267, 542, 9900, 10087, 10101, 16493, 22486, 22510
\sys_if_rand_exist_p: .............. 267, 9900
\sys_if_shell: .................. 116
\sys_if_shell:TF .................. 116, 9982, 10126, 32005
\sys_if_shell_p: .............. 116, 10126
\sys_if_shell_restricted:TF .......... 116, 10126
\sys_if_shell_restricted_p: .............. 116, 10126
\sys_if_shell_unrestricted:TF .......... 116, 10126
\sys_if_shell_unrestricted_p: .............. 116, 10126
\c_sys_jobname_str .................. 114, 163, 549, 10051, 32362
\sys_load_backend:n ............. 117, 117, 9902
\sys_load_debug: .................. 117, 9961
\sys_load_deprecation: ............. 117, 9961
\c_sys_minute_int .............. 114, 10053
\c_sys_month_int .............. 114, 10053
\c_sys_output_str .............. 115, 10146
\c_sys_platform_str .............. 115, 9902, 14548, 14569
\sys_rand_seed: .............. 86, 115, 215, 10085
\c_sys_shell_escape_int ........... 116, 10114, 10129, 10131, 10133
\sys_shell_now:n .............. 116, 10017
\sys_shell_shipout:n .............. 116, 10030
\c_sys_year_int .............. 114, 10053
sys internal commands:
\g__sys_backend_tl .............. 9912, 9913, 9914, 10160
\_sys_const:nn .............. 9867, 9897, 9900, 10128, 10130, 10132, 10155, 10157
\g__sys_debug_Bool ........ 9959, 9963, 9965
\g__sys_deprecation_Bool ......... 1190, 9959, 9969, 9971
\_sys_everyjob:n .............. 10043, 10051, 10053, 10085, 10099, 10114, 10126, 10135
\g__sys_everyjob_tl .............. 10043
\_sys_finalise:n .............. 10137, 10146, 10161, 10174
\g__sys_finalise_tl .............. 10137
\_sys_get:nn .............. 9975
\_sys_get_do:W .............. 9975
\_sys_internal_tl .............. 9974
\_sys_load_backend_check:N ........ 9902
\c__sys_marker_tl .............. 9974, 9998, 10010
\c_sys_shell_stream_int .............. 10015, 10027, 10040
\_sys_tmp:w .............. 10056, 10077, 10079, 10080, 10081, 10082
syst commands:
\c_syst_last_allocated_tokens ........ 23026
T
\texttt{t} .............. 29521, 31387
\texttt{tabskip} .............. 554
\texttt{tagcode} .............. 798, 1667
tan .............. 213
tand .............. 213
\texttt{tate} .............. 1258, 2077
\texttt{tbaselineshift} .............. 2059, 2785
\texttt{temp} .............. 164, 170, 175, 178, 179, 186, 191, 194
\TeX and \LaTeX 2e commands:
\texttt{\&} .............. 5900
\texttt{\@end} .............. 297, 1289, 1290
\texttt{\@hyphen} .............. 1293
\texttt{\@input} .............. 1294
\texttt{\@italiccrr} .............. 1295
\texttt{\@shipout} .............. 1297, 2988
\texttt{\@tracingfonts} .............. 298, 1333
\texttt{\@underline} .............. 1296
\texttt{\@addtofilelist} .............. 14358
\texttt{\@changedcmd} .............. 29831
\texttt{\@classoptionslist} .............. 10176, 10178, 10180
\texttt{\@currentcmd} .............. 29828
<table>
<thead>
<tr>
<th>Command</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\tex_preadisplaygapfactor:D</td>
<td>989</td>
</tr>
<tr>
<td>\tex_preadisplaypenalty:D</td>
<td>512</td>
</tr>
<tr>
<td>\tex_preadisplaysize:D</td>
<td>513</td>
</tr>
<tr>
<td>\tex_preexhyphenchar:D</td>
<td>990, 1364</td>
</tr>
<tr>
<td>\tex_prehyphenchar:D</td>
<td>991, 1365</td>
</tr>
<tr>
<td>\tex_preleapenalty:D</td>
<td>992</td>
</tr>
<tr>
<td>\tex_prebottompenalty:D</td>
<td>514</td>
</tr>
<tr>
<td>\tex_predepth:D</td>
<td>515</td>
</tr>
<tr>
<td>\tex_prevgraf:D</td>
<td>516</td>
</tr>
<tr>
<td>\tex_refract:D</td>
<td></td>
</tr>
<tr>
<td>\tex_refraction:D</td>
<td></td>
</tr>
<tr>
<td>\tex_refractionfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_refractionpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_refractionsize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relax:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxbuffer:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstyle:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxvalue:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxwidth:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretch:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxshrink:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrink:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkfactor:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinkpenalty:D</td>
<td></td>
</tr>
<tr>
<td>\tex_relaxstretchshrinksize:D</td>
<td></td>
</tr>
</tbody>
</table>
Index

1334

23743, 23976, 24238, 24396, 25048, 31198, 31250, 31875, 31876, 32662 \use_none:nn

20, 396, 401, 411, 411, 498, 2226, 2306, 2314, 2385, 3899, 4527, 4671, 4844, 5010, 5027, 6080, 6411, 8243, 8427, 9655, 10216, 10524, 13573, 13617, 16766, 16849, 16853, 16857, 16861, 22184, 25610, 26246, 31199 \use_none:nnn

20, 412, 2226, 3667, 3682, 4111, 5051, 13574, 16848, 16852, 16856, 16860, 17501, 23774 \use_none:nnnn

...... 20, 2226, 13575, 14820, 31201 \use_none:nnnnn

20, 325, 646, 748, 2226, 13576, 13586, 16992, 17026, 17052, 17060, 19102 \use_none:nnnnnn

...... 20, 2226, 2438, 13577, 32126 \use_none:nnnnnnn

20, 748, 2226, 16994, 17028, 17054, 17062, 17385, 19553 \use_none:nnnnn

...... 20, 329, 2226, 2347, 3748 \use_none:nnnnnnnnnnn

20, 2226 \use_none:nnnnnnnnnn

...... 21, 49, 40, 327, 2219, 3975, 3990 \use_none:nnnnnnnnnnnnnnn

21, 2219 \use_none:nnnnnnnnnnnnnnnnnnn

...... 21, 2219 \use_none:nnnnnnnnnnnnnnnnnnnnn

21, 49, 40, 327, 2219, 3975, 3990 \use_none:nnnnnnnnnnnnnnnnnnnnn

...... 21, 49, 40, 327, 2219, 3975, 3990 \use_none:nnnnnnnnnnnnnnn

...... 21, 49, 40, 327, 2219, 3975, 3990 \use_none:nnnnnnnnnnnnnnnnn

...... 21, 486, 461, 2219 \use_none:nnnnnnnnnnnnnnnnnnn

...... 21, 486, 461, 2219 \use_none:nnnnnnnnnnnnnnnnnnn

...... 42, 42, 4256 \useboxresource

1033, 1696 \usefont

10321 \useimageresource

1034, 1697 \Usekeyed

1198, 2033 \Usekeyedwithdelims

1199, 2034 \Ustack

1200, 2035 \Ustartdisplaymath

1201, 2036 \Ustartmath

1202, 2037 \Ustopdisplaymath

1203, 2038 \Ustopmath

1204, 2039 \Usubscript

1205, 2040 \Usuperscript

1206, 2041 \utext commands:

\utext_binbinspacing:D ...... 1887 \utext_char:D .................. 1877
\utext_charcat:D ............ 1878
\utext_closebinspacing:D .... 1902 \utext_closeclospacing:D ...... 1904
\utext_closecrosspacing:D ... 1906 \utext_closeinnerspacing:D .. 1907
\utext_closeopenspacing:D .. 1908 \utext_closeordpacing:D .... 1909
\utext_closepunctspacing:D .. 1911 \utext_closerelsPacing:D .... 1912
\utext_connectordotpacing:D ... 1916
\utext_delscode:D .......... 1879
\utext_delscodenumput:D .... 1880
\utext_delimiter:D .......... 1881
\utext_delimiterover:D ...... 1882
\utext_delimiterunder:D ..... 1883
\utext_fraction: D .......... 1917
\utext_fractionnum: D ...... 1919
\utext_fractionnumv: D ..... 1921
\utext_fractionnumvgap: D .. 1922
\utext_fractionnumvgap: D .. 1923
\utext_fractionrule: D ..... 1924
\utext_hextensible:D ...... 1884
\utext_innerbinspacing:D ... 1925
\utext_innerclospacing:D ... 1927
\utext_innercrosspacing:D .. 1929
\utext_innerinnerspacing:D .. 1930
\utext_inneropenspacing:D .. 1931
\utext_innerordpacing:D ... 1932
\utext_innerpunctspacing:D .. 1934
\utext_innerrelsPacing:D ... 1935
\utext_limitabovev: D ...... 1936
\utext_limitabov: D ....... 1937
\utext_limitabover: D ...... 1938
\utext_limitabovevgap: D .. 1939
\utext_limitabovev: D ...... 1940
\utext_limitbelowv: D ...... 1941
\utext_limitbelowv: D ...... 1942
\utext_limitbelowvgap: D .. 1943
\utext_limitbelowv: D ...... 1944
\utext_limitbelowv: D ...... 1945
Index

\TeXcountfeatures 819, 1708 \TeXOTcountfeatures 855, 1744
\TeXcountglyphs 820, 1709 \TeXOTcountlanguages 856, 1745
\TeXcountselectors 821, 1710 \TeXOTcountscripts 857, 1746
\TeXcountvariations 822, 1711 \TeXOTfeaturetag 858, 1747
\TeXdashbreakstate 824, 1713 \TeXOTlanguagetag 859, 1748
\TeXdefaultencoding 823, 1712 \TeXOTscripttag 860, 1749
\TeXfeaturecode 825, 1714 \TeXpdffile 861, 1750
\TeXfeaturename 826, 1715 \TeXpdfpagecount 862, 1751
\TeXfindfeaturebyname 827, 1716 \TeXpicfile 863, 1752
\TeXfindselectorbyname 829, 1718 \TeXrevision 864, 1762
\TeXfindvariationbyname 831, 1720 \TeXselectorname 865, 1753
\TeXfirstfontchar 833, 1722 \TeXtracingfonts 866, 1754
\TeXglyphname 834, 1723 \TeXupwardsmode 867, 1755
\TeXgenerateactualtext 835, 1724 \TeXuseglyphmetrics 868, 1756
\TeXglyph 837, 1726 \TeXVariation 869, 1757
\TeXglyphbounds 838, 1727 \TeXVariationdefault 870, 1758
\TeXglyphindex 839, 1728 \TeXVariationmax 871, 1759
\TeXglyphname 840, 1729 \TeXVariationmin 872, 1760
\TeXinputencoding 841, 1730 \TeXVariationname 873, 1761
\TeXinputnormalization 842, 1731 \TeXVersion 874, 1763
\TeXinterchartokenstate 844, 1733 \xkanjiskip 1263, 2080
\TeXinterchartoks 846, 1735 \xleaders 606
\TeXisdefaultselector 847, 1736 \xspaceskip 1264, 2081
\TeXisexclusivefeature 849, 1738 \xscode 607
\TeXlastfontchar 851, 1740 \Y
\TeXlinebreaklocale 853, 1742 \ybaselineshift 1265, 2082
\TeXlinebreakpenalty 854, 1743 \year 608, 1423, 10082
\TeXlinebreakskip 852, 1741 \yoko 1266, 2083