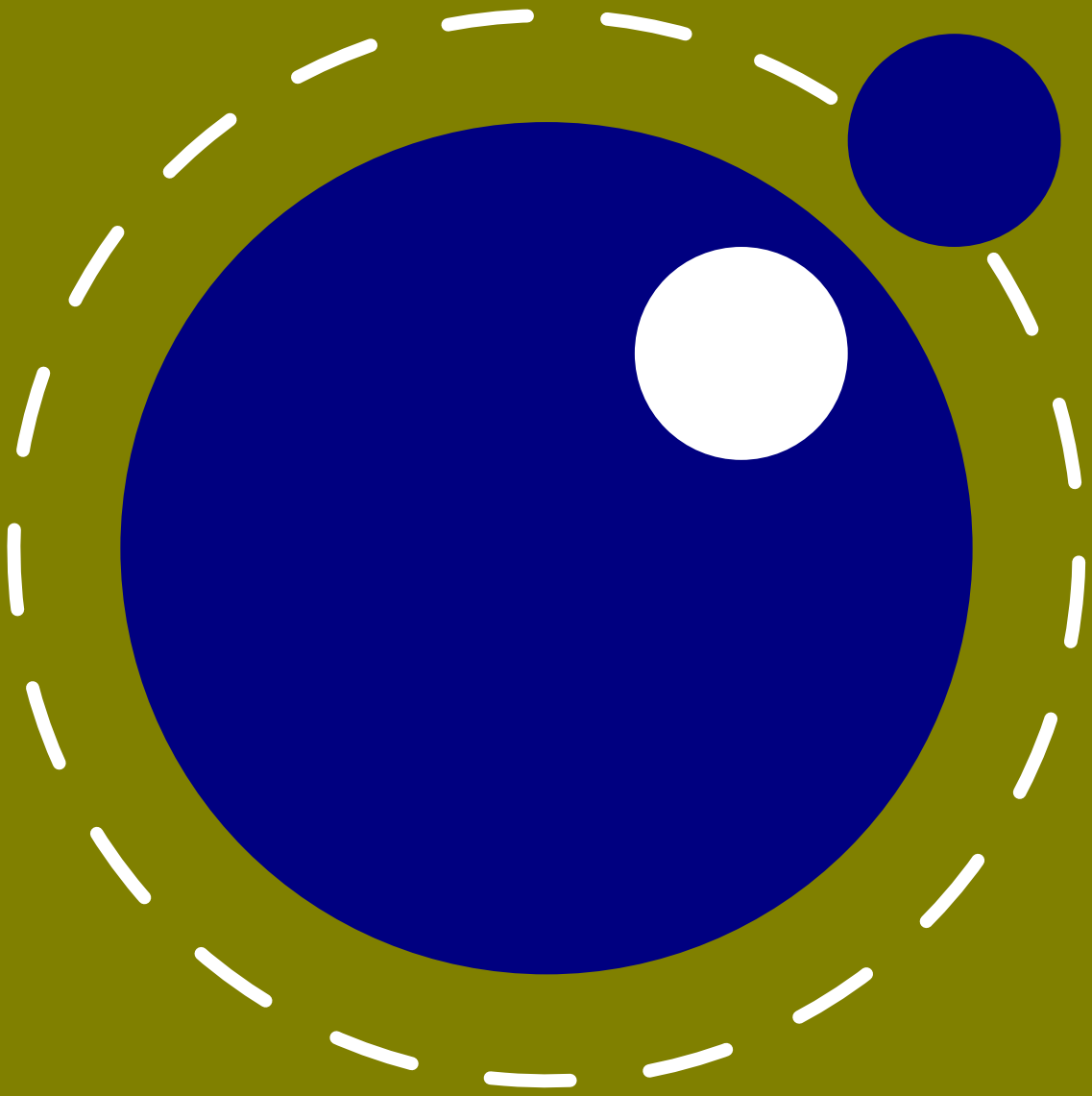
LuaMetaTEX Reference Manual



October 2021
Version 2.09.24

LuaMetaTEX Reference Manual

copyright : **LuaTEX development team**
: **ConTEXt development team**
more info : **www.luatex.org**
: **contextgarden.net**
version : **October 13, 2021**

Contents

Introduction **11**

1 The internals **13**

2 Differences with LuaTEX **17**

3 The original engines **27**
3.1 The merged engines 27
3.1.1 The rationale 27
3.1.2 Changes from TEX 3.1415926... 27
3.1.3 Changes from ε-TEX 2.2 28
3.1.4 Changes from pdfTEX 1.40 29
3.1.5 Changes from Aleph RC4 30
3.1.6 Changes from standard web2c 30
3.2 Implementation notes 31
3.2.1 Memory allocation 31
3.2.2 Sparse arrays 31
3.2.3 Simple single-character csnames 31
3.2.4 Binary file reading 31
3.2.5 Tabs and spaces 32
3.2.6 Logging 32
3.2.7 Parsing 33

4 Using LuaMetaTEX **35**
4.1 Initialization 35
4.1.1 LuaMetaTEX as a Lua interpreter 35
4.1.2 Other commandline processing 35
4.2 Lua behaviour 37
4.2.1 The Lua version 37
4.2.2 Locales 37
4.3 Lua modules 37
4.4 Testing 38

5 Basic TEX enhancements **39**
5.1 Introduction 39
5.1.1 Primitive behaviour 39
5.1.2 Rationale 39
5.1.3 Version information 40
5.2 Unicode text support 41
5.2.1 Extended ranges 41
5.2.2 `\Uchar` 42
5.2.3 Extended tables 42

5.3 Attributes 44
5.3.1 Nodes 44
5.3.2 Attribute registers 45
5.3.3 Box attributes 45
5.4 Lua related primitives 47
5.4.1 \directlua 47
5.4.2 \luaescapestring 48
5.4.3 \luafunction, \luafunctioncall and \luadef 49
5.4.4 \luabytecode and \luabytecodecall 49
5.5 Catcode tables 50
5.5.1 Catcodes 50
5.5.2 \catcodetable 50
5.5.3 \initcatcodetable 50
5.5.4 \savecatcodetable 51
5.6 Tokens, commands and strings 51
5.6.1 \scantextokens and \tokenized 51
5.6.2 \toksapp, \tokspre, \etoksapp, \etokspre, \gtoksapp, \gtokspre, \xtoksapp, \xtokspre 51
5.6.3 \csstring, \begincsname and \lastnamedcs 52
5.6.4 \clearmarks, \flushmarks, \currentmarks 52
5.6.5 \alignmark, \aligntab, \aligncontent and \tabsize 53
5.6.6 \letcharcode 53
5.6.7 \lettonothing and \glettonothing 53
5.6.8 \glet 53
5.6.9 \defcsname, \edefcsname, \edefcsname and \xdefcsname 54
5.6.10 \expanded 54
5.6.11 \ignorepars 54
5.6.12 \futureexpand, \futureexpandis, \futureexpandisap 54
5.6.13 \aftergrouped 54
5.7 Conditions 55
5.7.1 \ifabsnum and \ifabsdim 55
5.7.2 \ifcmpnum, \ifcmpdim, \ifnumval, \ifdimval, \ifchknum and \ifchkdim 56
5.7.3 \ifmathstyle and \ifmathparameter 57
5.7.4 \ifempty 57
5.7.5 \ifrelax 57
5.7.6 \ifboolean 57
5.7.7 \iftok and \ifcstok 57
5.7.8 \ifarguments, \ifparameters and \ifparameter 58
5.7.9 \ifcondition 58
5.7.10 \orelse and \orunless 59
5.7.11 \ifflags 60
5.8 Boxes, rules and leaders 61
5.8.1 \outputbox 61
5.8.2 \hrule, \vrule, \nohrule and \novrule 61
5.8.3 \vsplit 62

5.8.4 Images and reused box objects 62
5.8.5 \hpack, \vpack and \tpack 63
5.8.6 \gleaders 63
5.9 Languages 63
5.9.1 \hyphenationmin 63
5.9.2 \boundary, \noboundary, \protrusionboundary and \wordboundary 64
5.10 Control and debugging 64
5.10.1 Tracing 64
5.10.2 \lastnodetype, \lastnodesubtype, \currentiftype 65
5.11 Files 65
5.11.1 File syntax 65
5.11.2 Writing to file 65
5.12 Math 66
5.13 Fonts 66
5.14 Directions 66
5.14.1 Two directions 66
5.14.2 How it works 66
5.14.3 Normalizing lines 68
5.14.4 Orientations 68
5.15 Keywords 69
5.16 Expressions and \numericscale 70
5.17 Macro arguments 70
5.18 Overload protection 71
5.19 Constants with \integerdef and \dimensiondef 73
5.20 Serialization with \todimension, \toscaled and \tointeger 73
5.21 Nodes 74

6 Fonts 75
6.1 Introduction 75
6.2 Defining fonts 75
6.3 Virtual fonts 79
6.4 Additional TEX commands 82
6.4.1 Font syntax 82
6.4.2 \fontid and \setfontid 82
6.4.3 \glyphoptions 82
6.4.4 \glyphxscale, \glyphyscale and \scaledfontdimen 83
6.4.5 \glyphxoffset, \glyphyoffset 83
6.4.6 \glyph 83
6.4.7 \nospaces 84
6.4.8 \protrusionboundary 84
6.5 The Lua font library 85
6.5.1 Introduction 85
6.5.2 Defining a font with define, addcharacters and setfont 85
6.5.3 Font ids: id, max and current 85
6.5.4 Glyph data: \glyphdatafield, \glyphscriptfield, \glyphstatefield 86

7 Languages, characters, fonts and glyphs 87
7.1 Introduction 87
7.2 Characters, glyphs and discretionaries 87
7.3 The main control loop 92
7.4 Loading patterns and exceptions 94
7.5 Applying hyphenation 96
7.6 Applying ligatures and kerning 97
7.7 Breaking paragraphs into lines 98
7.8 The language library 98
7.8.1 new and id 98
7.8.2 hyphenation 99
7.8.3 clearhyphenation and clean 99
7.8.4 patterns and clearpatterns 99
7.8.5 hyphenationmin 100
7.8.6 [pre|post][ex|]hyphenchar 100
7.8.7 hyphenate 100
7.8.8 [set|get]hjcode 100

8 Math 103
8.1 Traditional alongside OpenType 103
8.2 Unicode math characters 103
8.3 Math styles 105
8.3.1 \mathstyle 105
8.3.2 \Ustack 106
8.3.3 The new \cramped ...style commands 106
8.4 Math parameter settings 108
8.4.1 Many new \Umath* primitives 108
8.4.2 Font-based math parameters 109
8.5 Math spacing 113
8.5.1 Setting inline surrounding space with \mathsurround and \mathsurroundskip 113
8.5.2 Pairwise spacing and \Umath...spacing commands 114
8.5.3 Local \frozen settings with 115
8.5.4 Checking a state with \ifmathparameter 116
8.5.5 Skips around display math and \mathdisplayskipmode 116
8.5.6 Nolimit correction with \mathnolimitsmode 116
8.5.7 Controlling math italic mess with \mathitalicsmode 117
8.5.8 Influencing script kerning with \mathscriptboxmode 117
8.5.9 Forcing fixed scripts with \mathscriptsmode 118
8.5.10 Penalties: \mathpenaltiesmode 119
8.5.11 Equation spacing: \matheqnogapstep 119
8.6 Math constructs 119
8.6.1 Unscaled fences and \mathdelimitersmode 119
8.6.2 Accent handling with \Umathaccent 120
8.6.3 Building radicals with \Uradical and \Uroot 121
8.6.4 Super- and subscripts 121

8.6.5 Scripts on extensibles: \Uunderdelimiter, \Uoverdelimiter, \Udelimiterover, \Udelimiterunder and \Uhextensible 122
8.6.6 Fractions and the new \Uskewed and \Uskewedwithdelims 123
8.6.7 Math styles: \Ustyle 124
8.6.8 Delimiters: \Uleft, \Umiddle and \Uright 125
8.6.9 Accents: \mathlimitsmode 125
8.7 Extracting values 125
8.7.1 Codes and using \Umathcode, \Umathcharclass, \Umathcharfam and \Umathcharslot 125
8.7.2 Last lines and \predisplaygapfactor 126
8.8 Math mode 126
8.8.1 Verbose versions of single-character math commands like \Usuperscript and \Usubscript 126
8.8.2 Script commands \Unosuperscript and \Unosubscript 127
8.8.3 Allowed math commands in non-math modes 127
8.9 Goodies 127
8.9.1 Flattening: \mathflattenmode 127
8.9.2 Less Tracing 128
8.10 Experiments 128
8.10.1 Prescripts with \Usuperprescript and Usubprescript 128
8.10.2 Prescripts with \Usuperprescript and Usubprescript 129

9 Building paragraphs and pages 131
9.1 Introduction 131
9.2 Paragraphs 131
9.3 Inserts 131
9.4 Marks 132
9.5 Migration 132
9.6 Pages 133

10 Nodes 135
10.1 Lua node representation 135
10.2 Main text nodes 136
10.2.1 hlist and vlist nodes 137
10.2.2 rule nodes 137
10.2.3 insert nodes 138
10.2.4 mark nodes 138
10.2.5 adjust nodes 139
10.2.6 disc nodes 139
10.2.7 math nodes 140
10.2.8 glue nodes 140
10.2.9 glue_spec nodes 141
10.2.10 kern nodes 141
10.2.11 penalty nodes 142
10.2.12 glyph nodes 142
10.2.13 boundary nodes 143
10.2.14 par nodes 143

10.2.15 dir nodes 144
10.2.16 Whatsits 144
10.2.17 Math noads 144
10.3 The node library 148
10.3.1 Introduction 148
10.3.2 Housekeeping 150
10.3.3 Manipulating lists 152
10.3.4 Glue handling 157
10.3.5 Attribute handling 158
10.3.6 Glyph handling 160
10.3.7 Packaging 161
10.3.8 Math 163
10.4 Two access models 164
10.5 Normalization 170
10.6 Properties 171

11 Lua callbacks 175
11.1 Registering callbacks 175
11.2 File related callbacks 176
11.2.1 find_format_file and find_log_file 176
11.2.2 open_data_file 176
11.3 Data processing callbacks 176
11.3.1 process_jobname 176
11.4 Node list processing callbacks 177
11.4.1 contribute_filter 177
11.4.2 buildpage_filter 177
11.4.3 build_page_insert 177
11.4.4 pre_linebreak_filter 178
11.4.5 linebreak_filter 179
11.4.6 append_to_vlist_filter 179
11.4.7 post_linebreak_filter 179
11.4.8 glyph_run 180
11.4.9 hpack_filter 180
11.4.10 vpack_filter 180
11.4.11 hpack_quality 181
11.4.12 vpack_quality 181
11.4.13 process_rule 181
11.4.14 pre_output_filter 181
11.4.15 hyphenate 182
11.4.16 ligaturing 182
11.4.17 kerning 182
11.4.18 insert_par 182
11.4.19 mlist_to_hlist 183

11.5 Information reporting callbacks 183
11.5.1 pre_dump 183
11.5.2 start_run 183
11.5.3 stop_run 183
11.5.4 intercept_tex_error, intercept_lua_error 184
11.5.5 show_error_message and show_warning_message 184
11.5.6 start_file 184
11.5.7 stop_file 184
11.5.8 wrapup_run 184
11.6 Font-related callbacks 185
11.6.1 define_font 185
11.6.2 show_whatsit 185

12 The TEX related libraries 187
12.1 The lua library 187
12.1.1 Version information 187
12.1.2 Table allocators 187
12.1.3 Bytecode registers 187
12.1.4 Introspection 188
12.2 The status library 188
12.3 The tex library 197
12.3.1 Introduction 197
12.3.2 Internal parameter values, set and get 197
12.3.3 Convert commands 199
12.3.4 Item commands 200
12.3.5 Accessing registers: set*, get* and is* 200
12.3.6 Character code registers: [get|set]*code[s] 202
12.3.7 Box registers: [get|set]box 203
12.3.8 triggerbuildpage 204
12.3.9 splitbox 204
12.3.10 Accessing math parameters: [get|set]math 204
12.3.11 Special list heads: [get|set]list 205
12.3.12 Semantic nest levels: getnest and ptr 206
12.3.13 Print functions 207
12.3.14 Helper functions 209
12.3.15 Functions for dealing with primitives 212
12.3.16 Core functionality interfaces 217
12.3.17 Functions related to synctex 219
12.4 The texconfig table 219
12.5 The texio library 220
12.5.1 write and writeselector 220
12.5.2 writenl and writeselectornl 221
12.5.3 setescape 221
12.5.4 closeinput 221

12.6 The token library 221
12.6.1 The scanner 221
12.6.2 Picking up one token 224
12.6.3 Creating tokens 224
12.6.4 Macros 225
12.6.5 Pushing back 226
12.6.6 Nota bene 227

13 The MetaPost library mplib 229
13.1 Introduction 229
13.2 Process management 229
13.2.1 new 229
13.2.2 getstatistics 231
13.2.3 execute 232
13.2.4 finish 232
13.2.5 settolerance and gettolerance 232
13.2.6 Errors 232
13.2.7 The scanner status 232
13.2.8 The hash 233
13.2.9 Callbacks 233
13.3 The end result 233
13.3.1 The figure 233
13.3.2 fill 234
13.3.3 outline 234
13.3.4 start_bounds, start_clip, start_group 235
13.3.5 stop_bounds, stop_clip, stop_group 235
13.4 Subsidiary table formats 235
13.4.1 Paths and pens 235
13.4.2 Colors 235
13.4.3 Transforms 236
13.4.4 Dashes 236
13.4.5 Pens and peninfo 236
13.4.6 Character size information 236
13.5 Scanners 237
13.6 Injectors 238
13.7 To be checked 240

14 The pdf related libraries 241
14.1 The pdfe library 241
14.1.1 Introduction 241
14.1.2 open, openfile, new, getstatus, close, unencrypt 241
14.1.3 getsize, getversion, getnofobjects, getnofpages 242
14.1.4 get[catalog|trailer|info] 242
14.1.5 getpage, getbox 242
14.1.6 get[string|integer|number|boolean|name] 243
14.1.7 get[dictionary|array|stream] 243
14.1.8 [open|close|readfrom|whole|]stream 243

14.1.9 getfrom[dictionary|array] 244
14.1.10 [dictionary|array]totable 244
14.1.11 getfromreference 245
14.2 Memory streams 245
14.3 The pdfscanner library 245

15 Extra libraries 247
15.1 Introduction 247
15.2 File and string readers: fio and type sio 247
15.3 md5 248
15.4 sha2 248
15.5 xzip 248
15.6 xmath 248
15.7 xcomplex 250
15.8 xdecimal 251
15.9 lfs 251
15.10 pngdecode 252
15.11 basexx 253
15.12 Multibyte string functions 253
15.13 Extra os library functions 254
15.14 The lua library functions 255

Primitive codes 257

Topics 277

Primitives 281

Callbacks 287

Nodes 289

Libraries 291

Statistics 299

Some remarks 301

Introduction

Around 2005 we started the LuaTEX project and it took about a decade to reach a state where we could consider the experiments to have reached a stable state. Pretty soon LuaTEX could be used in production, even if some of the interfaces evolved, but ConTEXt was kept in sync so that was not really a problem. In 2018 the functionality was more or less frozen. Of course we might add some features in due time but nothing fundamental will change as we consider version 1.10 to be reasonable feature complete. Among the reasons is that this engine is now used outside ConTEXt too which means that we cannot simply change much without affecting other macro packages.

In reaching that state some decisions were delayed because they didn't go well with a current stable version. This is why at the 2018 ConTEXt meeting those present agreed that we could move on with a follow up tagged MetaTEX, a name we already had in mind for a while, but as Lua is an important component, it got expanded to LuaMetaTEX. This follow up is a lightweight companion to LuaTEX that will be maintained alongside. More about the reasons for this follow up as well as the philosophy behind it can be found in the document(s) describing the development. During LuaTEX development I kept track of what happened in a series of documents, parts of which were published as articles in user group journals, but all are in the ConTEXt distribution. I did the same with the development of LuaMetaTEX.

The LuaMetaTEX engine is, as said, a follow up on LuaTEX. Just as we have ConTEXt MkII for pdfTEX and XETEX, we have MkIV for LuaTEX so for LuaMetaTEX we have yet another version of ConTEXt: LMTX. By freezing MkII, and at some point freezing MkIV, we can move on as we like, but we try to remain downward compatible where possible, something that the user interface makes possible. Although LuaMetaTEX can be used for production we can also use it for possibly drastic experiments but without affecting LuaTEX. Because we can easily adapt ConTEXt to support both, no other macro package will be harmed when (for instance) the interface that the engine provides change as part of an experiment or cleanup of code. Of course, when we consider something to be useful, it can be ported back to LuaTEX, but only when there are good reasons for doing so and when no compatibility issues are involved.

By now the code of these two related engines differs a lot so in retrospect it makes less sense to waste time on porting back. When considering this follow up one consideration was that a lean and mean version with an extension mechanism is a bit closer to original TEX. Of course, because we also have new primitives, this is not entirely true. The basic algorithms remain the same but code got reshuffled and because we expose internal names of variables and such that is reflected in the code base (like more granularity in nodes and token commands). Delegating tasks to Lua already meant that some aspects, especially system dependent ones, no longer made sense and therefore had consequences for the interface at the system level. In LuaMetaTEX more got delegated, like all file related operations. The penalty of moving even more responsibility to Lua has been compensated by (hopefully) harmless optimization of code in the engine and some more core functionality. In the process system dependencies have been minimalized.

One side effect of opening up is that what normally is hidden gets exposed and this is also true for all kind of codes that are used internally to distinguish states and properties of commands, tokens, nodes and more. Especially during development these can change but the good news is that they can be queried so on can write in code independent ways (in LuaTEX node id's are

an example). That also means more interface related commands, so again lean and mean is not applicable here, especially because the detailed control over the text, math, font and language subsystems also results in additional commands to query their state. And, as the MetaPost got extended, that subsystem is on the one hand leaner and meaner because backend code was dropped but on the other hand got a larger code base due to opening up the scanner and adding a feedback mechanism.

This manual started as an adaptation of the LuaTEX manual and therefore looks similar. Some chapters are removed, others were added and the rest has been (and will be further) adapted. It also discusses the (main) differences. Some of the new primitives or functions that show up in LuaMetaTEX might show up in LuaTEX at some point, but most will be exclusive to LuaMetaTEX, so don't take this manual as reference for LuaTEX! As long as we're experimenting we can change things at will but as we keep ConTEXt LMTX synchronized users normally won't notice this. Often you can find examples of usage in ConTEXt related documents and the source code so that serves a reference too. More detailed explanations can be found in documents in the ConTEXt distribution, if only because there we can present features in the perspective of useability.

For ConTEXt users the LuaMetaTEX engine will become the default. As mentioned, the ConTEXt variant for this engine is tagged LMTX. The pair can be used in production, just as with LuaTEX and MkIV. In fact, most users will probably not really notice the difference. In some cases there will be a drop in performance, due to more work being delegated to Lua, but on the average performance is much be better, due to some changes below the hood of the engine. Memory consumption is also less. The timeline of development is roughly: from 2018 upto 2020 engine development, 2019 upto 2021 the stepwise code split between MkIV and LMTX, while in 2021 and 2022 we will (mostly) freeze MkIV and LMTX will be the default.

As this follow up is closely related to ConTEXt development, and because we expect stock LuaTEX to be used outside the ConTEXt proper, there will be no special mailing list nor coverage (or pollution) on the LuaTEX related mailing lists. We have the ConTEXt mailing lists for that. In due time the source code will be part of the regular ConTEXt distribution so that is then also the reference implementation: if needed users can compile the binary themselves.

This manual sometimes refers to LuaTEX, especially when we talk of features common to both engine, as well as to LuaMetaTEX, when it is more specific to the follow up. A substantial amount of time went into the transition and more will go in, so if you want to complain about LuaMetaTEX, don't bother me. Of course, if you really need professional support with these engines (or TEX in general), you can always consider contacting the developers.

Hans Hagen

LuaMetaTEX Banner	:	luametatex 2.0924 / 20210914
LuaMetaTEX Version	:	October 13, 2021
ConTEXt Version	:	LMTX 2021.10.10 23:43
LuaTEX Team	:	Hans Hagen, Hartmut Henkel, Taco Hoekwater, Luigi Scarso
LuaMetaTEX Team	:	Hans Hagen, Alan Braslau, Mojca Miklavec and Wolfgang Schuster

1 The internals

This is a reference manual and not a tutorial. This means that we discuss changes relative to traditional TEX and also present new (or extended) functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later. Because the LuaTEX and LuaMetaTEX engines open up TEX there's suddenly quite some more to explain, especially about the way a (to be) typeset stream moves through the machinery. However, discussing all that in detail makes not much sense, because deep knowledge is only relevant for those who write code not possible with regular TEX and who are already familiar with these internals (or willing to spend time on figuring it out).

So, the average user doesn't need to know much about what is in this manual. For instance fonts and languages are normally dealt with in the macro package that you use. Messing around with node lists is also often not really needed at the user level. If you do mess around, you'd better know what you're dealing with. Reading “The TEX Book” by Donald Knuth is a good investment of time then also because it's good to know where it all started. A more summarizing overview is given by “TEX by Topic” by Victor Eijkhout. You might want to peek in “The ε-TEX manual” too.

But ... if you're here because of Lua, then all you need to know is that you can call it from within a run. If you want to learn the language, just read the well written Lua book. The macro package that you use probably will provide a few wrapper mechanisms but the basic `\directlua` command that does the job is:

```
\directlua{tex.print("Hi there")}
```

You can put code between curly braces but if it's a lot you can also put it in a file and load that file with the usual Lua commands. If you don't know what this means, you definitely need to have a look at the Lua book first.

If you still decide to read on, then it's good to know what nodes are, so we do a quick introduction here. If you input this text:

```
Hi There ...
```

eventually we will get a linked lists of nodes, which in ascii art looks like:

```
H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e ...
```

When we have a paragraph, we actually get something like this, where a `par` node stores some metadata and is followed by a `hlist` flagged as indent box:

```
[par] <=> [hlist] <=> H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e ...
```

Each character becomes a so called glyph node, a record with properties like the current font, the character code and the current language. Spaces become glue nodes. There are many node types and nodes can have many properties but that will be discussed later. Each node points back to a previous node or next node, given that these exist. Sometimes multiple characters are represented by one glyph (shape), so one can also get:

```
[par] <=> [hlist] <=> H <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

And maybe some characters get positioned relative to each other, so we might see:

```
[par] <=> [hlist] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

Actually, the above representation is one view, because in LuaMetaTEX we can choose for this:

```
[par] <=> [glue] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

where glue (currently fixed) is used instead of an empty hlist (think of a `\hbox`). Options like this are available because want a certain view on these lists from the Lua end and the result being predicable is part of that.

It's also good to know beforehand that TEX is basically centered around creating paragraphs and pages. The par builder takes a list and breaks it into lines. At some point horizontal blobs are wrapped into vertical ones. Lines are so called boxes and can be separated by glue, penalties and more. The page builder accumulates lines and when feasible triggers an output routine that will take the list so far. Constructing the actual page is not part of TEX but done using primitives that permit manipulation of boxes. The result is handled back to TEX and flushed to a (often pdf) file.

```
\setbox\scratchbox\vbox\bgroup
    line 1\par line 2
\egroup

\showbox\scratchbox
```

The above code produces the next log lines that reveal how the engines sees a paragraph (wrapped in a `\vbox`):

```
1:4: > \box257=
1:4: \vbox[normal][16=1,17=1,47=1], width 483.69687, height 27.58083, depth 0.1416, direction l2r
1:4: .\list
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...\list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, emergencystretch 12.0, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006C l
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U+000031 1
```

```
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt
1:4: ..\glue[par][16=1,17=1,47=1] 5.44995pt plus 1.81665pt minus 1.81665pt
1:4: ..\glue[baseline][16=1,17=1,47=1] 6.79396pt
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...\list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemer-
its 10000, linepenalty 10, doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty
2000, brokenpenalty 100, emergencystretch 12.0, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30:
DejaVuSerif @ 10.0pt>, glyph U+00006C l
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30:
DejaVuSerif @ 10.0pt>, glyph U+000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30:
DejaVuSerif @ 10.0pt>, glyph U+00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30:
DejaVuSerif @ 10.0pt>, glyph U+000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30:
DejaVuSerif @ 10.0pt>, glyph U+000032 2
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt
```

The LuaTEX engine provides hooks for Lua code at nearly every reasonable point in the process: collecting content, hyphenating, applying font features, breaking into lines, etc. This means that you can overload TEX's natural behaviour, which still is the benchmark. When we refer to 'callbacks' we means these hooks. The TEX engine itself is pretty well optimized but when you kick in much Lua code, you will notices that performance drops. Don't blame and bother the authors with performance issues. In ConTEXt over 50% of the time can be spent in Lua, but so far we didn't get many complaints about efficiency. Adding more callbacks makes no sense, also because at some point the performance hit gets too large. There are plenty of ways to achieve goals. For that reason: take remarks about LuaTEX, features, potential, performance etc. with a natural grain of salt.

Where plain TEX is basically a basic framework for writing a specific style, macro packages like ConTEXt and LATEX provide the user a whole lot of additional tools to make documents look good. They hide the dirty details of font management, language support, turning structure into typeset results, wrapping pages, including images, and so on. You should be aware of the fact that when you hook in your own code to manipulate lists, this can interfere with the macro package that you use. Each successive step expects a certain result and if you mess around to much, the engine eventually might bark and quit. It can even crash, because testing everywhere for what users can do wrong is no real option.

When you read about nodes in the following chapters it's good to keep in mind what commands relate to them. Here are a few:

COMMAND	NODE	EXPLANATION
\hbox	hlist	horizontal box
\vbox	vlist	vertical box with the baseline at the bottom
\vtop	vlist	vertical box with the baseline at the top
\hskip	glue	horizontal skip with optional stretch and shrink
\vskip	glue	vertical skip with optional stretch and shrink
\kern	kern	horizontal or vertical fixed skip
\discretionary	disc	hyphenation point (pre, post, replace)
\char	glyph	a character
\hrule	rule	a horizontal rule
\vrule	rule	a vertical rule
\textdirection	dir	a change in text direction

Whatever we feed into TEX at some point becomes a token which is either interpreted directly or stored in a linked list. A token is just a number that encodes a specific command (operator) and some value (operand) that further specifies what that command is supposed to do. In addition to an interface to nodes, there is an interface to tokens, as later chapters will demonstrate.

Text (interspersed with macros) comes from an input medium. This can be a file, token list, macro body cq. arguments, some internal quantity (like a number), Lua, etc. Macros get expanded. In the process TEX can enter a group. Inside the group, changes to registers get saved on a stack, and restored after leaving the group. When conditionals are encountered, another kind of nesting happens, and again there is a stack involved. Tokens, expansion, stacks, input levels are all terms used in the next chapters. Don't worry, they loose their magic once you use TEX a lot. You have access to most of the internals and when not, at least it is possible to query some state we're in or level we're at.

When we talk about pack(ag)ing it can mean two things. When TEX has consumed some tokens that represent text they are added to the current list. When the text is put into a so called \hbox (for instance a line in a paragraph) it (normally) first gets hyphenated, next ligatures are build, and finally kerns are added. Each of these stages can be overloaded using Lua code. When these three stages are finished, the dimension of the content is calculated and the box gets its width, height and depth. What happens with the box depends on what macros do with it.

The other thing that can happen is that the text starts a new paragraph. In that case some information is stored in a leading par node. Then indentation is appended and the paragraph ends with some glue. Again the three stages are applied but this time afterwards, the long line is broken into lines and the result is either added to the content of a box or to the main vertical list (the running text so to say). This is called par building. At some point TEX decides that enough is enough and it will trigger the page builder. So, building is another concept we will encounter. Another example of a builder is the one that turns an intermediate math list into something typeset.

Wrapping something in a box is called packing. Adding something to a list is described in terms of contributing. The more complicated processes are wrapped into builders. For now this should be enough to enable you to understand the next chapters. The text is not as enlightening and entertaining as Don Knuths books, sorry.

2 Differences with LuaTEX

As LuaMetaTEX is a leaner and meaner LuaTEX. This means that substantial parts and dependencies are gone: quite some font code, all backend code with related frontend code and of course image and font inclusion. There is also new functionality which makes for less lean but in the end we still have less, also in terms of dependencies. This chapter will discuss what is gone. We start with the primitives that were dropped.

fonts	`\letterspacefont` `\copyfont` `\expandglyphsinfont` `\ignoreligaturesinfont` `\tagcode` `\leftghost` `\rightghost`
backend	`\dviextension` `\dvivariable` `\dvifeedback` `\pdfextension` `\pdfvariable` `\pdffeedback` `\dviextension` `\draftmode` `\outputmode`
dimensions	`\pageleftoffset` `\pagerightoffset` `\pagetopoffset` `\pagebottomoffset` `\pageheight` `\pagewidth`
resources	`\saveboxresource` `\useboxresource` `\lastsavedboxresourceindex` `\saveimageresource` `\useimageresource` `\lastsavedimageresourceindex` `\lastsavedimageresourcepages`
positioning	`\savepos` `\lastxpos` `\lastypos`
directions	`\textdir` `\linedir` `\mathdir` `\pardir` `\pagedir` `\bodydir` `\pagedirection` `\bodydirection`
randomizer	`\randomseed` `\setrandomseed` `\normaldeviate` `\uniformdeviate`
utilities	`\synctex`
extensions	`\latelua` `\lateluafunction` `\openout` `\write` `\closeout` `\openin` `\read` `\readline` `\closein` `\ifeof`
control	`\suppressfontnotfounderror` `\suppresslongerror` `\suppressprimitiveerror` `\suppressmathparerror` `\suppressifcsnameerror` `\suppressoutererror` `\mathoption`
system	`\primitive` `\ifprimitive` `\formatname`
ignored	`\long` `\outer` `\mag`

The resources and positioning primitives are actually useful but can be defined as macros that (via Lua) inject nodes in the input that suit the macro package and backend. The three–letter direction primitives are gone and the numeric variants are now leading. There is no need for page and body related directions and they don't work well in LuaTEX anyway. We only have two directions left. Because we can hook in Lua functions that get information about what is expected (consumer or provider) there are plenty possibilities for adding functionality using this scripting langauge.

The primitive related extensions were not that useful and reliable so they have been removed. There are some new variants that will be discussed later. The `\outer` and `\long` prefixes are gone as they don't make much sense nowadays and them becoming dummies opened the way to something new: control sequence properties that permit protection against as well as controlled overloading of definitions. I don't think that (ConTEXt) users will notice these prefixes being gone. The definition and parsing related `\suppress..` features are now default and can't be changed so related primitives are gone.

The `\shipout` primitive does no ship out but just erases the content of the box unless of course

that has happened already in another way. A macro package should implement its own backend and related shipout. Talking of backend, the extension primitives that relate to backends can be implemented as part of a backend design using generic whatsits. There is only one type of whatsit now. In fact we're now closer to original TEX with respect to the extensions.

The `img` library has been removed as it's rather bound to the backend. The `slunicode` library is also gone. There are some helpers in the string library that can be used instead and one can write additional Lua code if needed. There is no longer a `pdf` backend library but we have an up to date pdf parsing library on board.

In the `node`, `tex` and `status` library we no longer have helpers and variables that relate to the backend. The LuaMetaTEX engine is in principle dvi and pdf unaware. There are, as mentioned, only generic whatsit nodes that can be used for some management related tasks. For instance you can use them to implement user nodes. More extensive status information is provided in the overhauled status library. All libraries have additional functionality and names of functions have been normalized (for as far as possible).

The margin kern nodes are gone and we now use regular kern nodes for them. As a consequence there are two extra subtypes indicating the injected left or right kern. The glyph field served no real purpose so there was no reason for a special kind of node.

The kpse library is no longer built-in, but one can use an external kpse library, assuming that it is present on the system, because the engine has a so called optional library interface to it. Because there is no backend, quite some file related callbacks could go away. The following file related callbacks remained (till now):

```
find_write_file find_format_file open_data_file
```

The callbacks related to errors are changed:

```
intercept_tex_error intercept_lua_error
show_error_message show_warning_message
```

There is a hook that gets called when one of the fundamental memory structures gets reallocated.

```
trace_memory
```

When you use the overload protect mechanisms, a callback can be plugged in to handle exceptions:

```
handle_overload
```

The (job) management hooks are kept:

```
process_jobname
start_run stop_run wrapup_run
pre_dump
start_file stop_file
```

Because we use a more generic whatsit model, there is a new callback:

```
show_whatsit
```

Because tracing boxes now reports a lot more information, we have a plug in for detail:

```
get_attribute
```

Being the core of extensibility, the typesetting callbacks of course stayed. This is what we ended up with:

```
alignment_filter, append_to_vlist_filter, begin_paragraph, build_page_insert,
buildpage_filter, contribute_filter, define_font, find_format_file,
find_log_file, get_attribute, glyph_run, handle_overload, hpack_filter,
hpack_quality, hyphenate, insert_par, intercept_lua_error, intercept_tex_error,
kerning, ligaturing, linebreak_filter, make_extensible, missing_character,
mlist_to_hlist, open_data_file, paragraph_context, post_linebreak_filter,
pre_dump, pre_linebreak_filter, pre_output_filter, process_jobname,
register_extensible, show_error_message, show_lua_call, show_warning_message,
show_whatsit, start_file, start_run, stop_file, stop_run, trace_memory,
vpack_filter, vpack_quality, wrapup_run
```

As in LuaTEX font loading happens with the following callback. This time it really needs to be set because there is no built-in font loader.

```
define_font
```

There are all kinds of subtle differences in the implementation, for instance we no longer intercept * and & as these were already replaced long ago in TEX engines by command line options. Talking of options, only a few are left. All input goes via Lua, even the console. One can program a terminal if needed.

We took our time for reaching a stable state in LuaTEX. Among the reasons is the fact that most was experimented with in ConTEXt, which we can adapt to the engine as we go. It took many years to decide what to keep and how to do things. Of course there are places when things can be improved but that most likely only happens in LuaMetaTEX. Contrary to what is sometimes suggested, the LuaTEX-ConTEXt MkIV combination (assuming matched versions) has been quite stable. It made no sense otherwise. Most ConTEXt functionality didn't change much at the user level. Of course there have been issues, as is natural with everything new and beta, but we have a fast update cycle.

The same is true for LuaMetaTEX and ConTEXt LMTX: it can be used for production as usual and in practice ConTEXt users tend to use the beta releases, which proves this. Of course, if you use low level features that are experimental you're on your own. Also, as with LuaTEX it might take many years before a long term stable is defined. The good news is that, when the source code has become part of the ConTEXt distribution, there is always a properly working, more or less long term stable, snapshot.

The error reporting subsystem has been redone quite a bit but is still fundamentally the same. We don't really assume interactive usage but if someone uses it, it might be noticed that it is not possible to backtrack or inject something. Of course it is no big deal to implement all that in Lua if needed. It removes a system dependency and makes for a bit cleaner code. In ConTEXt we quit on an error simply because one has to fix source anyway and runs are fast enough. Logging provides more detail and new primitives can be used to prevent clutter in tracing (the more complex a macro package becomes, the more extreme tracing becomes).

There are new primitives as well as some extensions to existing primitive functionality. These are described in following chapters but there might be hidden treasures in the binary. If you locate them, don't automatically assume them to stay, some might be part of experiments! There are for instance a few csname related definers, we have integer and dimension constants, the macro argument parser can be brought in tolerant mode, the repertoire of conditionals has been extended, some internals can be controlled (think of normalization of lines, hyphenation etc.), and macros can be protected against user overload. Not all is discussed in detail in this manual but there are introductions in the ConTEXt distribution that explain them. But the TEX kernel is of course omnipresent.

The following primitives are available in LuaTEX but not in LuaMetaTEX. Some of these are emulated in ConTEXt.

```
automatichyphenmode
bodydir
bodydirection
boxdir
breakafterdirmode
closein
closeout
compoundhyphenmode
copyfont
discretionaryligaturemode
draftmode
dviextension
dvifeedback
dvivariable
eTeXVersion
eTeXglueshrinkorder
eTeXgluestretchorder
eTeXminorversion
eTeXrevision
eTeXversion
expandglyphsinfont
fixupboxesmode
glyphdimensionsmode
hoffset
hyphenationbounds
hyphenpenaltymode
ifeof
ifprimitive
ignoreligaturesinfont
immediateassigned
immediateassignment
insertht
lastsavedboxresourceindex
lastsavedimageresourceindex
lastsavedimageresourcepages
lastxpos
lastypos
latelua
lateluafunction
leftghost
letterspacefont
linedir
mag
mathdefaultsmode
mathdir
mathoption
nokerns
noligs
nolocaldirs
nolocalwhatsits
normaldeviate
openin
openout
outputmode
pagebottomoffset
pagedir
pagedirection
pageheight
pageleftoffset
pagerightoffset
pagetopoffset
pagewidth
pardir
pdfextension
pdffeedback
pdfvariable
primitive
randomseed
```

```
read
readline
rightghost
saveboxresource
saveimageresource
savepos
setrandomseed
shapemode
special
suppressfontnotfounderror
suppressifcsnameerror
suppresslongerror
suppressmathparerror
suppressoutererror
suppressprimitiveerror
synctex
tagcode
textdir
tracingscantokens
uniformdeviate
useboxresource
useimageresource
voffset
write
```

The following primitives are available in LuaMetaTEX only. At some point in time some might be added to LuaTEX.

```
UUskewed
UUskewedwithdelims
Uabove
Uabovewithdelims
Uatop
Uatopwithdelims
Umathaccentbaseheight
Umathaccentvariant
Umathadapttoleft
Umathadapttoright
Umathbotaccentvariant
Umathclass
Umathdegreevariant
Umathdelimiterovervariant
Umathdelimiterundervariant
Umathdenominatorvariant
Umathextrasubpreshift
Umathextrasubshift
Umathextrasuppreshift
Umathextrasupshift
Umathfractionvariant
Umathhextensiblevariant
Umathlimits
Umathnoaxis
Umathnolimits
Umathnumeratorvariant
Umathopenupdepth
Umathopenupheight
Umathoverdelimitervariant
Umathoverlayaccentvariant
Umathoverlinevariant
Umathphantom
Umathradicalvariant
Umathspacebeforescript
Umathspacingmode
Umathstackvariant
Umathsubscriptvariant
Umathsuperscriptvariant
Umathtopaccentvariant
Umathunderdelimitervariant
Umathunderlinevariant
Umathvextensiblevariant
Umathvoid
Unosubprescript
Unosuperprescript
Uover
Uoverwithdelims
Ustyle
Usubprescript
Usuperprescript
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
afterassigned
aftergrouped
aliased
aligncontent
atendofgroup
atendofgrouped
automigrationmode
autoparagraphmode
beginlocalcontrol
```

beginsimplegroup
boxattribute
boxorientation
boxtotal
boxxmove
boxxoffset
boxymove
boxyoffset
currentmarks
defcsname
dimensiondef
dimexpression
edefcsname
endsimplegroup
enforced
everybeforepar
everytab
expand
expandafterpars
expandafterspaces
expandcstoken
expandtoken
flushmarks
fontmathcontrol
fontspecifiedname
fontspecifiedsize
fonttextcontrol
frozen
futurecsname
futuredef
futureexpand
futureexpandis
futureexpandisap
gdefcsname
gletcsname
glettonothing
gluespecdef
glyph
glyphdatafield
glyphoptions
glyphscale
glyphscriptfield
glyphscriptscale
glyphscriptscriptscale
glyphstatefield
glyphtextscale
glyphxoffset
glyphxscale
glyphyoffset
glyphyscale
hccode
hyphenationmode
ifarguments
ifboolean
ifchkdim
ifchknum
ifcmpdim
ifcmpnum
ifcstok
ifdimexpression
ifdimval
ifempty
ifflags
ifhaschar
ifhastok
ifhastoks
ifhasxtoks
ifinsert
ifmathparameter
ifmathstyle
ifnumexpression
ifnumval
ifparameter
ifparameters
ifrelax
iftok
ignorearguments
ignorepars
immutable
insertbox
insertcopy
insertdepth
insertdistance
insertheight
insertheights
insertlimit
insertmaxdepth
insertmode
insertmultiplier
insertpenalty
insertprogress
insertstorage

```
insertstoring
insertunbox
insertuncopy
insertwidth
instance
integerdef
lastarguments
lastchkdim
lastchknum
lastnodesubtype
lastparcontext
letcsname
letfrozen
letprotected
lettonothing
localcontrol
localcontrolled
mathcontrolmode
mathfontcontrol
mathscale
meaningfull
meaningless
mugluespecdef
mutable
noaligned
norelax
normalizelinemode
numericscale
numexpression
orelse
orunless
overloaded
overloadmode
overshoot
pageboundary
pagevsize
parametercount
parametermark
parattribute
parfillleftskip
permanent
retokenized
scaledfontdimen
semiexpanded
semiprotected
shapingpenaltiesmode
shapingpenalty
shownodedetails
snapshotpar
supmarkmode
swapcsvalues
tabsize
thewithoutunit
todimension
tointeger
tokenized
tolerant
toscaled
tracingalignments
tracingexpressions
tracinghyphenation
tracinginserts
tracinglevels
tracingmarks
tracingmath
undent
unhpack
unletfrozen
unletprotected
untraced
unvpack
wrapuppar
xdefcsname
```

As part of a bit more consistency some function names also changed. Names with an _ got that removed (as that was the minority). It's easy to provide a back mapping if needed (just alias the functions).

Todo: only mention the LuaTEX ones.

LIBRARY	OLD NAME	NEW NAME	COMMENT
language	clear_patterns	clearpatterns	
	clear_hyphenation	clearhyphenation	

mplib	italcor	italic	
	pen_info	peninfo	
	solve_path	solvepath	
texio	write_nl	writenl	old name stays
node	protect_glyph	protectglyph	
	protect_glyphs	protectglyphs	
	unprotect_glyph	unprotectglyph	
	unprotect_glyphs	unprotectglyphs	
	end_of_math	endofmath	
	mlist_to_hlist	mlisttohlist	
	effective_glue	effectiveglue	
	has_glyph	hasglyph	
	first_glyph	firstglyph	
	has_field	hasfield	
	copy_list	copylist	
	flush_node	flushnode	
	flush_list	flushlist	
	insert_before	insertbefore	
	insert_after	insertafter	
	last_node	lastnode	
	is_zero_glue	iszeroglue	
	make_extensible	makeextensible	
	uses_font	usesfont	
	is_char	ischar	
	is_direct	isdirect	
	is_glyph	isglyph	
	is_node	isnode	
token	scan_keyword	scankeyword	
	scan_keywordcs	scankeywordcs	
	scan_int	scanint	
	scan_real	scanreal	
	scan_float	scanfloat	
	scan_dimen	scandimen	
	scan_glue	scanglue	
	scan_toks	scantoks	
	scan_code	scancode	
	scan_string	scanstring	
	scan_argument	scanargument	
	scan_word	scanword	
	scan_csname	scancsname	
	scan_list	scanlist	
	scan_box	scanbox	

It's all part of trying to make the code base consistent but it is sometimes a bit annoying. However, that's why we develop this engine independent of the LuaTEX code base. It's anyway a change that has been on my todo list for quite a while because those inconsistencies annoyed

me.

3 The original engines

3.1 The merged engines

3.1.1 The rationale

The first version of LuaTEX, made by Hartmut after we discussed the possibility of an extension language, only had a few extra primitives and it was largely the same as pdfTEX. It was presented to the public in 2005. As part of the Oriental TEX project, Taco merged some parts of Aleph into the code and some more primitives were added. Then we started more fundamental experiments. After many years, when the engine had become more stable, the decision was made to clean up the rather hybrid nature of the program. This means that some primitives were promoted to core primitives, often with a different name, and that others were removed. This also made it possible to start cleaning up the code base, which showed decades of stepwise additions to original TEX. In chapter 5 we discuss some new primitives, here we will cover most of the adapted ones.

During more than a decade stepwise new functionality was added and after 10 years the more of less stable version 1.0 was presented. But we continued and after some 15 years the LuaMetaTEX follow up entered its first testing stage. But before details about the engine are discussed in successive chapters, we first summarize where we started from. Keep in mind that in LuaMetaTEX we have a bit less than in LuaTEX, so this section differs from the one in the LuaTEX manual.

Besides the expected changes caused by new functionality, there are a number of not-so-expected changes. These are sometimes a side-effect of a new (conflicting) feature, or, more often than not, a change necessary to clean up the internal interfaces. These will also be mentioned.

Again we stress that *this is not a TEX manual, nor a tutorial*. If you are unfamiliar with TEX first play a little with a macro package, take a look at the TEX book, make yourself familiar with the concepts and macro language. That will likely take days and not hours. Also, many of the new concepts introduced in LuaTEX and LuaMetaTEX are explained in documents that come with the ConTEXt distribution, articles and presentations. It doesn't pay of to repeat that here, especially not in a time when users often search instead of read from cover to cover.

3.1.2 Changes from TEX 3.1415926...

Of course it all starts with traditional TEX. Even if we started with pdfTEX, most still comes from original Knuthian TEX. But we divert a bit.

- The current code base is written in C, not Pascal. The original web documentation is kept when possible and not wrapped in tagged comments. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like `tex`, `lua`, `languages`, `fonts`, `libraries`, etc. There are some artifacts of the conversion to C, but these got (and get) removed stepwise. The documentation, which actually comes from the mix of engines (via so called change files), is a mix of what authors of the engines wove into the source, and most is of course from Don Knuths original. In LuaMetaTEX we try to stay as

close as possible to the original so that the documentation of the fundamentals behind TEX by Don Knuth still applies. However, because we use C, some documentation is a bit off. Also, most global variables are now collected in structures, but the original names and level of abstraction were mostly kept. On the other hand, opening up had its impact on the code, so that makes some documentation a bit off too. Adapting that all will take time.

- See chapter 7 for many small changes related to paragraph building, language handling and hyphenation. The most important change is that adding a brace group in the middle of a word (like in `of{}fice`) does not prevent ligature creation. Also, the hyphenation, ligature building and kerning has been split so that we can hook in alternative or extra code wherever we like. There are various options to control discretionary injection and related penalties are now integrated in these nodes. Language information is now bound to glyphs. The number of languages in LuaMetaTEX is smaller than in LuaTEX. Control over discretionaries is more granular and now managed by less variables.
- There is no pool file, all strings are embedded during compilation. This also removed some memory constraints. We kept token and node memory management because it is convenient and efficient but parts were reimplemented in order to remove some constraints. Token memory management is largely the same. All the other large memory structures, like those related to nesting, the save stack, input levels, the hash table and table of equivalents, etc. now all start out small and are enlarged when needed, where maxima are controlled in the usual way. In principle the initial memory footprint is smaller while at the same time we can go real large. Because we have wide memory words some data (arrays) used for housekeeping could be reorganized a bit.
- The specifier `plus 1 fillll` does not generate an error. The extra 'l' is simply typeset.
- The upper limit to `\endlinechar` and `\newlinechar` is 127.
- Because the backend is not built-in, the magnification (`\mag`) primitive is gone. A `\shipout` command just discards the content of the given box. The write related primitives have to be implemented in the used macro package using Lua. None of the pdfTEX derived primitives is present.
- Because there is no font loader, a Lua variant is free to either support or not the Omega `ofm` file format. As there are hardly any such fonts it probably makes no sense. There is plenty of control over the way glyphs get treated and scaling of fonts and glyphs is also more dynamic.
- There is more control over some (formerly hard-coded) math properties. In fact, there is a whole extra bit of math related code because we need to deal with OpenType fonts. The math processing has been adapted to the new (dynamic) font and glyph scaling features.
- The `\outer` and `\long` prefixed are silently ignored. It is permitted to use `\par` in math.
- The lack of a backend means that some primitives related to it are not implemented. This is no big deal because it is possible to use the scanner library to implement them as needed, which depends on the macro package and backend.
- The math style related primitives can use numbers as well as symbolic names. There is some more (control over) math anyway, which is a side effect of supporting OpenType math.

3.1.3 Changes from ε-TEX 2.2

Being the de-facto standard extension of course we provide the ε-TEX features, but with a few small adaptations.

- The ε-TEX functionality is always present and enabled so the prepended asterisk or `-etex` switch for iniTEX is not needed.
- The TEXXeT extension is not present, so the primitives `\TeXXeTstate`, `\beginR`, `\beginL`, `\endR` and `\endL` are missing. Instead we used the Omega/Aleph approach to directionality as starting point, albeit it has been changed quite a bit, so that we're probably not that far from TEXXeT.
- Some of the tracing information that is output by ε-TEX's `\tracingassigns` and `\tracingrestores` is not there. Also keep in mind that tracing doesn't involve what Lua does.
- Register management in LuaMetaTEX uses the Omega/Aleph model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat & sparse model from ε-TEX.
- Because we have more nodes, conditionals, etc. the ε-TEX status related variables are adapted to LuaMetaTEX: we use different 'constants', but that should be no problem because any sane macro package uses abstraction.
- The `\scantokens` primitive is now using the same mechanism as Lua print-to-TEX uses, which simplifies the code. There is a little performance hit but it will not be noticed in ConTEXt, because we never use this primitive.
- Because we don't use change files on top of original TEX, the integration of ε-TEX functionality is bit more natural, code wise.
- The `\readline` primitive has to be implemented in Lua. This is a side effect of delegating all file io.
- Most of the code is rewritten but the original primitives are still tagged as coming from ε-TEX.

3.1.4 Changes from pdfTEX 1.40

Because we want to produce pdf the most natural starting point was the popular pdfTEX program. We inherit the stable features, dropped most of the experimental code and promoted some functionality to core LuaTEX functionality which in turn triggered renaming primitives. However, as the backend was dropped, not that much from pdfTEX is present any more. Basically all we now inherit from pdfTEX is expansion and protrusion but even that has been adapted. So don't expect LuaMetaTEX to be compatible.

- The experimental primitives `\ifabsnum` and `\ifabsdim` have been promoted to core primitives.
- The primitives `\ifincsname`, `\expanded` and `\quitvmode` have become core primitives.
- As the hz (expansion) and protrusion mechanism are part of the core the related primitives `\lpcode`, `\rpcode`, `\efcode`, `\leftmarginkern`, `\rightmarginkern` are promoted to core primitives. The two commands `\protrudechars` and `\adjustspacing` control these processes.
- In LuaMetaTEX three extra primitives can be used to overload the font specific settings: `\adjustspacingstep` (max: 100), `\adjustspacingstretch` (max: 1000) and `\adjustspacingshrink` (max: 500).
- The hz optimization code has been partially redone so that we no longer need to create extra font instances. The front- and backend have been decoupled and the glyph and kern nodes carry the used values. In LuaTEX that made a more efficient generation of pdf code possible. It also resulted in much cleaner code. The backend code is gone, but of course the information

is still carried around.

- When `\adjustspacing` has value 2, hz optimization will be applied to glyphs and kerns. When the value is 3, only glyphs will be treated. A value smaller than 2 disables this feature.
- When `\protrudechars` has a value larger than zero characters at the edge of a line can be made to hang out. A value of 2 will take the protrusion into account when breaking a paragraph into lines. A value of 3 will try to deal with right-to-left rendering; this is a still experimental feature.
- The pixel multiplier dimension `\pxdimen` has be inherited as core primitive.
- The primitive `\tracingfonts` is now a core primitive but doesn't relate to the backend.

3.1.5 Changes from Aleph RC4

In LuaTEX we took the 32 bit aspects and much of the directional mechanisms and merged it into the pdfTEX code base as starting point for further development. Then we simplified directionality, fixed it and opened it up. In LuaMetaTEX not that much of the later is left. We only have two horizontal directions. Instead of vertical directions we introduce an orientation model bound to boxes.

The already reduced-to-four set of directions now only has two members: left-to-right and right-to-left. They don't do much as it is the backend that has to deal with them. When paragraphs are constructed a change in horizontal direction is irrelevant for calculating the dimensions. So, basically most that we do is registering state and passing that on till the backend can do something with it.

Here is a summary of inherited functionality:

- The ^^ notation has been extended: after ^^^^ four hexadecimal characters are expected and after ^^^^^^ six hexadecimal characters have to be given. The original TEX interpretation is still valid for the ^^ case but the four and six variants do no backtracking, i.e. when they are not followed by the right number of hexadecimal digits they issue an error message. Because ^^^ is a normal TEX case, we don't support the odd number of ^^^^^ either.
- Glues *immediately after* direction change commands are not legal breakpoints. There is a bit more sanity testing for the direction state. This can be configured.
- The placement of math formula numbers is direction aware and adapts accordingly. Boxes carry directional information but rules don't.
- There are no direction related primitives for page and body directions. The paragraph, text and math directions are specified using primitives that take a number. The three letter codes are dropped.

3.1.6 Changes from standard web2c

The LuaMetaTEX codebase is not dependent on the web2c framework. The interaction with the file system and tds is up to Lua. There still might be traces but eventually the code base should be lean and mean. The MetaPost library is coded in cweb and in order to be independent from related tools, conversion to C is done with a Lua script ran by, surprise, LuaMetaTEX.

3.2 Implementation notes

3.2.1 Memory allocation

The single internal memory heap that traditional TEX used for tokens and nodes is split into two separate arrays. Each of these will grow dynamically when needed. Internally a token or node is an index into these arrays. This permits for an efficient implementation and is also responsible for the performance of the core. All other data structures are mostly the same but managed dynamically too. Because we operate in a 64 bit world, the parallel table of equivalents needed for managing levels, is gone. Anyhow, the original documentation in TEX The Program mostly applies!

3.2.2 Sparse arrays

The `\mathcode`, `\delcode`, `\catcode`, `\sfcode`, `\lccode` and `\uccode` (and the new `\hjcode`) tables are now sparse arrays that are implemented in C. They are no longer part of the TEX 'equivalence table' and because each had 1.1 million entries with a few memory words each, this makes a major difference in memory usage. Performance is not really hurt by this.

The `\catcode`, `\sfcode`, `\lccode`, `\uccode` and `\hjcode` assignments don't show up when using the ε-TEX tracing routines `\tracingassigns` and `\tracingrestores` but we don't see that as a real limitation. It also saves a lot of clutter.

The glyph ids within a font are also managed by means of a sparse array as glyph ids can go up to index $2^{21} - 1$ but these are never accessed directly so again users will not notice this.

3.2.3 Simple single-character csnames

Single-character commands are no longer treated specially in the internals, they are stored in the hash just like the multiletter control sequences. This is a side effect of going Unicode and utf. Where using 256 slots in an array add no burden supporting the whole Unicode range is a waste of space. Therefore, also active characters are internally implemented as a special type of multi-letter control sequences that uses a prefix that is otherwise impossible to obtain.

The code that displays control sequences explicitly checks if the length is one when it has to decide whether or not to add a trailing space.

3.2.4 Binary file reading

All input now goes via Lua: files loaded with `\input` as well as files that are opened with `\openin`. Actually the later has to be implemented in terms of macros and Lua calls. This also means that compared to LuaTEX the internal handling of input has been changed but users won't notice that.

Setting a callback is expected now. Although reading input natively using `getc` calls is more efficient, we now fetch lines from Lua, put them in a buffer and then pick successive bytes (keep in mind that we read utf) from that. The performance is quite ok, also because Lua is fast, todays

operating systems cache, and storage media have become very fast. Also, TEX is spending more time messing around with what it has input than actually reading input.

3.2.5 Tabs and spaces

We conform to the way other TEX engines handle trailing tabs and spaces. For decades trailing tabs and spaces (before a newline) were removed from the input but this behaviour was changed in September 2017 to only handle spaces. We are aware that this can introduce compatibility issues in existing workflows but because we don't want too many differences with upstream TEXLive we just follow up on that patch (which is a functional one and not really a fix). It is up to macro packages maintainers to deal with possible compatibility issues and in LuaMetaTEX they can do so via the callbacks that deal with reading from files.

The previous behaviour was a known side effect and (as that kind of input normally comes from generated sources) it was normally dealt with by adding a comment token to the line in case the spaces and/or tabs were intentional and to be kept. We are aware of the fact that this contradicts some of our other choices but consistency with other engines. We still stick to our view that at the log level we can (and might be) more incompatible. We already expose some more details anyway.

3.2.6 Logging

When detailed logging is enabled more detail is output with respect to what nodes are involved. This is a side effect of the core nodes having more detailed subtype information. The benefit of more detail wins from any wish to be byte compatible in the logging. One can always write additional logging in Lua.

The information that goes into the log file can be different from LuaTEX, and might even differ a bit more in the future. The main reason is that inside the engine we have more granularity, which for instance means that we output subtype and attribute related information when nodes are printed. Of course we could have offered a compatibility mode but it serves no purpose. Over time there have been many subtle changes to control logs in the TEX ecosystems so another one is bearable.

In a similar fashion, there is a bit different behaviour when TEX expects input, which in turn is a side effect of removing the interception of * and & which made for cleaner code (quite a bit had accumulated as side effect of continuous adaptations in the TEX ecosystems). There was already code that was never executed, simply as side effect of the way LuaTEX initializes itself (one needs to enable classes of primitives for instance). Keep in mind that over time system dependencies have been handles with TEX change files, the web2c infrastructure, kpse features, compilation variables and flags, etc. In LuaMetaTEX we try to minimize all that.

When it became unavoidable that we output more detail, it also became clear that it made no sense to stay log and trace compatible. Some is controlled by parameters in order to stay close the original, but ConTEXt is configured such that we benefit from the new possibilities. Examples are that in addition to `\meaning` we have `\meaningfull` that also exposes macro properties, and `\meaningless` that only exposes the body. The `\untraced` prefix will suppress some in the log, and we set `\tracinglevels` to 3 in order to get details about the input and grouping level. When there's less shown than expected keep in mind that LuaMetaTEX has a somewhat optimized

saving and restoring of meanings so less can happen which is reflected in tracing. When node lists are serialized (as with `\showbox`) some nodes, like discretionaries report more detail. The compact serializer, used for instance to signal overfull boxes, also shows a bit more detail with respect to non-content nodes. I math more is shown if only because we have more control and additional mechanisms.

3.2.7 Parsing

Token parsers have been upgraded for the sake of Lua, `\csname` handling has been extended, macro definitions can be more flexible so there code was adapted, more conditionals also brought some changes. But we build upon the (reorganized) TEX foundation so the basics can definitely be recognized.

Because of interfacing in Lua the internal token and node organization has been normalized (read: we cannot cheat because all is kind of visible). On the one hand this can come with a performance penalty but that is more than compensated by extensions, optimized parsers and such. Still the fact that we are utf based (32 bit) makes the machinery slower than the 8 bit original. The reworked LuaMetaTEX engine is substantially faster than the LuaTEX predecessor.

The handling of conditionals has been adapted so that we can have flatter branches (`\orelse` cum suis). This again has some consequences for parsing. Because parsing alignments is rather interwoven in general parsing and expansion the handling of related primitives has been slightly adapted (also for the sake of Lua interfacing) and dealing with `\noalign` situations is a bit more convenient.

This are just a few of the adaptations and most of this happened stepwise with testing in the ConTEXt code base. It will be clear that LuaMetaTEX is a quite different extension to the original. You're warned.

4 Using LuaMetaTEX

4.1 Initialization

4.1.1 LuaMetaTEX as a Lua interpreter

Although LuaMetaTEX is primarily meant as a TEX engine, it can also serve as a stand alone Lua interpreter. There are two ways to make LuaMetaTEX behave like a standalone Lua interpreter. The first method uses the command line option `--luaonly` followed by a filename. The second is more automatic: if the only non-option argument (file) on the commandline has the extension `lmt` or `lua`. The `luc` extension has been dropped because bytecode compiled files are not portable and one can always load indirect. The `lmt` suffix is more ConTEXt specific and makes it possible to have files for LuaTEX and LuaMetaTEX alongside.

In this mode, it will set Lua's `arg[0]` to the found script name, pushing preceding options in negative values and the rest of the command line in the positive values, just like the Lua interpreter does.

LuaMetaTEX will exit immediately after executing the specified Lua script and is, in effect, a somewhat bulky stand alone Lua interpreter with a bunch of extra preloaded libraries. But we really want to keep the binary small, if possible below the 3MB which is okay for a script engine.

When no argument is given, LuaMetaTEX will look for a Lua file with the same name as the binary and run that one when present. This makes it possible to use the engine as a stub. For instance, in ConTEXt a symlink from `mtxrun` to type luametatex will run the `mtxrun.lmt` or `mtxrun.lua` script when present in the same path as the binary itself. As mentioned before first checking for (ConTEXt) `lmt` files permits different files for different engines in the same path.

4.1.2 Other commandline processing

When the LuaMetaTEX executable starts, it looks for the `--lua` command line option. If there is no `--lua` option, the command line is interpreted in a similar fashion as the other TEX engines. All options are accepted but only some are understood by LuaMetaTEX itself:

COMMANDLINE ARGUMENT	EXPLANATION
`--credits`	display credits and exit
`--fmt=FORMAT`	load the format file `FORMAT`
`--help`	display help and exit
`--ini`	be `iniluatex`, for dumping formats
`--jobname=STRING`	set the job name to `STRING`
`--lua=FILE`	load and execute a Lua initialization script
`--version`	display version and exit

There are less options than with LuaTEX, because one has to deal with them in Lua anyway. There are no options to enter a safer mode or control executing programs. This can easily be achieved with a startup Lua script.

Next the initialization script is loaded and executed. From within the script, the entire command line is available in the Lua table `arg`, beginning with `arg[0]`, containing the name of the executable. As consequence warnings about unrecognized options are suppressed.

Command line processing happens very early on. So early, in fact, that none of TEX's initializations have taken place yet. The Lua libraries that don't deal with TEX are initialized rather soon so you have these available.

LuaMetaTEX allows some of the command line options to be overridden by reading values from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document for more details on which ones exactly).

The value to use for `\jobname` is decided as follows:

- If `--jobname` is given on the command line, its argument will be the value for `\jobname`, without any changes. The argument will not be used for actual input so it need not exist. The `--jobname` switch only controls the `\jobname` setting.
- Otherwise, `\jobname` will be the name of the first file that is read from the file system, with any path components and the last extension (the part following the last `.`) stripped off.
- There is an exception to the previous point: if the command line goes into interactive mode (by starting with a command) and there are no files input via `\everyjob` either, then the `\jobname` is set to `texput` as a last resort.

So let's summarize this. The handling of what is called jobname is a bit complex. There can be explicit names set on the command line but when not set they can be taken from the `texconfig` table.

startup filename	`--lua`	a Lua file	
startup jobname	`--jobname`	a TEX tex	`texconfig.jobname`
startup dumpname	`--fmt`	a format file	`texconfig.formatname`

These names are initialized according to `--luaonly` or the first filename seen in the list of options. Special treatment of & and * as well as interactive startup is gone but we still enter TEX via an forced `\input` into the input buffer.[1]

When we are in TEX mode at some point the engine needs a filename, for instance for opening a log file. At that moment the set jobname becomes the internal one and when it has not been set which internalized to jobname but when not set becomes `texput`. When you see a `texput.log` file someplace on your system it normally indicates a bad run.

When running on MS Windows the command line, filenames, environment variable access etc. internally uses the current code page but to the user is exposed as utf8. Normally users won't notice this.

There is an extra options `--permitloadlib` that needs to be given when you load external libraries via Lua. Although you could manage this via Lua itself in a startup script, the reason for having this as option is the wish for security (at some point that became a demand for LuaTEX), so this might give an extra feeling of protection.

[1] This might change at some point into an explicit loading triggered via Lua.

4.2 Lua behaviour

4.2.1 The Lua version

We currently use Lua 5.4 and will follow developments of the language but normally with some delay. Therefore the user needs to keep an eye on (subtle) differences in successive versions of the language. Here is an example of one aspect.

Luas `tostring` function (and `string.format`) may return values in scientific notation, thereby confusing the TEX end of things when it is used as the right-hand side of an assignment to a `\dimen` or `\count`. The output of these serializers also depend on the Lua version, so in Lua 5.3 you can get different output than from 5.2. It is best not to depend the automatic cast from string to number and vise versa as this can change in future versions.

4.2.2 Locales

In stock Lua, many things depend on the current locale. In LuaMetaTEX, we can't do that, because it makes documents unportable. While LuaMetaTEX is running if forces the following locale settings:

```
LC_CTYPE=C
LC_COLLATE=C
LC_NUMERIC=C
```

There is no way to change that as it would interfere badly with the often language specific conversions needed at the TEX end.

4.3 Lua modules

Of course the regular Lua modules are present. In addition we provide the `lpeg` library by Roberto Ierusalimschy, This library is not Unicode-aware, but interprets strings on a byte-per-byte basis. This mainly means that `lpeg.S` cannot be used with utf8 characters that need more than one byte, and thus `lpeg.S` will look for one of those two bytes when matching, not the combination of the two. The same is true for `lpeg.R`, although the latter will display an error message if used with multibyte characters. Therefore `lpeg.R('aä')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, `ä` is two 'characters' (bytes), so `aä` totals three. In practice this is no real issue and with some care you can deal with Unicode just fine.

There are some more libraries present. These are discussed on a later chapter. For instance we embed `luasocket` but contrary to LuaTEX don't embed the related Lua code. The `luafilesystem` module has been replaced by a more efficient one that also deals with the MS Windows file and environment properties better (Unicode support in MS Windows dates from before utf8 became dominant so we need to deal with wide Unicode16).

There are more extensive math libraries and there are libraries that deal with encryption and compression. There are also some optional libraries that we do interface but that are loaded on

demand. The interfaces are as minimal as can be because we so much in Lua, which also means that one can tune behaviour to usage better.

4.4 Testing

For development reasons you can influence the used startup date and time. By setting the `start_time` variable in the `texconfig` table; as with other variables we use the internal name there. When Universal Time is needed, set the entry `use_utc_time` in the `texconfig` table.

In ConTEXt we provide the command line argument `--nodates` that does a bit more than disabling dates; it avoids time dependent information in the output file for instance.

5 Basic TEX enhancements

5.1 Introduction

5.1.1 Primitive behaviour

From day one, LuaTEX has offered extra features compared to the superset of pdfTEX, which includes ε-TEX, and Aleph. This has not been limited to the possibility to execute Lua code via `\directlua`, but LuaTEX also adds functionality via new TEX-side primitives or extensions to existing ones. The same is true for LuaMetaTEX. Some primitives have `luatex` in their name and there will be no `luametatex` variants. This is because we consider LuaMetaTEX to be LuaTEX2$^+$.

Contrary to the LuaTEX engine LuaMetaTEX enables all its primitives. You can clone (a selection of) primitives with a different prefix, like this:

```
\directlua { tex.enableprimitives('normal',tex.extraprimitives()) }
```

The `extraprimitives` function returns the whole list or a subset, specified by one or more keywords `core`, `tex`, `etex` or `luatex`. When you clone all primitives you can also do this:

```
\directlua { tex.enableprimitives('normal',true) }
```

But be aware that the curly braces may not have the proper `\catcode` assigned to them at this early time (giving a 'Missing number' error), so it may be needed to put these assignments before the above line:

```
\catcode `\{ = 1
\catcode `\} = 2
```

More fine-grained primitives control is possible and you can look up the details in section 12.3.15. There are only three kinds of primitives: `tex`, `etex` and `luatex` but a future version might drop this and no longer make that distinction as it no longer serves a purpose apart from the fact that it reveals some history.

5.1.2 Rationale

One can argue that TEX should stay as it is but over decades usage of this program has evolved and resulted in large macro packages that often need to rely on what the TEX books calls 'dirty tricks'. When you look deep down in the code of ConTEXt MkII, MkIV and MkXL (aka LMTX) you will see plenty of differences but quite a bit of the functionality in the most recent versions is also available in MkII. Of course more has been added over time, and some mechanisms could be made more efficient and reliable but plenty was possible.

So, when you see something done in ConTEXt LMTX using new LuaMetaTEX primitives you can assume that somehow the same is done in ConTEXt MkIV. We don't really need LuaMetaTEX instead of LuaTEX. Among the main reasons for still going for this new engine are:

- some new primitives make for less tracing and tracing has become rather verbose over years (just try `\tracingall`); examples are the new macro argument handling and some new hooks
- some new primitives permits more efficient coding and have a positive impact on performance (this sort of compensates a performance hit due to delegating work to Lua)
- other primitives are there because they make the code look better; good examples are the extensions to conditionals; they remove the necessity for all kind of (somewhat unnatural) middle layers; take local control as example
- a few primitives make complex and demanding mechanism a bit easier to grasp and explain; think of alignments, inserts and marks
- more access from the Lua end to TEX internals: a few more callbacks, more options, more robust interfaces, etc

I realize that new primitives also can make some TEX code look less threatening to new users. It removes a bit of hackery and limits the level of guru that comes with showing off the mastery of expansion and lookahead. So be it. I wonder if those objecting to some of the extensions (with the argument that they are not needed, and ConTEXt MkIV is proof of that) can resist using them. I admit that it sometimes hurt to throw away good working but cumbersome code that took a while to evolve, but I also admit that I favor long distance traveling by bike or car over riding horseback.

It took a few years for LuaMetaTEX to evolve to what it is now and most extensions are not there "because they were easy" or "could be done". If that were the case, there would be plenty more. In many aspects it has been a balancing act and much also relates to looking at the ConTEXt source code (TEX as well as Lua) and wondering why it looks that way. It is also driven by the fact that I want to be able to explain to users why things are done in a certain way. In fact, I want users to be able to look at the code and understand it (apart from maybe a few real dirty low level helpers that are also dirty because of performance reasons). Just take this into account when reading on.

And yes, there are still a few possibilities I want to explore ...

5.1.3 Version information

5.1.3.1 `\luatexbanner`, `\luatexversion` and `\luatexrevision`

There are three primitives to test the version of LuaTEX (and LuaMetaTEX):

PRIMITIVE	VALUE	EXPLANATION
`\luatexbanner`	This is LuaMetaTeX, Version 2.09.24	the banner reported on the console
`\luatexversion`	209	major and minor number combined
`\luatexrevision`	24	the revision number

A version is defined as follows:

- The major version is the integer result of `\luatexversion` divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with `\the` or `\number` depending on the context.
- The minor version is a number running from 0 upto 99.

- The revision is reported by `\luatexrevision`. Contrary to other engines in LuaMetaTEX is also a number so one needs to prefix it with `\the` or `\number`.[2]
- The full version number consists of the major version (`X`), minor version (`YY`) and revision (`ZZ`), separated by dots, so `X.YY.ZZ`.

The LuaMetaTEX version number starts at 2 in order to prevent a clash with LuaTEX, and the version commands are the same. This is a way to indicate that these projects are related.

The `status` library also provides some information including what we get with the three mentioned primitives:

FIELD	VALUE
`filename`	E:/context/manuals/mkiv/external/luametatex/luametatex-enhancements.tex
`banner`	This is LuaMetaTeX, Version 2.09.24
`luatex_engine`	luametatex
`luatex_version`	209
`luatex_revision`	24
`luatex_verbose`	2.09.24
`copyright`	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
`development_id`	20210914
`format_id`	596
`used_compiler`	gcc

5.2 Unicode text support

5.2.1 Extended ranges

Text input and output is now considered to be Unicode text, so input characters can use the full range of Unicode ($2^{20} + 2^{16} - 1 = 0x10FFFF$). Later chapters will talk of characters and glyphs. Although these are not interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside the engine there is no clear separation between the two concepts. Because the subtype of a glyph node can be changed in Lua it is up to the user. Subtypes larger than 255 indicate that font processing has happened.

A few primitives are affected by this, all in a similar fashion: each of them has to accommodate for a larger range of acceptable numbers. For instance, `\char` now accepts values between 0 and 1,114,111. This should not be a problem for well-behaved input files, but it could create incompatibilities for input that would have generated an error when processed by older TEX-based engines. The affected commands with an altered initial (left of the equal sign) or secondary (right of the equal sign) value are: `\char`, `\lccode`, `\uccode`, `\hjcode`, `\catcode`, `\sfcode`, `\efcode`, `\lpcode`, `\rpcode`, `\chardef`.

[2] In the past it always was good to prefix the revision with `\number` anyway, just to play safe, although there have for instance been times that pdfTEX had funny revision indicators that at some point ended up as letters due to the internal conversions.

As far as the core engine is concerned, all input and output to text files is utf-8 encoded. Input files can be pre-processed using the `reader` callback. This will be explained in section ??. Normalization of the Unicode input is on purpose not built-in and can be handled by a macro package during callback processing. We have made some practical choices and the user has to live with those.

Output in byte-sized chunks can be achieved by using characters just outside of the valid Unicode range, starting at the value 1,114,112 (0x110000). When the time comes to print a character c >= 1,114,112, LuaTEX will actually print the single byte corresponding to c minus 1,114,112.

Contrary to other TEX engines, the output to the terminal is as-is so there is no escaping with ^^. We operate in a utf universe. Because we operate in a C universum, zero characters are special but because we also live in a Unicode galaxy that is no real problem.

5.2.2 `\Uchar`

The expandable command `\Uchar` reads a number between 0 and 1,114,111 and expands to the associated Unicode character.

5.2.3 Extended tables

All traditional TEX and ε-TEX registers can be 16-bit numbers. The affected commands are:

`\count`	`\countdef`	`\box`	`\wd`
`\dimen`	`\dimendef`	`\unhbox`	`\ht`
`\skip`	`\skipdef`	`\unvbox`	`\dp`
`\muskip`	`\muskipdef`	`\copy`	`\setbox`
`\marks`	`\toksdef`	`\unhcopy`	`\vsplit`
`\toks`	`\insert`	`\unvcopy`	

Fonts are loaded via Lua and a minimal amount of information is kept at the TEX end. Sharing resources is up to the loaders. The engine doesn't really care about what a character (or glyph) number represents (a Unicode or index) as it only is interested in dimensions.

In TEX the number of registers is 256 and ε-TEX bumped that to 32K. One reason for a fixed number is that these registers are fast ways to store data and therefore are part of the main lookup table (used for data and pointers to data as well as save and restore housekeeping). In LuaTEX the number was bumped to 64K but one can argue that less would also do. In order to keep the default memory footprint reasonable, in LuaMetaTEX the number of languages, fonts and marks is limited. The size of some tables can be limited by configuration settings, so they can start out small and grow till configured maximum which is smaller than the absolute maximum. The following table shows all kind of defaults as reported by `status.getconstants()`.

`awful_bad`	1073741823
`decent_criterium`	12
`default_catcode_table`	-1
`default_deadcycles`	25

default_eqno_gap_step	1000
default_hangafter	1
default_output_box	255
default_pre_display_gap	2000
default_rule	26214
default_space_factor	1000
default_tolerance	10000
deplorable	100000
eject_penalty	-10000
ignore_depth	-65536000
infinite_bad	10000
infinite_penalty	10000
infinity	2147483647
large_width_excess	7230584
loose_criterium	99
max_bytecode_index	65535
max_cardinal	4294967295
max_category_code	15
max_char_code	15
max_character_code	1114111
max_data_value	2097151
max_dimen	1073741823
max_function_reference	2097151
max_half_value	32767
max_halfword	1073741823
max_integer	2147483647
max_mark_index	9999
max_math_class_code	7
max_math_family_index	255
max_n_of_bytecodes	65536
max_n_of_catcode_tables	256
max_n_of_fonts	100000
max_n_of_languages	10000
max_n_of_marks	10000
max_n_of_math_families	256
max_n_of_registers	65536
max_newline_character	127
max_quarterword	65535
max_register_index	65535
max_size_of_word	1024
max_space_factor	32767
min_cardinal	0
min_data_value	0
min_dimen	-1073741823
min_halfword	-1073741823
min_infinity	-2147483647

min_integer	-2147483647
min_quarterword	0
min_space_factor	0
no_catcode_table	-2
null	0
null_flag	-1073741824
null_font	0
one_bp	65781
preset_rule_thickness	1073741824
running_rule	-1073741824
small_stretchability	1663497
tex_eqtb_size	590042
tex_hash_prime	131041
tex_hash_size	131072
two	131072
unity	65536
unused_attribute_value	-2147483647
unused_script_value	0
unused_state_value	0
zero_glue	0

Because we have additional ways to store integers, dimensions and glue, we might actually decide to decrease the maximum of the registers: if 64K is not enough, and you work around it, then likely 32K might do as well. Also, we have Lua to store massive amounts of data. One can argue that saving some 1.5MB memory (when we go halfway) is not worth the effort in a time when you have to close a browser in order to free the gigabytes it consumes, but there is no reason not to be lean and mean: a more conservative approach to start with creates headroom for going wild later.

5.3 Attributes

5.3.1 Nodes

When TEX reads input it will interpret the stream according to the properties of the characters. Some signal a macro name and trigger expansion, others open and close groups, trigger math mode, etc. What's left over becomes the typeset text. Internally we get a linked list of nodes. Characters become `glyph` nodes that have for instance a `font` and `char` property and `\kern 10pt` becomes a `kern` node with a `width` property. Spaces are alien to TEX as they are turned into `glue` nodes. So, a simple paragraph is mostly a mix of sequences of `glyph` nodes (words) and `glue` nodes (spaces). A node can have a subtype so that it can be recognized as for instance a space related glue.

The sequences of characters at some point are extended with `disc` nodes that relate to hyphenation. After that font logic can be applied and we get a list where some characters can be replaced, for instance multiple characters can become one ligature, and font kerns can be injected. This is driven by the font properties.

Boxes (like \hbox and \vbox) become hlist or vlist nodes with width, height, depth and shift properties and a pointer list to its actual content. Boxes can be constructed explicitly or can be the result of subprocesses. For instance, when lines are broken into paragraphs, the lines are a linked list of hlist nodes, possibly with glue and penalties in between.

Internally nodes have a number. This number is actually an index in the memory used to store nodes.

So, to summarize: all that you enter as content eventually becomes a node, often as part of a (nested) list structure. They have a relative small memory footprint and carry only the minimal amount of information needed. In traditional TEX a character node only held the font and slot number, in LuaTEX we also store some language related information, the expansion factor, etc. Now that we have access to these nodes from Lua it makes sense to be able to carry more information with a node and this is where attributes kick in.

It is important to keep in mind that there are situations where nodes get created in the current context. For instance, when TEX builds a paragraph or page or constructs math formulas, it does add nodes and giving these the current attributes makes no sense and can even give weird side effects. In these cases, the attributes are inherited from neighbouring nodes.

5.3.2 Attribute registers

Attributes are a completely new concept in LuaTEX. Syntactically, they behave a lot like counters: attributes obey TEX's nesting stack and can be used after \the etc. just like the normal \count registers.

\attribute ⟨16-bit number⟩ ⟨optional equals⟩ ⟨32-bit number⟩
\attributedef ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩

Conceptually, an attribute is either 'set' or 'unset'. Unset attributes have a special negative value to indicate that they are unset, that value is the lowest legal value: -"7FFFFFFF in hexadecimal, a.k.a. −2147483647 in decimal. It follows that the value -"7FFFFFFF cannot be used as a legal attribute value, but you *can* assign -"7FFFFFFF to 'unset' an attribute. All attributes start out in this 'unset' state in iniTEX.

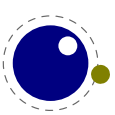
Attributes can be used as extra counter values, but their usefulness comes mostly from the fact that the numbers and values of all 'set' attributes are attached to all nodes created in their scope. These can then be queried from any Lua code that deals with node processing. Further information about how to use attributes for node list processing from Lua is given in chapter 10.

Attributes are stored in a sorted (sparse) linked list that are shared when possible. This permits efficient testing and updating. You can define many thousands of attributes but normally such a large number makes no sense and is also not that efficient because each node carries a (possibly shared) link to a list of currently set attributes. But they are a convenient extension and one of the first extensions we implemented in LuaTEX.

In LuaMetaTEX we try to minimize the memory footprint and creation of these attribute lists more aggressive sharing them. This feature is still somewhat experimental.

5.3.3 Box attributes

Nodes typically receive the list of attributes that is in effect when they are created. This moment



can be quite asynchronous. For example: in paragraph building, the individual line boxes are created after the \par command has been processed, so they will receive the list of attributes that is in effect then, not the attributes that were in effect in, say, the first or third line of the paragraph.

Similar situations happen in LuaTEX regularly. A few of the more obvious problematic cases are dealt with: the attributes for nodes that are created during hyphenation, kerning and ligaturing borrow their attributes from their surrounding glyphs, and it is possible to influence box attributes directly.

When you assemble a box in a register, the attributes of the nodes contained in the box are unchanged when such a box is placed, unboxed, or copied. In this respect attributes act the same as characters that have been converted to references to glyphs in fonts. For instance, when you use attributes to implement color support, each node carries information about its eventual color. In that case, unless you implement mechanisms that deal with it, applying a color to already boxed material will have no effect. Keep in mind that this incompatibility is mostly due to the fact that separate specials and literals are a more unnatural approach to colors than attributes.

It is possible to fine-tune the list of attributes that are applied to a hbox, vbox or vtop by the use of the keyword attr. The attr keyword(s) should come before a to or spread, if that is also specified. An example is:

```
\attribute997=123
\attribute998=456
\setbox0=\hbox {Hello}
\setbox2=\hbox attr 999 = 789 attr 998 = -"7FFFFFFF{Hello}
```

Box 0 now has attributes 997 and 998 set while box 2 has attributes 997 and 999 set while the nodes inside that box will all have attributes 997 and 998 set. Assigning the maximum negative value causes an attribute to be ignored.

To give you an idea of what this means at the Lua end, take the following code:

```
for b=0,2,2 do
  for a=997, 999 do
    tex.sprint("box ", b, " : attr ",a," : ",tostring(tex.box[b]     [a]))
    tex.sprint("\\quad\\quad")
    tex.sprint("list ",b, " : attr ",a," : ",tostring(tex.box[b].list[a]))
    tex.sprint("\\par")
  end
end
```

Later we will see that you can access properties of a node. The boxes here are so called hlist nodes that have a field list that points to the content. Because the attributes are a list themselves you can access them by indexing the node (here we do that with [a]). Running this snippet gives:

```
box 0 : attr 997 : 123   list 0 : attr 997 : 123
box 0 : attr 998 : 456   list 0 : attr 998 : 456
box 0 : attr 999 : nil   list 0 : attr 999 : nil
```

```
box 2 : attr 997 : 123   list 2 : attr 997 : 123
box 2 : attr 998 : nil   list 2 : attr 998 : 456
box 2 : attr 999 : 789   list 2 : attr 999 : nil
```

Because some values are not set we need to apply the `tostring` function here so that we get the word `nil`.

A special kind of box is `\vcenter`. This one also can have attributes. When one or more are set these plus the currently set attributes are bound to the resulting box. In regular TEX these centered boxes are only permitted in math mode, but in LuaMetaTEX there is no error message and the box the height and depth are equally divided. Of course in text mode there is no math axis related offset applied.

It is possible to change or add to the attributes assigned to a box with `\boxattribute`:

```
\boxattribute 0 123 456
```

You can set attributes of the current paragraph specification node with `\parattribute`:

```
\parattribute 123 456
```

5.4 Lua related primitives

5.4.1 \directlua

In order to merge Lua code with TEX input, a few new primitives are needed. The primitive `\directlua` is used to execute Lua code immediately. The syntax is

`\directlua` ⟨general text⟩

The ⟨general text⟩ is expanded fully, and then fed into the Lua interpreter. After reading and expansion has been applied to the ⟨general text⟩, the resulting token list is converted to a string as if it was displayed using `\the\toks`. On the Lua side, each `\directlua` block is treated as a separate chunk. In such a chunk you can use the `local` directive to keep your variables from interfering with those used by the macro package.

The conversion to and from a token list means that you normally can not use Lua line comments (starting with `--`) within the argument. As there typically will be only one 'line' the first line comment will run on until the end of the input. You will either need to use TEX-style line comments (starting with %), or change the TEX category codes locally. Another possibility is to say:

```
\begingroup
\endlinechar=10
\directlua ...
\endgroup
```

Then Lua line comments can be used, since TEX does not replace line endings with spaces. Of course such an approach depends on the macro package that you use.

The `\directlua` command is expandable. Since it passes Lua code to the Lua interpreter its expansion from the TEX viewpoint is usually empty. However, there are some Lua functions that

produce material to be read by TEX, the so called print functions. The most simple use of these is `tex.print(<string> s)`. The characters of the string `s` will be placed on the TEX input buffer, that is, 'before TEX's eyes' to be read by TEX immediately. For example:

```
\count10=20
a\directlua{tex.print(tex.count[10]+5)}b
```

expands to

a25b

Here is another example:

```
$\pi = \directlua{tex.print(math.pi)}$
```

will result in

$\pi = 3.1415926535898$

Note that the expansion of `\directlua` is a sequence of characters, not of tokens, contrary to all TEX commands. So formally speaking its expansion is null, but it collects material in a new level on the input stack to be immediately read by TEX after the Lua call as finished. It is a bit like ε-TEX's `\scantokens`, which now uses the same mechanism. For a description of print functions look at section 12.3.13.

Because the ⟨general text⟩ is a chunk, the normal Lua error handling is triggered if there is a problem in the included code. The Lua error messages should be clear enough, but the contextual information is often suboptimal because it can come from deep down, and TEX has no knowledge about what you do in Lua. Often, you will only see the line number of the right brace at the end of the code.

While on the subject of errors: some of the things you can do inside Lua code can break up LuaMetaTEX pretty bad. If you are not careful while working with the node list interface, you may even end up with errors or even crashes from within the TEX portion of the executable.

5.4.2 `\luaescapestring`

This primitive converts a TEX token sequence so that it can be safely used as the contents of a Lua string: 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. The token sequence is fully expanded.

`\luaescapestring` ⟨general text⟩

Most often, this command is not actually the best way to deal with the differences between TEX and Lua. In very short bits of Lua code it is often not needed, and for longer stretches of Lua code it is easier to keep the code in a separate file and load it using Lua's `dofile`:

```
\directlua { dofile("mysetups.lua") }
```

5.4.3 \luafunction, \luafunctioncall and \luadef

The \directlua commands involves tokenization of its argument (after picking up an optional name or number specification). The tokenlist is then converted into a string and given to Lua to turn into a function that is called. The overhead is rather small but when you have millions of calls it can have some impact. For this reason there is a variant call available: \luafunction. This command is used as follows:

```
\directlua {
    local t = lua.get_functions_table()
    t[1] = function() tex.print("!") end
    t[2] = function() tex.print("?") end
}

\luafunction1
\luafunction2
```

Of course the functions can also be defined in a separate file. There is no limit on the number of functions apart from normal Lua limitations. Of course there is the limitation of no arguments but that would involve parsing and thereby give no gain. The function, when called in fact gets one argument, being the index, so in the following example the number 8 gets typeset.

```
\directlua {
    local t = lua.get_functions_table()
    t[8] = function(slot) tex.print(slot) end
}
```

The \luafunctioncall primitive does the same but is unexpandable, for instance in an \edef. In addition LuaTEX provides a definer:

```
                \luadef\MyFunctionA 1
          \global\luadef\MyFunctionB 2
\protected\global\luadef\MyFunctionC 3
```

You should really use these commands with care. Some references get stored in tokens and assume that the function is available when that token expands. On the other hand, as we have tested this functionality in relative complex situations normal usage should not give problems.

It makes sense to delegate the implementation of the primitives to Lua.

5.4.4 \luabytecode and \luabytecodecall

Analogue to the function callers discussed in the previous section we have byte code callers. Again the call variant is unexpandable.

```
\directlua {
    lua.bytecode[9998] = function(s)
        tex.sprint(s*token.scan_int())
```

```
    end
    lua.bytecode[5555] = function(s)
        tex.sprint(s*token.scan_dimen())
    end
}
```

This works with:

```
\luabytecode    9998 5  \luabytecode    5555 5sp
\luabytecodecall9998 5  \luabytecodecall5555 5sp
```

The variable `s` in the code is the number of the byte code register that can be used for diagnostic purposes. The advantage of bytecode registers over function calls is that they are stored in the format (but without upvalues).

It makes sense to delegate the implementation of the primitives to Lua.

5.5 Catcode tables

5.5.1 Catcodes

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single statement. You can have lots of different tables, but if you need a dozen you might wonder what you're doing. This subsystem is backward compatible: if you never use the following commands, your document will not notice any difference in behaviour compared to traditional TEX. The contents of each catcode table is independent from any other catcode table, and its contents is stored and retrieved from the format file.

5.5.2 `\catcodetable`

The primitive `\catcodetable` switches to a different catcode table. Such a table has to be previously created using one of the two primitives below, or it has to be zero. Table zero is initialized by iniTEX.

`\catcodetable` ⟨15-bit number⟩

5.5.3 `\initcatcodetable`

`\initcatcodetable` ⟨15-bit number⟩

The primitive `\initcatcodetable` creates a new table with catcodes identical to those defined by iniTEX. The new catcode table is allocated globally: it will not go away after the current group has ended. If the supplied number is identical to the currently active table, an error is raised. The initial values are:

CATCODE	CHARACTER	EQUIVALENT	CATEGORY
0	\		escape

5	^^M	return	car_ret
9	^^@	null	ignore
10	<space>	space	spacer
11	a – z		letter
11	A – Z		letter
12	everything else		other
14	%		comment
15	^^?	delete	invalid_char

5.5.4 \savecatcodetable

\savecatcodetable ⟨15-bit number⟩

\savecatcodetable copies the current set of catcodes to a new table with the requested number. The definitions in this new table are all treated as if they were made in the outermost level. Again, the new table is allocated globally: it will not go away after the current group has ended. If the supplied number is the currently active table, an error is raised.

5.6 Tokens, commands and strings

5.6.1 \scantextokens and \tokenized

The syntax of \scantextokens is identical to \scantokens. This primitive is a slightly adapted version of ε-TEX's \scantokens. The differences are:

- The last (and usually only) line does not have a \endlinechar appended.
- \scantextokens never raises an EOF error, and it does not execute \everyeof tokens.
- There are no '... while end of file ...' error tests executed. This allows the expansion to end on a different grouping level or while a conditional is still incomplete.

The implementation in LuaMetaTEX is different in the sense that it uses the same methods as printing from Lua to TEX does. Therefore, in addition to the two commands we also have this expandable command:

\tokenized ... \tokenized catcodetable ⟨number⟩ ...

The ε-TEX command \tracingscantokens has been dropped in the process as that was interwoven with the old code.

5.6.2 \toksapp, \tokspre, \etoksapp, \etokspre, \gtoksapp, \gtokspre, \xtoksapp, \xtokspre

Instead of:

```
\toks0\expandafter{\the\toks0 foo}
```

you can use:

```
\etoksapp0{foo}
```

The pre variants prepend instead of append, and the e variants expand the passed general text. The g and x variants are global.

5.6.3 \csstring, \begincsname and \lastnamedcs

These are somewhat special. The \csstring primitive is like \string but it omits the leading escape character. This can be somewhat more efficient than stripping it afterwards.

The \begincsname primitive is like \csname but doesn't create a relaxed equivalent when there is no such name. It is equivalent to

```
\ifcsname foo\endcsname
  \csname foo\endcsname
\fi
```

The advantage is that it saves a lookup (don't expect much speedup) but more important is that it avoids using the \if test. The \lastnamedcs is one that should be used with care. The above example could be written as:

```
\ifcsname foo\endcsname
  \lastnamedcs
\fi
```

This is slightly more efficient than constructing the string twice (deep down in LuaTEX this also involves some utf8 juggling), but probably more relevant is that it saves a few tokens and can make code a bit more readable.

5.6.4 \clearmarks, \flushmarks, \currentmarks

The \clearmarks primitive complements the ε-TEX mark primitives and clears a mark class completely, resetting all three connected mark texts to empty. It is an immediate command (no synchronization node is used).

\clearmarks ⟨16-bit number⟩

The \flushmarks variant is delayed but puts a (mark) node in the list as signal (we could have gone for a keyword to \marks instead).

\flushmarks ⟨16-bit number⟩

In addition to the three mark fetch commands, we also have access to the last set mark in the given class.

\currentmarks ⟨16-bit number⟩

Marks can be traced with \tracingmarks. When set to 1 the page builder shows the set values, and when set to a higher value details about collecting them are shown.

5.6.5 \alignmark, \aligntab, \aligncontent and \tabsize

The primitive \alignmark duplicates the functionality of # inside alignment preambles, while \aligntab duplicates the functionality of &. The \aligncontent primitive directly refers to an entry so that one does not get repeated.

Alignments can be traced with \tracingalignments. When set to 1 basics usage is shown, for instance of \noalign but more interesting is 2 or more: you then get the preambles reported.

The \halign (tested) and \valign (yet untested) primitives accept a few keywords in addition to to and spread:

KEYWORD	EXPLANATION
attr	set the given attribute to the given value
callback	trigger the alignment_filter callback
discard	discard zero \tabskip's
noskips	don't even process zero \tabskip's
reverse	reverse the final rows

In the preamble the \tabsize primitive can be used to set the width of a column. By doing so one can avoid using a box in the preamble which, combined with the sparse tabskip features, is a bit easier on memory when you produce tables that span hundreds of pages and have a dozen columns.

5.6.6 \letcharcode

This primitive can be used to assign a meaning to an active character, as in:

```
\def\foo{bar} \letcharcode123=\foo
```

This can be a bit nicer than using the uppercase tricks (using the property of \uppercase that it treats active characters special).

5.6.7 \lettonothing and \glettonothing

This primitive is equivalent to:

```
\protected\def\lettonothing#1{\def#1{}}
```

and although it might feel faster (only measurable with millions of calls) it's mostly there because it is easier on tracing (less clutter). An advantage over letting to an empty predefined macro is also that in tracing we keep seeing the name (relaxing would show the relax equivalent).

5.6.8 \glet

This primitive is similar to:

```
\protected\def\glet{\global\let}
```

but faster (only measurable with millions of calls) and probably more convenient (after all we also have \gdef).

5.6.9 \defcsname, \edefcsname, \edefcsname and \xdefcsname

Although we can implement these primitives easily using macros it makes sense, given the popularity of \csname to have these as primitives. It also saves some \expandafter usage and it looks a bit better in the source.

```
\def\gdefcsname foo\endcsname{oof}
```

5.6.10 \expanded

The \expanded primitive takes a token list and expands its content which can come in handy: it avoids a tricky mix of \expandafter and \noexpand. You can compare it with what happens inside the body of an \edef. The \immediateassignment and \immediateassigned commands are gone because we have the more powerful local control commands. They are a tad slower but this mechanism isn't used that much anyway. Inside an \edef you can use the \immediate prefix anyway, so if you really want these primitives back you can say:

```
\let\immediateassignment\immediate
\let\immediateassigned  \localcontrolled
```

5.6.11 \ignorepars

This primitive is like \ignorespaces but also skips paragraph ending commands (normally \par and empty lines).

5.6.12 \futureexpand, \futureexpandis, \futureexpandisap

These commands are used as:

```
\futureexpand\sometoken\whenfound\whennotfound
```

When there is no match and a space was gobbled a space will be put back. The is variant doesn't do that while the isap even skips \pars, These characters stand for ‘ignorespaces’ and ‘ignorespacesandpars’.

5.6.13 \aftergrouped

There is a new experimental feature that can inject multiple tokens to after the group ends. An example demonstrate its use:

```
{
    \aftergroup A \aftergroup B \aftergroup C
```

```
test 1 : }

{
    \aftergrouped{What comes next 1}
    \aftergrouped{What comes next 2}
    \aftergrouped{What comes next 3}
test 2 : }

{
    \aftergroup A \aftergrouped{What comes next 1}
    \aftergroup B \aftergrouped{What comes next 2}
    \aftergroup C \aftergrouped{What comes next 3}
test 3 : }

{
    \aftergrouped{What comes next 1} \aftergroup A
    \aftergrouped{What comes next 2} \aftergroup B
    \aftergrouped{What comes next 3} \aftergroup C
test 4 : }
```

This gives:

test 1 : ABC
test 2 : What comes next 1What comes next 2What comes next 3
test 3 : AWhat comes next 1BWhat comes next 2CWhat comes next 3
test 4 : What comes next 1AWhat comes next 2BWhat comes next 3C

5.7 Conditions

5.7.1 `\ifabsnum` and `\ifabsdim`

There are two tests that we took from pdfTEX:

```
\ifabsnum -10 = 10
    the same number
\fi
\ifabsdim -10pt = 10pt
    the same dimension
\fi
```

This gives

```
the same number the same dimension
```

5.7.2 \ifcmpnum, \ifcmpdim, \ifnumval, \ifdimval, \ifchknum and \ifchkdim

New are the ones that compare two numbers or dimensions:

```
\ifcmpnum 5 8 less \or equal \else more \fi
\ifcmpnum 5 5 less \or equal \else more \fi
\ifcmpnum 8 5 less \or equal \else more \fi

less equal more
```

and

```
\ifcmpdim 5pt 8pt less \or equal \else more \fi
\ifcmpdim 5pt 5pt less \or equal \else more \fi
\ifcmpdim 8pt 5pt less \or equal \else more \fi

less equal more
```

There are also some number and dimension tests. All four expose the `\else` branch when there is an error, but two also report if the number is less, equal or more than zero.

```
\ifnumval  -123  \or < \or = \or > \or ! \else ? \fi
\ifnumval     0  \or < \or = \or > \or ! \else ? \fi
\ifnumval   123  \or < \or = \or > \or ! \else ? \fi
\ifnumval   abc  \or < \or = \or > \or ! \else ? \fi

\ifdimval -123pt \or < \or = \or > \or ! \else ? \fi
\ifdimval    0pt \or < \or = \or > \or ! \else ? \fi
\ifdimval  123pt \or < \or = \or > \or ! \else ? \fi
\ifdimval  abcpt \or < \or = \or > \or ! \else ? \fi

< = > !
< = > !

\ifchknum  -123  \or okay \else bad \fi
\ifchknum     0  \or okay \else bad \fi
\ifchknum   123  \or okay \else bad \fi
\ifchknum   abc  \or okay \else bad \fi

\ifchkdim -123pt \or okay \else bad \fi
\ifchkdim    0pt \or okay \else bad \fi
\ifchkdim  123pt \or okay \else bad \fi
\ifchkdim  abcpt \or okay \else bad \fi

okay okay okay bad
okay okay okay bad
```

The last checked values are available in \lastchknum and \lastchkdim. These don't obey grouping.

5.7.3 \ifmathstyle and \ifmathparameter

These two are variants on \ifcase where the first one operates with values in ranging from zero (display style) to seven (cramped script script style) and the second one can have three values: a parameter is zero, has a value or is unset. The \ifmathparameter primitive takes a proper parameter name and a valid style identifier (a primitive identifier or number). The \ifmathstyle primitive is equivalent to \ifcase \mathstyle.

5.7.4 \ifempty

This primitive tests for the following token (control sequence) having no content. Assuming that \empty is indeed empty, the following two are equivalent:

```
\ifempty\whatever
\ifx\whatever\empty
```

There is no real performance gain here, it's more one of these extensions that lead to less clutter in tracing.

5.7.5 \ifrelax

This primitive complements \ifdefined, \ifempty and \ifcsname so that we have all reasonable tests directly available.

5.7.6 \ifboolean

This primitive tests for non-zero, so the next variants are similar

```
       \ifcase   <integer>.F.\else .T.\fi
\unless\ifcase   <integer>.T.\else .F.\fi
       \ifboolean<integer>.T.\else .F.\fi
```

5.7.7 \iftok and \ifcstok

Comparing tokens and macros can be done with \ifx. Two extra test are provided in LuaMetaTEX:

```
\def\ABC{abc} \def\DEF{def} \def\PQR{abc} \newtoks\XYZ \XYZ {abc}

\iftok{abc}{def}\relax  (same) \else [different] \fi
\iftok{abc}{abc}\relax  [same] \else (different) \fi
\iftok\XYZ {abc}\relax  [same] \else (different) \fi
```

```
\ifcstok\ABC \DEF\relax (same) \else [different] \fi
\ifcstok\ABC \PQR\relax [same] \else (different) \fi
\ifcstok{abc}\ABC\relax [same] \else (different) \fi

[different][same][same]
[different][same][same]
```

You can check if a macro is defined as protected with `\ifprotected` while frozen macros can be tested with `\iffrozen`. A provisional `\ifusercmd` tests will check if a command is defined at the user level (and this one might evolve).

5.7.8 \ifarguments, \ifparameters and \ifparameter

These are part of the extended macro argument parsing features. The `\ifarguments` condition is like an `\ifcase` where the number is the picked up number of arguments. The number reflects the *last* count, so successive macro expansions will adapt the value. The `\ifparameters` counts till the first empty parameter and the `\ifparameter` (singular) takes a parameter reference (like #2) and again is an `\ifcase` where zero means a bad reference, one a non-empty argument and two an empty one. A typical usage is:

```
\def\foo#1#2%
  {\ifparameter#1\or one\fi
   \ifparameter#2\or two\fi}
```

No expansion of arguments takes place here but you can use a test like this:

```
\def\foo#1#2%
  {\iftok{#1}{}\else one\fi
   \iftok{#2}{}\else two\fi}
```

5.7.9 \ifcondition

This is a somewhat special one. When you write macros conditions need to be properly balanced in order to let TEX's fast branch skipping work well. This new primitive is basically a no–op flagged as a condition so that the scanner can recognize it as an if-test. However, when a real test takes place the work is done by what follows, in the next example `\something`.

```
\unexpanded\def\something#1#2%
  {\edef\tempa{#1}%
   \edef\tempb{#2}
   \ifx\tempa\tempb}

\ifcondition\something{a}{b}%
    \ifcondition\something{a}{a}%
        true 1
    \else
        false 1
```

```
    \fi
\else
    \ifcondition\something{a}{a}%
        true 2
    \else
        false 2
    \fi
\fi
```

If you are familiar with MetaPost, this is a bit like `vardef` where the macro has a return value. Here the return value is a test.

Experiments with something `\ifdef` actually worked ok but were rejected because in the end it gave no advantage so this generic one has to do. The `\ifcondition` test is basically is a no-op except when branches are skipped. However, when a test is expected, the scanner gobbles it and the next test result is used. Here is an other example:

```
\def\mytest#1%
  {\ifabsdim#1>0pt\else
     \expandafter \unless
   \fi
   \iftrue}

\ifcondition\mytest{10pt}\relax non-zero \else zero \fi
\ifcondition\mytest {0pt}\relax non-zero \else zero \fi

non-zero zero
```

The last expansion in a macro like `\mytest` has to be a condition and here we use `\unless` to negate the result.

5.7.10 `\orelse` and `\orunless`

Sometimes you have successive tests that, when laid out in the source lead to deep trees. The `\ifcase` test is an exception. Experiments with `\ifcasex` worked out fine but eventually were rejected because we have many tests so it would add a lot. As LuaMetaTEX permitted more experiments, eventually an alternative was cooked up, one that has some restrictions but is relative lightweight. It goes like this:

```
\ifnum\count0<10
    less
\orelse\ifnum\count0=10
    equal
\else
    more
\fi
```

The `\orelse` has to be followed by one of the if test commands, except `\ifcondition`, and there

can be an \unless in front of such a command. These restrictions make it possible to stay in the current condition (read: at the same level). If you need something more complex, using \orelse is probably unwise anyway. In case you wonder about performance, there is a little more checking needed when skipping branches but that can be neglected. There is some gain due to staying at the same level but that is only measurable when you runs tens of millions of complex tests and in that case it is very likely to drown in the real action. It's a convenience mechanism, in the sense that it can make your code look a bit easier to follow.

There is a nice side effect of this mechanism. When you define:

```
\def\quitcondition{\orelse\iffalse}
```

you can do this:

```
\ifnum\count0<10
    less
\orelse\ifnum\count0=10
    equal
    \quitcondition
    indeed
\else
    more
\fi
```

Of course it is only useful at the right level, so you might end up with cases like

```
\ifnum\count0<10
    less
\orelse\ifnum\count0=10
    equal
    \ifnum\count2=30
        \expandafter\quitcondition
    \fi
    indeed
\else
    more
\fi
```

The \orunless variant negates the next test, just like \unless. In some cases these commands look at the next token to see if it is an if-test so a following negation will not work (read: making that work would complicate the code and hurt efficiency too). Side note: interesting is that in ConTEXt we hardly use this kind of negation.

5.7.11 \ifflags

This checker deal with control sequences. You can check if a command is a protected one, that is, defined with the \protected prefix. A command is frozen when it has been defined with the \frozen prefix. Beware: only macros can be frozen. A user command is a command that is not

part of the predefined set of commands. This is an experimental command. The flag values can be queried with `tex.getflagvalues`.

5.8 Boxes, rules and leaders

5.8.1 `\outputbox`

This integer parameter allows you to alter the number of the box that will be used to store the page sent to the output routine. Its default value is 255, and the acceptable range is from 0 to 65535.

```
\outputbox = 12345
```

5.8.2 `\hrule`, `\vrule`, `\nohrule` and `\novrule`

Both rule drawing commands take an optional `xoffset` and `yoffset` parameter. The displacement is virtual and not taken into account when the dimensions are calculated. A rule is specified in the usual way:

There is however a catch. The keyword scanners in LuaMetaTEX are implemented slightly different. When TEX scans a keyword it will (case insensitive) scan for a whole keyword. So, it scans for `height` and when it doesn't find it it will scan for `depth` etc. When it does find a keyword in this case it expects a dimension next. When that criterium is not met it will issue an error message.

In order to avoid look ahead failures like that it is recommended to end the specification with `\relax`. A glue specification is an other example where a `\relax` makes sense when look ahead issues are expected and actually there in traditional scanning the order of keywords can also matter. In any case, when no valid keyword is seen the characters scanned so far are pushed back in the input.

The main reason for using an adapted scanner is that we always permit repetition (consistency) and accept an arbitrary order. Because we have more keywords to process the scanner quits at a partial failure. This prevents some push back and also gives an earlier warning. Interesting is that some ConTEXt users ran into error messages due to a missing `\relax` and found out that their style has a potential flaw with respect to look ahead. One can be lucky for years.

Back to rules, there are some extra keywords, two deal with an offset, and four provide margins. The margins are a bit special because `left` and `top` are the same as are `right` and `bottom`. They influence the edges and these depend on it being a horizontal or vertical rule.

Two new primitives were introduced: `\nohrule` and `\novrule`. These can be used to reserve space. This is often more efficient than creating an empty box with fake dimensions. Of course this assumes that the backend implements them being invisible but still taking space.

5.8.3 \vsplit

The \vsplit primitive has to be followed by a specification of the required height. As alternative for the to keyword you can use upto to get a split of the given size but result has the natural dimensions then.

```
\vsplit 123 to   10cm % final box has the required height
\vsplit 123 upto 10cm % final box has its natural height
```

5.8.4 Images and reused box objects

In original TEX image support is dealt with via specials. It's not a native feature of the engine. All that TEX cares about is dimensions, so in practice that meant: using a box with known dimensions that wraps a special that instructs the backend to include an image. The wrapping is needed because a special itself is a whatsit and as such has no dimensions.

In pdfTEX a special whatsit for images was introduced and that one *has* dimensions. As a consequence, in several places where the engine deals with the dimensions of nodes, it now has to check the details of whatsits. By inheriting code from pdfTEX, the LuaTEX engine also had that property. However, at some point this approach was abandoned and a more natural trick was used: images (and box resources) became a special kind of rules, and as rules already have dimensions, the code could be simplified.

When direction nodes and (formerly local) par nodes also became first class nodes, whatsits again became just that: nodes representing whatever you want, but without dimensions, and therefore they could again be ignored when dimensions mattered. And, because images were disguised as rules, as mentioned, their dimensions automatically were taken into account. This separation between front and backend cleaned up the code base already quite a bit.

In LuaMetaTEX we still have the image specific subtypes for rules, but the engine never looks at subtypes of rules. That was up to the backend. This means that image support is not present in LuaMetaTEX. When an image specification was parsed the special properties, like the filename, or additional attributes, were stored in the backend and all that LuaTEX does is registering a reference to an image's specification in the rule node. But, having no backend means nothing is stored, which in turn would make the image inclusion primitives kind of weird.

Therefore you need to realize that contrary to LuaTEX, *in LuaMetaTEX support for images and box reuse is not built in*! However, we can assume that an implementation uses rules in a similar fashion as LuaTEX does. So, you can still consider images and box reuse to be core concepts. Here we just mention the primitives that LuaTEX provides. They are not available in the engine but can of course be implemented in Lua.

COMMAND	EXPLANATION
\saveboxresource	save the box as an object to be included later
\saveimageresource	save the image as an object to be included later
\useboxresource	include the saved box object here (by index)
\useimageresource	include the saved image object here (by index)
\lastsavedboxresourceindex	the index of the last saved box object

`\lastsavedimageresourceindex`	the index of the last saved image object
`\lastsavedimageresourcepages`	the number of pages in the last saved image object

An implementation probably should accept the usual optional dimension parameters for `\use...resource` in the same format as for rules. With images, these dimensions are then used instead of the ones given to `\useimageresource` but the original dimensions are not overwritten, so that a `\useimageresource` without dimensions still provides the image with dimensions defined by `\saveimageresource`. These optional parameters are not implemented for `\saveboxresource`.

```
\useimageresource width 20mm height 10mm depth 5mm \lastsavedimageresourceindex
\useboxresource   width 20mm height 10mm depth 5mm \lastsavedboxresourceindex
```

Examples or optional entries are `attr` and `resources` that accept a token list, and the `type` key. When set to non-zero the `/Type` entry is omitted. A value of 1 or 3 still writes a `/BBox`, while 2 or 3 will write a `/Matrix`. But, as said: this is entirely up to the backend. Generic macro packages (like `tikz`) can use these assumed primitives so one can best provide them. It is probably, for historic reasons, the only more or less standardized image inclusion interface one can expect to work in all macro packages.

5.8.5 \hpack, \vpack and \tpack

These three primitives are the equivalents of `\hbox`, `\vbox` and `\vtop` but they don't trigger the packaging related callbacks. Of course one never know if content needs a treatment so using them should be done with care. Apart from accepting more keywords (and therefore options) the normal box behave the same as before. The `\vcenter` builder also works in text mode.

5.8.6 \gleaders

This type of leaders is anchored to the origin of the box to be shipped out. So they are like normal `\leaders` in that they align nicely, except that the alignment is based on the *largest* enclosing box instead of the *smallest*. The `g` stresses this global nature.

5.9 Languages

5.9.1 \hyphenationmin

This primitive can be used to set the minimal word length, so setting it to a value of 5 means that only words of 6 characters and more will be hyphenated, of course within the constraints of the `\lefthyphenmin` and `\righthyphenmin` values (as stored in the glyph node). This primitive accepts a number and stores the value with the language.

5.9.2 \boundary, \noboundary, \protrusionboundary and \wordboundary

The \noboundary command is used to inject a whatsit node but now injects a normal node with type boundary and subtype 0. In addition you can say:

```
x\boundary 123\relax y
```

This has the same effect but the subtype is now 1 and the value 123 is stored. The traditional ligature builder still sees this as a cancel boundary directive but at the Lua end you can implement different behaviour. The added benefit of passing this value is a side effect of the generalization. The subtypes 2 and 3 are used to control protrusion and word boundaries in hyphenation and have related primitives.

5.10 Control and debugging

5.10.1 Tracing

If \tracingonline is larger than 2, the node list display will also print the node number of the nodes as well as set attributes (these can be made verbose by a callback). We have only a generic whatsit but again a callback can be used to provide detail. So, when a box is shown in ConTEXt you will see quite a lot more than in other engines. Because nodes have more fields, more is shown anyway, and for nodes that have sublists (like discretionaries) these are also shown. All that could have been delegated to Lua but it felt wrong to not made that a core engine feature.

When bit 1 of \tracinglevels is set the current level is prepended to tracing lines in the log and when bit 2 is set the input level is prepended. You can set both bits and get both numbers prepended. In ConTEXt we default to the value 3, so you get prefixes like 3:4: followed by a space.

When \tracingcommands is larger than 3 the mode switch will be not be prefixed to the {command} but get its own [line].

When \tracinglevels variable is set to 3 the group and input level are shown, a value of 1 or 2 shows only one of them (in ConTEXt we default to 3).

When \tracinghyphenation is set to 1 duplicate patterns are reported (in ConTEXt we default to that) and higher values will also show details about the Lua hyphenation (exception) feedback loop discussed elsewhere.

When set to 1 the \tracingmath variable triggers the reporting of the mode (inline or display) an mlist is processed.

Because in LuaTEX the saving and restoring of locally redefined macros and set variables is optimized a bit in order to prevent redundant stack usage, there will be less tracing visible.

Also, because we have a more extensive macro argument parser, a fast path (and less storage demands) for macros with no arguments, and flags that can be set for macros the way macros are traced can be different in details (we therefore have for instance \meaningfull and \meaningless as variants of \meaning).

5.10.2 \lastnodetype, \lastnodesubtype, \currentiftype

The ε-TEX command \lastnodetype returns the node codes as used in the engine. You can query the numbers at the Lua end if you need the actual values. The parameter \internalcodesmode is no longer provided as compatibility switch because LuaTEX has more cq. some different nodes and it makes no sense to be incompatible with the Lua end of the engine. The same is true for \currentiftype, as we have more conditionals and also use a different order. The \lastnodesubtype is a bonus and again reports the codes used internally. During development these might occasionally change, but eventually they will be stable.

5.11 Files

5.11.1 File syntax

LuaMetaTEX will accept a braced argument as a file name:

```
\input {plain}
\openin 0 {plain}
```

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument. Keep in mind that as side effect of delegating io to Lua the \openin primitive is not provided by the engine and has to be implemented by the macro package. This also means that the limit on the number of open files is not enforced by the engine.

The \tracingfonts primitive that has been inherited from pdfTEX has been adapted to support variants in reporting the font. The reason for this extension is that a csname not always makes sense. The zero case is the default.

VALUE	REPORTED
0	\foo xyz
1	\foo (bar)
2	<bar> xyz
3	<bar @ ..pt> xyz
4	<id>
5	<id: bar>
6	<id: bar @ ..pt> xyz

5.11.2 Writing to file

Writing to a file in TEX has two forms: delayed and immediate. Delayed writing means that the to be written text is anchored in the node list and flushed by the backend. As all io is delegated to Lua, this also means that it has to deal with distinction. In LuaTEX the number of open files was already bumped to 127, but in LuaMetaTEX it depends on the macro package. The special meaning of channel 18 was already dropped in LuaTEX because we have os.execute.

5.12 Math

We will cover math extensions in its own chapter because not only the font subsystem and spacing model have been enhanced (thereby introducing many new primitives) but also because some more control has been added to existing functionality. Much of this relates to the different approaches of traditional TEX fonts and OpenType math.

5.13 Fonts

Like math, we will cover fonts extensions in its own chapter. Here we stick to mentioning that loading fonts is different in LuaMetaTEX. As in LuaTEX we have the extra primitives `\fontid` and `\setfontid`, `\noligs` and `\nokerns`, and `\nospaces`. The other new primitives in LuaTEX have been dropped.

5.14 Directions

5.14.1 Two directions

The directional model in LuaMetaTEX is a simplified version the the model used in LuaTEX. In fact, not much is happening at all: we only register a change in direction.

5.14.2 How it works

The approach is that we try to make node lists balanced but also try to avoid some side effects. What happens is quite intuitive if we forget about spaces (turned into glue) but even there what happens makes sense if you look at it in detail. However that logic makes in-group switching kind of useless when no properly nested grouping is used: switching from right to left several times nested, results in spacing ending up after each other due to nested mirroring. Of course a sane macro package will manage this for the user but here we are discussing the low level injection of directional information.

This is what happens:

```
\textdirection 1 nur {\textdirection 0 run \textdirection 1 NUR} nur
```

This becomes stepwise:

```
injected: [push 1]nur {[push 0]run [push 1]NUR} nur
balanced: [push 1]nur {[push 0]run [pop 0][push 1]NUR[pop 1]} nur[pop 0]
result  : run {RUNrun } run
```

And this:

```
\textdirection 1 nur {nur \textdirection 0 run \textdirection 1 NUR} nur
```

becomes:

```
injected: [+TRT]nur {nur [+TLT]run [+TRT]NUR} nur
balanced: [+TRT]nur {nur [+TLT]run [-TLT][+TRT]NUR[-TRT]} nur[-TRT]
result  : run {run RUNrun } run
```

Now, in the following examples watch where we put the braces:

```
\textdirection 1 nur {{\textdirection 0 run} {\textdirection 1 NUR}} nur
```

This becomes:

run RUN run run

Compare this to:

```
\textdirection 1 nur {{\textdirection 0 run }{\textdirection 1 NUR}} nur
```

Which renders as:

run RUNrun run

So how do we deal with the next?

```
\def\ltr{\textdirection 0\relax}
\def\rtl{\textdirection 1\relax}

run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```

It gets typeset as:

run run RUNrun RUNrun run
run run runRUN runRUN run

We could define the two helpers to look back, pick up a skip, remove it and inject it after the dir node. But that way we loose the subtype information that for some applications can be handy to be kept as-is. This is why we now have a variant of `\textdirection` which injects the balanced node before the skip. Instead of the previous definition we can use:

```
\def\ltr{\linedirection 0\relax}
\def\rtl{\linedirection 1\relax}
```

and this time:

```
run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```

comes out as a properly spaced:

run run RUN run RUN run run
run run run RUN run RUN run

Anything more complex that this, like combination of skips and penalties, or kerns, should be handled in the input or macro package because there is no way we can predict the expected

behaviour. In fact, the `\linedirection` is just a convenience extra which could also have been implemented using node list parsing.

Directions are complicated by the fact that they often need to work over groups so a separate grouping related stack is used. A side effect is that there can be paragraphs with only a local par node followed by direction synchronization nodes. Paragraphs like that are seen as empty paragraphs and therefore ignored. Because `\noindent` doesn't inject anything but a `\indent` injects an box, paragraphs with only an indent and directions are handles and paragraphs with content. When indentation is normalized a paragraph with an indentation skip is seen as content.

5.14.3 Normalizing lines

The original TEX machinery was never meant to be opened up. As a consequence a constructed line can have different layouts. There can be left- and/or right skips and hanging indentation or parshape can result in a shift and adapted width. In LuaTEX glue got subtypes so we can recognize the left-, right and parfill skips, but still there is no hundred percent certainty about the shape.

In LuaMetaTEX lines can be normalized. This is optional because we want to preserve the original (for comparison) and is controlled by `\normalizelinemode`. That variable actually drives some more. An earlier version provided a few more granular options (for instance: does a leftskip comes before or after a left hanging indentation) but in the end that was dropped. Because this normalization only is seen at the Lua end there is no need to go into much detail here.

At this moment a line has this pattern: left parfill, left hang, left skip, indentation, content, right hang, right skip, right parfill. Of course the indentation and fill skips are not present in every line.

Control over normalization happens via the mentioned mode variable and here is what the engine provides right now. We use a bitmap:

VALUE	REPORTED
`0x0001`	normalize line as described above
`0x0002`	use a skip for parindent instead of a box
`0x0004`	swap hangindent in l2r mode
`0x0008`	swap parshape in l2r mode
`0x0010`	put breaks after dir in l2r mode
`0x0020`	remove margin kerns (pdfTEX left-over)
`0x0040`	if needed clip width and use correction kern

Setting the bit enables the related normalization. More features might be added in future releases.

5.14.4 Orientations

As mentioned, the difference with LuaTEX is that we only have numeric directions and that there are only two: left-to-right (`0`) and right-to-left (`1`). The direction of a box is set with `direction`.

In addition to that boxes can now have an `orientation` keyword followed by optional `xoffset`

and/or `yoffset` keywords. The offsets don't have consequences for the dimensions. The alternatives `xmove` and `ymove` on the contrary are reflected in the dimensions. Just play with them. The offsets and moves only are accepted when there is also an orientation, so no time is wasted on testing for these rarely used keywords. There are related primitives `\box...` that set these properties.

As these are experimental it will not be explained here (yet). They are covered in the descriptions of the development of LuaMetaTEX: articles and/or documents in the ConTEXt distribution. For now it is enough to know that the orientation can be up, down, left or right (rotated) and that it has some anchoring variants. Combined with the offsets this permits macro writers to provide solutions for top-down and bottom-up writing directions, something that is rather macro package specific and used for scripts that need manipulations anyway. The 'old' vertical directions were never okay and therefore not used.

There are a couple of properties in boxes that you can set and query but that only really take effect when the backend supports them. When usage on ConTEXt shows that is't okay, they will become official, so we just mention them: `\boxdirection`, `\boxattr`, `\boxorientation`, `\boxxoffset`, `\boxyoffset`, `\boxxmove`, `\boxymove` and `\boxtotal`.

This is still somewhat experimental and will be documented in more detail when I've used it more in ConTEXt and the specification is frozen. This might take some time (and user input).

5.15 Keywords

Some primitives accept one or more keywords and LuaMetaTEX adds some more. In order to deal with this efficiently the keyword scanner has been optimized, where even the context was taken into account. As a result the scanner was quite a bit faster. This kind of optimization was a graduate process the eventually ended up in what we have now. In traditional TEX (and also LuaTEX) the order of keywords is sometimes mixed and sometimes prescribed. In most cases only one occurrence is permitted. So, for instance, this is valid in LuaTEX:

```
\hbox attr 123 456 attr 123 456 spread 10cm { }
\hrule width 10cm depth 3mm
\hskip 3pt plus 2pt minus 1pt
```

The `attr` comes before the `spread`, rules can have multiple mixed dimension specifiers, and in glue the optional `minus` part always comes last. The last two commands are famous for look ahead side effects which is why macro packages will end them with something not keyword, like `\relax`, when needed.

In LuaMetaTEX the following is okay. Watch the few more keywords in box and rule specifications.

```
\hbox reverse to 10cm attr 123 456 orientation 4 xoffset 10pt spread 10cm { }
\hrule xoffset 10pt width 10cm depth 3mm
\hskip 3pt minus 1pt plus 2pt
```

Here the order is not prescribed and, as demonstrated with the box specifier, for instance dimensions (specified by `to` or `spread` can be overloaded by later settings. In case you wonder if that breaks compatibility: in some way it does but bad or sloppy keyword usage breaks a run

anyway. For instance `minuscule` results in `minus` with no dimension being seen. So, in the end the user should not noticed it and when a user does, the macro package already had an issue that had to be fixed.

5.16 Expressions and `\numericscale`

The `*expr` parsers now accept `:` as operator for integer division (the `/` operators does rounding. This can be used for division compatible with `\divide`. I'm still wondering if adding a couple of bit operators makes sense (for integers).

The `\numericscale` parser is kind of special (and might evolve). For now it converts a following number in a scale value as often used in TEX, where 1000 means scaling by 1.0. The trick is in the presence of a digit (or comma): 1.234 becomes 1234 but 1234 stays 1234 and from this you can deduce that 12.34 becomes 123400. Internally TEX calculates with integers, but this permits the macro package to provide an efficient mix.

5.17 Macro arguments

Again this is experimental and (used and) discussed in document that come with the ConTEXt distribution. When defining a macro you can do this:

```
\def\foo(#1)#2{...}
```

Here the first argument between parentheses is mandate. But the magic prefix `\tolerant` makes that limitation go away:

```
\tolerant\def\foo(#1)#2{...}
```

A variant is this:

```
\tolerant\def\foo(#1)#*(#2){...}
```

Here we have two optional arguments, possibly be separated by spaces. There are more parsing options:

`+`	keep the braces
`-`	discard and don't count the argument
`/`	remove leading an trailing spaces and pars
`=`	braces are mandate
`_`	braces are mandate and kept
`^`	keep leading spaces
`1-9`	an argument
`0`	discard but count the argument
`*`	ignore spaces
`.`	ignore pars and spaces
`,`	push back space when no match

:	pick up scanning here
;	quit scanning

For the moment we leave it to your fantasy what these options do. Most probably only make sense when you write a bit more complex macros. Just try to imagine what this does:

```
\permanent\tolerant\global\protected\def\foo(#1)#*#;[#2]#:#3{...}
```

Of course complex combinations can be confusing because after all TEX is parsing for (multi-token) delimiters and will happily gobble the whole file if you are not careful. You can quit scanning if you want:

```
\mymacro 123\ignorearguments
```

which of course only makes sense when used in a nested call where an already picked up arguments is processed further. A not (yet) discussed feature of the parser is that it will happily skip tokens that have the (probably seldom used) ignored characters property.

When you use tracing or see error messages arguments defined using for instance #= will have their usual number in the macro body, so you need to keep track of the numbers.

All this is rather easy on the engine and although it might have a little impact on performance this has been compensated by some more efficiency in the macro parser and engine in general and of course you can gain back some by using these features.

5.18 Overload protection

There is an experimental overload protection mechanism that we will test for a while before declaring it stable. The reason for that is that we need to adapt the ConTEXt code base in order to test its usefulness. Protection is achieved via prefixes. Depending on the value of the `\overloadmode` variable warnings or errors will be triggered. Examples of usage can be found in some documents that come with ConTEXt, so here we just stick to the basics.

```
\mutable  \def\foo{...}
\immutable\def\foo{...}
\permanent\def\foo{...}
\frozen   \def\foo{...}
\aliased  \def\foo{...}
```

A `\mutable` macro can always be changed contrary to an `\immutable` one. For instance a macro that acts as a variable is normally `\mutable`, while a constant can best be immutable. It makes sense to define a public core macro as `\permanent`. Primives start out a `\permanent` ones but with a primitive property instead.

```
          \let\relaxone  \relax 1: \meaningfull\relaxone
\aliased  \let\relaxtwo  \relax 2: \meaningfull\relaxtwo
\permanent\let\relaxthree\relax 3: \meaningfull\relaxthree
```

The \meaningfull primitive is like \meaning but report the properties too. The \meaningless companion reports the body of a macro. Anyway, this typesets:

```
1: \relax
2: primitive \relax
3: permanent \relax
```

So, the \aliased prefix copies the properties. Keep in mind that a macro package can redefine primitives, but \relax is an unlikely candidate.

There is an extra prefix \noaligned that flags a macro as being valid for \noalign compatible usage (which means that the body must contain that one. The idea is that we then can do this:

```
\permanent\protected\noaligned\def\foo{\noalign{...}} % \foo is unexpandable
```

that is: we can have protected macros that don't trigger an error in the parser where there is a look ahead for \noalign which is why normally protection doesn't work well. So: we have macro flagged as permanent (overload protection), being protected (that is, not expandable by default) and a valid equivalent of the noalign primitive. Of course we can also apply the \global and \tolerant prefixes here. The complete repertoire of extra prefixes is:

frozen	a macro that has to be redefined in a managed way
permanent	a macro that had better not be redefined
primitive	a primitive that normally will not be adapted
immutable	a macro or quantity that cannot be changed, it is a constant
mutable	a macro that can be changed no matter how well protected it is
instance	a macro marked as (for instance) be generated by an interface
noaligned	the macro becomes acceptable as \noalign alias
overloaded	when permitted the flags will be adapted
enforced	all is permitted (but only in zero mode or ini mode)
aliased	the macro gets the same flags as the original
untraced	the macro gets a different treatment in tracing

The not yet discussed \instance is just a flag with no special meaning which can be used as classifier. The \frozen also protects against overload which brings amount of blockers to four.

To what extent the engine will complain when a property is changed in a way that violates the flags depends on the parameter \overloadmode. When this parameter is set to zero no checking takes place. More interesting are values larger than zero. If that is the case, when a control sequence is flagged as mutable, it is always permitted to change. When it is set to immutable one can never change it. The other flags determine the kind of checking done. Currently the following overload values are used:

		immutable	**permanent**	**primitive**	**frozen**	**instance**
1	warning	*	*	*		
2	error	*	*	*		
3	warning	*	*	*	*	
4	error	*	*	*	*	

5	warning	*	*	*	*	*
6	error	*	*	*	*	*

The even values (except zero) will abort the run. A value of 255 will freeze this parameter. At level five and above the `\instance` flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

The `\overloaded` prefix can be used to overload a frozen macro. The `\enforced` is more powerful and forces an overload but that prefix is only effective in ini mode or when it's embedded in the body of a macro or token list at ini time unless of course at runtime the mode is zero.

So far for a short explanation. More details can be found in the ConTEXt documentation where we can discuss it in a more relevant perspective. It must be noted that this feature only makes sense a controlled situation, that is: user modules or macros of unpredictable origin will probably suffer from warnings and errors when de mode is set to non zero. In ConTEXt we're okay unless of course users redefine instances but there a warning or error is kind of welcome.

There is an extra prefix `\untraced` that will suppress the meaning when tracing so that the macro looks more like a primitive. It is still somewhat experimental so what gets displayed might change.

5.19 Constants with `\integerdef` and `\dimensiondef`

It is rather common to store constant values in a register or character definition.

```
\newcount\MyConstantA \MyConstantA 123
\newdimen\MyConstantB \MyConstantB 123pt
\chardef \MyConstantC \MyConstantC 123
```

But in LuaMetaTEX we also can do this:

```
\integerdef  \MyConstantC 456
\dimensiondef\MyConstantD 456pt
```

These two are stored as efficient as a register but don't occupy a register slot. They can be set as above, need `\the` for serializations and are seen as valid number or dimension when needed.

Experiments with constant strings made the engine source more complex than I wanted so that features was rejected. Of course we can use the prefixes mentioned in a previous section.

5.20 Serialization with `\todimension`, `\toscaled` and `\tointeger`

These three serializers take a verbose or symbolic quantity:

```
\todimension 10pt   \todimension \scratchdimen    % with unit
\toscaled    10pt   \toscaled    \scratchdimen    % without unit
\tointeger   10     \tointeger   \scratchcounter
```

This is particularly handy in cases where you don't know what you deal with, for instance when a value is stored in a macro. Using `\the` could fail there while:

```
\the\dimexpr10pt\relax
```

is often overkill and gives more noise in a trace.

5.21 Nodes

The ε-TEX primitive `\lastnodetype` is not honest in reporting the internal numbers as it uses its own values. But you can set `\internalcodesmode` to a non-zero value to get the real id's instead. In addition there is `\lastnodesubtype`.

Another last one is `\lastnamedcs` which holds the last match but this one should be used with care because one never knows if in the meantime something else ‘last’ has been seen.

6 Fonts

6.1 Introduction

Only traditional font support is built in, anything more needs to be implemented in Lua. This conforms to the LuaTEX philosophy. When you pass a font to the frontend only the dimensions matter, as these are used in typesetting, and optionally ligatures and kerns when you rely on the built-in font handler. For math some extra data is needed, like information about extensibles and next in size glyphs. You can of course put more information in your Lua tables because when such a table is passed to TEX only that what is needed is filtered from it.

Because there is no built-in backend, virtual font information is not used. If you want to be compatible you'd better make sure that your tables are okay, and in that case you can best consult the LuaTEX manual. For instance, parameters like `extend` are backend related and the standard LuaTEX backend sets the standard here.

6.2 Defining fonts

All TEX fonts are represented to Lua code as tables, and internally as C structures. All keys in the table below are saved in the internal font structure if they are present in the table passed to `font.define`. When the callback is set, which is needed for `\font` to work, its function gets the name and size passed, and it has to return a valid font identifier (a positive number).

For the engine to work well, the following information has to be present at the font level:

KEY	VALUE TYPE	DESCRIPTION
`name`	string	metric (file) name
`original`	string	the name used in logging and feedback
`designsize`	number	expected size (default: 655360 == 10pt)
`size`	number	the required scaling (by default the same as designsize)
`characters`	table	the defined glyphs of this font
`fonts`	table	locally used fonts
`parameters`	hash	default: 7 parameters, all zero
`stretch`	number	the 'stretch'
`shrink`	number	the 'shrink'
`step`	number	the 'step'
`textcontrol`	bitset	this controls various code paths in the text engine
`hyphenchar`	number	default: TEX's `\hyphenchar`
`skewchar`	number	default: TEX's `\skewchar`
`nomath`	boolean	this key allows a minor speedup for text fonts; if it is present and true, then LuaTEX will not check the character entries for math-specific keys
`oldmath`	boolean	this key flags a font as representing an old school TEX math font and disables the OpenType code path

mathcontrol	bitset	this controls various code paths in the math engine, like enforcing the traditional code path
compactmath	boolean	experimental: use the smaller chain to locate a character
textscale	number	scale applied to math text
scriptscale	number	scale applied to math script
scriptscriptscale	number	scale applied to math script script

The parameters is a hash with mixed key types. There are seven possible string keys, as well as a number of integer indices (these start from 8 up). The seven strings are actually used instead of the bottom seven indices, because that gives a nicer user interface.

The names and their internal remapping are:

NAME	REMAPPING
slant	1
space	2
space_stretch	3
space_shrink	4
x_height	5
quad	6
extra_space	7

The characters table is a Lua hash table where the keys are integers. When a character in the input is turned into a glyph node, it gets a character code that normally refers to an entry in that table. For proper paragraph building and math rendering the following fields can be present in an entry in the characters table. You can of course add all kind of extra fields. The engine only uses those that it needs for typesetting a paragraph or formula. The subtables that define ligatures and kerns are also hashes with integer keys, and these indices should point to entries in the main characters table.

Providing ligatures and kerns this way permits TEX to construct ligatures and add inter-character kerning. However, normally you will use an OpenType font in combination with Lua code that does this. In ConTEXt we have base mode that uses the engine, and node mode that uses Lua. A monospaced font normally has no ligatures and kerns and is normally not processed at all.

KEY	TYPE	DESCRIPTION
width	number	width in sp (default 0)
height	number	height in sp (default 0)
depth	number	depth in sp (default 0)
italic	number	italic correction in sp (default 0)
top_accent	number	top accent alignment place in sp (default zero)
bot_accent	number	bottom accent alignment place, in sp (default zero)
left_protruding	number	left protruding factor (\lpcode)
right_protruding	number	right protruding factor (\rpcode)
expansion_factor	number	expansion factor (\efcode)
next	number	‘next larger’ character index
extensible	table	constituent parts of an extensible recipe
vert_variants	table	constituent parts of a vertical variant set

horiz_variants	table	constituent parts of a horizontal variant set
kerns	table	kerning information
ligatures	table	ligaturing information
mathkern	table	math cut-in specifications

For example, here is the character 'f' (decimal 102) in the font `cmr10 at 10pt`. The numbers that represent dimensions are in scaled points.

```
[102] = {
    ["width"]  = 200250,
    ["height"] = 455111,
    ["depth"]  = 0,
    ["italic"] = 50973,
    ["kerns"]  = {
        [63] = 50973,
        [93] = 50973,
        [39] = 50973,
        [33] = 50973,
        [41] = 50973
    },
    ["ligatures"] = {
        [102] = { ["char"] = 11, ["type"] = 0 },
        [108] = { ["char"] = 13, ["type"] = 0 },
        [105] = { ["char"] = 12, ["type"] = 0 }
    }
}
```

Two very special string indexes can be used also: `left_boundary` is a virtual character whose ligatures and kerns are used to handle word boundary processing. `right_boundary` is similar but not actually used for anything (yet).

The values of `top_accent`, `bot_accent` and `mathkern` are used only for math accent and superscript placement, see page 103 in this manual for details. The values of `left_protruding` and `right_protruding` are used only when `\protrudechars` is non-zero. Whether or not `expansion_factor` is used depends on the font's global expansion settings, as well as on the value of `\adjustspacing`.

A math character can have a `next` field that points to a next larger shape. However, the presence of `extensible` will overrule `next`, if that is also present. The `extensible` field in turn can be overruled by `vert_variants`, the OpenType version. The `extensible` table is very simple:

KEY	TYPE	DESCRIPTION
top	number	top character index
mid	number	middle character index
bot	number	bottom character index
rep	number	repeatable character index

The `horiz_variants` and `vert_variants` are arrays of components. Each of those components

is itself a hash of up to five keys:

KEY	TYPE	EXPLANATION
glyph	number	The character index. Note that this is an encoding number, not a name.
extender	number	One (1) if this part is repeatable, zero (0) otherwise.
start	number	The maximum overlap at the starting side (in scaled points).
end	number	The maximum overlap at the ending side (in scaled points).
advance	number	The total advance width of this item. It can be zero or missing, then the natural size of the glyph for character component is used.

The kerns table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value right_boundary), with the values of the kerning to be applied, in scaled points.

The ligatures table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value right_boundary), with the values being yet another small hash, with two fields:

KEY	TYPE	DESCRIPTION
type	number	the type of this ligature command, default 0
char	number	the character index of the resultant ligature

The char field in a ligature is required. The type field inside a ligature is the numerical or string value of one of the eight possible ligature types supported by TEX. When TEX inserts a new ligature, it puts the new glyph in the middle of the left and right glyphs. The original left and right glyphs can optionally be retained, and when at least one of them is kept, it is also possible to move the new 'insertion point' forward one or two places. The glyph that ends up to the right of the insertion point will become the next 'left'.

TEXTUAL (KNUTH)	NUMBER	STRING	RESULT
l + r =: n	0	=:	|n
l + r =:| n	1	=:|	|nr
l + r |=: n	2	|=:	|ln
l + r |=:| n	3	|=:|	|lnr
l + r =:|> n	5	=:|>	n|r
l + r |=:> n	6	|=:>	l|n
l + r |=:|> n	7	|=:|>	l|nr
l + r |=:|>> n	11	|=:|>>	ln|r

The default value is 0, and can be left out. That signifies a 'normal' ligature where the ligature replaces both original glyphs. In this table the | indicates the final insertion point.

The mathcontrol bitset is mostly there for experimental purposes. Because there is inconsistency in the OpenType math fonts with respect to for instance glyph dimensions, it is possible to force the traditional code path. We just mention the possible flags:

VALUE	EFFECT
0x0001	over_rule

0x0002	under_rule
0x0004	radical_rule
0x0008	fraction_rule
0x0010	accent_skew_half
0x0020	accent_skew_apply
0x0040	accent_italic_kern
0x0080	delimiter_italic_kern
0x0100	ord_italic_kern
0x0200	char_italic_width
0x0400	char_italic_no_rebox
0x0800	boxed_no_italic_kern
0x1000	no_staircase_kern
0x2000	text_italic_kern

Compact math is an experimental feature. The smaller field in a character definition of a text character can point to a script character that itself can point to a scriptscript one. When set the `textscale`, `scriptscale` and `scriptscriptscale` is applied to those.

The `textcontrol` field is used to control some aspects of text processing. More options might be added in the future.

VALUE	EFFECT
0x0001	collapse_hyphens

In ConTEXt these are interfaced via pseudo features. The math control flags of a font can be overloaded by `\mathcontrolmode` on the spot and the set controls of a font can be queried by `\fontmathcontrol`. The text control flags in a font always win over the ones set by other parameters, like `\hyphenationmode`. They can be queried with `\fonttextcontrol`.

6.3 Virtual fonts

Virtual fonts have been introduced to overcome limitations of good old TEX. They were mostly used for providing a direct mapping from for instance accented characters onto a glyph. The backend was responsible for turning a reference to a character slot into a real glyph, possibly constructed from other glyphs. In our case there is no backend so there is also no need to pass this information through TEX. But it can of course be part of the font information and because it is a kind of standard, we describe it here.

A character is virtual when it has a `commands` array as part of the data. A virtual character can itself point to virtual characters but be careful with nesting as you can create loops and overflow the stack (which often indicates an error anyway).

At the font level there can be a an (indexed) `fonts` table. The values are one- or two-key hashes themselves, each entry indicating one of the base fonts in a virtual font. In case your font is referring to itself in for instance a virtual font, you can use the `slot` command with a zero font reference, which indicates that the font itself is used. So, a table looks like this:

```
fonts = {
```

```
  { name = "ptmr8a", size = 655360 },
  { name = "psyr", size = 600000 },
  { id = 38 }
}
```

The first referenced font (at index 1) in this virtual font is `ptrmr8a` loaded at 10pt, and the second is `psyr` loaded at a little over 9pt. The third one is a previously defined font that is known to LuaTEX as font id 38. The array index numbers are used by the character command definitions that are part of each character.

The `commands` array is a hash where each item is another small array, with the first entry representing a command and the extra items being the parameters to that command. The frontend is only interested in the dimensions, ligatures and kerns of a font, which is the reason why the TEX engine didn't have to be extended when virtual fonts showed up: dealing with it is up to the driver that comes after the backend. In pdfTEX and LuaTEX that driver is integrated so there the backend also deals with virtual fonts. The first block in the next table is what the standard mentions. The `special` command is indeed special because it is an extension container. The mentioned engines only support pseudo standards where the content starts with `pdf:`. The last block is LuaTEX specific and will not be found in native fonts. These entries can be used in virtual fonts that are constructed in Lua.

But ... in LuaMetaTEX there is no backend built in but we might assume that the one provided deals with these entries. However, a provided backend can provide more and that is indeed what happens in ConTEXt. There, because we no longer have compacting (of passed tables) and unpacking (when embedding) of these tables going on we stay in the Lua domain. None of the virtual specification is ever seen in the engine.

COMMAND	ARGUMENTS	TYPE	DESCRIPTION
`font`	1	number	select a new font from the local `fonts` table
`char`	1	number	typeset this character number from the current font, and move right by the character's width
`push`	0		save current position
`pop`	0		pop position
`rule`	2	2 numbers	output a rule $ht * wd$, and move right.
`down`	1	number	move down on the page
`right`	1	number	move right on the page
`special`	1	string	output a driver directive
`nop`	0		do nothing
`slot`	2	2 numbers	a shortcut for the combination of a font and char command
`node`	1	node	output this node (list), and move right by the width of this list
`pdf`	2	2 strings	output a pdf literal, the first string is one of `origin`, `page`, `text`, `font`, `direct` or `raw`; if you have one string only `origin` is assumed

lua	1	string, function	execute a Lua script when the glyph is embedded; in case of a function it gets the font id and character code passed
image	1	image	output an image (the argument can be either an <image> variable or an image_spec table)
comment	any	any	the arguments of this command are ignored

When a font id is set to 0 then it will be replaced by the currently assigned font id. This prevents the need for hackery with future id's.

The pdf option also accepts a mode keyword in which case the third argument sets the mode. That option will change the mode in an efficient way (passing an empty string would result in an extra empty lines in the pdf file. This option only makes sense for virtual fonts. The font mode only makes sense in virtual fonts. Modes are somewhat fuzzy and partially inherited from pdfTEX.

MODE	DESCRIPTION
origin	enter page mode and set the position
page	enter page mode
text	enter text mode
font	enter font mode (kind of text mode, only in virtual fonts)
always	finish the current string and force a transform if needed
raw	finish the current string

You always need to check what pdf code is generated because there can be all kind of interferences with optimization in the backend and fonts are complicated anyway. Here is a rather elaborate glyph commands example using such keys:

```
...
commands = {
    { "push" },                     -- remember where we are
    { "right", 5000 },              -- move right about 0.08pt
    { "font", 3 },                  -- select the fonts[3] entry
    { "char", 97 },                 -- place character 97 (ASCII 'a')
 -- { "slot", 2, 97 },              -- an alternative for the previous two
    { "pop" },                      -- go all the way back
    { "down", -200000 },            -- move upwards by about 3pt
    { "special", "pdf: 1 0 0 rg" }  -- switch to red color
 -- { "pdf", "origin", "1 0 0 rg" } -- switch to red color (alternative)
    { "rule", 500000, 20000 }       -- draw a bar
    { "special", "pdf: 0 g" }       -- back to black
 -- { "pdf", "origin", "0 g" }      -- back to black (alternative)
}
...
```

The default value for font is always 1 at the start of the commands array. Therefore, if the virtual font is essentially only a re-encoding, then you do usually not have created an explicit 'font' command in the array.

Rules inside of `commands` arrays are built up using only two dimensions: they do not have depth. For correct vertical placement, an extra `down` command may be needed.

Regardless of the amount of movement you create within the `commands`, the output pointer will always move by exactly the width that was given in the `width` key of the character hash. Any movements that take place inside the `commands` array are ignored on the upper level.

The special can have a `pdf:`, `pdf:origin:`, `pdf:page:`, `pdf:direct:` or `pdf:raw:` prefix. When you have to concatenate strings using the `pdf` command might be more efficient.

6.4 Additional TEX commands

6.4.1 Font syntax

LuaTEX will accept a braced argument as a font name:

```
\font\myfont = {cmr10}
```

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument.

6.4.2 `\fontid` and `\setfontid`

```
\fontid\font
```

This primitive expands into a number. The currently used font id is 13. Here are some more:[3]

STYLE	COMMAND	FONT ID
normal	`\tf`	13
bold	`\bf`	**17**
italic	`\it`	*22*
bold italic	`\bi`	***23***

These numbers depend on the macro package used because each one has its own way of dealing with fonts. They can also differ per run, as they can depend on the order of loading fonts. For instance, when in ConTEXt virtual math Unicode fonts are used, we can easily get over a hundred ids in use. Not all ids have to be bound to a real font, after all it's just a number.

The primitive `\setfontid` can be used to enable a font with the given id, which of course needs to be a valid one.

6.4.3 `\glyphoptions`

In LuaTEX the `\noligs` and `\nokerns` primitives suppress these features but in LuaMetaTEX these primitives are gone. They are replace by a more generic control primitive `\glyphoptions`. This

[3] Contrary to LuaTEX this is now a number so you need to use `\number` or `\the`. The same is true for some other numbers and dimensions that for some reason ended up in the serializer that produced a sequence of tokens.

numerical parameter is a bitset with the following fields:

VALUE	EFFECT
0x01	prevent left ligature
0x02	prevent right ligature
0x04	block left kern
0x08	block right kern
0x10	don't apply expansion
0x20	don't apply protrusion
0x40	apply xoffset to width
0x80	apply yoffset to height and depth

The effects speak for themselves. They provide detailed control over individual glyph, this because the current value of this option is stored with glyphs.

6.4.4 \glyphxscale, \glyphyscale and \scaledfontdimen

The two scale parameters control the current scaling. They are traditional TEX integer parameters that operate independent of each other. The scaling is reflected in the dimensions of glyphs as well as in the related font dimensions, which means that units like ex and em work as expected. If you query a font dimensions with \fontdimen you get the raw value but with \scaledfontdimen you get the useable value.

6.4.5 \glyphxoffset, \glyphyoffset

These two parameters control the horizontal and vertical shift of glyphs with, when applied to a stretch of them, the horizontal offset probably being the least useful.

6.4.6 \glyph

This command is a variation in \char that takes keywords:

KEYWORD	EFFECT	type
xoffset	(virtual) horizontal shift	dimension
yoffset	(virtual) vertical shift	dimension
xscale	horizontal scaling	integer
yscale	vertical scaling	integer
options	glyph options	bitset
font	font	identifier
id	font	integer

The values default to the currently set values. Here is a ConTEXt example:

```
\ruledhbox{
    \ruledhbox{\glyph yoffset 1ex options 0 123}
    \ruledhbox{\glyph xoffset .5em yoffset 1ex options "C0 125}
```

```
    \ruledhbox{baseline\glyphyoffset 1ex \glyphxscale 800 \glyphyscale\glyphxs-
cale raised}
}
```

Visualized:

{ } baseline raised

6.4.7 \nospaces

This new primitive can be used to overrule the usual `\spaceskip` related heuristics when a space character is seen in a text flow. The value 1 triggers no injection while 2 results in injection of a zero skip. In figure 6.1 we see the results for four characters separated by a space.

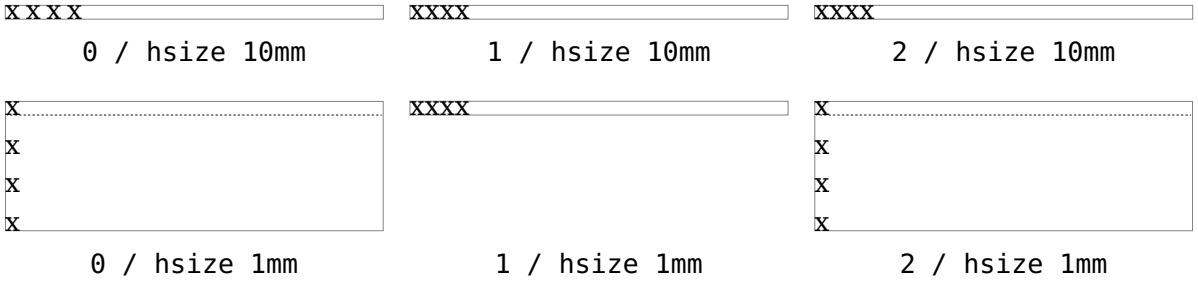


Figure 6.1 The `\nospaces` options.

6.4.8 \protrusionboundary

The protrusion detection mechanism is enhanced a bit to enable a bit more complex situations. When protrusion characters are identified some nodes are skipped:

- zero glue
- penalties
- empty discretionaries
- normal zero kerns
- rules with zero dimensions
- math nodes with a surround of zero
- dir nodes
- empty horizontal lists
- local par nodes
- inserts, marks and adjusts
- boundaries
- whatsits

Because this can not be enough, you can also use a protrusion boundary node to make the next node being ignored. When the value is 1 or 3, the next node will be ignored in the test when locating a left boundary condition. When the value is 2 or 3, the previous node will be ignored when locating a right boundary condition (the search goes from right to left). This permits protrusion combined with for instance content moved into the margin:

```
\protrusionboundary1\llap{!\quad}«Who needs protrusion?»
```

6.5 The Lua font library

6.5.1 Introduction

The Lua font library is reduced to a few commands. Contrary to LuaTEX there is no loading of tfm or vf files. The explanation of the following commands is in the LuaTEX manual.

FUNCTION	DESCRIPTION
`current`	returns the id of the currently active font
`max`	returns the last assigned font identifier
`setfont`	enables a font setfont (sets the current font id)
`addcharacters`	adds characters to a font
`define`	defined a font
`id`	returns the id that relates to a command name

For practical reasons the management of font identifiers is still done by TEX but it can become an experiment to delegate that to Lua as well.

6.5.2 Defining a font with `define`, `addcharacters` and `setfont`

Normally you will use a callback to define a font but there's also a Lua function that does the job.

```
id = font.define(<table> f)
```

Within reasonable bounds you can extend a font after it has been defined. Because some properties are best left unchanged this is limited to adding characters.

```
font.addcharacters(<number n>, <table> f)
```

The table passed can have the fields `characters` which is a (sub)table like the one used in define, and for virtual fonts a `fonts` table can be added. The characters defined in the `characters` table are added (when not yet present) or replace an existing entry. Keep in mind that replacing can have side effects because a character already can have been used. Instead of posing restrictions we expect the user to be careful. The `setfont` helper is a more drastic replacer and only works when a font has not been used yet.

6.5.3 Font ids: `id`, `max` and `current`

```
<number> i = font.id(<string> csname)
```

This returns the font id associated with `csname`, or −1 if `csname` is not defined.

```
<number> i = font.max()
```

This is the largest used index so far. The currently active font id can be queried or set with:

```
<number> i = font.current()
font.current(<number> i)
```

6.5.4 Glyph data: \glyphdatafield, \glyphscriptfield, \glyphstatefield

These primitives can be used to set an additional glyph properties. Of course it's very macro package dependant what is done with that. It started with just the first one as experiment, simply because we had some room left in the glyph data structure. It's basically an single attribute. Then, when we got rid of the ligature pointer we could either drop it or use that extra field for some more, and because ConTEXt already used the data field, that is what happened. The script and state fields are shorts, that is, they run from zero to `0xFFFF` where we assume that zero means 'unset'. Although they can be used for whatever purpose their use in ConTEXt is fixed.

7 Languages, characters, fonts and glyphs

7.1 Introduction

LuaTEX's internal handling of the characters and glyphs that eventually become typeset is quite different from the way TEX82 handles those same objects. The easiest way to explain the difference is to focus on unrestricted horizontal mode (i.e. paragraphs) and hyphenation first. Later on, it will be easy to deal with the differences that occur in horizontal and math modes.

In TEX82, the characters you type are converted into `char` node records when they are encountered by the main control loop. TEX attaches and processes the font information while creating those records, so that the resulting 'horizontal list' contains the final forms of ligatures and implicit kerning. This packaging is needed because we may want to get the effective width of for instance a horizontal box.

When it becomes necessary to hyphenate words in a paragraph, TEX converts (one word at time) the `char` node records into a string by replacing ligatures with their components and ignoring the kerning. Then it runs the hyphenation algorithm on this string, and converts the hyphenated result back into a 'horizontal list' that is consecutively spliced back into the paragraph stream. Keep in mind that the paragraph may contain unboxed horizontal material, which then already contains ligatures and kerns and the words therein are part of the hyphenation process.

Those `char` node records are somewhat misnamed, as they are glyph positions in specific fonts, and therefore not really 'characters' in the linguistic sense. There is no language information inside the `char` node records at all. Instead, language information is passed along using `language whatsit` nodes inside the horizontal list.

In LuaTEX, the situation is quite different. The characters you type are always converted into `glyph` node records with a special subtype to identify them as being intended as linguistic characters. LuaTEX stores the needed language information in those records, but does not do any font-related processing at the time of node creation. It only stores the index of the current font and a reference to a character in that font.

When it becomes necessary to typeset a paragraph, LuaTEX first inserts all hyphenation points right into the whole node list. Next, it processes all the font information in the whole list (creating ligatures and adjusting kerning), and finally it adjusts all the subtype identifiers so that the records are 'glyph nodes' from now on.

7.2 Characters, glyphs and discretionaries

TEX82 (including pdfTEX) differentiates between `char` nodes and `lig` nodes. The former are simple items that contained nothing but a 'character' and a 'font' field, and they lived in the same memory as tokens did. The latter also contained a list of components, and a subtype indicating whether this ligature was the result of a word boundary, and it was stored in the same place as other nodes like boxes and kerns and glues. In LuaMetaTEX we no longer keep the list of components with the glyph node.

In LuaTEX, these two types are merged into one, somewhat larger structure called a `glyph` node. Besides having the old character, font, and component fields there are a few more, like 'attr' that we will see in section 10.2.12, these nodes also contain a subtype, that codes four main types and two additional ghost types. For ligatures, multiple bits can be set at the same time (in case of a single-glyph word).

- `character`, for characters to be hyphenated: the lowest bit (bit 0) is set to 1.
- `glyph`, for specific font glyphs: the lowest bit (bit 0) is not set.
- `ligature`, for constructed ligatures bit 1 is set.

The `glyph` nodes also contain language data, split into four items that were current when the node was created: the `\setlanguage` (15 bits), `\lefthyphenmin` (8 bits), `\righthyphenmin` (8 bits), and `\uchyph` (1 bit).

Incidentally, LuaTEX allows 16383 separate languages, and words can be 256 characters long. The language is stored with each character. You can set `\firstvalidlanguage` to for instance 1 and make thereby language 0 an ignored hyphenation language.

The new primitive `\hyphenationmin` can be used to signal the minimal length of a word. This value is stored with the (current) language.

Because the `\uchyph` value is saved in the actual nodes, its handling is subtly different from TEX82: changes to `\uchyph` become effective immediately, not at the end of the current partial paragraph.

Typeset boxes now always have their language information embedded in the nodes themselves, so there is no longer a possible dependency on the surrounding language settings. In TEX82, a mid-paragraph statement like `\unhbox0` would process the box using the current paragraph language unless there was a `\setlanguage` issued inside the box. In LuaTEX, all language variables are already frozen.

In traditional TEX the process of hyphenation is driven by `lccodes`. In LuaTEX we made this dependency less strong. There are several strategies possible. When you do nothing, the currently used `lccodes` are used, when loading patterns, setting exceptions or hyphenating a list.

When you set `\savinghyphcodes` to a value greater than zero the current set of `lccodes` will be saved with the language. In that case changing a `lccode` afterwards has no effect. However, you can adapt the set with:

```
\hjcode`a=`a
```

This change is global which makes sense if you keep in mind that the moment that hyphenation happens is (normally) when the paragraph or a horizontal box is constructed. When `\savinghyphcodes` was zero when the language got initialized you start out with nothing, otherwise you already have a set.

When a `\hjcode` is greater than 0 but less than 32 is indicates the to be used length. In the following example we map a character (`x`) onto another one in the patterns and tell the engine that `œ` counts as two characters. Because traditionally zero itself is reserved for inhibiting hyphenation, a value of 32 counts as zero.

Here are some examples (we assume that French patterns are used):

```
                                foobar  foo-bar
\hjcode `x=`o                   fxxbar  fxx-bar
\lefthyphenmin 3                œdipus  œdi-pus
\lefthyphenmin 4                œdipus  œdipus
\hjcode `œ=2                    œdipus  œdi-pus
\hjcode `i=32 \hjcode `d=32     œdipus  œdipus
```

Carrying all this information with each glyph would give too much overhead and also make the process of setting up these codes more complex. A solution with `hjcode` sets was considered but rejected because in practice the current approach is sufficient and it would not be compatible anyway.

Beware: the values are always saved in the format, independent of the setting of `\savinghyphcodes` at the moment the format is dumped.

A boundary node normally would mark the end of a word which interferes with for instance discretionary injection. For this you can use the `\wordboundary` as a trigger. Here are a few examples of usage:

```
discrete---discrete
```

dis-
crete—
dis-
crete

```
discrete\discretionary{}{}{---}discrete
```

discrete
discrete

```
discrete\wordboundary\discretionary{}{}{---}discrete
```

dis-
crete
discrete

```
discrete\wordboundary\discretionary{}{}{---}\wordboundary discrete
```

dis-
crete
dis-
crete

```
discrete\wordboundary\discretionary{---}{}{}\wordboundary discrete
```

dis-
crete—
dis-
crete

We only accept an explicit hyphen when there is a preceding glyph and we skip a sequence of explicit hyphens since that normally indicates a `--` or `---` ligature in which case we can in a worse case usage get bad node lists later on due to messed up ligature building as these dashes are ligatures in base fonts. This is a side effect of separating the hyphenation, ligaturing and kerning steps.

The start and end of a sequence of characters is signalled by a `glue`, `penalty`, `kern` or `boundary` node. But by default also a `hlist`, `vlist`, `rule`, `dir`, `whatsit`, `insert`, and `adjust` node indicate a start or end. You can omit the last set from the test by setting flags in `\hyphenationmode`:

VALUE	BEHAVIOUR
	not strict
64	strict start
128	strict end
192	strict start and strict end

The word start is determined as follows:

NODE	BEHAVIOUR
boundary	yes when wordboundary
hlist	when the start bit is set
vlist	when the start bit is set
rule	when the start bit is set
dir	when the start bit is set
whatsit	when the start bit is set
glue	yes
math	skipped
glyph	exhyphenchar (one only) : yes (so no – —)
otherwise	yes

The word end is determined as follows:

NODE	BEHAVIOUR
boundary	yes
glyph	yes when different language
glue	yes
penalty	yes
kern	yes when not italic (for some historic reason)
hlist	when the end bit is set
vlist	when the end bit is set
rule	when the end bit is set
dir	when the end bit is set
whatsit	when the end bit is set
ins	when the end bit is set
adjust	when the end bit is set

Figures 7.1 upto 7.5 show some examples. In all cases we set the min values to 1 and make sure

that the words hyphenate at each character.

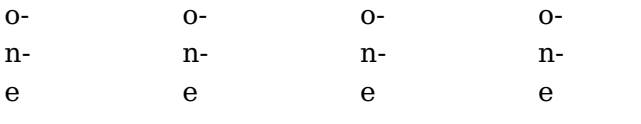


Figure 7.1 one

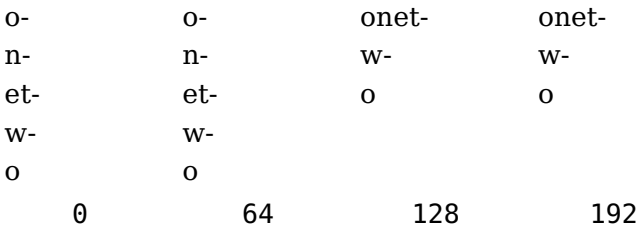


Figure 7.2 one\null two

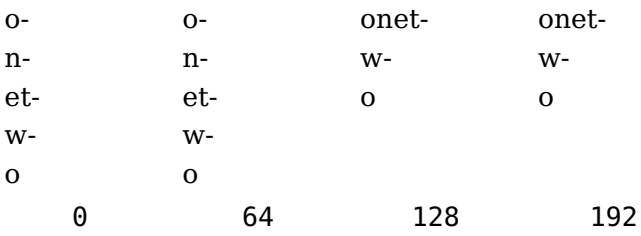


Figure 7.3 \null one\null two

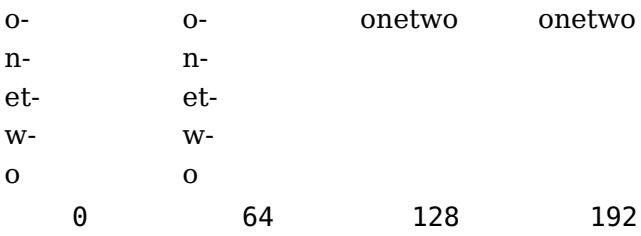


Figure 7.4 one\null two\null

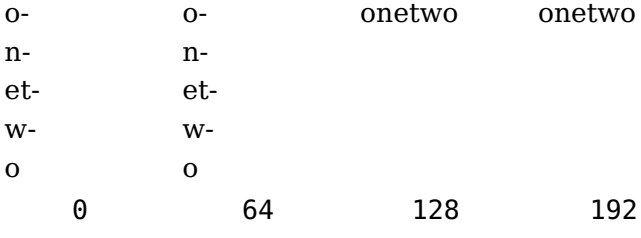


Figure 7.5 \null one\null two\null

In traditional TEX ligature building and hyphenation are interwoven with the line break mechanism. In LuaTEX these phases are isolated. As a consequence we deal differently with (a sequence of) explicit hyphens. We already have added some control over aspects of the hyphenation and yet another one concerns automatic hyphens (e.g. - characters in the input).

Hyphenation and discretionary injection is driven by a mode parameter which is a bitset made

from the following values, some of which we saw in the previous examples.

1	honour (normal) `\discretionary`'s
2	turn `-` into (automatic) discretionaries
4	turn `\-` into (explicit) discretionaries
8	hyphenate (syllable) according to language
16	hyphenate uppercase characters too (replaces `\uchyph`
32	permit break at an explicit hyphen (border cases)
64	traditional TEX compatibility wrt the start of a word
128	traditional TEX compatibility wrt the end of a word
256	use `\automatichyphenpenalty`
512	use `\explicithyphenpenalty`
1024	turn glue in discretionaries into kerns
2048	okay, let's be even more tolerant in discretionaries
4096	and again we're more permissive
16384	controls how successive explicit discretionaries are handled in base mode
8192	treat all discretionaries equal when breaking lines (in all three passes)
32768	kick in the handler (experiment)
65536	feedback compound snippets

Some of these options are still experimental, simply because not all aspects and side effects have been explored. You can find some experimental use cases in ConTEXt.

7.3 The main control loop

In LuaTEX's main loop, almost all input characters that are to be typeset are converted into `glyph` node records with subtype ‘character’, but there are a few exceptions.

1. The `\accent` primitive creates nodes with subtype ‘glyph’ instead of ‘character’: one for the actual accent and one for the accentee. The primary reason for this is that `\accent` in TEX82 is explicitly dependent on the current font encoding, so it would not make much sense to attach a new meaning to the primitive's name, as that would invalidate many old documents and macro packages. A secondary reason is that in TEX82, `\accent` prohibits hyphenation of the current word. Since in LuaTEX hyphenation only takes place on ‘character’ nodes, it is possible to achieve the same effect. Of course, modern Unicode aware macro packages will not use the `\accent` primitive at all but try to map directly on composed characters.
   This change of meaning did happen with `\char`, that now generates ‘glyph’ nodes with a character subtype. In traditional TEX there was a strong relationship between the 8-bit input encoding, hyphenation and glyphs taken from a font. In LuaTEX we have utf input, and in most cases this maps directly to a character in a font, apart from glyph replacement in the font engine. If you want to access arbitrary glyphs in a font directly you can always use Lua to do so, because fonts are available as Lua table.
2. All the results of processing in math mode eventually become nodes with ‘glyph’ subtypes. In fact, the result of processing math is just a regular list of glyphs, kerns, glue, penalties, boxes etc.
3. Automatic discretionaries are handled differently. TEX82 inserts an empty discretionary after

sensing an input character that matches the \hyphenchar in the current font. This test is wrong in our opinion: whether or not hyphenation takes place should not depend on the current font, it is a language property.[4]

In LuaTEX, it works like this: if LuaTEX senses a string of input characters that matches the value of the new integer parameter \exhyphenchar, it will insert an explicit discretionary after that series of nodes. Initially TEX sets the \exhyphenchar=`\-. Incidentally, this is a global parameter instead of a language-specific one because it may be useful to change the value depending on the document structure instead of the text language.

The insertion of discretionaries after a sequence of explicit hyphens happens at the same time as the other hyphenation processing, *not* inside the main control loop.

The only use LuaTEX has for \hyphenchar is at the check whether a word should be considered for hyphenation at all. If the \hyphenchar of the font attached to the first character node in a word is negative, then hyphenation of that word is abandoned immediately. This behaviour is added for backward compatibility only, and the use of \hyphenchar=-1 as a means of preventing hyphenation should not be used in new LuaTEX documents.

4. The \setlanguage command no longer creates whatsits. The meaning of \setlanguage is changed so that it is now an integer parameter like all others. That integer parameter is used in \glyph_node creation to add language information to the glyph nodes. In conjunction, the \language primitive is extended so that it always also updates the value of \setlanguage.
5. The \noboundary command (that prohibits word boundary processing where that would normally take place) now does create nodes. These nodes are needed because the exact place of the \noboundary command in the input stream has to be retained until after the ligature and font processing stages.
6. There is no longer a main_loop label in the code. Remember that TEX82 did quite a lot of processing while adding char_nodes to the horizontal list? For speed reasons, it handled that processing code outside of the 'main control' loop, and only the first character of any 'word' was handled by that 'main control' loop. In LuaTEX, there is no longer a need for that (all hard work is done later), and the (now very small) bits of character-handling code have been moved back inline. When \tracingcommands is on, this is visible because the full word is reported, instead of just the initial character.

Because we tend to make hard coded behaviour configurable a few new primitives have been added:

```
\hyphenpenaltymode
\automatichyphenpenalty
\explicithyphenpenalty
```

The usage of these penalties is controlled by the \hyphenationmode flags 256 and 512 and when these are not set \exhyphenpenalty is used.

You can use the \tracinghyphenation variable to get a bit more information about what happens.

[4] When TEX showed up we didn't have Unicode yet and being limited to eight bits meant that one sometimes had to compromise between supporting character input, glyph rendering, hyphenation.

VALUE	EFFECT
1	report redundant pattern (happens by default in LuaTEX)
2	report words that reach the hyphenator and got treated
3	show the result of a hyphenated word (a node list)

7.4 Loading patterns and exceptions

Although we keep the traditional approach towards hyphenation (which is still superior) the implementation of the hyphenation algorithm in LuaTEX is quite different from the one in TEX82.

After expansion, the argument for `\patterns` has to be proper utf8 with individual patterns separated by spaces, no `\char` or `\chardefd` commands are allowed. The current implementation is quite strict and will reject all non-Unicode characters. Likewise, the expanded argument for `\hyphenation` also has to be proper utf8, but here a bit of extra syntax is provided:

1. Three sets of arguments in curly braces (`{}{}{}`) indicate a desired complex discretionary, with arguments as in `\discretionary`'s command in normal document input.
2. A `-` indicates a desired simple discretionary, cf. `\-` and `\discretionary{-}{}{}` in normal document input.
3. Internal command names are ignored. This rule is provided especially for `\discretionary`, but it also helps to deal with `\relax` commands that may sneak in.
4. An `=` indicates a (non-discretionary) hyphen in the document input.

The expanded argument is first converted back to a space-separated string while dropping the internal command names. This string is then converted into a dictionary by a routine that creates key-value pairs by converting the other listed items. It is important to note that the keys in an exception dictionary can always be generated from the values. Here are a few examples:

VALUE	IMPLIED KEY (INPUT)	EFFECT
`ta-ble`	table	`ta\-ble` (= `ta\discretionary{-}{}{}ble`)
`ba{k-}{}{c}ken`	backen	`ba\discretionary{k-}{}{c}ken`

The resultant patterns and exception dictionary will be stored under the language code that is the present value of `\language`.

In the last line of the table, you see there is no `\discretionary` command in the value: the command is optional in the TEX-based input syntax. The underlying reason for that is that it is conceivable that a whole dictionary of words is stored as a plain text file and loaded into LuaTEX using one of the functions in the Lua `language` library. This loading method is quite a bit faster than going through the TEX language primitives, but some (most?) of that speed gain would be lost if it had to interpret command sequences while doing so.

It is possible to specify extra hyphenation points in compound words by using `{-}{}{-}` for the explicit hyphen character (replace `-` by the actual explicit hyphen character if needed). For example, this matches the word ‘multi-word-boundaries’ and allows an extra break inbetween ‘boun’ and ‘daries’:

```
\hyphenation{multi{-}{}{-}word{-}{}{-}boun-daries}
```

The motivation behind the ε-TEX extension `\savinghyphcodes` was that hyphenation heavily depended on font encodings. This is no longer true in LuaTEX, and the corresponding primitive is basically ignored. Because we now have `\hjcode`, the case related codes can be used exclusively for `\uppercase` and `\lowercase`.

The three curly brace pair pattern in an exception can be somewhat unexpected so we will try to explain it by example. The pattern `foo{}{}{x}bar` pattern creates a lookup `fooxbar` and the pattern `foo{}{}{}bar` creates `foobar`. Then, when a hit happens there is a replacement text (`x`) or none. Because we introduced penalties in discretionary nodes, the exception syntax now also can take a penalty specification. The value between square brackets is a multiplier for `\exceptionpenalty`. Here we have set it to 10000 so effectively we get 30000 in the example.

`x{a-}{-b}{}x{a-}{-b}{}x{a-}{-b}{}x{a-}{-b}{}xx`			
`10em`	`3em`	`0em`	`6em`
`123 xxxxxx 123`	`123 xxa- -bxa- -bxa- -bxx 123`	`123 xa- -bxa- -bxa- -bxx 123`	`123 xxxxxx xxxxxx xxa- -bxxxx xxa- -bxxxx 123`

`x{a-}{-b}{}x{a-}{-b}{}[3]x{a-}{-b}{}[1]x{a-}{-b}{}xx`			
`10em`	`3em`	`0em`	`6em`
`123 xxxxxx 123`	`123 xa- -bxxxa- -bxx 123`	`123 xa- -bxxxa- -bxx 123`	`123 xxxxa- -bxx xxxxxx xxxxxx xa- -bxxxxx 123`

`z{a-}{-b}{z}{a-}{-b}{z}{a-}{-b}{z}{a-}{-b}{z}z`			
`10em`	`3em`	`0em`	`6em`
`123 zzzzzz 123`	`123 za- -bza- -bza- -b 123`	`123 za- -bza- -bza- -b a- -b23`	`123 zzzzzz zzzzzz zzza- -bzz zzzzzz 123`

z{a-}{-b}{z}{a-}{-b}{z}[3]{a-}{-b}{z}[1]{a-}{-b}{z}z			
10em	3em	0em	6em
123 zzzzzz 123	123 za- -bzzzz 123	123 za- -bzzzz a- -b23	123 zzzzzz zzzzzz za- -bzzzz a- -bzzzzz 123

7.5 Applying hyphenation

The internal structures LuaTEX uses for the insertion of discretionaries in words is very different from the ones in TEX82, and that means there are some noticeable differences in handling as well.

First and foremost, there is no 'compressed trie' involved in hyphenation. The algorithm still reads pattern files generated by Patgen, but LuaTEX uses a finite state hash to match the patterns against the word to be hyphenated. This algorithm is based on the 'libhnj' library used by OpenOffice, which in turn is inspired by TEX.

There are a few differences between LuaTEX and TEX82 that are a direct result of the implementation:

- LuaTEX happily hyphenates the full Unicode character range.
- Pattern and exception dictionary size is limited by the available memory only, all allocations are done dynamically. The trie-related settings in `texmf.cnf` are ignored.
- Because there is no 'trie preparation' stage, language patterns never become frozen. This means that the primitive `\patterns` (and its Lua counterpart `language.patterns`) can be used at any time, not only in iniTEX.
- Only the string representation of `\patterns` and `\hyphenation` is stored in the format file. At format load time, they are simply re-evaluated. It follows that there is no real reason to preload languages in the format file. In fact, it is usually not a good idea to do so. It is much smarter to load patterns no sooner than the first time they are actually needed.
- LuaTEX uses the language-specific variables `\prehyphenchar` and `\posthyphenchar` in the creation of implicit discretionaries, instead of TEX82's `\hyphenchar`, and the values of the language-specific variables `\preexhyphenchar` and `\postexhyphenchar` for explicit discretionaries (instead of TEX82's empty discretionary).
- The value of the two counters related to hyphenation, `\hyphenpenalty` and `\exhyphenpenalty`, are now stored in the discretionary nodes. This permits a local overload for explicit `\discretionary` commands. The value current when the hyphenation pass is applied is used. When no callbacks are used this is compatible with traditional TEX. When you apply the Lua `language.hyphenate` function the current values are used.
- The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the `hyph_size` setting is not used either.

Because we store penalties in the disc node the `\discretionary` command has been extended to accept an optional penalty specification, so you can do the following:

```
\hsize1mm
1:foo{\hyphenpenalty 10000\discretionary{}{}{}bar\par
2:foo\discretionary penalty 10000 {}{}{}bar\par
3:foo\discretionary{}{}{}bar\par
```

This results in:

1:foobar

2:foobar

3:foo
bar

Inserted characters and ligatures inherit their attributes from the nearest glyph node item (usually the preceding one, but the following one for the items inserted at the left-hand side of a word).

Word boundaries are no longer implied by font switches, but by language switches. One word can have two separate fonts and still be hyphenated correctly (but it can not have two different languages, the `\setlanguage` command forces a word boundary).

All languages start out with `\prehyphenchar=`\-`, `\posthyphenchar=0`, `\preexhyphenchar=0` and `\postexhyphenchar=0`. When you assign the values of one of these four parameters, you are actually changing the settings for the current `\language`, this behaviour is compatible with `\patterns` and `\hyphenation`.

LuaTEX also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in TEX82). Longer words are ignored right now, but eventually either the limitation will be removed or perhaps it will become possible to silently ignore the excess characters (this is what happens in TEX82, but there the behaviour cannot be controlled).

If you are using the Lua function `language.hyphenate`, you should be aware that this function expects to receive a list of 'character' nodes. It will not operate properly in the presence of 'glyph', 'ligature', or 'ghost' nodes, nor does it know how to deal with kerning.

7.6 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, LuaTEX will process the list to convert the 'character' nodes into 'glyph' and 'ligature' nodes. This is actually done in two stages: first all ligatures are processed, then all kerning information is applied to the result list. But those two stages are somewhat dependent on each other: If the used font makes it possible to do so, the ligaturing stage adds virtual 'character' nodes to the word boundaries in the list. While doing so, it removes and interprets `\noboundary` nodes. The kerning stage deletes those word boundary items after it is done with them, and it does the same for 'ghost' nodes. Finally, at the end of the kerning stage, all remaining 'character' nodes are converted to 'glyph' nodes.

This separation is worth mentioning because, if you overrule from Lua only one of the two callbacks related to font handling, then you have to make sure you perform the tasks normally done by LuaTEX itself in order to make sure that the other, non-overruled, routine continues to function properly.

Although we could improve the situation the reality is that in modern OpenType fonts ligatures can be constructed in many ways: by replacing a sequence of characters by one glyph, or by selectively replacing individual glyphs, or by kerning, or any combination of this. Add to that contextual analysis and it will be clear that we have to let Lua do that job instead. The generic font handler that we provide (which is part of ConTEXt) distinguishes between base mode (which essentially is what we describe here and which delegates the task to TEX) and node mode (which deals with more complex fonts.

In so called base mode, where TEX does the work, the ligature construction (normally) goes in small steps. An `f` followed by an `f` becomes an `ff` ligatures and that one followed by an `i` can become a `ffi` ligature. The situation can be complicated by hyphenation points between these characters. When there are several in a ligature collapsing happens. Flag `"4000` in the `\hyphenationmode` variable determines if this happens lazy or greedy, i.e. the first hyphen wins or the last one does. In practice a ConTEXt user won't have to deal with this because most fonts are processed in node mode.

7.7 Breaking paragraphs into lines

This code is almost unchanged, but because of the above-mentioned changes with respect to discretionaries and ligatures, line breaking will potentially be different from traditional TEX. The actual line breaking code is still based on the TEX82 algorithms, and there can be no discretionaries inside of discretionaries. But, as patterns evolve and font handling can influence discretionaries, you need to be aware of the fact that long term consistency is not an engine matter only.

But that situation is now fairly common in LuaTEX, due to the changes to the ligaturing mechanism. And also, the LuaTEX discretionary nodes are implemented slightly different from the TEX82 nodes: the `no_break` text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list. In traditional TEX the discretionary node contains a counter indicating how many nodes to skip, but in LuaTEX we store the pre, post and replace text in the discretionary node.

The combined effect of these two differences is that LuaTEX does not always use all of the potential breakpoints in a paragraph, especially when fonts with many ligatures are used. Of course kerning also complicates matters here.

7.8 The `language` library

7.8.1 `new` and `id`

This library provides the interface to LuaTEX's structure representing a language, and the associated functions.

```
<language> l = language.new()
<language> l = language.new(<number> id)
```

This function creates a new userdata object. An object of type `<language>` is the first argument

to most of the other functions in the `language` library. These functions can also be used as if they were object methods, using the colon syntax. Without an argument, the next available internal id number will be assigned to this object. With argument, an object will be created that links to the internal language with that id number.

```
<number> n = language.id(<language> l)
```

The number returned is the internal `\language` id number this object refers to.

7.8.2 hyphenation

You can load exceptions with:

```
<string> n = language.hyphenation(<language> l)
language.hyphenation(<language> l, <string> n)
```

When no string is given (the first example) a string with all exceptions is returned.

7.8.3 clearhyphenation and clean

This either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 7.4.

```
language.clearhyphenation(<language> l)
```

This call clears the exception dictionary (string) for this language.

```
<string> n = language.clean(<language> l, <string> o)
<string> n = language.clean(<string> o)
```

This function creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 7.4. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

7.8.4 patterns and clearpatterns

```
<string> n = language.patterns(<language> l)
language.patterns(<language> l, <string> n)
```

This adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 7.4.

```
language.clearpatterns(<language> l)
```

This can be used to clear the pattern dictionary for a language.

7.8.5 hyphenationmin

This function sets (or gets) the value of the TEX parameter `\hyphenationmin`.

```
n = language.hyphenationmin(<language> l)
language.hyphenationmin(<language> l, <number> n)
```

7.8.6 [pre|post][ex|]hyphenchar

```
<number> n = language.prehyphenchar(<language> l)
language.prehyphenchar(<language> l, <number> n)

<number> n = language.posthyphenchar(<language> l)
language.posthyphenchar(<language> l, <number> n)
```

These two are used to get or set the 'pre-break' and 'post-break' hyphen characters for implicit hyphenation in this language. The intial values are decimal 45 (hyphen) and decimal 0 (indicating emptiness).

```
<number> n = language.preexhyphenchar(<language> l)
language.preexhyphenchar(<language> l, <number> n)

<number> n = language.postexhyphenchar(<language> l)
language.postexhyphenchar(<language> l, <number> n)
```

These gets or set the 'pre-break' and 'post-break' hyphen characters for explicit hyphenation in this language. Both are initially decimal 0 (indicating emptiness).

7.8.7 hyphenate

The next call inserts hyphenation points (discretionary nodes) in a node list. If `tail` is given as argument, processing stops on that node. Currently, `success` is always true if `head` (and `tail`, if specified) are proper nodes, regardless of possible other errors.

```
<boolean> success = language.hyphenate(<node> head)
<boolean> success = language.hyphenate(<node> head, <node> tail)
```

Hyphenation works only on 'characters', a special subtype of all the glyph nodes with the node subtype having the value 1. Glyph modes with different subtypes are not processed. See section 7.2 for more details.

7.8.8 [set|get]hjcode

The following two commands can be used to set or query hj codes:

```
language.sethjcode(<language> l, <number> char, <number> usedchar)
<number> usedchar = language.gethjcode(<language> l, <number> char)
```

When you set a hjcode the current sets get initialized unless the set was already initialized due to `\savinghyphcodes` being larger than zero.

8 Math

8.1 Traditional alongside OpenType

At this point there is no difference between LuaMetaTEX and LuaTEX with respect to math. The handling of mathematics in LuaTEX differs quite a bit from how TEX82 (and therefore pdfTEX) handles math. First, LuaTEX adds primitives and extends some others so that Unicode input can be used easily. Second, all of TEX82's internal special values (for example for operator spacing) have been made accessible and changeable via control sequences. Third, there are extensions that make it easier to use OpenType math fonts. And finally, there are some extensions that have been proposed or considered in the past that are now added to the engine.

8.2 Unicode math characters

Character handling is now extended up to the full Unicode range (the `\U` prefix), which is compatible with XETEX.

The math primitives from TEX are kept as they are, except for the ones that convert from input to math commands: `mathcode`, and `delcode`. These two now allow for a 21-bit character argument on the left hand side of the equals sign.

Some of the new LuaTEX primitives read more than one separate value. This is shown in the tables below by a plus sign.

The input for such primitives would look like this:

```
\def\overbrace{\Umathaccent 0 1 "23DE }
```

The altered TEX82 primitives are:

PRIMITIVE	MIN	MAX		MIN	MAX
\mathcode	0	10FFFF	=	0	8000
\delcode	0	10FFFF	=	0	FFFFFF

The unaltered ones are:

PRIMITIVE	MIN	MAX
\mathchardef	0	8000
\mathchar	0	7FFF
\mathaccent	0	7FFF
\delimiter	0	7FFFFFF
\radical	0	7FFFFFF

For practical reasons `\mathchardef` will silently accept values larger that `0x8000` and interpret it as `\Umathcharnumdef`. This is needed to satisfy older macro packages.

The following new primitives are compatible with XETEX:

PRIMITIVE	MIN	MAX		MIN	MAX
\Umathchardef	0+0+0	7+FF+10FFFF			
\Umathcharnumdef[5]	-80000000	7FFFFFFF			
\Umathcode	0	10FFFF	=	0+0+0	7+FF+10FFFF
\Udelcode	0	10FFFF	=	0+0	FF+10FFFF
\Umathchar	0+0+0	7+FF+10FFFF			
\Umathaccent	0+0+0	7+FF+10FFFF			
\Udelimiter	0+0+0	7+FF+10FFFF			
\Uradical	0+0	FF+10FFFF			
\Umathcharnum	-80000000	7FFFFFFF			
\Umathcodenum	0	10FFFF	=	-80000000	7FFFFFFF
\Udelcodenum	0	10FFFF	=	-80000000	7FFFFFFF

Specifications typically look like:

```
\Umathchardef\xx="1"0"456
\Umathcode   123="1"0"789
```

The new primitives that deal with delimiter-style objects do not set up a ‘large family’. Selecting a suitable size for display purposes is expected to be dealt with by the font via the `\Umathoperatorsize` parameter.

For some of these primitives, all information is packed into a single signed integer. For the first two (`\Umathcharnum` and `\Umathcodenum`), the lowest 21 bits are the character code, the 3 bits above that represent the math class, and the family data is kept in the topmost bits. This means that the values for math families 128–255 are actually negative. For `\Udelcodenum` there is no math class. The math family information is stored in the bits directly on top of the character code. Using these three commands is not as natural as using the two- and three-value commands, so unless you know exactly what you are doing and absolutely require the speedup resulting from the faster input scanning, it is better to use the verbose commands instead.

The `\Umathaccent` command accepts optional keywords to control various details regarding math accents. See section 8.6.2 below for details.

There are more new primitives and all of these will be explained in following sections:

PRIMITIVE	VALUE RANGE (IN HEX)
\Uroot	0 + 0–FF + 10FFFF
\Uoverdelimiter	0 + 0–FF + 10FFFF
\Uunderdelimiter	0 + 0–FF + 10FFFF
\Udelimiterover	0 + 0–FF + 10FFFF
\Udelimiterunder	0 + 0–FF + 10FFFF

8.3 Math styles

8.3.1 \mathstyle

It is possible to discover the math style that will be used for a formula in an expandable fashion (while the math list is still being read). To make this possible, LuaTEX adds the new primitive: `\mathstyle`. This is a 'convert command' like e.g. `\romannumeral`: its value can only be read, not set. Beware that contrary to LuaTEX this is now a proper number so you need to use `\number` o r`\the` in order to serialize it.

The returned value is between 0 and 7 (in math mode), or −1 (all other modes). For easy testing, the eight math style commands have been altered so that they can be used as numeric values, so you can write code like this:

```
\ifnum\mathstyle=\textstyle
    \message{normal text style}
\else \ifnum\mathstyle=\crampedtextstyle
    \message{cramped text style}
\fi \fi
```

Sometimes you won't get what you expect so a bit of explanation might help to understand what happens. When math is parsed and expanded it gets turned into a linked list. In a second pass the formula will be build. This has to do with the fact that in order to determine the automatically chosen sizes (in for instance fractions) following content can influence preceding sizes. A side effect of this is for instance that one cannot change the definition of a font family (and thereby reusing numbers) because the number that got used is stored and used in the second pass (so changing `\fam 12` mid-formula spoils over to preceding use of that family).

The style switching primitives like `\textstyle` are turned into nodes so the styles set there are frozen. The `\mathchoice` primitive results in four lists being constructed of which one is used in the second pass. The fact that some automatic styles are not yet known also means that the `\mathstyle` primitive expands to the current style which can of course be different from the one really used. It's a snapshot of the first pass state. As a consequence in the following example you get a style number (first pass) typeset that can actually differ from the used style (second pass). In the case of a math choice used ungrouped, the chosen style is used after the choice too, unless you group.

```
[a:\number\mathstyle]\quad
\bgroup
\mathchoice
    {\bf \scriptstyle       (x:d :\number\mathstyle)}
    {\bf \scriptscriptstyle (x:t :\number\mathstyle)}
    {\bf \scriptscriptstyle (x:s :\number\mathstyle)}
    {\bf \scriptscriptstyle (x:ss:\number\mathstyle)}
\egroup
\quad[b:\number\mathstyle]\quad
\mathchoice
    {\bf \scriptstyle       (y:d :\number\mathstyle)}
```

```
    {\bf \scriptscriptstyle (y:t :\number\mathstyle)}
    {\bf \scriptscriptstyle (y:s :\number\mathstyle)}
    {\bf \scriptscriptstyle (y:ss:\number\mathstyle)}
\quad[c:\number\mathstyle]\quad
\bgroup
\mathchoice
    {\bf \scriptstyle       (z:d :\number\mathstyle)}
    {\bf \scriptscriptstyle (z:t :\number\mathstyle)}
    {\bf \scriptscriptstyle (z:s :\number\mathstyle)}
    {\bf \scriptscriptstyle (z:ss:\number\mathstyle)}
\egroup
\quad[d:\number\mathstyle]
```

This gives:

$[a:0]$ (**x**:**d**:**4**) $[b:0]$ (**y**:**d**:**4**) $[c:0]$ (**z**:**s**:**6**) $[d:0]$

$[a:2]$ (**x**:**t**:**6**) $[b:2]$ (**y**:**t**:**6**) $[c:2]$ (**z**:**ss**:**6**) $[d:2]$

Using `\begingroup` ... `\endgroup` instead gives:

$[a:0]$ (**x**:**d**:**4**) $[b:0]$ (**y**:**s**:**6**) $[c:0]$ (**z**:**ss**:**6**) $[d:0]$

$[a:2]$ (**x**:**t**:**6**) $[b:2]$ (**y**:**ss**:**6**) $[c:2]$ (**z**:**ss**:**6**) $[d:2]$

This might look wrong but it's just a side effect of `\mathstyle` expanding to the current (first pass) style and the number being injected in the list that gets converted in the second pass. It all makes sense and it illustrates the importance of grouping. In fact, the math choice style being effective afterwards has advantages. It would be hard to get it otherwise.

8.3.2 \Ustack

There are a few math commands in TEX where the style that will be used is not known straight from the start. These commands (`\over`, `\atop`, `\overwithdelims`, `\atopwithdelims`) would therefore normally return wrong values for `\mathstyle`. To fix this, LuaTEX introduces a special prefix command: `\Ustack`:

```
$\Ustack {a \over b}$
```

The `\Ustack` command will scan the next brace and start a new math group with the correct (numerator) math style.

8.3.3 The new \cramped ...style commands

LuaTEX has four new primitives to set the cramped math styles directly:

```
\crampeddisplaystyle
\crampedtextstyle
```

```
\crampedscriptstyle
\crampedscriptscriptstyle
```

These additional commands are not all that valuable on their own, but they come in handy as arguments to the math parameter settings that will be added shortly.

In Eijkhouts "TEX by Topic" the rules for handling styles in scripts are described as follows:

- In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are in script style.
- Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.
- In an `..\over..` formula in any style the numerator and denominator are taken from the next smaller style.
- The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- Formulas under a `\sqrt` or `\overline` are in cramped style.

In LuaTEX one can set the styles in more detail which means that you sometimes have to set both normal and cramped styles to get the effect you want. (Even) if we force styles in the script using `\scriptstyle` and `\crampedscriptstyle` we get this:

STYLE	EXAMPLE
default	$b^{x=x\,x}_{x=x\,x}$
script	$b^{x=x\,x}_{x=x\,x}$
crampedscript	$b^{x=x\,x}_{x=x\,x}$

Now we set the following parameters

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
```

This gives a different result:

STYLE	EXAMPLE
default	$b^{x\quad=x\quad x}_{x=x\,x}$
script	$b^{x\quad=x\quad x}_{x\quad=x\quad x}$
crampedscript	$b^{x=x\,x}_{x=x\,x}$

But, as this is not what is expected (visually) we should say:

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
\Umathordrelspacing\crampedscriptstyle=30mu
\Umathordordspacing\crampedscriptstyle=30mu
```

Now we get:

STYLE	EXAMPLE
default	$b_{x}^{x} = x \quad x$
script	$b_{x}^{x} = x \quad x$
crampedscript	$b_{x}^{x} = x \quad x$

8.4 Math parameter settings

8.4.1 Many new \Umath* primitives

In LuaTEX, the font dimension parameters that TEX used in math typesetting are now accessible via primitive commands. In fact, refactoring of the math engine has resulted in turning some hard codes properties into parameters.

PRIMITIVE NAME	DESCRIPTION
\Umathquad	the width of 18 mu's
\Umathaxis	height of the vertical center axis of the math formula above the baseline
\Umathoperatorsize	minimum size of large operators in display mode
\Umathoverbarkern	vertical clearance above the rule
\Umathoverbarrule	the width of the rule
\Umathoverbarvgap	vertical clearance below the rule
\Umathunderbarkern	vertical clearance below the rule
\Umathunderbarrule	the width of the rule
\Umathunderbarvgap	vertical clearance above the rule
\Umathradicalkern	vertical clearance above the rule
\Umathradicalrule	the width of the rule
\Umathradicalvgap	vertical clearance below the rule
\Umathradicaldegreebefore	the forward kern that takes place before placement of the radical degree
\Umathradicaldegreeafter	the backward kern that takes place after placement of the radical degree
\Umathradicaldegreeraise	this is the percentage of the total height and depth of the radical sign that the degree is raised by; it is expressed in percents, so 60% is expressed as the integer 60
\Umathstackvgap	vertical clearance between the two elements in an \atop stack
\Umathstacknumup	numerator shift upward in \atop stack
\Umathstackdenomdown	denominator shift downward in \atop stack
\Umathfractionrule	the width of the rule in a \over
\Umathfractionnumvgap	vertical clearance between the numerator and the rule
\Umathfractionnumup	numerator shift upward in \over
\Umathfractiondenomvgap	vertical clearance between the denominator and the rule
\Umathfractiondenomdown	denominator shift downward in \over
\Umathfractiondelsize	minimum delimiter size for \...withdelims
\Umathlimitabovevgap	vertical clearance for limits above operators

\Umathlimitabovebgap	vertical baseline clearance for limits above operators
\Umathlimitabovekern	space reserved at the top of the limit
\Umathlimitbelowvgap	vertical clearance for limits below operators
\Umathlimitbelowbgap	vertical baseline clearance for limits below operators
\Umathlimitbelowkern	space reserved at the bottom of the limit
\Umathoverdelimitervgap	vertical clearance for limits above delimiters
\Umathoverdelimiterbgap	vertical baseline clearance for limits above delimiters
\Umathunderdelimitervgap	vertical clearance for limits below delimiters
\Umathunderdelimiterbgap	vertical baseline clearance for limits below delimiters
\Umathsubshiftdrop	subscript drop for boxes and subformulas
\Umathsubshiftdown	subscript drop for characters
\Umathsupshiftdrop	superscript drop (raise, actually) for boxes and subformulas
\Umathsupshiftup	superscript raise for characters
\Umathsubsupshiftdown	subscript drop in the presence of a superscript
\Umathsubtopmax	the top of standalone subscripts cannot be higher than this above the baseline
\Umathsupbottommin	the bottom of standalone superscripts cannot be less than this above the baseline
\Umathsupsubbottommax	the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
\Umathsubsupvgap	vertical clearance between super- and subscript
\Umathspacebeforescript	additional space added before a super- or subprescript (bonus setting)
\Umathspaceafterscript	additional space added after a super- or subscript
\Umathconnectoroverlapmin	minimum overlap between parts in an extensible recipe

Each of the parameters in this section can be set by a command like this:

```
\Umathquad\displaystyle=1em
```

they obey grouping, and you can use `\the\Umathquad\displaystyle` if needed.

8.4.2 Font-based math parameters

While it is nice to have these math parameters available for tweaking, it would be tedious to have to set each of them by hand. For this reason, LuaTEX initializes a bunch of these parameters whenever you assign a font identifier to a math family based on either the traditional math font dimensions in the font (for assignments to math family 2 and 3 using tfm-based fonts like `cmsy` and `cmex`), or based on the named values in a potential `MathConstants` table when the font is loaded via Lua. If there is a `MathConstants` table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the `MathConstants` tables in the last assigned family sets all parameters.

In the table below, the one-letter style abbreviations and symbolic tfm font dimension names match those used in the TEXbook. Assignments to `\textfont` set the values for the cramped and uncramped display and text styles, `\scriptfont` sets the script styles, and `\scriptscriptfont` sets the scriptscript styles, so we have eight parameters for three font sizes. In the tfm case,

assignments only happen in family 2 and family 3 (and of course only for the parameters for which there are font dimensions).

Besides the parameters below, LuaTEX also looks at the 'space' font dimension parameter. For math fonts, this should be set to zero.

VARIABLE / STYLE	TFM / OPENTYPE
\Umathaxis	axis_height AxisHeight
[6] **\Umathoperatorsize** D, D'	– DisplayOperatorMinHeight
[9] **\Umathfractiondelsize** D, D'	delim1 FractionDelimiterDisplayStyleSize
[9] **\Umathfractiondelsize** T, T', S, S', SS, SS'	delim2 FractionDelimiterSize
\Umathfractiondenomdown D, D'	denom1 FractionDenominatorDisplayStyleShiftDown
\Umathfractiondenomdown T, T', S, S', SS, SS'	denom2 FractionDenominatorShiftDown
\Umathfractiondenomvgap D, D'	3*default_rule_thickness FractionDenominatorDisplayStyleGapMin
\Umathfractiondenomvgap T, T', S, S', SS, SS'	default_rule_thickness FractionDenominatorGapMin
\Umathfractionnumup D, D'	num1 FractionNumeratorDisplayStyleShiftUp
\Umathfractionnumup T, T', S, S', SS, SS'	num2 FractionNumeratorShiftUp
\Umathfractionnumvgap D, D'	3*default_rule_thickness FractionNumeratorDisplayStyleGapMin
\Umathfractionnumvgap T, T', S, S', SS, SS'	default_rule_thickness FractionNumeratorGapMin
\Umathfractionrule	default_rule_thickness FractionRuleThickness
\Umathskewedfractionhgap	math_quad/2 SkewedFractionHorizontalGap
\Umathskewedfractionvgap	math_x_height SkewedFractionVerticalGap
\Umathlimitabovebgap	big_op_spacing3 UpperLimitBaselineRiseMin
[1] **\Umathlimitabovekern**	big_op_spacing5 0
\Umathlimitabovevgap	big_op_spacing1 UpperLimitGapMin
\Umathlimitbelowbgap	big_op_spacing4

		LowerLimitBaselineDropMin
[1]	**\Umathlimitbelowkern**	big_op_spacing5 0
	\Umathlimitbelowvgap	big_op_spacing2 LowerLimitGapMin
	\Umathoverdelimitervgap	big_op_spacing1 StretchStackGapBelowMin
	\Umathoverdelimiterbgap	big_op_spacing3 StretchStackTopShiftUp
	\Umathunderdelimitervgap	big_op_spacing2 StretchStackGapAboveMin
	\Umathunderdelimiterbgap	big_op_spacing4 StretchStackBottomShiftDown
	\Umathoverbarkern	default_rule_thickness OverbarExtraAscender
	\Umathoverbarrule	default_rule_thickness OverbarRuleThickness
	\Umathoverbarvgap	3*default_rule_thickness OverbarVerticalGap
[1]	**\Umathquad**	math_quad <font_size(f)>
	\Umathradicalkern	default_rule_thickness RadicalExtraAscender
[2]	**\Umathradicalrule**	<not set> RadicalRuleThickness
[3]	**\Umathradicalvgap**	default_rule_thickness+abs(math_x_height)/4
	D, D'	RadicalDisplayStyleVerticalGap
[3]	**\Umathradicalvgap**	default_rule_thickness+abs(default_rule_thickness)/4
	T, T', S, S', SS, SS'	RadicalVerticalGap
[2]	**\Umathradicaldegreebefore**	<not set> RadicalKernBeforeDegree
[2]	**\Umathradicaldegreeafter**	<not set> RadicalKernAfterDegree
[2,7]	**\Umathradicaldegreeraise**	<not set> RadicalDegreeBottomRaisePercent
[4]	**\Umathspaceafterscript**	script_space SpaceAfterScript
	\Umathstackdenomdown	denom1
	D, D'	StackBottomDisplayStyleShiftDown
	\Umathstackdenomdown	denom2
	T, T', S, S', SS, SS'	StackBottomShiftDown
	\Umathstacknumup	num1

	D, D'	StackTopDisplayStyleShiftUp
	\Umathstacknumup	num3
	T, T', S, S', SS, SS'	StackTopShiftUp
	\Umathstackvgap	7*default_rule_thickness
	D, D'	StackDisplayStyleGapMin
	\Umathstackvgap	3*default_rule_thickness
	T, T', S, S', SS, SS'	StackGapMin
	\Umathsubshiftdown	sub1
		SubscriptShiftDown
	\Umathsubshiftdrop	sub_drop
		SubscriptBaselineDropMin
[8]	**\Umathsubsupshiftdown**	–
		SubscriptShiftDownWithSuperscript
	\Umathsubtopmax	abs(math_x_height*4)/5
		SubscriptTopMax
	\Umathsubsupvgap	4*default_rule_thickness
		SubSuperscriptGapMin
	\Umathsupbottommin	abs(math_x_height/4)
		SuperscriptBottomMin
	\Umathsupshiftdrop	sup_drop
		SuperscriptBaselineDropMax
	\Umathsupshiftup	sup1
	D	SuperscriptShiftUp
	\Umathsupshiftup	sup2
	T, S, SS,	SuperscriptShiftUp
	\Umathsupshiftup	sup3
	D', T', S', SS'	SuperscriptShiftUpCramped
	\Umathsupsubbottommax	abs(math_x_height*4)/5
		SuperscriptBottomMaxWithSubscript
	\Umathunderbarkern	default_rule_thickness
		UnderbarExtraDescender
	\Umathunderbarrule	default_rule_thickness
		UnderbarRuleThickness
	\Umathunderbarvgap	3*default_rule_thickness
		UnderbarVerticalGap
[5]	**\Umathconnectoroverlapmin**	0
		MinConnectorOverlap

Note 1: OpenType fonts set `\Umathlimitabovekern` and `\Umathlimitbelowkern` to zero and set `\Umathquad` to the font size of the used font, because these are not supported in the `MATH` table,

Note 2: Traditional tfm fonts do not set `\Umathradicalrule` because TEX82 uses the height of the radical instead. When this parameter is indeed not set when LuaTEX has to typeset a radical, a backward compatibility mode will kick in that assumes that an oldstyle TEX font is used.

Also, they do not set `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. These are then automatically initialized to 5/18quad, −10/18quad, and 60.

Note 3: If tfm fonts are used, then the `\Umathradicalvgap` is not set until the first time LuaTEX has to typeset a formula because this needs parameters from both family 2 and family 3. This provides a partial backward compatibility with TEX82, but that compatibility is only partial: once the `\Umathradicalvgap` is set, it will not be recalculated any more.

Note 4: When tfm fonts are used a similar situation arises with respect to `\Umathspaceafterscript`: it is not set until the first time LuaTEX has to typeset a formula. This provides some backward compatibility with TEX82. But once the `\Umathspaceafterscript` is set, `\scriptspace` will never be looked at again.

Note 5: Traditional tfm fonts set `\Umathconnectoroverlapmin` to zero because TEX82 always stacks extensibles without any overlap.

Note 6: The `\Umathoperatorsize` is only used in `\displaystyle`, and is only set in OpenType fonts. In tfm font mode, it is artificially set to one scaled point more than the initial attempt's size, so that always the 'first next' will be tried, just like in TEX82.

Note 7: The `\Umathradicaldegreeraise` is a special case because it is the only parameter that is expressed in a percentage instead of a number of scaled points.

Note 8: `SubscriptShiftDownWithSuperscript` does not actually exist in the 'standard' OpenType math font Cambria, but it is useful enough to be added.

Note 9: `FractionDelimiterDisplayStyleSize` and `FractionDelimiterSize` do not actually exist in the 'standard' OpenType math font Cambria, but were useful enough to be added.

8.5 Math spacing

8.5.1 Setting inline surrounding space with `\mathsurround` and `\mathsurroundskip`

Inline math is surrounded by (optional) `\mathsurround` spacing but that is a fixed dimension. There is now an additional parameter `\mathsurroundskip`. When set to a non-zero value (or zero with some stretch or shrink) this parameter will replace `\mathsurround`. By using an additional parameter instead of changing the nature of `\mathsurround`, we can remain compatible. In the meantime a bit more control has been added via `\mathsurroundmode`. This directive can take 6 values with zero being the default behaviour.

```
\mathsurround    10pt
\mathsurroundskip20pt
```

MODE	XXX	X X X	EFFECT
0	x *x* x	x *x* x	obey `\mathsurround` when `\mathsurroundskip` is 0pt
1	x *x*x	x *x* x	only add skip to the left
2	x*x* x	x *x* x	only add skip to the right
3	x *x* x	x *x* x	add skip to the left and right

4	x x x x x x	ignore the skip setting, obey `\mathsurround`
5	xxx x x x	disable all spacing around math
6	x x x x x x	only apply `\mathsurroundskip` when also spacing
7	x x x x x x	only apply `\mathsurroundskip` when no spacing

Anything more fancy, like checking the beginning or end of a paragraph (or edges of a box) would not be robust anyway. If you want that you can write a callback that runs over a list and analyzes a paragraph. Actually, in that case you could also inject glue (or set the properties of a math node) explicitly. So, these modes are in practice mostly useful for special purposes and experiments (they originate in a tracker item). Keep in mind that this glue is part of the math node and not always treated as normal glue: it travels with the begin and end math nodes. Also, method 6 and 7 will zero the skip related fields in a node when applicable in the first occasion that checks them (linebreaking or packaging).

8.5.2 Pairwise spacing and `\Umath...spacing` commands

Besides the parameters mentioned in the previous sections, there are also 64 new primitives to control the math spacing table (as explained in Chapter 18 of the TEXbook). The primitive names are a simple matter of combining two math atom types, but for completeness' sake, here is the whole list:

```
\Umathordordspacing
\Umathordopspacing
\Umathordbinspacing
\Umathordrelspacing
\Umathordopenspacing
\Umathordclosespacing
\Umathordpunctspacing
\Umathordinnerspacing
\Umathopordspacing
\Umathopopspacing
\Umathopbinspacing
\Umathoprelspacing
\Umathopopenspacing
\Umathopclosespacing
\Umathoppunctspacing
\Umathopinnerspacing
\Umathbinordspacing
\Umathbinopspacing
\Umathbinbinspacing
\Umathbinrelspacing
\Umathbinopenspacing
\Umathbinclosespacing
\Umathbinpunctspacing
\Umathbininnerspacing
\Umathrelordspacing
\Umathrelopspacing
\Umathrelbinspacing
\Umathrelrelspacing
\Umathrelopenspacing
\Umathrelclosespacing
\Umathrelpunctspacing
\Umathrelinnerspacing
\Umathopenordspacing
\Umathopenopspacing
\Umathopenbinspacing
\Umathopenrelspacing
\Umathopenopenspacing
\Umathopenclosespacing
\Umathopenpunctspacing
\Umathopeninnerspacing
\Umathcloseordspacing
\Umathcloseopspacing
\Umathclosebinspacing
\Umathcloserelspacing
\Umathcloseopenspacing
\Umathcloseclosespacing
\Umathclosepunctspacing
\Umathcloseinnerspacing
\Umathpunctordspacing
\Umathpunctopspacing
```

```
\Umathpunctbinspacing                \Umathinneropspacing
\Umathpunctrelspacing                \Umathinnerbinspacing
\Umathpunctopenspacing               \Umathinnerrelspacing
\Umathpunctclosespacing              \Umathinneropenspacing
\Umathpunctpunctspacing              \Umathinnerclosespacing
\Umathpunctinnerspacing              \Umathinnerpunctspacing
\Umathinnerordspacing                \Umathinnerinnerspacing
```

These parameters are of type `\muskip`, so setting a parameter can be done like this:

```
\Umathopordspacing\displaystyle=4mu plus 2mu
```

They are all initialized by `initex` to the values mentioned in the table in Chapter 18 of the TEXbook.

Note 1: For ease of use as well as for backward compatibility, `\thinmuskip`, `\medmuskip` and `\thickmuskip` are treated specially. In their case a pointer to the corresponding internal parameter is saved, not the actual `\muskip` value. This means that any later changes to one of these three parameters will be taken into account.

Note 2: Careful readers will realise that there are also primitives for the items marked * in the TEXbook. These will not actually be used as those combinations of atoms cannot actually happen, but it seemed better not to break orthogonality. They are initialized to zero.

8.5.3 Local `\frozen` settings with

Math is processed in two passes. The first pass is needed to intercept for instance `\over`, one of the few TEX commands that actually has a preceding argument. There are often lots of curly braces used in math and these can result in a nested run of the math sub engine. However, you need to be aware of the fact that some properties are kind of global to a formula and the last setting (for instance a family switch) wins. This also means that a change (or again, the last one) in math parameters affects the whole formula. In LuaMetaTEX we have changed this model a bit. One can argue that this introduces an incompatibility but it's hard to imagine a reason for setting the parameters at the end of a formula run and assume that they also influence what goes in front.

```
$
                                                x \Usubscript  {-}
     \frozen\Umathsubshiftdown\textstyle  0pt   x \Usubscript  {0}
    {\frozen\Umathsubshiftdown\textstyle  5pt   x \Usubscript  {5}}
                                                x \Usubscript  {0}
    {\frozen\Umathsubshiftdown\textstyle 15pt   x \Usubscript {15}}
                                                x \Usubscript  {0}
    {\frozen\Umathsubshiftdown\textstyle 20pt   x \Usubscript {20}}
                                                x \Usubscript  {0}
     \frozen\Umathsubshiftdown\textstyle 10pt   x \Usubscript {10}
                                                x \Usubscript  {0}
$
```

The `\frozen` prefix does the magic: it injects information in the math list about the set parameter.

In LuaTEX 1.10+ the last setting, the `10pt` drop wins, but in LuaMetaTEX you will see each local setting taking effect. The implementation uses a new node type, parameters nodes, so you might encounter these in an unprocessed math list. The result looks as follows:

$$x\_x_0x_5x_0x_{15}x_0x_{20}x_0x_{10}x_0$$

8.5.4 Checking a state with `\ifmathparameter`

When you adapt math parameters it might make sense to see if they are set at all. When a parameter is unset its value has the maximum dimension value and you might for instance mistakenly multiply that value to open up things a bit, which gives unexpected side effects. For that reason there is a convenient checker: `\ifmathparameter`. This test primitive behaves like an `\ifcase`, with:

VALUE	MEANING
0	the parameter value is zero
1	the parameter is set
2	the parameter is unset

8.5.5 Skips around display math and `\mathdisplayskipmode`

The injection of `\abovedisplayskip` and `\belowdisplayskip` is not symmetrical. An above one is always inserted, also when zero, but the below is only inserted when larger than zero. Especially the latter makes it sometimes hard to fully control spacing. Therefore LuaTEX comes with a new directive: `\mathdisplayskipmode`. The following values apply:

VALUE	MEANING
0	normal TEX behaviour
1	always (same as 0)
2	only when not zero
3	never, not even when not zero

8.5.6 Nolimit correction with `\mathnolimitsmode`

There are two extra math parameters `\Umathnolimitsupfactor` and `\Umathnolimitsubfactor` that were added to provide some control over how limits are spaced (for example the position of super and subscripts after integral operators). They relate to an extra parameter `\mathnolimitsmode`. The half corrections are what happens when scripts are placed above and below. The problem with italic corrections is that officially that correction italic is used for above/below placement while advanced kerns are used for placement at the right end. The question is: how often is this implemented, and if so, do the kerns assume correction too. Anyway, with this parameter one can control it.

	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	${}_1\int^0$
mode	0	1	2	3	4	8000
superscript	0	font	0	0	+ic/2	0
subscript	-ic	font	0	-ic/2	-ic/2	8000ic/1000

When the mode is set to one, the math parameters are used. This way a macro package writer can decide what looks best. Given the current state of fonts in ConTEXt we currently use mode 1 with factor 0 for the superscript and 750 for the subscripts. Positive values are used for both parameters but the subscript shifts to the left. A `\mathnolimitsmode` larger that 15 is considered to be a factor for the subscript correction. This feature can be handy when experimenting.

8.5.7 Controlling math italic mess with `\mathitalicsmode`

The `\mathitalicsmode` parameter can be set to 1 to force italic correction before noads that represent some more complex structure (read: everything that is not an ord, bin, rel, open, close, punct or inner). We show a Cambria example.

`\mathitalicsmode = 0` $\left|T^1\right|$ $|T|$ $T+1$ $T\frac{1}{2}$ $T\sqrt{1}$
`\mathitalicsmode = 1` $\left|T^1\right|$ $|T|$ $T+1$ $T\frac{1}{2}$ $T\sqrt{1}$

This kind of parameters relate to the fact that italic correction in OpenType math is bound to fuzzy rules. So, control is the solution.

8.5.8 Influencing script kerning with `\mathscriptboxmode`

If you want to typeset text in math macro packages often provide something `\text` which obeys the script sizes. As the definition can be anything there is a good chance that the kerning doesn't come out well when used in a script. Given that the first glyph ends up in an `\hbox` we have some control over this. And, as a bonus we also added control over the normal sublist kerning. The `\mathscriptboxmode` parameter defaults to 1.

VALUE	MEANING
0	forget about kerning
1	kern math sub lists with a valid glyph
2	also kern math sub boxes that have a valid glyph
3	only kern math sub boxes with a boundary node present

Here we show some examples. Of course this doesn't solve all our problems, if only because some fonts have characters with bounding boxes that compensate for italics, while other fonts can lack kerns.

	`$T_{\tf fluff}$`	`$T_{\tf fluff}$`	`$T_{\text{fluff}}$`	`$T_{\text{fluff}}$`	`$T_{\text{\boundary1 fluff}}$`
	mode 0	**mode 1**	**mode 1**	**mode 2**	**mode 3**
modern	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
lucidaot	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
pagella	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}

cambria	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
dejavu	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}

Kerning between a character subscript is controlled by `\mathscriptcharmode` which also defaults to 1.

Here is another example. Internally we tag kerns as italic kerns or font kerns where font kerns result from the staircase kern tables. In 2018 fonts like Latin Modern and Pagella rely on cheats with the boundingbox, Cambria uses staircase kerns and Lucida a mixture. Depending on how fonts evolve we might add some more control over what one can turn on and off.

normal	modern	T_{f}	γ_{e}	γ_{ee}	T_{fluff}
	pagella	T_{f}	γ_{e}	γ_{ee}	T_{fluff}
	cambria	T_{f}	γ_{e}	γ_{ee}	T_{fluff}
	lucidaot	T_{f}	γ_{e}	γ_{ee}	T_{fluff}
bold	modern	$\boldsymbol{T_{f}}$	$\boldsymbol{\gamma_{e}}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	pagella	$\boldsymbol{T_{f}}$	$\boldsymbol{\gamma_{e}}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	cambria	$\boldsymbol{T_{f}}$	$\boldsymbol{\gamma_{e}}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	lucidaot	$\boldsymbol{T_{f}}$	$\boldsymbol{\gamma_{e}}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$

8.5.9 Forcing fixed scripts with `\mathscriptsmode`

We have three parameters that are used for this fixed anchoring:

PARAMETER	REGISTER
d	`\Umathsubshiftdown`
u	`\Umathsupshiftup`
s	`\Umathsubsupshiftdown`

When we set `\mathscriptsmode` to a value other than zero these are used for calculating fixed positions. This is something that is needed for instance for chemistry. You can manipulate the mentioned variables to achieve different effects.

MODE	DOWN	UP	EXAMPLE
0	dynamic	dynamic	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$
1	d	u	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$
2	s	u	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$
3	s	$u + s - d$	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$
4	$d + (s - d)/2$	$u + (s - d)/2$	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$
5	d	$u + s - d$	$\mathrm{CH}_2 + \mathrm{CH}_2^+ + \mathrm{CH}_2^2$

The value of this parameter obeys grouping but applies to the whole current formula.

8.5.10 Penalties: \mathpenaltiesmode

Only in inline math penalties will be added in a math list. You can force penalties (also in display math) by setting:

```
\mathpenaltiesmode = 1
```

This primnitive is not really needed in LuaTEX because you can use the callback `mlist_to_hlist` to force penalties by just calling the regular routine with forced penalties. However, as part of opening up and control this primitive makes sense. As a bonus we also provide two extra penalties:

```
\prebinoppenalty = -100 % example value
\prerelpenalty   =  900 % example value
```

They default to inifinite which signals that they don't need to be inserted. When set they are injected before a binop or rel noad. This is an experimental feature.

8.5.11 Equation spacing: \matheqnogapstep

By default TEX will add one quad between the equation and the number. This is hard coded. A new primitive can control this:

```
\matheqnogapstep = 1000
```

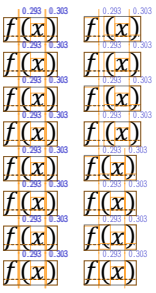
Because a math quad from the math text font is used instead of a dimension, we use a step to control the size. A value of zero will suppress the gap. The step is divided by 1000 which is the usual way to mimmick floating point factors in TEX.

8.6 Math constructs

8.6.1 Unscaled fences and \mathdelimitersmode

The `\mathdelimitersmode` primitive is experimental and deals with the following (potential) problems. Three bits can be set. The first bit prevents an unwanted shift when the fence symbol is not scaled (a cambria side effect). The second bit forces italic correction between a preceding character ordinal and the fenced subformula, while the third bit turns that subformula into an ordinary so that the same spacing applies as with unfenced variants. Here we show Cambria (with `\mathitalicsmode` enabled).

```
\mathdelimitersmode = 0
\mathdelimitersmode = 1
\mathdelimitersmode = 2
\mathdelimitersmode = 3
\mathdelimitersmode = 4
\mathdelimitersmode = 5
\mathdelimitersmode = 6
\mathdelimitersmode = 7
```



So, when set to 7 fenced subformulas with unscaled delimiters come out the same as unfenced ones. This can be handy for cases where one is forced to use `\left` and `\right` always because of unpredictable content. As said, it's an experimental feature (which somehow fits in the exceptional way fences are dealt with in the engine). The full list of flags is given in the next table:

VALUE	MEANING
"01	don't apply the usual shift
"02	apply italic correction when possible
"04	force an ordinary subformula
"08	no shift when a base character
"10	only shift when an extensible

The effect can depend on the font (and for Cambria one can use for instance `"16`).

Sometimes you might want to act upon the size of a delimiter, something that is not really possible because of the fact that they are calculated *after* most has been typeset already. For this we have two keyword: `phantom` and `void`. In both cases the symbol is replaced by an empty rule, in the first case all three dimensions are preserved in the last case only the height and depth.

```
\startformula
    x\mathlimop{\Uvextensible         \Udelimiter 5 0 "222B}_1^2 x
\stopformula
\vskip-9ex
\startformula \red
    x\mathlimop{\Uvextensible phantom \Udelimiter 5 0 "222B}_1^2 x
\stopformula
\vskip-9ex
\startformula \blue
    x\mathlimop{\Uvextensible void    \Udelimiter 5 0 "222B}_1^2 x
\stopformula
```

In typeset form this looks like:

$$x \int_1^2 x$$

8.6.2 Accent handling with `\Umathaccent`

LuaTEX supports both top accents and bottom accents in math mode, and math accents stretch automatically (if this is supported by the font the accent comes from, of course). Bottom and combined accents as well as fixed-width math accents are controlled by optional keywords following `\Umathaccent`.

The keyword `bottom` after `\Umathaccent` signals that a bottom accent is needed, and the keyword `both` signals that both a top and a bottom accent are needed (in this case two accents need to be specified, of course).

Then the set of three integers defining the accent is read. This set of integers can be prefixed by

the fixed keyword to indicate that a non-stretching variant is requested (in case of both accents, this step is repeated).

A simple example:

```
\Umathaccent both fixed 0 0 "20D7 fixed 0 0 "20D7 {example}
```

If a math top accent has to be placed and the accentee is a character and has a non-zero top_accent value, then this value will be used to place the accent instead of the \skewchar kern used by TEX82.

The top_accent value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own top_accent line coincides with the one from the accentee. If the top_accent value of the accent is zero, then half the width of the accent followed by its italic correction is used instead.

The vertical placement of a top accent depends on the x_height of the font of the accentee (as explained in the TEXbook), but if a value turns out to be zero and the font had a MathConstants table, then AccentBaseHeight is used instead.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

Possible locations are top, bottom, both and center. When no location is given top is assumed. An additional parameter fraction can be specified followed by a number; a value of for instance 1200 means that the criterium is 1.2 times the width of the nucleus. The fraction only applies to the stepwise selected shapes and is mostly meant for the overlay location. It also works for the other locations but then it concerns the width.

8.6.3 Building radicals with \Uradical and \Uroot

The new primitive \Uroot allows the construction of a radical noad including a degree field. Its syntax is an extension of \Uradical:

```
\Uradical <fam integer> <char integer> <radicand>
\Uroot    <fam integer> <char integer> <degree> <radicand>
```

The placement of the degree is controlled by the math parameters \Umathradicaldegreebefore, \Umathradicaldegreeafter, and \Umathradicaldegreeraise. The degree will be typeset in \scriptscriptstyle.

8.6.4 Super- and subscripts

The character fields in a Lua-loaded OpenType math font can have a ‘mathkern’ table. The format of this table is the same as the ‘mathkern’ table that is returned by the fontloader library, except that all height and kern values have to be specified in actual scaled points.

When a super- or subscript has to be placed next to a math item, LuaTEX checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OpenType fonts (as opposed to legacy TEX fonts), then LuaTEX

will use the OpenType math algorithm for deciding on the horizontal placement of the super- or subscript.

This works as follows:

- The vertical position of the script is calculated.
- The default horizontal position is flat next to the base character.
- For superscripts, the italic correction of the base character is added.
- For a superscript, two vertical values are calculated: the bottom of the script (after shifting up), and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.
- For each of these two locations:
  - find the math kern value at this height for the base (for a subscript placement, this is the bottom_right corner, for a superscript placement the top_right corner)
  - find the math kern value at this height for the script (for a subscript placement, this is the top_left corner, for a superscript placement the bottom_left corner)
  - add the found values together to get a preliminary result.
- The horizontal kern to be applied is the smallest of the two results from previous step.

The math kern value at a specific height is the kern value that is specified by the next higher height and kern pair, or the highest one in the character (if there is no value high enough in the character), or simply zero (if the character has no math kern pairs at all).

8.6.5 Scripts on extensibles: \Uunderdelimiter, \Uoverdelimiter, \Udelimiterover, \Udelimiterunder and \Uhextensible

The primitives `\Uunderdelimiter` and `\Uoverdelimiter` allow the placement of a subscript or superscript on an automatically extensible item and `\Udelimiterunder` and `\Udelimiterover` allow the placement of an automatically extensible item as a subscript or superscript on a nucleus. The input:

```
$\Uoverdelimiter  0 "2194 {\hbox{\strut  overdelimiter}}$
$\Uunderdelimiter 0 "2194 {\hbox{\strut underdelimiter}}$
$\Udelimiterover  0 "2194 {\hbox{\strut  delimiterover}}$
$\Udelimiterunder 0 "2194 {\hbox{\strut delimiterunder}}$
```

will render this:

overdelimiter delimiterover delimiterunder
underdelimiter

The vertical placements are controlled by `\Umathunderdelimiterbgap`, `\Umathunderdelimitervgap`, `\Umathoverdelimiterbgap`, and `\Umathoverdelimitervgap` in a similar way as limit placements on large operators. The superscript in `\Uoverdelimiter` is typeset in a suitable scripted style, the subscript in `\Uunderdelimiter` is cramped as well.

These primitives accepts an optional `width` specification. When used the also optional keywords `left`, `middle` and `right` will determine what happens when a requested size can't be met (which can happen when we step to successive larger variants).

An extra primitive `\Uhextensible` is available that can be used like this:

```
$\Uhextensible width 10cm 0 "2194$
```

This will render this:

$\longleftrightarrow$

Here you can also pass options, like:

```
$\Uhextensible width 1pt middle 0 "2194$
```

This gives:

$\leftrightarrow$

LuaTEX internally uses a structure that supports OpenType 'MathVariants' as well as tfm 'extensible recipes'. In most cases where font metrics are involved we have a different code path for traditional fonts end OpenType fonts.

8.6.6 Fractions and the new \Uskewed and \Uskewedwithdelims

The `\abovewithdelims` command accepts a keyword `exact`. When issued the extra space relative to the rule thickness is not added. One can of course use the `\Umathfraction..gap` commands to influence the spacing. Also the rule is still positioned around the math axis.

```
$$ { {a} \abovewithdelims() exact 4pt {b} }$$
```

The math parameter table contains some parameters that specify a horizontal and vertical gap for skewed fractions. Of course some guessing is needed in order to implement something that uses them. And so we now provide a primitive similar to the other fraction related ones but with a few options so that one can influence the rendering. Of course a user can also mess around a bit with the parameters `\Umathskewedfractionhgap` and `\Umathskewedfractionvgap`.

The syntax used here is:

```
{ {1} \Uskewed / <options> {2} }
{ {1} \Uskewedwithdelims / () <options> {2} }
```

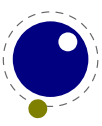
where the options can be `noaxis` and `exact`. By default we add half the axis to the shifts and by default we zero the width of the middle character. For Latin Modern the result looks as follows:

	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact`	$x + {}^{a}\!/_{exactb} + x$	$x + {}^{1}\!/_{exact2} + x$	$x + ({}^{a}\!/_{exactb}) + x$	$x + ({}^{1}\!/_{exact2}) + x$
`noaxis`	$x + {}^{a}\!/_{noaxisb} + x$	$x + {}^{1}\!/_{noaxis2} + x$	$x + ({}^{a}\!/_{noaxisb}) + x$	$x + ({}^{1}\!/_{noaxis2}) + x$
`exact noaxis`	$x + {}^{a}\!/_{exactnoaxisb} + x$	$x + {}^{1}\!/_{exactnoaxis2} + x$	$x + ({}^{a}\!/_{exactnoaxisb}) + x$	$x + ({}^{1}\!/_{exactnoaxis2}) + x$

The `\over` and related primitives have the form:

```
{{top}\over{bottom}}
```

For convenience, which also avoids some of the trickery that makes this 'looking back' possible,



the LuaMetaTEX also provides this variant:

```
\Uover{top}{bottom}
```

The optional arguments are also supported but we have one extra option: `style`. The style is applied to the numerator and denominator.

```
\Uover style \scriptstyle {top} {bottom}
```

The complete list of these commands is: `\Uabove`, `\Uatop`, `\Uover`, `\Uabovewithdelims`, `\Uatop-withdelims`, `\Uoverwithdelims`, `\UUskewed`, `\UUskewedwithdelims`. As with other extensions we use a leading `U` and because we already had extra skew related primitives we end up with a `UU` there. This obscurity is not that big an issue because normally such primitives are wrapped in a macro. Here are a few examples:

```
$\Uover {                    1234} {                    5678} $\quad
$\Uover {\textstyle          1234} {\textstyle          5678} $\quad
$\Uover {\scriptstyle        1234} {\scriptstyle        5678} $\quad
$\Uover {\scriptscriptstyle 1234} {\scriptscriptstyle 5678} $\blank

$\Uover                           {1234} {5678} $\quad
$\Uover style \textstyle          {1234} {5678} $\quad
$\Uover style \scriptstyle        {1234} {5678} $\quad
$\Uover style \scriptscriptstyle {1234} {5678} $\blank
```

These render as: $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$

$\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$

8.6.7 Math styles: `\Ustyle`

This primitive accepts a style identifier:

```
\Ustyle \displaystyle
```

This in itself is not spectacular because it is equivalent to

```
\displaystyle
```

Both commands inject a style node and change the current style. However, as in other places where LuaMetaTEX expects a style you can also pass a number in the range zero upto seven (like the ones reported by the primitive `\mathstyle`). So, the next few lines give identical results:

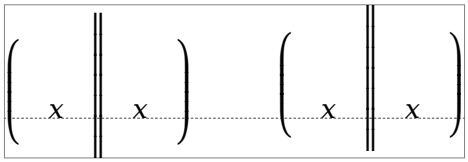
Like: 0_7 0_7 0_7. Values outside the valid range are ignored.

There is an extra option `norule` that can be used to suppress the rule while keeping the spacing compatible.

8.6.8 Delimiters: \Uleft, \Umiddle and \Uright

Normally you will force delimiters to certain sizes by putting an empty box or rule next to it. The resulting delimiter will either be a character from the stepwise size range or an extensible. The latter can be quite differently positioned than the characters as it depends on the fit as well as the fact whether the used characters in the font have depth or height. Commands like (plain TEXs) \big need to use this feature. In LuaTEX we provide a bit more control by three variants that support optional parameters height, depth and axis. The following example uses this:

```
\Uleft   height 30pt depth 10pt      \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt      \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt      \Udelimiter "0 "0 "000029
\quad \quad \quad
\Uleft   height 30pt depth 10pt axis \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt axis \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt axis \Udelimiter "0 "0 "000029
```



The keyword exact can be used as directive that the real dimensions should be applied when the criteria can't be met which can happen when we're still stepping through the successively larger variants. When no dimensions are given the noaxis command can be used to prevent shifting over the axis.

You can influence the final class with the keyword class which will influence the spacing. The numbers are the same as for character classes.

8.6.9 Accents: \mathlimitsmode

When you use \limits or \nolimits without scripts spacing might get messed up. This can be prevented by setting \mathlimitsmode to a non-zero value.

8.7 Extracting values

8.7.1 Codes and using \Umathcode, \Umathcharclass, \Umathcharfam and \Umathcharslot

You can extract the components of a math character. Say that we have defined:

```
\Umathcode 1 2 3 4
```

then

```
[\Umathcharclass1] [\Umathcharfam1] [\Umathcharslot1]
```

will return:

```
[2] [3] [4]
```

These commands are provided as convenience. Before they come available you could do the following:

```
\def\Umathcharclass{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[1])}
\relax}
\def\Umathcharfam{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[2])}
\relax}
\def\Umathcharslot{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[3])}
\relax}
```

8.7.2 Last lines and \predisplaygapfactor

There is a new primitive to control the overshoot in the calculation of the previous line in mid-paragraph display math. The default value is 2 times the em width of the current font:

```
\predisplaygapfactor=2000
```

If you want to have the length of the last line independent of math i.e. you don't want to revert to a hack where you insert a fake display math formula in order to get the length of the last line, the following will often work too:

```
\def\lastlinelength{\dimexpr
    \directlua {tex.sprint (
        (nodes.dimensions(node.tail(tex.lists.page_head).list))
    )}sp
\relax}
```

8.8 Math mode

8.8.1 Verbose versions of single-character math commands like \Usuperscript and \Usubscript

LuaTEX defines six new primitives that have the same function as ^, _, $, and $$:

PRIMITIVE	EXPLANATION
\Usuperscript	duplicates the functionality of ^
\Usubscript	duplicates the functionality of _
\Ustartmath	duplicates the functionality of $, when used in non-math mode.
\Ustopmath	duplicates the functionality of $, when used in inline math mode.
\Ustartdisplaymath	duplicates the functionality of $$, when used in non-math mode.
\Ustopdisplaymath	duplicates the functionality of $$, when used in display math mode.

The \Ustopmath and \Ustopdisplaymath primitives check if the current math mode is the correct one (inline vs. displayed), but you can freely intermix the four mathon/mathoff commands with explicit dollar sign(s).

8.8.2 Script commands \Unosuperscript and \Unosubscript

These two commands result in super- and subscripts but with the current style (at the time of rendering). So,

```
$
    x\Usuperscript  {1}\Usubscript  {2} =
    x\Unosuperscript{1}\Unosubscript{2} =
    x\Usuperscript  {1}\Unosubscript{2} =
    x\Unosuperscript{1}\Usubscript  {2}
$
```

results in $x_2^1 = x_2^1 = x_2^1 = x_2^1$.

8.8.3 Allowed math commands in non-math modes

The commands \mathchar, and \Umathchar and control sequences that are the result of \mathchardef or \Umathchardef are also acceptable in the horizontal and vertical modes. In those cases, the \textfont from the requested math family is used.

8.9 Goodies

8.9.1 Flattening: \mathflattenmode

The TEX math engine collapses ord noads without sub- and superscripts and a character as nucleus, which has the side effect that in OpenType mode italic corrections are applied (given that they are enabled).

```
\switchtobodyfont[modern]
$V \mathbin{\mathbin{v}} V$\par
$V \mathord{\mathord{v}} V$\par
```

This renders as:

$V\,v\,V$
$V\,vV$

When we set `\mathflattenmode` to 31 we get:

$V\,\,v\,V$
$V\,vV$

When you see no difference, then the font probably has the proper character dimensions and no italic correction is needed. For Latin Modern (at least till 2018) there was a visual difference. In that respect this parameter is not always needed unless of course you want efficient math lists anyway.

You can influence flattening by adding the appropriate number to the value of the mode parameter. The default value is 1.

MODE	CLASS
1	ord
2	bin
4	rel
8	punct
16	inner

8.9.2 Less Tracing

Because there are quite some math related parameters and values, it is possible to limit tracing. Only when `tracingassigns` and/or `tracingrestores` are set to 2 or more they will be traced.

8.10 Experiments

There are a couple of experimental features. They will stay but details might change, for instance more control over spacing. We just show some examples and let your imagination work it out. First we have prescripts:

8.10.1 Prescripts with `\Usuperprescript` and `Usubprescript`

```
\hbox{$
    {\tf X}^1_2^^3__4 \quad
    {\tf X}^1  ^^3    \quad
    {\tf X}  _1   __4 \quad
    {\tf X}    ^^3    \quad
    {\tf X}       __4 \quad
    {\tf X}^^3    __4
$}
```

The question is: are these double super and subscript triggers the way to go? Anyway, you need to have them either being active (which in ConTEXt then boils down to them being other characters), or say `\supmarkmode = 1` to disable the normal multiple ^ treatment (a value larger than 1 will also disable that in text mode).

$^{3}_{4}X^{1}_{2}$ $^{3}X^{1}$ $_{4}X_{1}$ ^{3}X $_{4}X$ $^{3}_{4}X$

The more explicit commands are:

```
\hbox{$
{\tf X}\Usuperscript{1}                                          \quad
{\tf X}                \Usubscript{2}                            \quad
{\tf X}\Usuperscript{1}\Usubscript{2}                            \quad
{\tf X}\Usuperscript{1}             \Usuperprescript{3}          \quad
{\tf X}                \Usubscript{2}                 \Usubprescript{4}\quad
{\tf X}\Usuperscript{1}\Usubscript{2}\Usuperprescript{3}\Usubprescript{4}\quad
{\tf X}                             \Usuperprescript{3}          \quad
{\tf X}                                            \Usubprescript{4}\quad
{\tf X}                             \Usuperprescript{3}\Usubprescript{4}
$}
```

These more verbose triggers can be used to build interfaces:

X^{1} X_{2} X^{1}_{2} $^{3}X^{1}$ $_{4}X_{2}$ $^{3}_{4}X^{1}_{2}$ ^{3}X $_{4}X$ $^{3}_{4}X$

8.10.2 Prescripts with \Usuperprescript and Usubprescript

You can change the class of a math character on the fly:

```
$x\mathopen   {!}+123+\mathclose  {!}x$
$x\Umathclass4 ! +123+\Umathclass5 ! x$
$x             ! +123+             ! x$
$x\mathclose  {!}+123+\mathopen   {!}x$
$x\Umathclass5 ! +123+\Umathclass4 ! x$
```

Watch how the spacing changes:

$x!+123+!x$
$x!+123+!x$
$x! + 123+!x$
$x! + 123 + !x$
$x! + 123 + !x$

9 Building paragraphs and pages

9.1 Introduction

There are some enhancements that relate to the way paragraphs and pages are built. In this chapter we will cover those. There can be a bit of overlap with other chapters. These enhancements are still somewhat experimental.

9.2 Paragraphs

This section will describe `\autoparagraphmode`*,* `\shapingpenaltiesmode`*,* `\shapingpenaltymode`*,* `\everybeforepar`*,* `\snapshotpar`*,* `\wrapuppar`*, enz. For the moment the manuals that come with ConTEXt have to do.*

9.3 Inserts

Inserts are tightly integrated into the page builder. Depending on penalties and available space they end up on the same page as were they got injected or they move to following pages, either or not split.

In traditional TEX inserts are controlled by registers. A quadruple of box, skip, dimen and count registers with the same number acts as an insert class. Details can be found in the TEXbook. A side effect of this is that we only have these four properties bound to class, other properties of inserts are driven by shared parameters. Another side effect is that register management has to make sure that these foursome get 'allocates' as set and not clashes with other register allocations.

In LuaMetaTEX you can set the `\insertmode` to a non zero value in which case inserts are not using the register pool but have their own (global) resources. For now this is mode driven (for compatibility reasons) and once set or when an insert has been accessed, this mode is frozen, so this parameter can be set very early in the macro package loading process.

PRIMITIVE	TRADITIONAL	EXPLANATION
`\insertdistance`	skip	the space before the first instance (on a page)
`\insertmultiplier`	count	a factor that is used to calculate the height used
`\insertlimit`	dimen	the maximum amount of space on a page to be taken
`\insertpenalty`	`\insertpenalties`	the floating penalty (used when set)
`\insertmaxdepth`	`\maxdepth`	the maximum split depth (used when set)
`\insertstorage`		signals that the insert has to be stored for later
`\insertheight`	`\ht` box	the accumulated height of the inserts so far
`\insertdepth`	`\dp` box	the current depth of the inserts so far
`\insertwidth`	`\wd` box	the width of the inserts

These primitives takes an insert class number. The `\insertpenalties` primitives is unchanged, as is the LuaTEX `\insertheights` one. When `\insertstoring` is set 1, all inserts that have their

storage flag set will be saved. Think of a multi column setup where inserts have to end up in the last column. If there are three columns, the first two will store inserts. Then when the last column is dealt with `\insertstoring` can be set to 2 and that will signal the builder that we will inject the inserts. In both cases, the value of this register will be set to zero so that it doesn't influence further processing. More details about these (probably experimental for a while) features can be found in documents that come with ConTEXt.

A limitation of inserts is that when they are buried too deep, a property they share with inserts, they become invisible This can be dealt with by the migration feature described in an upcoming section.

The LuaMetaTEX engine has some tracing built in that is enabled by setting `\tracinginserts` to a positive value.

9.4 Marks

Marks are kind of signal nodes in the list that refer to stored token lists. When a page has been split off and is handed over to the output routine these signals are resolved into first, top and bottom mark references that can (for instance) be used for running headers.

In ε-TEX the standard TEX primitives `\mark`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark` and `\splitbotmark` have been extended with plural forms that accent a number before the token list. That number indicates a mark class.

A problem with marks is that one cannot really reset them. Mark states are kept in the node lists and only periodically the state is snapshot into the global state variables. The LuaTEX engine can reset these global states with `\clearmarks` but that's only half a solution. In LuaMetaTEX we have `\flushmarks` which, like `\marks`, puts a node in the list that does a reset. This permits implementing controlled resets of specific marks at the cost of a possible interfering mode, but that can normally be dealt with rather well.

Another problem with marks is that when they are buried too deep, a property they share with inserts, they become invisible. This can be dealt with by the migration feature described in the next section.

The LuaMetaTEX engine has some tracing built in that is enabled by setting `\tracingmarks` to a positive value.

9.5 Migration

A new primitive `\automigrationmode` can be used to let deeply burried marks and inserts bubble up to the outer level.

VALUE	EXPLANATION
1	migrate marks in the par builder
2	migrate inserts in the par builder
4	migrate prebox material in the page builder
8	migrate postbox material in the page builder

If you want to migrate marks and inserts you need to set al these flags. Migrated marks and inserts end up as post-box properties and will be handled in the page builder as such. At the Lua end you can add pre- and post-box material too.

9.6 Pages

The page builder can triggered by (for instance) a penalty but you can also use `\pageboundary`. This will trigger the page builder but not leave anything behind.

10 Nodes

10.1 Lua node representation

TEX's nodes are represented in Lua as userdata objects with a variable set of fields or by a numeric identifier when requested. When you print a node userdata object you will see these numbers. In the following syntax tables the type of such a userdata object is represented as ⟨node⟩.

The return values of `node.types` are: `hlist` (0), `vlist` (1), `rule` (2), `insert` (3), `mark` (4), `adjust` (5), `boundary` (6), `disc` (7), `whatsit` (8), `par` (9), `dir` (10), `math` (11), `glue` (12), `kern` (13), `penalty` (14), `style` (15), `choice` (16), `parameter` (17), `noad` (18), `radical` (19), `fraction` (20), `accent` (21), `fence` (22), `math_char` (23), `math_text_char` (24), `sub_box` (25), `sub_mlist` (26), `delimiter` (27), `glyph` (28), `unset` (29), `align_record` (31), `attribute_list` (32), `attribute` (33), `glue_spec` (34), `temp` (35) and `split` (36)

You can ask for a list of fields with `node.fields` and for valid subtypes with `node.subtypes`. The `node.values` function reports some used values. Valid arguments are `glue`, `style` and `math`. Keep in mind that the setters normally expect a number, but this helper gives you a list of what numbers matter. For practical reason the `pagestate` values are also reported with this helper, but they are backend specific.

The return values of `node.values("glue")` are: `normal` (0), `fi` (1), `fil` (2), `fill` (3) and `filll` (4)

The return values of `node.values("style")` are: `display` (0), `crampeddisplay` (1), `text` (2), `crampedtext` (3), `script` (4), `crampedscript` (5), `scriptscript` (6) and `crampedscriptscript` (7)

The return values of `node.values("math")` are: `quad` (0), `axis` (1), `accentbaseheight` (2), `spacingmode` (3), `operatorsize` (4), `overbarkern` (5), `overbarrule` (6), `overbarvgap` (7), `underbarkern` (8), `underbarrule` (9), `underbarvgap` (10), `radicalkern` (11), `radicalrule` (12), `radicalvgap` (13), `radicaldegreebefore` (14), `radicaldegreeafter` (15), `radicaldegreeraise` (16), `stackvgap` (17), `stacknumup` (18), `stackdenomdown` (19), `fractionrule` (20), `fractionnumvgap` (21), `fractionnumup` (22), `fractiondenomvgap` (23), `fractiondenomdown` (24), `fractiondelsize` (25), `skewedfractionhgap` (26), `skewedfractionvgap` (27), `limitabovevgap` (28), `limitabovebgap` (29), `limitabovekern` (30), `limitbelowvgap` (31), `limitbelowbgap` (32), `limitbelowkern` (33), `nolimitsubfactor` (34), `nolimitsupfactor` (35), `underdelimitervgap` (36), `underdelimiterbgap` (37), `overdelimitervgap` (38), `overdelimiterbgap` (39), `subshiftdrop` (40), `supshiftdrop` (41), `subshiftdown` (42), `subsupshiftdown` (43), `subtopmax` (44), `supshiftup` (45), `supbottommin` (46), `supsubbottommax` (47), `subsupvgap` (48), `spacebeforescript` (49), `spaceafterscript` (50), `connectoroverlapmin` (51), `extrasuperscriptshift` (52), `extrasubscriptshift` (53), `extrasuperprescriptshift` (54), `extrasubprescriptshift` (55), `ordordspacing` (56), `ordopspacing` (57), `ordbinspacing` (58), `ordrelspacing` (59), `ordopenspacing` (60), `ordclosespacing` (61), `ordpunctspacing` (62), `ordinnerspacing` (63), `opordspacing` (64), `opopspacing` (65), `opbinspacing` (66), `oprelspacing` (67), `opopenspacing`

(68), opclosespacing (69), oppunctspacing (70), opinnerspacing (71), binordspacing (72), binopspacing (73), binbinspacing (74), binrelspacing (75), binopenspacing (76), binclosespacing (77), binpunctspacing (78), bininnerspacing (79), relordspacing (80), relopspacing (81), relbinspacing (82), relrelspacing (83), relopenspacing (84), relclosespacing (85), relpunctspacing (86), relinnerspacing (87), openordspacing (88), openopspacing (89), openbinspacing (90), openrelspacing (91), openopenspacing (92), openclosespacing (93), openpunctspacing (94), openinnerspacing (95), closeordspacing (96), closeopspacing (97), closebinspacing (98), closerelspacing (99), closeopenspacing (100), closeclosespacing (101), closepunctspacing (102), closeinnerspacing (103), punctordspacing (104), punctopspacing (105), punctbinspacing (106), punctrelspacing (107), punctopenspacing (108), punctclosespacing (109), punctpunctspacing (110), punctinnerspacing (111), innerordspacing (112), inneropspacing (113), innerbinspacing (114), innerrelspacing (115), inneropenspacing (116), innerclosespacing (117), innerpunctspacing (118), innerinnerspacing (119), overlinevariant (120), underlinevariant (121), overdelimitervariant (122), underdelimitervariant (123), delimiterovervariant (124), delimiterundervariant (125), hextensiblevariant (126), vextensiblevariant (127), fractionvariant (128), radicalvariant (129), accentvariant (130), degreevariant (131), topaccentvariant (132), botaccentvariant (133), overlayaccentvariant (134), numeratorvariant (135), denominatorvariant (136), superscriptvariant (137), subscriptvariant (138) and stackvariant (139)

The return values of `node.values("pagestate")` are:

10.2 Main text nodes

These are the nodes that comprise actual typesetting commands. A few fields are present in all nodes regardless of their type, these are:

FIELD	TYPE	EXPLANATION
next	node	the next node in a list, or nil
id	number	the node's type (id) number
subtype	number	the node subtype identifier

The subtype is sometimes just a dummy entry because not all nodes actually use the subtype, but this way you can be sure that all nodes accept it as a valid field name, and that is often handy in node list traversal. In the following tables next and id are not explicitly mentioned.

Besides these three fields, almost all nodes also have an attr field, and there is a also a field called prev. That last field is always present, but only initialized on explicit request: when the function node.slide is called, it will set up the prev fields to be a backwards pointer in the argument node list. By now most of TEX's node processing makes sure that the prev nodes are valid but there can be exceptions, especially when the internal magic uses a leading temp nodes to temporarily store a state.

The LuaMetaTEX engine provides a lot of freedom and it is up to the user to make sure that the node lists remain sane. There are some safeguards but there can be cases where the engine just quits out of frustration. And, of course you can make the engine crash.

10.2.1 hlist and vlist nodes

These lists share fields and subtypes although some subtypes can only occur in horizontal lists while others are unique for vertical lists. The possible fields are attr, depth, direction, doffset, glue_order, glue_set, glue_sign, height, hoffset, list, orientation, shift, state, width, woffset, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	accent, alignment, box, cell, container, degree, denominator, equation, equationnumber, fraction, hdelimiter, hextensible, indent, line, math, mathchar, modifier, nucleus, numerator, over, overdelimiter, radical, scripts, sub, sup, under, underdelimiter, unknown, vdelimiter and vextensible
attr	node	list of attributes
width	number	the width of the box
height	number	the height of the box
depth	number	the depth of the box
direction	number	the direction of this box, see 10.2.15
shift	number	a displacement perpendicular to the character (hlist) or line (vlist) progression direction
glue_order	number	a number in the range [0, 4], indicating the glue order
glue_set	number	the calculated glue ratio
glue_sign	number	0 = normal, 1 = stretching, 2 = shrinking
list	node	the first node of the body of this list

The orientation, woffset, hoffset, doffset, xoffset and yoffset fields are special. They can be used to make the backend rotate and shift boxes which can be handy in for instance vertical typesetting. Because they relate to (and depend on the) the backend they are not discussed here (yet).

A warning: never assign a node list to the list field unless you are sure its internal link structure is correct, otherwise an error may result.

Note: the field name head and list are both valid. Sometimes it makes more sense to refer to a list by head, sometimes list makes more sense.

10.2.2 rule nodes

Contrary to traditional TEX, LuaTEX has more \hrule and \vrule subtypes because we also use rules to store reuseable objects and images. User nodes are invisible and can be intercepted by a callback. The supported fields are attr, data, depth, height, left, right, width, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	box, empty, fraction, image, normal, outline, over, radical, under and user
attr	node	list of attributes

width	number	the width of the rule where the special value −1073741824 is used for 'running' glue dimensions
height	number	the height of the rule (can be negative)
depth	number	the depth of the rule (can be negative)
left	number	shift at the left end (also subtracted from width)
right	number	(subtracted from width)
dir	string	the direction of this rule, see 10.2.15
index	number	an optional index that can be referred to
transform	number	an private variable (also used to specify outline width)

The left and type right keys are somewhat special (and experimental). When rules are auto adapting to the surrounding box width you can enforce a shift to the right by setting left. The value is also subtracted from the width which can be a value set by the engine itself and is not entirely under user control. The right is also subtracted from the width. It all happens in the backend so these are not affecting the calculations in the frontend (actually the auto settings also happen in the backend). For a vertical rule left affects the height and right affects the depth. There is no matching interface at the TEX end (although we can have more keywords for rules it would complicate matters and introduce a speed penalty.) However, you can just construct a rule node with Lua and write it to the TEX input. The outline subtype is just a convenient variant and the transform field specifies the width of the outline.

The xoffset and yoffset fields are special. They can be used to shift rules. Because they relate to (and depend on the) the backend they are not discussed here (yet).

10.2.3 insert nodes

This node relates to the \insert primitive and support the fields: attr, cost, depth, height, list and spec.

FIELD	TYPE	EXPLANATION
subtype	number	the insertion class
attr	node	list of attributes
cost	number	the penalty associated with this insert
height	number	height of the insert
depth	number	depth of the insert
list	node	the first node of the body of this insert

There is a set of extra fields that concern the associated glue: width, stretch, stretch_order, shrink and shrink_order. These are all numbers.

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may result. You can use list instead (often in functions you want to use local variable with similar names and both names are equally sensible).

10.2.4 mark nodes

This one relates to the \mark primitive and only has a few fields: attr, class and mark.

FIELD	TYPE	EXPLANATION
subtype	number	unused
attr	node	list of attributes
class	number	the mark class
mark	table	a table representing a token list

10.2.5 adjust nodes

This node comes from \vadjust primitive and has fields: attr and list.

FIELD	TYPE	EXPLANATION
subtype	number	normal and pre
attr	node	list of attributes
list	node	adjusted material

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be the result.

10.2.6 disc nodes

The \discretionary and \-, the - character but also the hyphenation mechanism produces these nodes. The available fields are: attr, options, penalty, post, pre and replace.

FIELD	TYPE	EXPLANATION
subtype	number	automatic, discretionary, explicit and regular
attr	node	list of attributes
pre	node	pointer to the pre-break text
post	node	pointer to the post-break text
replace	node	pointer to the no-break text
penalty	number	the penalty associated with the break, normally \hyphenpenalty or \exhyphenpenalty

The subtype numbers 4 and 5 belong to the 'of-f-ice' explanation given elsewhere. These disc nodes are kind of special as at some point they also keep information about breakpoints and nested ligatures.

The pre, post and replace fields at the Lua end are in fact indirectly accessed and have a prev pointer that is not nil. This means that when you mess around with the head of these (three) lists, you also need to reassign them because that will restore the proper prev pointer, so:

```
pre = d.pre
-- change the list starting with pre
d.pre = pre
```

Otherwise you can end up with an invalid internal perception of reality and LuaMetaTEX might even decide to crash on you. It also means that running forward over for instance pre is ok but

backward you need to stop at pre. And you definitely must not mess with the node that prev points to, if only because it is not really a node but part of the disc data structure (so freeing it again might crash LuaMetaTEX).

10.2.7 math nodes

Math nodes represent the boundaries of a math formula, normally wrapped into $ signs. The following fields are available: attr, shrink, shrink_order, stretch, stretch_order, surround and width.

FIELD	TYPE	EXPLANATION
subtype	number	beginmath and endmath
attr	node	list of attributes
surround	number	width of the \mathsurround kern
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

The glue fields only kick in when the surround fields is zero.

10.2.8 glue nodes

Skips are about the only type of data objects in traditional TEX that are not a simple value. They are inserted when TEX sees a space in the text flow but also by \hskip and \vskip. The structure that represents the glue components of a skip internally is called a glue_spec. In LuaMetaTEX we don't use the spec itself but just its values. A glue node has the fields: attr, font, leader, shrink, shrink_order, stretch, stretch_order and width.

FIELD	TYPE	EXPLANATION
subtype	number	abovedisplayshortskip, abovedisplayskip, baselineskip, belowdisplayshortskip, belowdisplayskip, cleaders, conditionalmathskip, correctionskip, gleaders, ignored, indentskip, intermathskip, leaders, lefthangskip, leftskip, lineskip, mathskip, medmuskip, muglue, parfillleftskip, parfillskip, parskip, righthangskip, rightskip, spaceskip, splittopskip, tabskip, thickmuskip, thinmuskip, topskip, userskip, xleaders, xspaceskip and zerospaceskip
attr	node	list of attributes
leader	node	pointer to a box or rule for leaders
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount

shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

Note that we use the key width in both horizontal and vertical glue. This suits the TEX internals well so we decided to stick to that naming.

The effective width of some glue subtypes depends on the stretch or shrink needed to make the encapsulating box fit its dimensions. For instance, in a paragraph lines normally have glue representing spaces and these stretch or shrink to make the content fit in the available space. The effectiveglue function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item. When you pass true as third argument the value will be rounded.

10.2.9 glue_spec nodes

Internally LuaMetaTEX (like its ancestors) also uses nodes to store data that is not seen in node lists. For instance the state of expression scanning (\dimexpr etc.) and conditionals (\ifcase etc.) is also kept in lists of nodes. A glue, which has five components, is stored in a node as well, so, where most registers store just a number, a skip register (of internal quantity) uses a pointer to a glue spec node. It has similar fields as glue nodes: shrink, shrink_order, stretch, stretch_order and width, which is not surprising because in the past (and other engines than LuaTEX) a glue node also has its values stored in a glue spec. This has some advantages because often the values are the same, so for instance spacing related skips were not resolved immediately but pointed to the current value of a space related internal register (like \spaceskip). But, in LuaTEX we do resolve these quantities immediately and we put the current values in the glue nodes.

FIELD	TYPE	EXPLANATION
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

You will only find these nodes in a few places, for instance when you query an internal quantity. In principle we could do without them as we have interfaces that use the five numbers instead. For compatibility reasons we keep glue spec nodes exposed but this might change in the future.

10.2.10 kern nodes

The \kern command creates such nodes but for instance the font and math machinery can also add them. There are not that many fields: attr, expansion_factor and kern.

FIELD	TYPE	EXPLANATION
subtype	number	accentkern, fontkern, italiccorrection, leftmarginkern, mathlistkern, rightmarginkern and userkern
attr	node	list of attributes

kern	number	fixed horizontal or vertical advance
expansion_factor	number	multiplier related to hz for font kerns

10.2.11 penalty nodes

The \penalty command is one that generates these nodes. It is one of the type of nodes often found in vertical lists. It has the fields: attr and penalty.

FIELD	TYPE	EXPLANATION
subtype	number	afterdisplaypenalty, beforedisplaypenalty, equationnumberpenalty, finalpenalty, linebreakpenalty, linepenalty, noadpenalty, userpenalty and wordpenalty
attr	node	list of attributes
penalty	number	the penalty value

The subtypes are just informative and TEX itself doesn't use them. When you run into an linebreakpenalty you need to keep in mind that it's a accumulation of club, widow and other relevant penalties.

10.2.12 glyph nodes

These are probably the mostly used nodes and although you can push them in the current list with for instance \char TEX will normally do it for you when it considers some input to be text. Glyph nodes are relatively large and have many fields: attr, char, data, depth, expansion_factor, font, height, hyphenate, language, left, lhmin, options, rhmin, right, script, state, total, uchyph, width, xoffset, xscale, yoffset and yscale.

FIELD	TYPE	EXPLANATION
subtype	number	bit field
attr	node	list of attributes
char	number	the character index in the font
font	number	the font identifier
language	number	the language identifier
left	number	the frozen \lefthyphenmnin value
right	number	the frozen \righthyphenmnin value
uchyph	boolean	the frozen \uchyph value
state	number	a user field (replaces the component list)
xoffset	number	a virtual displacement in horizontal direction
yoffset	number	a virtual displacement in vertical direction
width	number	the (original) width of the character
height	number	the (original) height of the character
depth	number	the (original) depth of the character
expansion_factor	number	the to be applied expansion_factor
data	number	a general purpose field for users (we had room for it)

The width, height and depth values are read-only. The expansion_factor is assigned in the par builder and used in the backend. Valid bits for the subtype field are:

BIT	MEANING
0	character
1	ligature
2	ghost
3	left
4	right

The expansion_factor has been introduced as part of the separation between front- and backend. It is the result of extensive experiments with a more efficient implementation of expansion. Early versions of LuaTEX already replaced multiple instances of fonts in the backend by scaling but contrary to pdfTEX in LuaTEX we now also got rid of font copies in the frontend and replaced them by expansion factors that travel with glyph nodes. Apart from a cleaner approach this is also a step towards a better separation between front- and backend.

The is_char function checks if a node is a glyph node with a subtype still less than 256. This function can be used to determine if applying font logic to a glyph node makes sense. The value nil gets returned when the node is not a glyph, a character number is returned if the node is still tagged as character and false gets returned otherwise. When nil is returned, the id is also returned. The is_glyph variant doesn't check for a subtype being less than 256, so it returns either the character value or nil plus the id. These helpers are not always faster than separate calls but they sometimes permit making more readable tests. The uses_font helpers takes a node and font id and returns true when a glyph or disc node references that font.

10.2.13 boundary nodes

This node relates to the \noboundary, \boundary, \protrusionboundary and \wordboundary primitives. These are small nodes: attr and data are the only fields.

FIELD	TYPE	EXPLANATION
subtype	number	cancel, protrusion, user and word
attr	node	list of attributes
data	number	values 0–255 are reserved

10.2.14 par nodes

This node is inserted at the start of a paragraph. You should not mess too much with this one. Valid fields are: attr, box_left, box_left_width, box_right, box_right_width, brokenpenalty, dir and interlinepenalty.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
pen_inter	number	local interline penalty (from \localinterlinepenalty)
pen_broken	number	local broken penalty (from \localbrokenpenalty)

dir	string	the direction of this par. see 10.2.15
box_left	node	the \localleftbox
box_left_width	number	width of the \localleftbox
box_right	node	the \localrightbox
box_right_width	number	width of the \localrightbox

A warning: never assign a node list to the box_left or box_right field unless you are sure its internal link structure is correct, otherwise an error may result.

10.2.15 dir nodes

Direction nodes mark parts of the running text that need a change of direction and the \textdirection command generates them. Again this is a small node, we just have attr, dir and level.

FIELD	TYPE	EXPLANATION
subtype	number	cancel and normal
attr	node	list of attributes
dir	string	the direction (0 = l2r, 1 = r2l)
level	number	nesting level of this direction

There are only two directions: left-to-right (0) and right-to-left (1). This is different from LuaTEX that has four directions.

10.2.16 Whatsits

A whatsit node is a real simple one and it only has a subtype. It is even less than a user node (which it actually could be) and uses hardly any memory. What you do with it it entirely up to you: it's is real minimalistic. You can assign a subtype and it has attributes. It is all up to the user how they are handled.

10.2.17 Math noads

10.2.17.1 The concept

These are the so–called 'noad's and the nodes that are specifically associated with math processing. When you enter a formula, TEX creates a node list with regular nodes and noads. Then it hands over the list the math processing engine. The result of that is a nodelist without noads. Most of the noads contain subnodes so that the list of possible fields is actually quite small. Math formulas are both a linked list and a tree. For instance in $e = mc^2$ there is a linked list e = m c but the c has a superscript branch that itself can be a list with branches.

First, there are the objects (the TEXbook calls them 'atoms') that are associated with the simple math objects: ord, op, bin, rel, open, close, punct, inner, over, under, vcenter. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation: attr, nucleus, options, sub and sup.

Many object fields in math mode are either simple characters in a specific family or math lists

or node lists: `math_char`, `math_text_char`, sub_box and `sub_mlist` and `delimiter`. These are endpoints and therefore the `next` and `prev` fields of these these subnodes are unused.

Some of the more elaborate noads have an option field. The values in this bitset are common:

MEANING	BITS
set	0x08
internal	0x00 + 0x08
internal	0x01 + 0x08
axis	0x02 + 0x08
no axis	0x04 + 0x08
exact	0x10 + 0x08
left	0x11 + 0x08
middle	0x12 + 0x08
right	0x14 + 0x08
no subscript	0x21 + 0x08
no superscript	0x22 + 0x08
no script	0x23 + 0x08

10.2.17.2 `math_char` and `math_text_char` subnodes

These are the most common ones, as they represent characters, and they both have the same fields: `attr`, `char`, `fam` and `options`.

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`char`	number	the character index
`fam`	number	the family number

The `math_char` is the simplest subnode field, it contains the character and family for a single glyph object. The family eventually resolves on a reference to a font. The `math_text_char` is a special case that you will not normally encounter, it arises temporarily during math list conversion (its sole function is to suppress a following italic correction).

10.2.17.3 `sub_box` and `sub_mlist` subnodes

These two subnode types are used for subsidiary list items. For `sub_box`, the `list` points to a 'normal' vbox or hbox. For `sub_mlist`, the `list` points to a math list that is yet to be converted. Their fields are: `attr` and `list`.

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`list`	node	list of nodes

A warning: never assign a node list to the `list` field unless you are sure its internal link structure is correct, otherwise an error is triggered.

10.2.17.4 delimiter subnodes

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the next and prev fields are unused, but we do have: attr, large_char, large_fam, small_char and small_fam.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
small_char	number	character index of base character
small_fam	number	family number of base character
large_char	number	character index of next larger character
large_fam	number	family number of next larger character

The fields large_char and large_fam can be zero, in that case the font that is set for the small_fam is expected to provide the large version as an extension to the small_char.

10.2.17.5 simple noad nodes

In these noads, the nucleus, sub and sup fields can branch of. Its fields are: attr, nucleus, options, sub and sup.

FIELD	TYPE	EXPLANATION
subtype	number	bin, close, inner, op, open, ord, over, punct, rel, under and vcenter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
options	number	bitset of rendering options

10.2.17.6 accent nodes

Accent nodes deal with stuff on top or below a math constructs. They support: accent, attr, bot_accent, fraction, nucleus, overlay_accent, sub, sup and top_accent.

FIELD	TYPE	EXPLANATION
subtype	number	bothflexible, fixedboth, fixedbottom and fixedtop
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
accent	kernel node	top accent
bot_accent	kernel node	bottom accent
fraction	number	larger step criterium (divided by 1000)

10.2.17.7 style nodes

These nodes are signals to switch to another math style. They are quite simple: attr and style. Currently the subtype is actually used to store the style but don't rely on that for the future. Fields are: attr and style.

FIELD	TYPE	EXPLANATION
style	string	contains the style

Valid styles are: display (0), crampeddisplay (1), text (2), crampedtext (3), script (4), crampedscript (5), scriptscript (6) and crampedscriptscript (7).

10.2.17.8 parameter nodes

These nodes are used to (locally) set math parameters: list, name, style and value. Fields are: list, name, style and value.

FIELD	TYPE	EXPLANATION
style	string	contains the style
name	string	defines the parameter
value	number	holds the value, in case of a muglue multiple

10.2.17.9 choice nodes

Of its fields attr, display, script, scriptscript and text most are lists. Warning: never assign a node list unless you are sure its internal link structure is correct, otherwise an error can occur.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
display	node	list of display size alternatives
text	node	list of text size alternatives
script	node	list of scriptsize alternatives
scriptscript	node	list of scriptscriptsize alternatives

10.2.17.10 radical nodes

Radical nodes are the most complex as they deal with scripts as well as constructed large symbols. Many fields: attr, degree, left, nucleus, options, sub, sup and width. Warning: never assign a node list to the nucleus, sub, sup, left, or degree field unless you are sure its internal link structure is correct, otherwise an error can be triggered.

FIELD	TYPE	EXPLANATION
subtype	number	radical, udelimiterover, udelimiterunder, uhextensible, uoverdelimiter, uradical, uroot and uunderdelimiter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
left	delimiter node	
degree	kernel node	only set by \Uroot
width	number	required width
options	number	bitset of rendering options

10.2.17.11 fraction nodes

Fraction nodes are also used for delimited cases, hence the left and right fields among: attr, denom, fam, left, middle, num, options, right and width.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	(optional) width of the fraction
num	kernel node	numerator
denom	kernel node	denominator
left	delimiter node	left side symbol
right	delimiter node	right side symbol
middle	delimiter node	middle symbol
options	number	bitset of rendering options

Warning: never assign a node list to the num, or denom field unless you are sure its internal link structure is correct, otherwise an error can result.

10.2.17.12 fence nodes

Fence nodes come in pairs but either one can be a dummy (this period driven empty fence). Fields are: attr, class, delimiter, depth, height, italic and options. Some of these fields are used by the renderer and might get adapted in the process.

FIELD	TYPE	EXPLANATION
subtype	number	left, middle, no, right and unset
attr	node	list of attributes
delimiter	delimiter node	delimiter specification
italic	number	italic correction
height	number	required height
depth	number	required depth
options	number	bitset of rendering options
class	number	spacing related class

10.3 The node library

10.3.1 Introduction

The node library provides methods that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, copy, delete, and insert node, the core objects within the typesetter. Nodes are represented in Lua as userdata. The various parts within a node can be accessed using named fields.

Each node has at least the three fields next, id, and subtype. The other available fields depend

on the `id`.

- The `next` field returns the userdata object for the next node in a linked list of nodes, or `nil`, if there is no next node.
- The `id` indicates TEX's 'node type'. The field `id` has a numeric value for efficiency reasons, but some of the library functions also accept a string value instead of `id`.
- The `subtype` is another number. It often gives further information about a node of a particular `id`.

Nodes can be compared to each other, but: you are actually comparing indices into the node memory. This means that equality tests can only be trusted under very limited conditions. It will not work correctly in any situation where one of the two nodes has been freed and/or reallocated: in that case, there will be false positives. The general approach to a node related callback is as follows:

- Assume that the node list that you get is okay and properly double linked. If for some reason the links are not right, you can apply `node.slide` to the list.
- When you insert a node, make sure you use a previously removed one, a new one or a copy. Don't simply inject the same node twice.
- When you remove a node, make sure that when this is permanent, you also free the node or list.
- Although you can fool the system, normally you will trigger an error when you try to copy a nonexisting node, or free an already freed node. There is some overhead involved in this checking but the current compromise is acceptable.
- When you're done, pass back (if needed) the result. It's your responsibility to make sure that the list is properly linked (you can play safe and again apply `node.slide`. In principle you can put nodes in a list that are not acceptable in the following up actions. Some nodes get ignored, others will trigger an error, and sometimes the engine will just crash.

So, from the above it will be clear then memory management of nodes has to be done explicitly by the user. Nodes are not 'seen' by the Lua garbage collector, so you have to call the node freeing functions yourself when you are no longer in need of a node (list). Nodes form linked lists without reference counting, so you have to be careful that when control returns back to LuaTEX itself, you have not deleted nodes that are still referenced from a `next` pointer elsewhere, and that you did not create nodes that are referenced more than once. Normally the setters and getters handle this for you.

A good example are discretionary nodes that themselves have three sublists. Internally they use special pointers, but the user never sees them because when you query them or set fields, this property is hidden and taken care of. You just see a list. But, when you mess with these sub lists it is your responsibility that it only contains nodes that are permitted in a discretionary.

There are statistics available with regards to the allocated node memory, which can be handy for tracing. Normally the amount of used nodes is not that large. Typesetting a page can involve thousands of them but most are freed when the page has been shipped out. Compared to other programs, node memory usage is not that excessive. So, if for some reason your application leaks nodes, if at the end of your run you lost as few hundred it's not a real problem. In fact, if you created boxes and made copies but not flushed them for good reason, your run will for sure end with used nodes and the statistics will mention that. The same is true for attributes and

skips (glue spec nodes): keeping the current state involves using nodes.

10.3.2 Housekeeping

10.3.2.1 types

This function returns an array that maps node id numbers to node type strings, providing an overview of the possible top-level `id` types.

```
<table> t = node.types()
```

When we issue this command, we get a table. The currently visible types are `{ [0] = "hlist", "vlist", "rule", "insert", "mark", "adjust", "boundary", "disc", "whatsit", "par", "dir", "math", "glue", "kern", "penalty", "style", "choice", "parameter", "noad", "radical", "fraction", "accent", "fence", "math_char", "math_text_char", "sub_box", "sub_mlist", "delimiter", "glyph", "unset", [31] = "align_record", [32] = "attribute_list", [33] = "attribute", [34] = "glue_spec", [35] = "temp", [36] = "split", }` where the numbers are the internal identifiers. Only those nodes are reported that make sense to users so there can be gaps in the range of numbers.

10.3.2.2 id and type

This converts a single type name to its internal numeric representation.

```
<number> id = node.id(<string> type)
```

The `node.id("glyph")` command returns the number `28` and `node.id("hlist")` returns `0` where the numbers don't relate to importance or some ordering; they just appear in the order that is handy for the engine. Commands like this are rather optimized so performance should be ok but you can of course always store the id in a Lua number.

The reverse operation is: `node.type` If the argument is a number, then the next function converts an internal numeric representation to an external string representation. Otherwise, it will return the string `node` if the object represents a node, and `nil` otherwise.

```
<string> type = node.type(<any> n)
```

The `node.type(4)` command returns the string `mark` and `node.id(99)` returns `nil` because there is no node with that id.

10.3.2.3 fields and hasfield

This function returns an indexed table with valid field names for a particular type of node.

```
<table> t = node.fields(<number|string> id)
```

The function accepts a string or number, so `node.fields ("glyph")` returns `{ [-1] = "prev", [0] = "next", "id", "subtype", "attr", "char", "font", "language", "lhmin", "rhmin", "uchyph", "state", "left", "right", "xoffset", "yoffset", "xscale", "yscale", "width", "height", "depth", "total", "expansion_factor", "data", "script", "hyphen-`

ate", "options", } and node.fields (12) gives { [-1] = "prev", [0] = "next", "id", "subtype", "attr", "leader", "width", "stretch", "shrink", "stretch_order", "shrink_order", "font", }.

The hasfield function returns a boolean that is only true if n is actually a node, and it has the field.

```
<boolean> t = node.hasfield(<node> n, <string> field)
```

This function probably is not that useful but some nodes don't have a subtype, attr or prev field and this is a way to test for that.

10.3.2.4 is_node

```
<boolean|integer> t = node.is_node(<any> item)
```

This function returns a number (the internal index of the node) if the argument is a userdata object of type <node> and false when no node is passed.

10.3.2.5 new

The new function creates a new node. All its fields are initialized to either zero or nil except for id and subtype. Instead of numbers you can also use strings (names). If you pass a second argument the subtype will be set too.

```
<node> n = node.new(<number|string> id)
<node> n = node.new(<number|string> id, <number|string> subtype)
```

As already has been mentioned, you are responsible for making sure that nodes created this way are used only once, and are freed when you don't pass them back somehow.

10.3.2.6 free, flushnode and flushlist

The next one frees node n from TEX's memory. Be careful: no checks are done on whether this node is still pointed to from a register or some next field: it is up to you to make sure that the internal data structures remain correct. Fields that point to nodes or lists are flushed too. So, when you used their content for something else you need to set them to nil first.

```
<node> next = node.free(<node> n)
flushnode(<node> n)
```

The free function returns the next field of the freed node, while the flushnode alternative returns nothing.

A list starting with node n can be flushed from TEX's memory too. Be careful: no checks are done on whether any of these nodes is still pointed to from a register or some next field: it is up to you to make sure that the internal data structures remain correct.

```
node.flushlist(<node> n)
```

When you free for instance a discretionary node, flushlist is applied to the pre, post, replace

so you don't need to do that yourself. Assigning them `nil` won't free those lists!

10.3.2.7 copy and copylist

This creates a deep copy of node `n`, including all nested lists as in the case of a hlist or vlist node. Only the `next` field is not copied.

```
<node> m = node.copy(<node> n)
```

A deep copy of the node list that starts at `n` can be created too. If `m` is also given, the copy stops just before node `m`.

```
<node> m = node.copylist(<node> n)
<node> m = node.copylist(<node> n, <node> m)
```

Note that you cannot copy attribute lists this way. However, there is normally no need to copy attribute lists as when you do assignments to the `attr` field or make changes to specific attributes, the needed copying and freeing takes place automatically. When you change a value of an attribute *in* a list, it will affect all the nodes that share that list.

10.3.2.8 write

```
node.write(<node> n)
```

This function will append a node list to TEX's 'current list'. The node list is not deep-copied! There is no error checking either! You might need to enforce horizontal mode in order for this to work as expected.

10.3.3 Manipulating lists

10.3.3.1 slide

This helper makes sure that the node list is double linked and returns the found tail node.

```
<node> tail = node.slide(<node> n)
```

In most cases TEX itself only uses `next` pointers but your other callbacks might expect proper `prev` pointers too. So, when you run into issues or are in doubt, apply the slide function before you return the list.

10.3.3.2 tail

```
<node> m = node.tail(<node> n)
```

Returns the last node of the node list that starts at `n`.

10.3.3.3 length and count

```
<number> i = node.length(<node> n)
<number> i = node.length(<node> n, <node> m)
```

Returns the number of nodes contained in the node list that starts at `n`. If `m` is also supplied it stops at `m` instead of at the end of the list. The node `m` is not counted.

```
<number> i = node.count(<number> id, <node> n)
<number> i = node.count(<number> id, <node> n, <node> m)
```

Returns the number of nodes contained in the node list that starts at `n` that have a matching `id` field. If `m` is also supplied, counting stops at `m` instead of at the end of the list. The node `m` is not counted. This function also accept string `id`'s.

10.3.3.4 remove

```
<node> head, current, removed =
    node.remove(<node> head, <node> current)
<node> head, current =
    node.remove(<node> head, <node> current, <boolean> true)
```

This function removes the node `current` from the list following `head`. It is your responsibility to make sure it is really part of that list. The return values are the new `head` and `current` nodes. The returned `current` is the node following the `current` in the calling argument, and is only passed back as a convenience (or `nil`, if there is no such node). The returned `head` is more important, because if the function is called with `current` equal to `head`, it will be changed. When the third argument is passed, the node is freed.

10.3.3.5 insertbefore

```
<node> head, new = node.insertbefore(<node> head, <node> current, <node> new)
```

This function inserts the node `new` before `current` into the list following `head`. It is your responsibility to make sure that `current` is really part of that list. The return values are the (potentially mutated) `head` and the node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

10.3.3.6 insertafter

```
<node> head, new = node.insertafter(<node> head, <node> current, <node> new)
```

This function inserts the node `new` after `current` into the list following `head`. It is your responsibility to make sure that `current` is really part of that list. The return values are the `head` and the node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

10.3.3.7 lastnode

```
<node> n = node.lastnode()
```

This function pops the last node from TEX's 'current list'. It returns that node, or `nil` if the current list is empty.

10.3.3.8 traverse

```
<node> t, id, subtype = node.traverse(<node> n)
```

This is a Lua iterator that loops over the node list that starts at `n`. Typically code looks like this:

```
for n in node.traverse(head) do
   ...
end
```

is functionally equivalent to:

```
do
  local n
  local function f (head,var)
    local t
    if var == nil then
       t = head
    else
       t = var.next
    end
    return t
  end
  while true do
    n = f (head, n)
    if n == nil then break end
    ...
  end
end
```

It should be clear from the definition of the function `f` that even though it is possible to add or remove nodes from the node list while traversing, you have to take great care to make sure all the `next` (and `prev`) pointers remain valid.

If the above is unclear to you, see the section 'For Statement' in the Lua Reference Manual.

10.3.3.9 traverse_id

```
<node> t, subtype = node.traverse_id(<number> id, <node> n)
```

This is an iterator that loops over all the nodes in the list that starts at `n` that have a matching `id` field.

See the previous section for details. The change is in the local function `f`, which now does an extra while loop checking against the upvalue `id`:

```
 local function f(head,var)
   local t
   if var == nil then
      t = head
```

```
   else
      t = var.next
   end
   while not t.id == id do
      t = t.next
   end
   return t
end
```

10.3.3.10 traverse_char and traverse_glyph

The `traverse_char` iterator loops over the `glyph` nodes in a list. Only nodes with a subtype less than 256 are seen.

```
<direct> n, font, char = node.direct.traverse_char(<direct> n)
```

The `traverse_glyph` iterator loops over a list and returns the list and filters all glyphs:

```
<direct> n, font, char = node.traverse_glyph(<direct> n)
```

These functions are only available for direct nodes.

10.3.3.11 traverse_list

This iterator loops over the `hlist` and `vlist` nodes in a list.

```
<direct> n, id, subtype, list = node.traverse_list(<direct> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. This function is only available for direct nodes.

10.3.3.12 traverse_content

This iterator loops over nodes that have content: `hlist`, `vlist`, `glue` with leaders, `glyphs`, `disc` and `rules` nodes.

```
<direct> n, id, subtype[, list|leader] = node.traverse_list(<direct> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. This function is only available for direct nodes.

10.3.3.13 Reverse traversing

The traversers also support backward traversal. An optional extra boolean triggers this. Yet another optional boolean will automatically start at the end of the given list.

```
\setbox0\hbox{1 2 3 4 5}

local l = tex.box[0].list
for n in node.traverse(l) do
```

```
    print("1>",n)
end
for n in node.traverse(l,true) do
    print("2>",n)
end
for n in node.traverse(l,true,true) do
    print("3>",n)
end
for n in node.traverse_id(nodes.nodecodes.glyph,l) do
    print("4>",n)
end
for n in node.traverse_id(nodes.nodecodes.glyph,l,true) do
    print("5>",n)
end
for n in node.traverse_id(nodes.nodecodes.glyph,l,true,true) do
    print("6>",n)
end
```

This produces something similar to this (the glyph subtype indicates that it has been processed by the font handlers):

```
1>     <node :    nil <=   1112 =>    590 : glyph 32768>
1>     <node :   1112 <=    590 =>   1120 : glue spaceskip>
1>     <node :    590 <=   1120 =>    849 : glyph 32768>
1>     <node :   1120 <=    849 =>   1128 : glue spaceskip>
1>     <node :    849 <=   1128 =>    880 : glyph 32768>
1>     <node :   1128 <=    880 =>   1136 : glue spaceskip>
1>     <node :    880 <=   1136 =>   1020 : glyph 32768>
1>     <node :   1136 <=   1020 =>   1144 : glue spaceskip>
1>     <node :   1020 <=   1144 =>    nil : glyph 32768>
2>     <node :    nil <=   1112 =>    590 : glyph 32768>
3>     <node :   1020 <=   1144 =>    nil : glyph 32768>
3>     <node :   1136 <=   1020 =>   1144 : glue spaceskip>
3>     <node :    880 <=   1136 =>   1020 : glyph 32768>
3>     <node :   1128 <=    880 =>   1136 : glue spaceskip>
3>     <node :    849 <=   1128 =>    880 : glyph 32768>
3>     <node :   1120 <=    849 =>   1128 : glue spaceskip>
3>     <node :    590 <=   1120 =>    849 : glyph 32768>
3>     <node :   1112 <=    590 =>   1120 : glue spaceskip>
3>     <node :    nil <=   1112 =>    590 : glyph 32768>
4>     <node :    nil <=   1112 =>    590 : glyph 32768>
4>     <node :    590 <=   1120 =>    849 : glyph 32768>
4>     <node :    849 <=   1128 =>    880 : glyph 32768>
4>     <node :    880 <=   1136 =>   1020 : glyph 32768>
4>     <node :   1020 <=   1144 =>    nil : glyph 32768>
5>     <node :    nil <=   1112 =>    590 : glyph 32768>
6>     <node :   1020 <=   1144 =>    nil : glyph 32768>
```

```
6>     <node :    880 <=   1136 =>   1020 : glyph 32768>
6>     <node :    849 <=   1128 =>    880 : glyph 32768>
6>     <node :    590 <=   1120 =>    849 : glyph 32768>
6>     <node :    nil <=   1112 =>    590 : glyph 32768>
```

10.3.3.14 find_node

This helper returns the location of the first match at or after node `n`:

```
<node> n = node.find_node(<node> n, <integer> subtype)
<node> n, subtype = node.find_node(<node> n)
```

10.3.4 Glue handling

10.3.4.1 setglue

You can set the five properties of a glue in one go. If a non-numeric value is passed the property becomes zero.

```
node.setglue(<node> n)
node.setglue(<node> n,width,stretch,shrink,stretch_order,shrink_order)
```

When you pass values, only arguments that are numbers are assigned so

```
node.setglue(n,655360,false,65536)
```

will only adapt the width and shrink.

When a list node is passed, you set the glue, order and sign instead.

10.3.4.2 getglue

The next call will return 5 values or nothing when no glue is passed.

```
<integer> width, <integer> stretch, <integer> shrink, <integer> stretch_order,
    <integer> shrink_order = node.getglue(<node> n)
```

When the second argument is false, only the width is returned (this is consistent with `tex.get`).

When a list node is passed, you get back the glue that is set, the order of that glue and the sign.

10.3.4.3 iszeroglue

This function returns `true` when the width, stretch and shrink properties are zero.

```
<boolean> isglue = node.iszeroglue(<node> n)
```

10.3.5 Attribute handling

10.3.5.1 Attributes

Assignments to attributes registers result in assigning lists with set attributes to nodes and the implementation is non-trivial because the value that is attached to a node is essentially a (sorted) sparse array of key-value pairs. It is generally easiest to deal with attribute lists and attributes by using the dedicated functions in the `node` library.

10.3.5.2 `attribute_list` nodes

An `attribute_list` item is used as a head pointer for a list of attribute items. It has only one user-visible field:

FIELD	TYPE	EXPLANATION
`next`	node	pointer to the first attribute

10.3.5.3 `attr` nodes

A normal node's attribute field will point to an item of type `attribute_list`, and the `next` field in that item will point to the first defined 'attribute' item, whose `next` will point to the second 'attribute' item, etc.

FIELD	TYPE	EXPLANATION
`next`	node	pointer to the next attribute
`number`	number	the attribute type id
`value`	number	the attribute value

As mentioned it's better to use the official helpers rather than edit these fields directly. For instance the `prev` field is used for other purposes and there is no double linked list.

10.3.5.4 `currentattr`

This returns the currently active list of attributes, if there is one.

```
<node> m = node.currentattr()
```

The intended usage of `currentattr` is as follows:

```
local x1 = node.new("glyph")
x1.attr = node.currentattr()
local x2 = node.new("glyph")
x2.attr = node.currentattr()
```

or:

```
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.currentattr()
```

```
x1.attr = ca
x2.attr = ca
```

The attribute lists are ref counted and the assignment takes care of incrementing the refcount. You cannot expect the value `ca` to be valid any more when you assign attributes (using `tex.setattribute`) or when control has been passed back to TEX.

10.3.5.5 hasattribute

```
<number> v = node.hasattribute(<node> n, <number> id)
<number> v = node.hasattribute(<node> n, <number> id, <number> val)
```

Tests if a node has the attribute with number `id` set. If `val` is also supplied, also tests if the value matches `val`. It returns the value, or, if no match is found, `nil`.

10.3.5.6 getattribute

```
<number> v = node.getattribute(<node> n, <number> id)
```

Tests if a node has an attribute with number `id` set. It returns the value, or, if no match is found, `nil`. If no `id` is given then the zero attributes is assumed.

10.3.5.7 findattribute

```
<number> v, <node> n = node.findattribute(<node> n, <number> id)
```

Finds the first node that has attribute with number `id` set. It returns the value and the node if there is a match and otherwise nothing.

10.3.5.8 setattribute

```
node.setattribute(<node> n, <number> id, <number> val)
```

Sets the attribute with number `id` to the value `val`. Duplicate assignments are ignored.

10.3.5.9 unsetattribute

```
<number> v =
    node.unsetattribute(<node> n, <number> id)
<number> v =
    node.unsetattribute(<node> n, <number> id, <number> val)
```

Unsets the attribute with number `id`. If `val` is also supplied, it will only perform this operation if the value matches `val`. Missing attributes or attribute-value pairs are ignored.

If the attribute was actually deleted, returns its old value. Otherwise, returns `nil`.

10.3.6 Glyph handling

10.3.6.1 firstglyph

```
<node> n = node.firstglyph(<node> n)
<node> n = node.firstglyph(<node> n, <node> m)
```

Returns the first node in the list starting at `n` that is a glyph node with a subtype indicating it is a glyph, or `nil`. If `m` is given, processing stops at (but including) that node, otherwise processing stops at the end of the list.

10.3.6.2 is_char and is_glyph

The subtype of a glyph node signals if the glyph is already turned into a character reference or not.

```
<boolean> b = node.is_char(<node> n)
<boolean> b = node.is_glyph(<node> n)
```

10.3.6.3 hasglyph

This function returns the first glyph or disc node in the given list:

```
<node> n = node.hasglyph(<node> n)
```

10.3.6.4 ligaturing

```
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n)
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n, <node> m)
```

Apply TEX-style ligaturing to the specified nodelist. The tail node `m` is optional. The two returned nodes `h` and `t` are the new head and tail (both `n` and `m` can change into a new ligature).

10.3.6.5 kerning

```
<node> h, <node> t, <boolean> success = node.kerning(<node> n)
<node> h, <node> t, <boolean> success = node.kerning(<node> n, <node> m)
```

Apply TEX-style kerning to the specified node list. The tail node `m` is optional. The two returned nodes `h` and `t` are the head and tail (either one of these can be an inserted kern node, because special kernings with word boundaries are possible).

10.3.6.6 unprotectglyph[s]

```
node.unprotectglyph(<node> n)
node.unprotectglyphs(<node> n,[<node> n])
```

Subtracts 256 from all glyph node subtypes. This and the next function are helpers to convert from `characters` to `glyphs` during node processing. The second argument is optional and indicates the end of a range.

10.3.6.7 protectglyph[s]

```
node.protectglyph(<node> n)
node.protectglyphs(<node> n,[<node> n])
```

Adds 256 to all glyph node subtypes in the node list starting at `n`, except that if the value is 1, it adds only 255. The special handling of 1 means that `characters` will become `glyphs` after subtraction of 256. A single character can be marked by the singular call. The second argument is optional and indicates the end of a range.

10.3.6.8 protrusionskippable

```
<boolean> skippable = node.protrusionskippable(<node> n)
```

Returns `true` if, for the purpose of line boundary discovery when character protrusion is active, this node can be skipped.

10.3.6.9 checkdiscretionary, checkdiscretionaries

When you fool around with disc nodes you need to be aware of the fact that they have a special internal data structure. As long as you reassign the fields when you have extended the lists it's ok because then the tail pointers get updated, but when you add to list without reassigning you might end up in trouble when the linebreak routine kicks in. You can call this function to check the list for issues with disc nodes.

```
node.checkdiscretionary(<node> n)
node.checkdiscretionaries(<node> head)
```

The plural variant runs over all disc nodes in a list, the singular variant checks one node only (it also checks if the node is a disc node).

10.3.6.10 flattendiscretionaries

This function will remove the discretionaries in the list and inject the replace field when set.

```
<node> head, count = node.flattendiscretionaries(<node> n)
```

10.3.7 Packaging

10.3.7.1 hpack

This function creates a new hlist by packaging the list that begins at node `n` into a horizontal box. With only a single argument, this box is created using the natural width of its components. In the three argument form, `info` must be either `additional` or `exactly`, and `w` is the additional (`\hbox spread`) or exact (`\hbox to`) width to be used. The second return value is the badness of the generated box.

```
<node> h, <number> b =
    node.hpack(<node> n)
```

```
<node> h, <number> b =
    node.hpack(<node> n, <number> w, <string> info)
<node> h, <number> b =
    node.hpack(<node> n, <number> w, <string> info, <string> dir)
```

Caveat: there can be unexpected side-effects to this function, like updating some of the `\marks` and `\inserts`. Also note that the content of `h` is the original node list `n`: if you call `node.free(h)` you will also free the node list itself, unless you explicitly set the `list` field to `nil` beforehand. And in a similar way, calling `node.free(n)` will invalidate `h` as well!

10.3.7.2 vpack

This function creates a new vlist by packaging the list that begins at node `n` into a vertical box. With only a single argument, this box is created using the natural height of its components. In the three argument form, `info` must be either `additional` or `exactly`, and `w` is the additional (`\vbox spread`) or exact (`\vbox to`) height to be used.

```
<node> h, <number> b =
    node.vpack(<node> n)
<node> h, <number> b =
    node.vpack(<node> n, <number> w, <string> info)
<node> h, <number> b =
    node.vpack(<node> n, <number> w, <string> info, <string> dir)
```

The second return value is the badness of the generated box. See the description of `hpack` for a few memory allocation caveats.

10.3.7.3 prepend_prevdepth

This function is somewhat special in the sense that it is an experimental helper that adds the interlinespace to a line keeping the baselineskip and lineskip into account.

```
<node> n, <number> delta =
    node.prepend_prevdepth(<node> n,<number> prevdepth)
```

10.3.7.4 dimensions, rangedimensions, naturalwidth

```
<number> w, <number> h, <number> d  =
    node.dimensions(<node> n)
<number> w, <number> h, <number> d  =
    node.dimensions(<node> n, <node> t)
```

This function calculates the natural in-line dimensions of the node list starting at node `n` and terminating just before node `t` (or the end of the list, if there is no second argument). The return values are scaled points. An alternative format that starts with glue parameters as the first three arguments is also possible:

```
<number> w, <number> h, <number> d  =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order,
```

```
        <node> n)
<number> w, <number> h, <number> d  =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order,
        <node> n, <node> t)
```

This calling method takes glue settings into account and is especially useful for finding the actual width of a sublist of nodes that are already boxed, for example in code like this, which prints the width of the space in between the `a` and `b` as it would be if `\box0` was used as-is:

```
\setbox0 = \hbox to 20pt {a b}

\directlua{print (node.dimensions(
    tex.box[0].glue_set,
    tex.box[0].glue_sign,
    tex.box[0].glue_order,
    tex.box[0].head.next,
    node.tail(tex.box[0].head)
)) }
```

You need to keep in mind that this is one of the few places in TEX where floats are used, which means that you can get small differences in rounding when you compare the width reported by `hpack` with `dimensions`.

The second alternative saves a few lookups and can be more convenient in some cases:

```
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first)
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first, <node> last)
```

A simple and somewhat more efficient variant is this:

```
<number> w =
    node.naturalwidth(<node> start, <node> stop)
```

10.3.8 Math

10.3.8.1 mlisttohlist

```
<node> h =
    node.mlisttohlist(<node> n, <string> display_type, <boolean> penalties)
```

This runs the internal mlist to hlist conversion, converting the math list in `n` into the horizontal list `h`. The interface is exactly the same as for the callback `mlisttohlist`.

10.3.8.2 end_of_math

```
<node> t = node.end_of_math(<node> start)
```

Looks for and returns the next `math_node` following the `start`. If the given node is a math end node this helper returns that node, else it follows the list and returns the next math endnote. If no such node is found nil is returned.

10.4 Two access models

Deep down in TEX a node has a number which is a numeric entry in a memory table. In fact, this model, where TEX manages memory is real fast and one of the reasons why plugging in callbacks that operate on nodes is quite fast too. Each node gets a number that is in fact an index in the memory table and that number often is reported when you print node related information. You go from userdata nodes and there numeric references and back with:

```
<integer> d = node.todirect(<node> n))
<node> n = node.tonode(<integer> d))
```

The userdata model is rather robust as it is a virtual interface with some additional checking while the more direct access which uses the node numbers directly. However, even with userdata you can get into troubles when you free nodes that are no longer allocated or mess up lists. if you apply `tostring` to a node you see its internal (direct) number and id.

The first model provides key based access while the second always accesses fields via functions:

```
nodeobject.char
getfield(nodenumber,"char")
```

If you use the direct model, even if you know that you deal with numbers, you should not depend on that property but treat it as an abstraction just like traditional nodes. In fact, the fact that we use a simple basic datatype has the penalty that less checking can be done, but less checking is also the reason why it's somewhat faster. An important aspect is that one cannot mix both methods, but you can cast both models. So, multiplying a node number makes no sense.

So our advice is: use the indexed (table) approach when possible and investigate the direct one when speed might be a real issue. For that reason LuaTEX also provide the `get*` and `set*` functions in the top level node namespace. There is a limited set of getters. When implementing this direct approach the regular index by key variant was also optimized, so direct access only makes sense when nodes are accessed millions of times (which happens in some font processing for instance).

We're talking mostly of getters because setters are less important. Documents have not that many content related nodes and setting many thousands of properties is hardly a burden contrary to millions of consultations.

Normally you will access nodes like this:

```
local next = current.next
if next then
    -- do something
end
```

Here `next` is not a real field, but a virtual one. Accessing it results in a metatable method being

called. In practice it boils down to looking up the node type and based on the node type checking for the field name. In a worst case you have a node type that sits at the end of the lookup list and a field that is last in the lookup chain. However, in successive versions of LuaTEX these lookups have been optimized and the most frequently accessed nodes and fields have a higher priority.

Because in practice the `next` accessor results in a function call, there is some overhead involved. The next code does the same and performs a tiny bit faster (but not that much because it is still a function call but one that knows what to look up).

```
local next = node.next(current)
if next then
    -- do something
end
```

In the direct namespace there are more helpers and most of them are accompanied by setters. The getters and setters are clever enough to see what node is meant. We don't deal with whatsit nodes: their fields are always accessed by name. It doesn't make sense to add getters for all fields, we just identifier the most likely candidates. In complex documents, many node and fields types never get seen, or seen only a few times, but for instance glyphs are candidates for such optimization. The `node.direct` interface has some more helpers.[5]

The `setdisc` helper takes three (optional) arguments plus an optional fourth indicating the subtype. Its `getdisc` takes an optional boolean; when its value is `true` the tail nodes will also be returned. The `setfont` helper takes an optional second argument, it being the character. The directmode setter `setlink` takes a list of nodes and will link them, thereby ignoring `nil` entries. The first valid node is returned (beware: for good reason it assumes single nodes). For rarely used fields no helpers are provided and there are a few that probably are used seldom too but were added for consistency. You can of course always define additional accessors using `getfield` and `setfield` with little overhead. When the second argument of `setattributelist` is `true` the current attribute list is assumed.

The `reverse` function reverses a given list. The `exchange` function swaps two nodes; it takes upto three arguments: a head node, and one or two to be swapped nodes. When there is no third argument, it will assume that the node following node is to be used. So we have:

```
head = node.direct.reverse(head)
head = node.direct.exchange(head,first,[second])
```

In ConTEXt some of the not performance-critical userdata variants are emulated in Lua and not in the engine, so we retain downward compatibility.

FUNCTION	NODE	DIRECT	emulated
checkdiscretionaries	–	+	+
checkdiscretionary	–	+	+
copylist	+	+	
copy	+	+	
count	–	+	+

[5] We can define the helpers in the node namespace with `getfield` which is about as efficient, so at some point we might provide that as module.

currentattributes	+	+	
dimensions	–	+	+
effectiveglue	–	+	+
endofmath	–	+	+
findattributerange	–	+	
findattribute	–	+	+
findnode	–	+	
firstglyph	–	+	+
flattendiscretionaries	–	+	+
flushlist	+	+	
flushnode	+	+	
free	+	+	
getattributes	–	+	
getattribute	+	+	
getpropertiestable	+	+	
getsynctexfields	–	+	
getattributelist	–	+	
getboth	–	+	
getbox	–	+	
getchar	–	+	
getdata	–	+	
getdepth	–	+	
getdirection	–	+	
getdisc	–	+	
getexpansion	–	+	
getfam	–	+	
getfield	+	+	
getfont	–	+	
getglue	–	+	+
getglyphdata	–	+	
getglyphdimensions	–	+	+
getglyphscript	–	+	
getglyphstate	–	+	
getheight	–	+	
getid	–	+	
getindex	–	+	
getkerndimension	–	+	+
getkern	–	+	
getlanguage	–	+	
getleader	–	+	
getlist	–	+	
getnext	–	+	
getnormalizedline	–	+	
getnucleus	–	+	
getoffsets	–	+	
getoptions	–	+	

getorientation	−	+	
getparstate	−	+	
getpenalty	−	+	
getpost	−	+	
getprev	−	+	
getpre	−	+	
getproperty	+	+	
getreplace	−	+	
getscales	−	+	
getscript	−	+	
getshift	−	+	
getstate	−	+	
getsubpre	−	+	
getsubtype	−	+	
getsub	−	+	
getsuppre	−	+	
getsup	−	+	
gettotal	+	+	
getwhd	−	+	
getwidth	−	+	
getxscale	−	+	
getxyscale	−	+	
getyscale	−	+	
hasattribute	+	+	
hasdimensions	−	+	
hasfield	+	+	
hasglyphoption	−	+	+
hasglyph	−	+	+
hpack	−	+	+
hyphenating	−	+	+
ignoremathskip	−	+	
insertafter	+	+	
insertbefore	+	+	
ischar	−	+	
isdirect	−	+	
isglyph	−	+	
isnextchar	−	+	
isnextglyph	−	+	
isnode	+	+	
isprevchar	−	+	
isprevglyph	−	+	
isvalid	−	+	
iszeroglue	−	+	+
kerning	−	+	+
lastnode	−	+	+
length	−	+	+

ligaturing	−	+	+
makeextensible	−	+	+
migrate	−	+	
mlisttohlist	−	+	+
naturalwidth	−	+	+
new	+	+	
protectglyphs	−	+	+
protectglyph	−	+	+
protrusionskippable	−	+	+
rangedimensions	−	+	+
remove	+	+	
setattributes	−	+	
setattribute	+	+	
setsynctexfields	−	+	
setattributelist	−	+	
setboth	−	+	
setbox	−	+	
setchar	−	+	
setdata	−	+	
setdepth	−	+	
setdirection	−	+	
setdisc	−	+	
setexpansion	−	+	
setfam	−	+	
setfield	+	+	
setfont	−	+	
setglue	+	+	
setglyphdata	−	+	
setglyphscript	−	+	
setglyphstate	−	+	
setheight	−	+	
setindex	−	+	
setkern	−	+	
setlanguage	−	+	
setleader	−	+	
setlink	−	+	
setlist	−	+	
setnext	−	+	
setnucleus	−	+	
setoffsets	−	+	
setoptions	−	+	
setorientation	−	+	
setpenalty	−	+	
setpost	−	+	
setprev	−	+	
setpre	−	+	

setproperty	+	+	
setreplace	−	+	
setscales	−	+	
setscript	−	+	
setshift	−	+	
setsplit	−	+	
setstate	−	+	
setsubpre	−	+	
setsubtype	−	+	
setsub	−	+	
setsuppre	−	+	
setsup	−	+	
setwhd	−	+	
setwidth	−	+	
slide	−	+	+
startofpar	−	+	
subtype	−	−	
tail	+	+	
todirect	−	+	
tonode	−	+	
tostring	+	−	
total	−	+	
tovaliddirect	−	+	
traverse_char	+	+	
traverse_content	+	+	
traverse_glyph	+	+	
traverse_id	+	+	
traverse_list	+	+	
traverse	+	+	
type	+	−	
unprotectglyphs	−	+	+
unprotectglyph	−	+	+
unsetattributes	−	+	
unsetattribute	+	+	
usedlist	−	+	+
usesfont	−	+	+
verticalbreak	−	+	
vpack	−	+	+
write	+	+	

The `node.next` and `node.prev` functions will stay but for consistency there are variants called `getnext` and `getprev`. We had to use `get` because `node.id` and `node.subtype` are already taken for providing meta information about nodes. Note: The getters do only basic checking for valid keys. You should just stick to the keys mentioned in the sections that describe node properties.

Some of the getters and setters handle multiple node types, given that the field is relevant. In that case, some field names are considered similar (like `kern` and `width`, or `data` and `value`). In

retrospect we could have normalized field names better but we decided to stick to the original (internal) names as much as possible. After all, at the Lua end one can easily create synonyms.

Some nodes have indirect references. For instance a math character refers to a family instead of a font. In that case we provide a virtual font field as accessor. So, `getfont` and `.font` can be used on them. The same is true for the `width`, `height` and `depth` of glue nodes. These actually access the spec node properties, and here we can set as well as get the values.

In some places LuaTEX can do a bit of extra checking for valid node lists and you can enable that with:

```
node.fix_node_lists(<boolean> b)
```

You can set and query the SyncTEX fields, a file number aka tag and a line number, for a glue, kern, hlist, vlist, rule and math nodes as well as glyph nodes (although this last one is not used in native SyncTEX).

```
node.setsynctexfields(<integer> f, <integer> l)
<integer> f, <integer> l =
    node.getsynctexfields(<node> n)
```

Of course you need to know what you're doing as no checking on sane values takes place. Also, the synctex interpreter used in editors is rather peculiar and has some assumptions (heuristics).

10.5 Normalization

As an experiment the lines resulting from paragraph construction can be normalized. There are several modes, that can be set and queried with:

```
node.direct.setnormalize(<integer> n)
<integer> n = node.direct.getnormalize()
```

The state of a line (a hlist) can be queried with:

```
<integer> leftskip, <integer> rightskip,
    <integer> lefthangskip, <integer> righthangskip,
    <node> head, <node> tail,
    <integer> parindent, <integer> parfillskip = node.direct.getnormalized()
```

The modes accumulate, so mode 4 includes 1 upto 3:

VALUE	EXPLANATION
1	left and right skips and directions
2	indentation and parfill skip
3	hanging indentation and par shapes
4	idem but before left and right skips
5	inject compensation for overflow

This is experimental code and might take a while to become frozen.

10.6 Properties

Attributes are a convenient way to relate extra information to a node. You can assign them at the TEX end as well as at the Lua end and consult them at the Lua end. One big advantage is that they obey grouping. They are linked lists and normally checking for them is pretty efficient, even if you use a lot of them. A macro package has to provide some way to manage these attributes at the TEX end because otherwise clashes in their usage can occur.

Each node also can have a properties table and you can assign values to this table using the `setproperty` function and get properties using the `getproperty` function. Managing properties is way more demanding than managing attributes.

Take the following example:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,"foo")
    print(node.getproperty(n))

    node.setproperty(n,"bar")
    print(node.getproperty(n))

    node.free(n)
}
```

This will print `foo` and `bar` which in itself is not that useful when multiple mechanisms want to use this feature. A variant is:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,{ one = "foo", two = "bar" })
    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

This time we store two properties with the node. It really makes sense to have a table as property because that way we can store more. But in order for that to work well you need to do it this way:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)

    if not t then
```

```
        t = { }
        node.setproperty(n,t)
    end
    t.one = "foo"
    t.two = "bar"

    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

Here our own properties will not overwrite other users properties unless of course they use the same keys. So, eventually you will end up with something:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)

    if not t then
        t = { }
        node.setproperty(n,t)
    end
    t.myself = { one = "foo", two = "bar" }

    print(node.getproperty(n).myself.one)
    print(node.getproperty(n).myself.two)

    node.free(n)
}
```

This assumes that only you use `myself` as subtable. The possibilities are endless but care is needed. For instance, the generic font handler that ships with ConTEXt uses the `injections` subtable and you should not mess with that one!

There are a few helper functions that you normally should not touch as user: `getpropertiestable` and will give the table that stores properties (using direct entries) and you can best not mess too much with that one either because LuaTEX itself will make sure that entries related to nodes will get wiped when nodes get freed, so that the Lua garbage collector can do its job. In fact, the main reason why we have this mechanism is that it saves the user (or macro package) some work. One can easily write a property mechanism in Lua where after a shipout properties gets cleaned up but it's not entirely trivial to make sure that with each freed node also its properties get freed, due to the fact that there can be nodes left over for a next page. And having a callback bound to the node deallocator would add way to much overhead.

When we copy a node list that has a table as property, there are several possibilities: we do the same as a new node, we copy the entry to the table in properties (a reference), we do a deep copy

of a table in the properties, we create a new table and give it the original one as a metatable. After some experiments (that also included timing) with these scenarios we decided that a deep copy made no sense, nor did nilling. In the end both the shallow copy and the metatable variant were both ok, although the second one is slower. The most important aspect to keep in mind is that references to other nodes in properties no longer can be valid for that copy. We could use two tables (one unique and one shared) or metatables but that only complicates matters.

When defining a new node, we could already allocate a table but it is rather easy to do that at the lua end e.g. using a metatable `__index` method. That way it is under macro package control. When deleting a node, we could keep the slot (e.g. setting it to false) but it could make memory consumption raise unneeded when we have temporary large node lists and after that only small lists. Both are not done.

So in the end this is what happens now: when a node is copied, and it has a table as property, the new node will share that table. If the second argument of `set_properties_mode` is `true` then a metatable approach is chosen: the copy gets its own table with the original table as metatable. If you use the generic font loader the mode is enabled that way.

A few more experiments were done. For instance: copy attributes to the properties so that we have fast access at the Lua end. In the end the overhead is not compensated by speed and convenience, in fact, attributes are not that slow when it comes to accessing them. So this was rejected.

Another experiment concerned a bitset in the node but again the gain compared to attributes was neglectable and given the small amount of available bits it also demands a pretty strong agreement over what bit represents what, and this is unlikely to succeed in the TEX community. It doesn't pay off.

Just in case one wonders why properties make sense: it is not so much speed that we gain, but more convenience: storing all kinds of (temporary) data in attributes is no fun and this mechanism makes sure that properties are cleaned up when a node is freed. Also, the advantage of a more or less global properties table is that we stay at the Lua end. An alternative is to store a reference in the node itself but that is complicated by the fact that the register has some limitations (no numeric keys) and we also don't want to mess with it too much.

174 Nodes

11 Lua callbacks

11.1 Registering callbacks

The callbacks are a moving target. Don't bother me with questions about them.

This library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kinds of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In mosty cases the second category is expected to behave similar to the built in functionality because in a next step specific data is expected. For instance, you can replace the hyphenation routine. The function gets a list that can be hyphenated (or not). The final list should be valid and is (normally) used for constructing a paragraph. Another function can replace the ligature builder and/or kerner. Doing something else is possible but in the end might not give the user the expected outcome.

The first thing you need to do is registering a callback:

```
id = callback.register(<string> callback_name, <function> func)
id = callback.register(<string> callback_name, nil)
id = callback.register(<string> callback_name, false)
```

Here the `callback_name` is a predefined callback name, see below. The function returns the internal `id` of the callback or `nil`, if the callback could not be registered.

LuaTEX internalizes the callback function in such a way that it does not matter if you redefine a function accidentally.

Callback assignments are always global. You can use the special value `nil` instead of a function for clearing the callback.

For some minor speed gain, you can assign the boolean `false` to the non-file related callbacks, doing so will prevent LuaTEX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know *exactly* what you are doing!

```
<table> info =
    callback.list()
```

The keys in the table are the known callback names, the value is a boolean where `true` means that the callback is currently set (active).

```
<function> f = callback.find(callback_name)
```

If the callback is not set, `find` returns `nil`. The `known` function can be used to check if a callback is supported.

```
if callback.known("foo") then ... end
```

11.2 File related callbacks

11.2.1 find_format_file and find_log_file

These callbacks are called as:

```
<string> actualname =
    function (<string> askedname)
```

The askedname is a format file for reading (the format file for writing is always opened in the current directory) or a log file for writing.

11.2.2 open_data_file

This callback function gets a filename passed:

```
<table> env = function (<string> filename)
```

The return value is either the boolean value false or a table with two functions. A mandate reader function fill be called once for each new line to be read, the optional close function will be called once LuaTEX is done with the file.

LuaTEX never looks at the rest of the table, so you can use it to store your private per-file data. Both the callback functions will receive the table as their only argument.

11.3 Data processing callbacks

11.3.1 process_jobname

This callback allows you to change the jobname given by \jobname in TEX and tex.jobname in Lua. It does not affect the internal job name or the name of the output or log files.

```
function(<string> jobname)
    return <string> adjusted_jobname
end
```

The only argument is the actual job name; you should not use tex.jobname inside this function or infinite recursion may occur. If you return nil, LuaTEX will pretend your callback never happened. This callback does not replace any internal code.

11.4 Node list processing callbacks

The description of nodes and node lists is in chapter 10.

11.4.1 contribute_filter

This callback is called when LuaTEX adds contents to list:

```
function(<string> extrainfo)
end
```

The string reports the group code. From this you can deduce from what list you can give a treat.

VALUE	EXPLANATION
pre_box	interline material is being added
pre_adjust	\vadjust material is being added
box	a typeset box is being added (always called)
adjust	\vadjust material is being added

11.4.2 buildpage_filter

This callback is called whenever LuaTEX is ready to move stuff to the main vertical list. You can use this callback to do specialized manipulation of the page building stage like imposition or column balancing.

```
function(<string> extrainfo)
end
```

The string extrainfo gives some additional information about what TEX's state is with respect to the 'current page'. The possible values for the buildpage_filter callback are:

VALUE	EXPLANATION
alignment	a (partial) alignment is being added
after_output	an output routine has just finished
new_graf	the beginning of a new paragraph
vmode_par	\par was found in vertical mode
hmode_par	\par was found in horizontal mode
insert	an insert is added
penalty	a penalty (in vertical mode)
before_display	immediately before a display starts
after_display	a display is finished
end	LuaTEX is terminating (it's all over)

11.4.3 build_page_insert

This callback is called when the pagebuilder adds an insert. There is not much control over this

mechanism but this callback permits some last minute manipulations of the spacing before an insert, something that might be handy when for instance multiple inserts (types) are appended in a row.

```
function(<number> n, <number> i)
    return <number> register
end
```

with

VALUE	EXPLANATION
n	the insert class
i	the order of the insert

The return value is a number indicating the skip register to use for the prepended spacing. This permits for instance a different top space (when `i` equals one) and intermediate space (when `i` is larger than one). Of course you can mess with the insert box but you need to make sure that LuaTEX is happy afterwards.

11.4.4 pre_linebreak_filter

This callback is called just before LuaTEX starts converting a list of nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

```
function(<node> head, <string> groupcode)
    return <node> newhead
end
```

The string called `groupcode` identifies the nodelist's context within TEX's processing. The range of possibilities is given in the table below, but not all of those can actually appear in `pre_linebreak_filter`, some are for the `hpack_filter` and `vpack_filter` callbacks that will be explained in the next two paragraphs.

VALUE	EXPLANATION
<empty>	main vertical list
hbox	`\hbox` in horizontal mode
adjusted_hbox	`\hbox` in vertical mode
vbox	`\vbox`
vtop	`\vtop`
align	`\halign` or `\valign`
disc	discretionaries
insert	packaging an insert
vcenter	`\vcenter`
local_box	`\localleftbox` or `\localrightbox`
split_off	top of a `\vsplit`
split_keep	remainder of a `\vsplit`

align_set	alignment cell
fin_row	alignment row

As for all the callbacks that deal with nodes, the return value can be one of three things:

- boolean true signals successful processing
- <node> signals that the 'head' node should be replaced by the returned node
- boolean false signals that the 'head' node list should be ignored and flushed from memory

This callback does not replace any internal code.

11.4.5 linebreak_filter

This callback replaces LuaTEX's line breaking algorithm.

```
function(<node> head, <boolean> is_display)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the main vertical list, the boolean argument is true if this paragraph is interrupted by a following math display.

If you return something that is not a <node>, LuaTEX will apply the internal linebreak algorithm on the list that starts at <head>. Otherwise, the <node> you return is supposed to be the head of a list of nodes that are all allowed in vertical mode, and at least one of those has to represent an \hbox. Failure to do so will result in a fatal error.

Setting this callback to false is possible, but dangerous, because it is possible you will end up in an unfixable 'deadcycles loop'.

11.4.6 append_to_vlist_filter

This callback is called whenever LuaTEX adds a box to a vertical list (the mirrored argument is obsolete):

```
function(<node> box, <string> locationcode, <number> prevdepth)
    return list [, prevdepth [, checkdepth ] ]
end
```

It is ok to return nothing or nil in which case you also need to flush the box or deal with it yourself. The prevdepth is also optional. Locations are box, alignment, equation, equation_number and post_linebreak. When the third argument returned is true the normal prevdepth correction will be applied, based on the first node.

11.4.7 post_linebreak_filter

This callback is called just after LuaTEX has converted a list of nodes into a stack of \hboxes.

```
function(<node> head, <string> groupcode)
    return <node> newhead
end
```

This callback does not replace any internal code.

11.4.8 glyph_run

When set this callback is triggered when TEX normally handles the ligaturing and kerning. In LuaTEX you use the `hpack_filter` and `per_linebreak_filter` callbacks for that (where each passes different arguments). This callback doesn't get triggered when there are no glyphs (in LuaTEX this optimization is controlled by a a variable).

```
function(<node> head, <string> groupcode, <number> direction])
    return <node> newhead
end
```

The traditional TEX font processing is bypassed so you need to take care of that with the helpers. (For the moment we keep the ligaturing and kerning callbacks but they are kind of obsolete.)

11.4.9 hpack_filter

This callback is called when TEX is ready to start boxing some horizontal mode material. Math items and line boxes are ignored at the moment.

```
function(<node> head, <string> groupcode, <number> size,
         <string> packtype [, <number> direction] [, <node> attributelist])
    return <node> newhead
end
```

The `packtype` is either `additional` or `exactly`. If `additional`, then the `size` is a `\hbox spread ...` argument. If `exactly`, then the `size` is a `\hbox to ...`. In both cases, the number is in scaled points.

This callback does not replace any internal code.

11.4.10 vpack_filter

This callback is called when TEX is ready to start boxing some vertical mode material. Math displays are ignored at the moment.

This function is very similar to the `hpack_filter`. Besides the fact that it is called at different moments, there is an extra variable that matches TEX's `\maxdepth` setting.

```
function(<node> head, <string> groupcode, <number> size, <string> packtype,
        <number> maxdepth [, <number> direction] [, <node> attributelist]))
    return <node> newhead
end
```

This callback does not replace any internal code.

11.4.11 hpack_quality

This callback can be used to intercept the overfull messages that can result from packing a horizontal list (as happens in the par builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
         <number> last)
    return <node> whatever
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed (when protrusion or expansion is enabled, this is an intermediate list). Optionally you can return a node, for instance an overfull rule indicator. That node will be appended to the list (just like TEX's own rule would).

11.4.12 vpack_quality

This callback can be used to intercept the overfull messages that can result from packing a vertical list (as happens in the page builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
         <number> last)
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed.

11.4.13 process_rule

This is an experimental callback. It can be used with rules of subtype 4 (user). The callback gets three arguments: the node, the width and the height. The callback can use `pdf.print` to write code to the pdf file but beware of not messing up the final result. No checking is done.

11.4.14 pre_output_filter

This callback is called when TEX is ready to start boxing the box 255 for `\output`.

```
function(<node> head, <string> groupcode, <number> size, <string> packtype,
        <number> maxdepth [, <number> direction])
    return <node> newhead
end
```

This callback does not replace any internal code.

11.4.15 hyphenate

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to insert discretionary nodes in the node list it receives.

Setting this callback to `false` will prevent the internal discretionary insertion pass.

11.4.16 ligaturing

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply ligaturing to the node list it receives.

You don't have to worry about return values because the `head` node that is passed on to the callback is guaranteed not to be a glyph_node (if need be, a temporary node will be prepended), and therefore it cannot be affected by the mutations that take place. After the callback, the internal value of the 'tail of the list' will be recalculated.

The `next` of `head` is guaranteed to be non-nil.

The `next` of `tail` is guaranteed to be nil, and therefore the second callback argument can often be ignored. It is provided for orthogonality, and because it can sometimes be handy when special processing has to take place.

Setting this callback to `false` will prevent the internal ligature creation pass.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTEX into panic mode.

11.4.17 kerning

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply kerning between the nodes in the node list it receives. See `ligaturing` for calling conventions.

Setting this callback to `false` will prevent the internal kern insertion pass.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTEX into panic mode.

11.4.18 insert_par

Each paragraph starts with a local par node that keeps track of for instance the direction. You can hook a callback into the creator:

```
function(<node> par, <string> location)
```

```
end
```

There is no return value and you should make sure that the node stays valid as otherwise TEX can get confused.

11.4.19 mlist_to_hlist

This callback replaces LuaTEX's math list to node list conversion algorithm.

```
function(<node> head, <string> display_type, <boolean> need_penalties)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the vertical or horizontal list, the string argument is either 'text' or 'display' depending on the current math mode, the boolean argument is `true` if penalties have to be inserted in this list, `false` otherwise.

Setting this callback to `false` is bad, it will almost certainly result in an endless loop.

11.5 Information reporting callbacks

11.5.1 pre_dump

```
function()
end
```

This function is called just before dumping to a format file starts. It does not replace any code and there are neither arguments nor return values.

11.5.2 start_run

```
function()
end
```

This callback replaces the code that prints LuaTEX's banner. Note that for successful use, this callback has to be set in the Lua initialization script, otherwise it will be seen only after the run has already started.

11.5.3 stop_run

```
function()
end
```

This callback replaces the code that prints LuaTEX's statistics and 'output written to' messages. The engine can still do housekeeping and therefore you should not rely on this hook for postprocessing the pdf or log file.

11.5.4 intercept_tex_error, intercept_lua_error

```
function()
end
```

This callback is run from inside the TEX error function, and the idea is to allow you to do some extra reporting on top of what TEX already does (none of the normal actions are removed). You may find some of the values in the `status` table useful. The TEX related callback gets two arguments: the current processing mode and a boolean indicating if there was a runaway.

11.5.5 show_error_message and show_warning_message

```
function()
end
```

These callback replaces the code that prints the error message. The usual interaction after the message is not affected.

11.5.6 start_file

```
function(category,filename)
end
```

This callback replaces the code that LuaTEX prints when a file is opened like `(filename` for regular files. The category is a number:

VALUE	MEANING
1	a normal data file, like a TEX source
2	a font map coupling font names to resources
3	an image file (`png`, `pdf`, etc)
4	an embedded font subset
5	a fully embedded font

11.5.7 stop_file

```
function(category)
end
```

This callback replaces the code that LuaTEX prints when a file is closed like the `)` for regular files.

11.5.8 wrapup_run

This callback is called after the pdf and log files are closed. Use it at your own risk.

11.6 Font-related callbacks

11.6.1 define_font

```
function(<string> name, <number> size)
    return <number> id
end
```

The string `name` is the filename part of the font specification, as given by the user.

The number `size` is a bit special:

- If it is positive, it specifies an 'at size' in scaled points.
- If it is negative, its absolute value represents a 'scaled' setting relative to the design size of the font.

The font can be defined with `font.define` which returns a font identifier that can be returned in the callback. So, contrary to LuaTEX, in LuaMetaTEX we only accept a number.

The internal structure of the `font` table that is passed to `font.define` is explained in chapter 6. That table is saved internally, so you can put extra fields in the table for your later Lua code to use. In alternative, `retval` can be a previously defined fontid. This is useful if a previous definition can be reused instead of creating a whole new font structure.

Setting this callback to `false` is pointless as it will prevent font loading completely but will nevertheless generate errors.

11.6.2 show_whatsit

Because we only have a generic whatsit it is up to the macro package to provide details when tracing them.

```
function(<node> whatsit, <number> indentation,
    <number> tracinglevel, <number> currentlevel, <number> inputlevel)
    -- no return value
end
```

The indentation tells how many periods are to be typeset if you want to be compatible with the rest of tracing. The tracinglevels indicates if the current level and|/or input level are shown cf. `\tracinglevels`. Of course one is free to show whatever in whatever way suits the whatsit best.

12 The TEX related libraries

12.1 The lua library

12.1.1 Version information

This version of the used Lua interpreter (currently Lua 5.4) can be queried with:

```
<string> v = lua.getversion()
```

The name of used startup file, if at all, is returned by:

```
<string> s = lua.getstartupfile()
```

For this document the reported value is:

c:/data/develop/tex-context/tex/texmf-cache/luametatex-cache/context/764bd4e1ce0f004ab3cec90018f8b80a/formats/luametatex/cont-en.lui

12.1.2 Table allocators

Sometimes performance (and memory usage) can benefit a little from it preallocating a table with newtable:

```
<table> t = lua.newtable(100,5000)
```

This preallocates 100 hash entries and 5000 index entries. The newindex function create an indexed table with preset values:

```
<table> t = lua.newindex(2500,true)
```

12.1.3 Bytecode registers

Lua registers can be used to store Lua code chunks. The accepted values for assignments are functions and nil. Likewise, the retrieved value is either a function or nil.

```
lua.bytecode[<number> n] = <function> f
<function> f = lua.bytecode[<number> n] % -- f()
```

The contents of the lua.bytecode array is stored inside the format file as actual Lua bytecode, so it can also be used to preload Lua code. The function must not contain any upvalues. The associated function calls are:

```
lua.setbytecode(<number> n, <function> f)
<function> f = lua.getbytecode(<number> n)
```

Note: Since a Lua file loaded using `loadfile(filename)` is essentially an anonymous function, a complete file can be stored in a bytecode register like this:

```
lua.setbytecode(n,loadfile(filename))
```

Now all definitions (functions, variables) contained in the file can be created by executing this bytecode register:

```
lua.callbytecode(n)
```

Note that the path of the file is stored in the Lua bytecode to be used in stack backtraces and therefore dumped into the format file if the above code is used in iniTEX. If it contains private information, i.e. the user name, this information is then contained in the format file as well. This should be kept in mind when preloading files into a bytecode register in iniTEX.

12.1.4 Introspection

The `getstacktop` function return a number indicating how full the Lua stack is. This function only makes sense as breakpoint when checking some mechanism going haywire.

There are four time related helpers. The `getruntime` function returns the time passed since startup. The `getcurrenttime` does what its name says. Just play with them to see how it pays off. The `getpreciseticks` returns a number that can be used later, after a similar call, to get a difference. The `getpreciseseconds` function gets such a tick (delta) as argument and returns the number of seconds. Ticks can differ per operating system, but one always creates a reference first and then deltas to this reference.

12.2 The status library

This contains a number of run-time configuration items that you may find useful in message reporting, as well as an iterator function that gets all of the names and values as a table.

```
<table> info = status.list()
```

The keys in the table are the known items, the value is the current value. There are toplevel items and items that are tables with subentries. The current list is:

TOPLEVEL STATISTICS	
banner	This is LuaMetaTeX, Version 2.09.24
copyright	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
development_id	20210914
filename	E:/context/manuals/mkiv/external/luametatex/luametatex-tex.tex
format_id	596
logfilename	luametatex.log
luatex_engine	luametatex
luatex_revision	24
luatex_verbose	2.09.24
luatex_version	209

permit_loadlib	false
run_state	1
used_compiler	gcc

BUFFERSTATE.*	
all	1000000
ini	-1
max	100000000
mem	1000000
min	1000000
ptr	0
set	10000000
stp	1000000
top	828

CALLBACKSTATE.*	
bytecode	598
count	202055
direct	1218
file	14216
function	53080
local	0
message	0
saved	129099
value	3844

ENGINESTATE.*	
banner	This is LuaMetaTeX, Version 2.09.24
copyright	Taco Hoekwater, Hans Hagen & Wolfgang Schuster
development_id	20210914
format_id	596
logfilename	luametatex.log
luatex_engine	luametatex
luatex_revision	24
luatex_verbose	2.09.24
luatex_version	209
permit_loadlib	false
run_state	1
tex_hash_size	131072
used_compiler	gcc

ERRORLINESTATE.*	
max	255
min	132

set	250
top	0

ERRORSTATE.*	
error	unset
errorcontext	unset
luaerror	unset

EXPANDSTATE.*	
max	1000000
min	10000
set	10000
top	10

EXTRASTATE.*	
all	0
ini	-1
max	-1
mem	0
min	-1
ptr	0
set	-1
stp	-1
top	0

FILESTATE.*	
all	16000
ini	-1
max	2000
mem	500
min	500
ptr	6
set	2000
stp	250
top	11

FONTSTATE.*	
all	8701240
ini	-1
max	100000
mem	8701240
min	250
ptr	27
set	100000

stp	250
top	250

HALFERRORLINESTATE.*	
max	255
min	80
set	234
top	0

HASHSTATE.*	
all	2400000
ini	0
max	2097152
mem	150000
min	150000
ptr	11739
set	250000
stp	100000
top	740042

INPUTSTATE.*	
all	320000
ini	-1
max	100000
mem	10000
min	10000
ptr	7
set	100000
stp	10000
top	48

INSERTSTATE.*	
all	10240
ini	-1
max	500
mem	320
min	10
ptr	7
set	250
stp	25
top	10

LANGUAGESTATE.*	
all	96
ini	0

max	10000
mem	96
min	250
ptr	0
set	250
stp	250
top	250

LOOKUPSTATE.*	
all	-1
ini	44736
max	2097152
mem	-1
min	150000
ptr	54429
set	250000
stp	100000
top	131074

LUASTATE.*	
bytecodebytes	16000
bytecodes	999
functionsize	32768
propertiessize	10000
statebytes	140807782
statebytesmax	160638256

MARKSTATE.*	
all	28800
ini	-1
max	10000
mem	1200
min	50
ptr	28
set	250
stp	50
top	50

NESTSTATE.*	
all	48000
ini	-1
max	10000
mem	1000
min	1000
ptr	0
set	10000

stp	1000
top	19

NODESTATE.*	
all	9000000
ini	0
max	100000000
mem	1000000
min	1000000
ptr	-154225
set	100000000
stp	500000
top	197658

PARAMETERSTATE.*	
all	80000
ini	-1
max	100000
mem	20000
min	20000
ptr	1
set	100000
stp	10000
top	53

POOLSTATE.*	
all	1061119
ini	844941
max	100000000
mem	1061119
min	10000000
ptr	-1
set	10000000
stp	1000000
top	-1

READSTATE.*	
filename	E:/context/manuals/mkiv/external/luametatex/luametatex-tex.tex
iocode	5
linenumber	210
skiplinenumber	163

SAVESTATE.*	
all	160000
ini	-1

max	500000
mem	10000
min	100000
ptr	162
set	500000
stp	10000
top	741

SPARSESTATE.*	
all	1683928
ini	-1
max	-1
mem	1683928
min	-1
ptr	-1
set	-1
stp	-1
top	-1

STRINGSTATE.*	
all	2400000
ini	2141905
max	2097152
mem	150000
min	150000
ptr	54446
set	500000
stp	100000
top	54446

TEXSTATE.*	
approximate	35909423

TOKENSTATE.*	
all	8000000
ini	490867
max	10000000
mem	1000000
min	1000000
ptr	1250701
set	10000000
stp	250000
top	682198

WARNINGSTATE.*	
warning	unset
warningtag	unset

There are also getters for the subtables. The whole repertoire of functions in the `status` table is: `getbufferstate`, `getcallbackstate`, `getconstants`, `geterrorlinestate`, `geterrorstate`, `getexpandstate`, `getextrastate`, `getfilestate`, `getfontstate`, `gethalferrorlinestate`, `gethashstate`, `getinputstate`, `getinsertstate`, `getlanguagestate`, `getlookupstate`, `getluastate`, `getmarkstate`, `getneststate`, `getnodestate`, `getparameterstate`, `getpoolstate`, `getreadstate`, `getsavestate`, `getsparsestate`, `getstringstate`, `gettexstate`, `gettokenstate`, `getwarningstate`, `iocodes`, `list`, `resetmessages`. The error and warning messages can be wiped with the `resetmessages` function. The states in subtables relate to memory management and are mostly there for development purposes.

The `getconstants` query gives back a table with all kind of internal quantities and again these are only relevant for diagnostic and development purposes. Many are good old TEX constants that are describes in the original documentation of the source but some are definitely LuaMetaTEX specific.

CONSTANTS.*	
awful_bad	1073741823
decent_criterium	12
default_catcode_table	-1
default_deadcycles	25
default_eqno_gap_step	1000
default_hangafter	1
default_output_box	255
default_pre_display_gap	2000
default_rule	26214
default_space_factor	1000
default_tolerance	10000
deplorable	100000
eject_penalty	-10000
ignore_depth	-65536000
infinite_bad	10000
infinite_penalty	10000
infinity	2147483647
large_width_excess	7230584
loose_criterium	99
max_bytecode_index	65535
max_cardinal	4294967295
max_category_code	15
max_char_code	15
max_character_code	1114111
max_data_value	2097151
max_dimen	1073741823

```
max_function_reference     2097151
max_half_value             32767
max_halfword               1073741823
max_integer                2147483647
max_mark_index             9999
max_math_class_code        7
max_math_family_index      255
max_n_of_bytecodes         65536
max_n_of_catcode_tables    256
max_n_of_fonts             100000
max_n_of_languages         10000
max_n_of_marks             10000
max_n_of_math_families     256
max_n_of_registers         65536
max_newline_character      127
max_quarterword            65535
max_register_index         65535
max_size_of_word           1024
max_space_factor           32767
min_cardinal               0
min_data_value             0
min_dimen                  -1073741823
min_halfword               -1073741823
min_infinity               -2147483647
min_integer                -2147483647
min_quarterword            0
min_space_factor           0
no_catcode_table           -2
null                       0
null_flag                  -1073741824
null_font                  0
one_bp                     65781
preset_rule_thickness      1073741824
running_rule               -1073741824
small_stretchability       1663497
tex_eqtb_size              590042
tex_hash_prime             131041
tex_hash_size              131072
two                        131072
unity                      65536
unused_attribute_value     -2147483647
unused_script_value        0
unused_state_value         0
zero_glue                  0
```

Most variables speak for themselves, some are more obscure. For instance the `run_state` vari-

able indicates what the engine is doing:

N	meaning	explanation
0	initializing	`--ini` mnode
1	updating	relates to `\overloadmode`
2	production	a regular (format driven) run

12.3 The tex library

12.3.1 Introduction

The `tex` table contains a large list of virtual internal TEX parameters that are partially writable.

The designation 'virtual' means that these items are not properly defined in Lua, but are only frontends that are handled by a metatable that operates on the actual TEX values. As a result, most of the Lua table operators (like `pairs` and #) do not work on such items.

At the moment, it is possible to access almost every parameter that you can use after `\the`, is a single token or is sort of special in TEX. This excludes parameters that need extra arguments, like `\the\scriptfont`. The subset comprising simple integer and dimension registers are writable as well as readable (like `\tracingcommands` and `\parindent`).

12.3.2 Internal parameter values, set and get

For all the parameters in this section, it is possible to access them directly using their names as index in the `tex` table, or by using one of the functions `tex.get` and `tex.set`.

The exact parameters and return values differ depending on the actual parameter, and so does whether `tex.set` has any effect. For the parameters that *can* be set, it is possible to use `global` as the first argument to `tex.set`; this makes the assignment global instead of local.

```
tex.set (["global",] <string> n, ...)
... = tex.get (<string> n)
```

Glue is kind of special because there are five values involved. The return value is a `glue_spec` node but when you pass `false` as last argument to `tex.get` you get the width of the glue and when you pass `true` you get all five values. Otherwise you get a node which is a copy of the internal value so you are responsible for its freeing at the Lua end. When you set a glue quantity you can either pass a `glue_spec` or upto five numbers.

Beware: as with regular Lua tables you can add values to the `tex` table. So, the following is valid:

```
tex.foo = 123
```

When you access a TEX parameter a look up takes place. For read–only variables that means that you will get something back, but when you set them you create a new entry in the table thereby making the original invisible.

There are a few special cases that we make an exception for: `prevdepth`, `prevgraf` and `spacefactor`. These normally are accessed via the `tex.nest` table:

```
tex.nest[tex.nest.ptr].prevdepth   = p
tex.nest[tex.nest.ptr].spacefactor = s
```

However, the following also works:

```
tex.prevdepth   = p
tex.spacefactor = s
```

Keep in mind that when you mess with node lists directly at the Lua end you might need to update the top of the nesting stack's `prevdepth` explicitly as there is no way LuaTEX can guess your intentions. By using the accessor in the `tex` tables, you get and set the values at the top of the nesting stack.

12.3.2.1 Integer parameters

The integer parameters accept and return Lua integers. In some cases the values are checked, trigger other settings or result in some immediate change of behaviour: `adjdemerits`, `adjustspacing`, `adjustspacingshrink`, `adjustspacingstep`, `adjustspacingstretch`, `automatichyphenpenalty`, `automigrationmode`, `autoparagraphmode`, `binoppenalty`, `brokenpenalty`, `catcodetable`, `clubpenalty`, `day`, `defaulthyphenchar`, `defaultskewchar`, `delimiterfactor`, `displaywidowpenalty`, `doublehyphendemerits`, `endlinechar`, `errorcontextlines`, `escapechar`, `exceptionpenalty`, `exhyphenchar`, `exhyphenpenalty`, `explicithyphenpenalty`, `fam`, `finalhyphendemerits`, `firstvalidlanguage`, `floatingpenalty`, `globaldefs`, `glyphdatafield`, `glyphoptions`, `glyphscale`, `glyphscriptfield`, `glyphscriptscale`, `glyphscriptscriptscale`, `glyphstatefield`, `glyphtextscale`, `glyphxscale`, `glyphyscale`, `hangafter`, `hbadness`, `holdinginserts`, `hyphenationmode`, `hyphenpenalty`, `interlinepenalty`, `language`, `lastlinefit`, `lefthyphenmin`, `linedirection`, `linepenalty`, `localbrokenpenalty`, `localinterlinepenalty`, `looseness`, `luacopyinputnodes`, `mathcontrolmode`, `mathdelimitersmode`, `mathdirection`, `mathdisplayskipmode`, `matheqnogapstep`, `mathflattenmode`, `mathfontcontrol`, `mathitalicsmode`, `mathnolimitsmode`, `mathpenaltiesmode`, `mathrulesfam`, `mathrulesmode`, `mathrulethicknessmode`, `mathscriptboxmode`, `mathscriptcharmode`, `mathscriptsmode`, `mathsurroundmode`, `maxdeadcycles`, `month`, `newlinechar`, `normalizelinemode`, `nospaces`, `outputbox`, `outputpenalty`, `overloadmode`, `pardirection`, `pausing`, `postdisplaypenalty`, `prebinoppenalty`, `predisplaydirection`, `predisplaygapfactor`, `predisplaypenalty`, `prerelpenalty`, `pretolerance`, `protrudechars`, `relpenalty`, `righthyphenmin`, `savinghyphcodes`, `savingvdiscards`, `setfontid`, `setlanguage`, `shapingpenaltiesmode`, `shapingpenalty`, `showboxbreadth`, `showboxdepth`, `shownodedetails`, `supmarkmode`, `textdirection`, `time`, `tolerance`, `tracingalignments`, `tracingassigns`, `tracingcommands`, `tracingexpressions`, `tracingfonts`, `tracinggroups`, `tracinghyphenation`, `tracingifs`, `tracinginserts`, `tracinglevels`, `tracinglostchars`, `tracingmacros`, `tracingmarks`, `tracingmath`, `tracingnesting`, `tracingonline`, `tracingoutput`, `tracingpages`, `tracingparagraphs`, `tracingrestores`, `tracingstats`, `uchyph`, `vbadness`, `widowpenalty`, `year`.

Some integer parameters are read only, because they are actually referring not to some internal integer register but to an engine property: `deadcycles`, `insertpenalties`, `parshape`, `in-`

terlinepenalties, clubpenalties, widowpenalties, displaywidowpenalties, prevgraf and spacefactor.

12.3.2.2 Dimension parameters

The dimension parameters accept Lua numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points. These are read-write: boxmaxdepth, delimitershortfall, displayindent, displaywidth, emergencystretch, glyphxoffset, glyphyoffset, hangindent, hfuzz, hsize, lineskiplimit, mathsurround, maxdepth, nulldelimiterspace, overfullrule, parindent, predisplaysize, pxdimen, scriptspace, splitmaxdepth, tabsize, vfuzz, vsize.

These are read-only: pagedepth, pagefilllstretch, pagefillstretch, pagefilstretch, pagegoal, pageshrink, pagestretch and pagetotal.

12.3.2.3 Direction parameters

The direction states can be queried with: gettextdir, getlinedir, getmathdir and getpardir. You can set them with settextdir, setlinedir, setmathdir and setpardir, commands that accept a number. You can also set these parameters as table key/values: textdirection, linedirection, mathdirection and pardirection, so the next code sets the text direction to r2l:

```
tex.textdirection = 1
```

12.3.2.4 Glue parameters

The internal glue parameters accept and return a userdata object that represents a glue_spec node: abovedisplayshortskip, abovedisplayskip, baselineskip, belowdisplayshortskip, belowdisplayskip, leftskip, lineskip, mathsurroundskip, parfillleftskip, parfillskip, parskip, rightskip, spaceskip, splittopskip, tabskip, topskip, xspaceskip.

12.3.2.5 Muglue parameters

All muglue parameters are to be used read-only and return a Lua string medmuskip, thickmuskip, thinmuskip.

12.3.2.6 Tokenlist parameters

The tokenlist parameters accept and return Lua strings. Lua strings are converted to and from token lists using \the \toks style expansion: all category codes are either space (10) or other (12). It follows that assigning to some of these, like 'tex.output', is actually useless, but it feels bad to make exceptions in view of a coming extension that will accept full-blown token strings. Here is the lot: errhelp, everybeforepar, everycr, everydisplay, everyeof, everyhbox, everyjob, everymath, everypar, everytab, everyvbox, output.

12.3.3 Convert commands

All 'convert' commands are read-only and return a Lua string. The supported commands at this

moment are: Uchar, csstring, directlua, expanded, fontname, fontspecifiedname, formatname, jobname, luabytecode, luaescapestring, luafunction, luatexbanner, meaning, meaningfull, meaningless, number, romannumeral, semiexpanded, string, todimension, tointeger, toscaled. You will get an error message if an operation is not (yet) permitted. Some take an string or number argument, just like at the TEX end some extra input is expected.

12.3.4 Item commands

All so called 'item' commands are read-only and return a number. The complete list of these commands is: Umathcharclass, Umathcharfam, Umathcharslot, badness, currentgrouplevel, currentgrouptype, currentifbranch, currentiflevel, currentiftype, dimexpr, dimexpression, fontchardp, fontcharht, fontcharic, fontcharwd, fontid, fontmathcontrol, fontspecifiedsize, fonttextcontrol, glueexpr, glueshrink, glueshrinkorder, gluestretch, gluestretchorder, gluetomu, inputlineno, insertprogress, lastarguments, lastchkdim, lastchknum, lastkern, lastnodesubtype, lastnodetype, lastparcontext, lastpenalty, lastskip, leftmarginkern, luatexrevision, luatexversion, mathscale, mathstyle, muexpr, mutoglue, numericscale, numexpr, numexpression, overshoot, parametercount, parshapedimen, parshapeindent, parshapelength, rightmarginkern. No all are currently supported but eventually that might be the case. Like the lists in previous sections, there are differences between LuaTEX and LuaMetaTEX, where some commands are organized differently in order to provide a consistent Lua interface.

12.3.5 Accessing registers: set*, get* and is*

TEX's attributes (\attribute), counters (\count), dimensions (\dimen), skips (\skip, \muskip) and token (\toks) registers can be accessed and written to using two times five virtual sub-tables of the tex table:

```
tex.attribute              tex.skip                  tex.muglue
tex.count                  tex.glue                  tex.toks
tex.dimen                  tex.muskip
```

It is possible to use the names of relevant \attributedef, \countdef, \dimendef, \skipdef, or \toksdef control sequences as indices to these tables:

```
tex.count.scratchcounter = 0
enormous = tex.dimen['maxdimen']
```

In this case, LuaTEX looks up the value for you on the fly. You have to use a valid \countdef (or \attributedef, or \dimendef, or \skipdef, or \toksdef), anything else will generate an error (the intent is to eventually also allow <chardef tokens> and even macros that expand into a number).

- The count registers accept and return Lua numbers.
- The dimension registers accept Lua numbers (in scaled points) or strings (with an included absolute dimension; em and ex and px are forbidden). The result is always a number in scaled

points.

- The token registers accept and return Lua strings. Lua strings are converted to and from token lists using `\the \toks` style expansion: all category codes are either space (10) or other (12).
- The skip registers accept and return `glue_spec` userdata node objects (see the description of the node interface elsewhere in this manual).
- The glue registers are just skip registers but instead of userdata are verbose.
- Like the counts, the attribute registers accept and return Lua numbers.

As an alternative to array addressing, there are also accessor functions defined for all cases, for example, here is the set of possibilities for `\skip` registers:

```
tex.setskip (["global",] <number> n, <node> s)
tex.setskip (["global",] <string> s, <node> s)
<node> s = tex.getskip (<number> n)
<node> s = tex.getskip (<string> s)
```

We have similar setters for `count`, `dimen`, `muskip`, and `toks`. Counters and dimen are represented by numbers, skips and muskips by nodes, and toks by strings.

Again the glue variants are not using the `glue-spec` userdata nodes. The `setglue` function accepts upto five arguments: width, stretch, shrink, stretch order and shrink order. Non-numeric values set the property to zero. The `getglue` function reports all five properties, unless the second argument is `false` in which case only the width is returned.

Here is an example using a threesome:

```
local d = tex.getdimen("foo")
if tex.isdimen("oof") then
    tex.setdimen("oof",d)
end
```

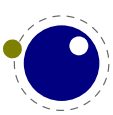
There are six extra skip (glue) related helpers:

```
tex.setglue (["global"], <number> n,
    width, stretch, shrink, stretch_order, shrink_order)
tex.setglue (["global"], <string> s,
    width, stretch, shrink, stretch_order, shrink_order)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<number> n)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<string> s)
```

The other two are `tex.setmuglue` and `tex.getmuglue`.

There are such helpers for `dimen`, `count`, `skip`, `muskip`, `box` and `attribute` registers but the glue ones are special because they have to deal with more properties.

As with the general `get` and `set` function discussed before, for the skip registers `getskip` returns a node and `getglue` returns numbers, while `setskip` accepts a node and `setglue` expects upto



5 numbers. Again, when you pass `false` as second argument to `getglue` you only get the width returned. The same is true for the `mu` variants `getmuskip`, `setmuskip`, `getmuskip` and`setmuskip`.

For tokens registers we have an alternative where a catcode table is specified:

```
tex.scantoks(0,3,"$e=mc^2$")
tex.scantoks("global",0,3,"$\int\limits^1_2$")
```

In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

There is a dedicated getter for marks: `getmark` that takes two arguments. The first argument is one of `top`, `bottom`, `first`, `splitbottom` or `splitfirst`, and the second argument is a marks class number. When no arguments are given the current maximum number of classes is returned.

When `tex.gettoks` gets an extra argument `true` it will return a table with userdata tokens.

12.3.6 Character code registers: `[get|set]*code[s]`

TEX's character code tables (`\lccode`, `\uccode`, `\sfcode`, `\catcode`, `\mathcode`, `\delcode`) can be accessed and written to using six virtual subtables of the `tex` table

```
tex.lccode                  tex.sfcode                  tex.mathcode
tex.uccode                  tex.catcode                 tex.delcode
```

The function call interfaces are roughly as above, but there are a few twists. `sfcodes` are the simple ones:

```
tex.setsfcode (["global",] <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
```

The function call interface for `lccode` and `uccode` additionally allows you to set the associated sibling at the same time:

```
tex.setlccode (["global"], <number> n, <number> lc)
tex.setlccode (["global"], <number> n, <number> lc, <number> uc)
<number> lc = tex.getlccode (<number> n)
tex.setuccode (["global"], <number> n, <number> uc)
tex.setuccode (["global"], <number> n, <number> uc, <number> lc)
<number> uc = tex.getuccode (<number> n)
```

The function call interface for `catcode` also allows you to specify a category table to use on assignment or on query (default in both cases is the current one):

```
tex.setcatcode (["global"], <number> n, <number> c)
tex.setcatcode (["global"], <number> cattable, <number> n, <number> c)
<number> lc = tex.getcatcode (<number> n)
<number> lc = tex.getcatcode (<number> cattable, <number> n)
```

The interfaces for `delcode` and `mathcode` use small array tables to set and retrieve values:

```
tex.setmathcode (["global"], <number> n, <table> mval )
<table> mval = tex.getmathcode (<number> n)
tex.setdelcode (["global"], <number> n, <table> dval )
<table> dval = tex.getdelcode (<number> n)
```

Where the table for `mathcode` is an array of 3 numbers, like this:

```
{
    <number> class,
    <number> family,
    <number> character
}
```

And the table for `delcode` is an array with 4 numbers, like this:

```
{
    <number> small_fam,
    <number> small_char,
    <number> large_fam,
    <number> large_char
}
```

You can also avoid the table:

```
tex.setmathcode (["global"], <number> n, <number> class,
    <number> family, <number> character)
class, family, char =
    tex.getmathcodes (<number> n)
tex.setdelcode (["global"], <number> n, <number> smallfam,
    <number> smallchar, <number> largefam, <number> largechar)
smallfam, smallchar, largefam, largechar =
    tex.getdelcodes (<number> n)
```

Normally, the third and fourth values in a delimiter code assignment will be zero according to `\Udelcode` usage, but the returned table can have values there (if the delimiter code was set using `\delcode`, for example). Unset `delcode`'s can be recognized because `dval[1]` is −1.

12.3.7 Box registers: `[get|set]box`

It is possible to set and query actual boxes, coming for instance from `\hbox`, `\vbox` or `\vtop`, using the node interface as defined in the `node` library:

```
tex.box
```

for array access, or

```
tex.setbox(["global",] <number> n, <node> s)
```

```
tex.setbox(["global",] <string> cs, <node> s)
<node> n = tex.getbox(<number> n)
<node> n = tex.getbox(<string> cs)
```

for function-based access. In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

Be warned that an assignment like

```
tex.box[0] = tex.box[2]
```

does not copy the node list, it just duplicates a node pointer. If `\box2` will be cleared by TEX commands later on, the contents of `\box0` becomes invalid as well. To prevent this from happening, always use `node.copy_list` unless you are assigning to a temporary variable:

```
tex.box[0] = node.copy_list(tex.box[2])
```

12.3.8 triggerbuildpage

You should not expect to much from the `triggerbuildpage` helpers because often TEX doesn't do much if it thinks nothing has to be done, but it might be useful for some applications. It just does as it says it calls the internal function that build a page, given that there is something to build.

12.3.9 splitbox

You can split a box:

```
local vlist = tex.splitbox(n,height,mode)
```

The remainder is kept in the original box and a packaged vlist is returned. This operation is comparable to the `\vsplit` operation. The mode can be `additional` or `exactly` and concerns the split off box.

12.3.10 Accessing math parameters: [get|set]math

It is possible to set and query the internal math parameters using:

```
tex.setmath(["global",] <string> n, <string> t, <number> n)
<number> n = tex.getmath(<string> n, <string> t)
```

As before an optional first parameter `global` indicates a global assignment.

The first string is the parameter name minus the leading 'Umath', and the second string is the style name minus the trailing 'style'. Just to be complete, the values for the math parameter name are:

```
quad                axis                operatorsize
overbarkern         overbarrule         overbarvgap
```

```
underbarkern          underbarrule          underbarvgap
radicalkern           radicalrule           radicalvgap
radicaldegreebefore   radicaldegreeafter    radicaldegreeraise
stackvgap             stacknumup            stackdenomdown
fractionrule          fractionnumvgap       fractionnumup
fractiondenomvgap     fractiondenomdown     fractiondelsize
limitabovevgap        limitabovebgap        limitabovekern
limitbelowvgap        limitbelowbgap        limitbelowkern
underdelimitervgap    underdelimiterbgap
overdelimitervgap     overdelimiterbgap
subshiftdrop          supshiftdrop          subshiftdown
subsupshiftdown       subtopmax             supshiftup
supbottommin          supsubbottommax       subsupvgap
spaceafterscript      connectoroverlapmin
ordordspacing         ordopspacing          ordbinspacing      ordrelspacing
ordopenspacing        ordclosespacing       ordpunctspacing    ordinnerspacing
opordspacing          opopspacing           opbinspacing       oprelspacing
opopenspacing         opclosespacing        oppunctspacing     opinnerspacing
binordspacing         binopspacing          binbinspacing      binrelspacing
binopenspacing        binclosespacing       binpunctspacing    bininnerspacing
relordspacing         relopspacing          relbinspacing      relrelspacing
relopenspacing        relclosespacing       relpunctspacing    relinnerspacing
openordspacing        openopspacing         openbinspacing     openrelspacing
openopenspacing       openclosespacing      openpunctspacing   openinnerspacing
closeordspacing       closeopspacing        closebinspacing    closerelspacing
closeopenspacing      closeclosespacing     closepunctspacing  closeinnerspacing
punctordspacing       punctopspacing        punctbinspacing    punctrelspacing
punctopenspacing      punctclosespacing     punctpunctspacing  punctinnerspacing
innerordspacing       inneropspacing        innerbinspacing    innerrelspacing
inneropenspacing      innerclosespacing     innerpunctspacing  innerinnerspacing
```

The values for the style parameter are:

```
display       crampeddisplay
text          crampedtext
script        crampedscript
scriptscript  crampedscriptscript
```

The value is either a number (representing a dimension or number) or a glue spec node representing a muskip for `ordordspacing` and similar spacing parameters.

12.3.11 Special list heads: `[get|set]list`

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.

FIELD	EXPLANATION
page_ins_head	circular list of pending insertions
contribute_head	the recent contributions
page_head	the current page content
hold_head	used for held-over items for next page
adjust_head	head of the current \vadjust list
pre_adjust_head	head of the current \vadjust pre list
page_discards_head	head of the discarded items of a page break
split_discards_head	head of the discarded items in a vsplit

The getter and setter functions are getlist and setlist. You have to be careful with what you set as TEX can have expectations with regards to how a list is constructed or in what state it is.

12.3.12 Semantic nest levels: getnest and ptr

The virtual table nest contains the currently active semantic nesting state. It has two main parts: a zero-based array of userdata for the semantic nest itself, and the numerical value ptr, which gives the highest available index. Neither the array items in nest[] nor ptr can be assigned to (as this would confuse the typesetting engine beyond repair), but you can assign to the individual values inside the array items, e.g. tex.nest[tex.nest.ptr].prevdepth.

tex.nest[tex.nest.ptr] is the current nest state, nest[0] the outermost (main vertical list) level. The getter function is getnest. You can pass a number (which gives you a list), nothing or top, which returns the topmost list, or the string ptr which gives you the index of the topmost list.

The known fields are:

KEY	TYPE	MODES	EXPLANATION
mode	number	all	the meaning of these numbers depends on the engine and sometimes even the version; you can use tex.getmodevalues() to get the mapping: positive values signal vertical, horizontal and math mode, while negative values indicate inner and inline variants
modeline	number	all	source input line where this mode was entered in, negative inside the output routine
head	node	all	the head of the current list
tail	node	all	the tail of the current list
prevgraf	number	vmode	number of lines in the previous paragraph
prevdepth	number	vmode	depth of the previous paragraph
spacefactor	number	hmode	the current space factor
direction	node	hmode	stack used for temporary storage by the line break algorithm
noad	node	mmode	used for temporary storage of a pending fraction numerator, for \over etc.
delimptr	node	mmode	used for temporary storage of the previous math delimiter, for \middle

mathdir	boolean	mmode	true when during math processing the \mathdirection is not the same as the surrounding \textdirection
mathstyle	number	mmode	the current \mathstyle

When a second string argument is given to the getnest, the value with that name is returned. Of course the level must be valid. When setnest gets a third argument that value is assigned to the field given as second argument.

12.3.13 Print functions

The tex table also contains the three print functions that are the major interface from Lua scripting to TEX. The arguments to these three functions are all stored in an in-memory virtual file that is fed to the TEX scanner as the result of the expansion of \directlua.

The total amount of returnable text from a \directlua command is only limited by available system ram. However, each separate printed string has to fit completely in TEX's input buffer. The result of using these functions from inside callbacks is undefined at the moment.

12.3.13.1 print

```
tex.print(<string> s, ...)
tex.print(<number> n, <string> s, ...)
tex.print(<table> t)
tex.print(<number> n, <table> t)
```

Each string argument is treated by TEX as a separate input line. If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional parameter can be used to print the strings using the catcode regime defined by \catcodetable n. If n is −1, the currently active catcode regime is used. If n is −2, the resulting catcodes are the result of \the \toks: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, if n is not a valid catcode table, then it is ignored, and the currently active catcode regime is used instead.

The very last string of the very last tex.print command in a \directlua will not have the \endlinechar appended, all others do.

12.3.13.2 sprint

```
tex.sprint(<string> s, ...)
tex.sprint(<number> n, <string> s, ...)
tex.sprint(<table> t)
tex.sprint(<number> n, <table> t)
```

Each string argument is treated by TEX as a special kind of input line that makes it suitable for use as a partial line input mechanism:

- TEX does not switch to the 'new line' state, so that leading spaces are not ignored.
- No \endlinechar is inserted.

- Trailing spaces are not removed. Note that this does not prevent TEX itself from eating spaces as result of interpreting the line. For example, in

  ```
  before\directlua{tex.sprint("\\relax")tex.sprint(" in between")}after
  ```

  the space before `in between` will be gobbled as a result of the 'normal' scanning of `\relax`.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional argument sets the catcode regime, as with `tex.print`. This influences the string arguments (or numbers turned into strings).

Although this needs to be used with care, you can also pass token or node userdata objects. These get injected into the stream. Tokens had best be valid tokens, while nodes need to be around when they get injected. Therefore it is important to realize the following:

- When you inject a token, you need to pass a valid token userdata object. This object will be collected by Lua when it no longer is referenced. When it gets printed to TEX the token itself gets copied so there is no interference with the Lua garbage collection. You manage the object yourself. Because tokens are actually just numbers, there is no real extra overhead at the TEX end.
- When you inject a node, you need to pass a valid node userdata object. The node related to the object will not be collected by Lua when it no longer is referenced. It lives on at the TEX end in its own memory space. When it gets printed to TEX the node reference is used assuming that node stays around. There is no Lua garbage collection involved. Again, you manage the object yourself. The node itself is freed when TEX is done with it.

If you consider the last remark you might realize that we have a problem when a printed mix of strings, tokens and nodes is reused. Inside TEX the sequence becomes a linked list of input buffers. So, `"123"` or `"\foo{123}"` gets read and parsed on the fly, while `<token userdata>` already is tokenized and effectively is a token list now. A `<node userdata>` is also tokenized into a token list but it has a reference to a real node. Normally this goes fine. But now assume that you store the whole lot in a macro: in that case the tokenized node can be flushed many times. But, after the first such flush the node is used and its memory freed. You can prevent this by using copies which is controlled by setting `\luacopyinputnodes` to a non-zero value. This is one of these fuzzy areas you have to live with if you really mess with these low level issues.

12.3.13.3 tprint

```
tex.tprint({<number> n, <string> s, ...}, {...})
```

This function is basically a shortcut for repeated calls to `tex.sprint(<number> n, <string> s, ...)`, once for each of the supplied argument tables.

12.3.13.4 cprint

This function takes a number indicating the to be used catcode, plus either a table of strings or an argument list of strings that will be pushed into the input stream.

```
tex.cprint( 1," 1: $&{\\foo}") tex.print("\\par") -- a lot of \bgroup s
tex.cprint( 2," 2: $&{\\foo}") tex.print("\\par") -- matching \egroup s
tex.cprint( 9," 9: $&{\\foo}") tex.print("\\par") -- all get ignored
tex.cprint(10,"10: $&{\\foo}") tex.print("\\par") -- all become spaces
tex.cprint(11,"11: $&{\\foo}") tex.print("\\par") -- letters
tex.cprint(12,"12: $&{\\foo}") tex.print("\\par") -- other characters
tex.cprint(14,"12: $&{\\foo}") tex.print("\\par") -- comment triggers
```

12.3.13.5 write

```
tex.write(<string> s, ...)
tex.write(<table> t)
```

Each string argument is treated by TEX as a special kind of input line that makes it suitable for use as a quick way to dump information:

- All catcodes on that line are either 'space' (for ' ') or 'character' (for all others).
- There is no `\endlinechar` appended.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

12.3.14 Helper functions

12.3.14.1 round

```
<number> n = tex.round(<number> o)
```

Rounds Lua number `o`, and returns a number that is in the range of a valid TEX register value. If the number starts out of range, it generates a 'number too big' error as well.

12.3.14.2 scale

```
<number> n = tex.scale(<number> o, <number> delta)
<table> n = tex.scale(table o, <number> delta)
```

Multiplies the Lua numbers `o` and `delta`, and returns a rounded number that is in the range of a valid TEX register value. In the table version, it creates a copy of the table with all numeric top–level values scaled in that manner. If the multiplied number(s) are of range, it generates 'number too big' error(s) as well.

Note: the precision of the output of this function will depend on your computer's architecture and operating system, so use with care! An interface to LuaTEX's internal, 100% portable scale function will be added at a later date.

12.3.14.3 number and romannumeral

These are the companions to the primitives `\number` and `\romannumeral`. They can be used like:

```
tex.print(tex.romannumeral(123))
```

12.3.14.4 fontidentifier and fontname

The first one returns the name only, the second one reports the size too.

```
tex.print(tex.fontname(tex.fontname))
tex.print(tex.fontname(tex.fontidentidier))
```

12.3.14.5 sp

```
<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)
```

Converts the number `o` or a string `s` that represents an explicit dimension into an integer number of scaled points.

For parsing the string, the same scanning and conversion rules are used that LuaTEX would use if it was scanning a dimension specifier in its TEX-like input language (this includes generating errors for bad values), expect for the following:

1. only explicit values are allowed, control sequences are not handled
2. infinite dimension units (`fil...`) are forbidden
3. `mu` units do not generate an error (but may not be useful either)

12.3.14.6 tex.getlinenumber and tex.setlinenumber

You can mess with the current line number:

```
local n = tex.getlinenumber()
tex.setlinenumber(n+10)
```

which can be shortcut to:

```
tex.setlinenumber(10,true)
```

This might be handy when you have a callback that reads numbers from a file and combines them in one line (in which case an error message probably has to refer to the original line). Interference with TEX's internal handling of numbers is of course possible.

12.3.14.7 error, show_context and gethelptext

```
tex.error(<string> s)
tex.error(<string> s, <table> help)
<string> s = tex.gethelptext()
```

This creates an error somewhat like the combination of `\errhelp` and `\errmessage` would. During this error, deletions are disabled.

The array part of the `help` table has to contain strings, one for each line of error help.

In case of an error the `show_context` function will show the current context where we're at (in the expansion).

12.3.14.8 getfamilyoffont

When you pass a proper family identifier the next helper will return the font currently associated with it.

```
<integer> id = font.getfamilyoffont(<integer> fam)
```

12.3.14.9 [set|get]interaction

The engine can be in one of four modes:

VALUE	mode	MEANING
0	batch	omits all stops and omits terminal output
1	nonstop	omits all stops
2	scroll	omits error stops
3	errorstop	stops at every opportunity to interact

The mode can be queried and set with:

```
<integer> i = tex.getinteraction()
tex.setinteraction(<integer> i)
```

12.3.14.10 runtoks and quittoks

Because of the fact that TEX is in a complex dance of expanding, dealing with fonts, typesetting paragraphs, messing around with boxes, building pages, and so on, you cannot easily run a nested TEX run (read nested main loop). However, there is an option to force a local run with `runtoks`. The content of the given token list register gets expanded locally after which we return to where we triggered this expansion, at the Lua end. Instead a function can get passed that does some work. You have to make sure that at the end TEX is in a sane state and this is not always trivial. A more complex mechanism would complicate TEX itself (and probably also harm performance) so this simple local expansion loop has to do.

```
tex.runtoks(<token register>)
tex.runtoks(<lua function>)
tex.runtoks(<macro name>)
tex.runtoks(<register name>)
```

When the `\tracingnesting` parameter is set to a value larger than 2 some information is reported about the state of the local loop. The return value indicates an error:

VALUE	meaning
0	no error
1	bad register number
2	unknown macro or register name
3	macro is unsuitable for runtoks (has arguments)

This function has two optional arguments in case a token register is passed:

```
tex.runtoks(<token register>,force,grouped,obeymode)
```

Inside for instance an `\edef` the `runtoks` function behaves (at least tries to) like it were an `\the`. This prevents unwanted side effects: normally in such an definition tokens remain tokens and (for instance) characters don't become nodes. With the second argument you can force the local main loop, no matter what. The third argument adds a level of grouping. The last argument tells the scanner to stay in the current mode.

You can quit the local loop with `\endlocalcontrol` or from the Lua end with `tex.quittoks`. In that case you end one level up! Of course in the end that can mean that you arrive at the main level in which case an extra end will trigger a redundancy warning (not an abort!).

12.3.14.11 forcehmode

An example of a (possible error triggering) complication is that TEX expects to be in some state, say horizontal mode, and you have to make sure it is when you start feeding back something from Lua into TEX. Normally a user will not run into issues but when you start writing tokens or nodes or have a nested run there can be situations that you need to run `forcehmode`. There is no recipe for this and intercepting possible cases would weaken LuaTEX's flexibility.

12.3.14.12 hashtokens

```
for i,v in pairs (tex.hashtokens()) do ... end
```

Returns a list of names. This can be useful for debugging, but note that this also reports control sequences that may be unreachable at this moment due to local redefinitions: it is strictly a dump of the hash table. You can use `token.create` to inspect properties, for instance when the `command` key in a created table equals 123, you have the `cmdname` value `undefined_cs`.

12.3.14.13 definefont

```
tex.definefont(<string> csname, <number> fontid)
tex.definefont(<boolean> global, <string> csname, <number> fontid)
```

Associates `csname` with the internal font number `fontid`. The definition is global if (and only if) `global` is specified and true (the setting of `globaldefs` is not taken into account).

12.3.15 Functions for dealing with primitives

12.3.15.1 enableprimitives

```
tex.enableprimitives(<string> prefix, <table> primitive names)
```

This function accepts a prefix string and an array of primitive names. For each combination of 'prefix' and 'name', the `tex.enableprimitives` first verifies that 'name' is an actual primitive (it must be returned by one of the `tex.extraprimitives` calls explained below, or part of TEX82, or `\directlua`). If it is not, `tex.enableprimitives` does nothing and skips to the next pair.

But if it is, then it will construct a csname variable by concatenating the 'prefix' and 'name', unless the 'prefix' is already the actual prefix of 'name'. In the latter case, it will discard the

'prefix', and just use 'name'.

Then it will check for the existence of the constructed csname. If the csname is currently undefined (note: that is not the same as `\relax`), it will globally define the csname to have the meaning: run code belonging to the primitive 'name'. If for some reason the csname is already defined, it does nothing and tries the next pair.

An example:

```
tex.enableprimitives('LuaTeX', {'formatname'})
```

will define `\LuaTeXformatname` with the same intrinsic meaning as the documented primitive `\formatname`, provided that the control sequences `\LuaTeXformatname` is currently undefined.

When LuaTEX is run with `--ini` only the TEX82 primitives and `\directlua` are available, so no extra primitives **at all**.

If you want to have all the new functionality available using their default names, as it is now, you will have to add

```
\ifx\directlua\undefined \else
    \directlua {tex.enableprimitives('',tex.extraprimitives ())}
\fi
```

near the beginning of your format generation file. Or you can choose different prefixes for different subsets, as you see fit.

Calling some form of `tex.enableprimitives` is highly important though, because if you do not, you will end up with a TEX82-lookalike that can run Lua code but not do much else. The defined csnames are (of course) saved in the format and will be available at runtime.

12.3.15.2 extraprimitives

```
<table> t = tex.extraprimitives(<string> s, ...)
```

This function returns a list of the primitives that originate from the engine(s) given by the requested string value(s). The possible values and their (current) return values are given in the following table. In addition the somewhat special primitives '`\ `', '`\/`' and '`-`' are defined.

NAME	VALUES
tex	above abovedisplayshortskip abovedisplayskip abovewithdelims accent adjdemerits advance afterassignment aftergroup aligncontent atop atopwithdelims badness baselineskip batchmode begingroup beginsimplegroup belowdisplayshortskip belowdisplayskip binoppenalty botmark box boxmaxdepth brokenpenalty catcode char chardef cleaders clubpenalty copy count countdef cr crcr csname day deadcycles def defaulthyphenchar defaultskewchar delcode delimiter delimiterfactor delimitershortfall dimen dimendef discretionary displayindent displaylimits displaystyle displaywidowpenalty displaywidth divide doublehyphendemerits dp dump edef else emergencystretch end endcsname endgroup endinput endlinechar endsimplegroup eqno errhelp errmessage errorcontextlines errorstopmode escapechar everycr everydisplay everyhbox everyjob everymath everypar everyvbox exhyphenchar exhyphenpenalty ex-

pandafter fam fi finalhyphendemerits firstmark floatingpenalty font fontdimen fontname fontspecifiedname futurelet gdef global globaldefs glyph halign hangafter hangindent hbadness hbox hfil hfill hfilneg hfuzz holdinginserts hrule hsize hskip hss ht hyphenation hyphenchar hyphenpenalty if ifcase ifcat ifdim iffalse ifhbox ifhmode ifinner ifmmode ifnum ifodd iftrue ifvbox ifvmode ifvoid ifx ignorespaces indent input inputlineno insert insertpenalties interlinepenalty jobname kern language lastbox lastkern lastpenalty lastskip lccode leaders left lefthyphenmin leftskip leqno let limits linepenalty lineskip lineskiplimit long looseness lower lowercase mark mathaccent mathbin mathchar mathchardef mathchoice mathclose mathcode mathinner mathop mathopen mathord mathpunct mathrel mathsurround maxdeadcycles maxdepth meaning meaningfull meaningless medmuskip message middle mkern month moveleft moveright mskip multiply muskip muskipdef newlinechar noalign noexpand noindent nolimits nonscript nonstopmode nulldelimiterspace nullfont number omit or outer output outputpenalty over overfullrule overline overshoot overwithdelimits pagedepth pagefilllstretch pagefillstretch pagefilstretch pagegoal pageshrink pagestretch pagetotal par parfillleftskip parfillskip parindent parshape parskip patterns pausing penalty postdisplaypenalty predisplaypenalty predisplaysize pretolerance prevdepth prevgraf radical raise relax relpenalty right righthyphenmin rightskip romannumeral scaledfontdimen scriptfont scriptscriptfont scriptscriptstyle scriptspace scriptstyle scrollmode setbox setlanguage sfcode shipout show showbox showboxbreadth showboxdepth showlists shownodedetails showthe skewchar skip skipdef spacefactor spaceskip span splitbotmark splitfirstmark splitmaxdepth splittopskip string tabskip textfont textstyle the thickmuskip thinmuskip time todimension tointeger toks toksdef tolerance topmark topskip toscaled tracingcommands tracinglostchars tracingmacros tracingonline tracingoutput tracingpages tracingparagraphs tracingrestores tracingstats uccode uchyph underline unhbox unhcopy unhpack unkern unpenalty unskip unvbox unvcopy unvpack uppercase vadjust valign vbadness vbox vcenter vfil vfill vfilneg vfuzz vrule vsize vskip vsplit vss vtop wd widowpenalty xdef xleaders xspaceskip year

core

etex botmarks clubpenalties currentgrouplevel currentgrouptype currentifbranch currentiflevel currentiftype detokenize dimexpr displaywidowpenalties everyeof firstmarks fontchardp fontcharht fontcharic fontcharwd glueexpr glueshrink glueshrinkorder gluestretch gluestretchorder gluetomu ifcsname ifdefined iffontchar interactionmode interlinepenalties lastlinefit lastnodetype marks muexpr mutoglue numexpr pagediscards parshapedimen parshapeindent parshapelength predisplaydirection protected savinghyphcodes savingvdiscards scantokens showgroups showifs showtokens splitbotmarks splitdiscards splitfirstmarks topmarks tracingassigns tracinggroups tracingifs tracinglevels tracingnesting unexpanded unless widowpenalties

luatex UUskewed UUskewedwithdelimits Uabove Uabovewithdelimits Uatop Uatopwithdelimits Uchar Udelcode Udelcodenum Udelimiter Udelimiterover Udelimiterunder Uhextensible Uleft Umathaccent Umathaccentbaseheight Umathaccentvariant Umath-

adapttoleft Umathadapttoright Umathaxis Umathbinbinspacing Umathbinclos-
espacing Umathbininnerspacing Umathbinopenspacing Umathbinopspacing Umath-
binordspacing Umathbinpunctspacing Umathbinrelspacing Umathbotaccentvari-
ant Umathchar Umathcharclass Umathchardef Umathcharfam Umathcharnum Umath-
charnumdef Umathcharslot Umathclass Umathclosebinspacing Umathcloseclos-
espacing Umathcloseinnerspacing Umathcloseopenspacing Umathcloseopspacing
Umathcloseordspacing Umathclosepunctspacing Umathcloserelspacing Umath-
code Umathcodenum Umathconnectoroverlapmin Umathdegreevariant Umathdelim-
iterovervariant Umathdelimiterundervariant Umathdenominatorvariant Umath-
extrasubpreshift Umathextrasubshift Umathextrasuppreshift Umathextrasup-
shift Umathfractiondelsize Umathfractiondenomdown Umathfractiondenomvgap
Umathfractionnumup Umathfractionnumvgap Umathfractionrule Umathfractionva-
riant Umathhextensiblevariant Umathinnerbinspacing Umathinnerclosespacing
Umathinnerinnerspacing Umathinneropenspacing Umathinneropspacing Umathin-
nerordspacing Umathinnerpunctspacing Umathinnerrelspacing Umathlimitabove-
bgap Umathlimitabovekern Umathlimitabovevgap Umathlimitbelowbgap Umath-
limitbelowkern Umathlimitbelowvgap Umathlimits Umathnoaxis Umathnolimits
Umathnolimitsubfactor Umathnolimitsupfactor Umathnumeratorvariant Umathop-
binspacing Umathopclosespacing Umathopenbinspacing Umathopenclosespacing
Umathopeninnerspacing Umathopenopenspacing Umathopenopspacing Umath-
openordspacing Umathopenpunctspacing Umathopenrelspacing Umathopenupdepth
Umathopenupheight Umathoperatorsize Umathopinnerspacing Umathopopenspac-
ing Umathopopspacing Umathopordspacing Umathoppunctspacing Umathoprelspac-
ing Umathordbinspacing Umathordclosespacing Umathordinnerspacing Umath-
ordopenspacing Umathordopspacing Umathordordspacing Umathordpunctspac-
ing Umathordrelspacing Umathoverbarkern Umathoverbarrule Umathoverbarv-
gap Umathoverdelimiterbgap Umathoverdelimitervariant Umathoverdelimiter-
vgap Umathoverlayaccentvariant Umathoverlinevariant Umathphantom Umath-
punctbinspacing Umathpunctclosespacing Umathpunctinnerspacing Umathpunc-
topenspacing Umathpunctopspacing Umathpunctordspacing Umathpunctpunctspac-
ing Umathpunctrelspacing Umathquad Umathradicaldegreeafter Umathradi-
caldegreebefore Umathradicaldegreeraise Umathradicalkern Umathradical-
rule Umathradicalvariant Umathradicalvgap Umathrelbinspacing Umathrel-
closespacing Umathrelinnerspacing Umathrelopenspacing Umathrelopspacing
Umathrelordspacing Umathrelpunctspacing Umathrelrelspacing Umathskewed-
fractionhgap Umathskewedfractionvgap Umathspaceafterscript Umathspace-
beforescript Umathspacingmode Umathstackdenomdown Umathstacknumup Umath-
stackvariant Umathstackvgap Umathsubscriptvariant Umathsubshiftdown Umath-
subshiftdrop Umathsubsupshiftdown Umathsubsupvgap Umathsubtopmax Umath-
supbottommin Umathsuperscriptvariant Umathsupshiftdrop Umathsupshiftup
Umathsupsubbottommax Umathtopaccentvariant Umathunderbarkern Umathunder-
barrule Umathunderbarvgap Umathunderdelimiterbgap Umathunderdelimiter-
variant Umathunderdelimitervgap Umathunderlinevariant Umathvextensible-
variant Umathvoid Umiddle Unosubprescript Unosubscript Unosuperprescript
Unosuperscript Uover Uoverdelimiter Uoverwithdelims Uradical Uright Uroot
Uskewed Uskewedwithdelims Ustack Ustartdisplaymath Ustartmath Ustopdis-

playmath Ustopmath Ustyle Usubprescript Usubscript Usuperprescript Usuperscript Uunderdelimiter Uvextensible adjustspacing adjustspacingshrink adjustspacingstep adjustspacingstretch afterassigned aftergrouped aliased alignmark aligntab atendofgroup atendofgrouped attribute attributedef automaticdiscretionary automatichyphenpenalty automigrationmode autoparagraphmode begincsname beginlocalcontrol boundary boxattribute boxdirection boxorientation boxtotal boxxmove boxxoffset boxymove boxyoffset catcodetable clearmarks crampeddisplaystyle crampedscriptscriptstyle crampedscriptstyle crampedtextstyle csstring currentmarks defcsname dimensiondef dimexpression directlua edefcsname efcode endlocalcontrol enforced etoksapp etokspre everybeforepar everytab exceptionpenalty expand expandafterpars expandafterspaces expandcstoken expanded expandtoken explicitdiscretionary explicithyphenpenalty firstvalidlanguage flushmarks fontid fontmathcontrol fontspecifiedsize fonttextcontrol formatname frozen futurecsname futuredef futureexpand futureexpandis futureexpandisap gdefcsname gleaders glet gletcsname glettonothing gluespecdef glyphdatafield glyphoptions glyphscale glyphscriptfield glyphscriptscale glyphscriptscriptscale glyphstatefield glyphtextscale glyphxoffset glyphxscale glyphyoffset glyphyscale gtoksapp gtokspre hccode hjcode hpack hyphenationmin hyphenationmode ifabsdim ifabsnum ifarguments ifboolean ifchkdim ifchknum ifcmpdim ifcmpnum ifcondition ifcstok ifdimexpression ifdimval ifempty ifflags ifhaschar ifhastok ifhastoks ifhasxtoks ifincsname ifinsert ifmathparameter ifmathstyle ifnumexpression ifnumval ifparameter ifparameters ifrelax iftok ignorearguments ignorepars immediate immutable initcatcodetable insertbox insertcopy insertdepth insertdistance insertheight insertheights insertlimit insertmaxdepth insertmode insertmultiplier insertpenalty insertprogress insertstorage insertstoring insertunbox insertuncopy insertwidth instance integerdef lastarguments lastchkdim lastchknum lastnamedcs lastnodesubtype lastparcontext leftmarginkern letcharcode letcsname letfrozen letprotected lettonothing linedirection localbrokenpenalty localcontrol localcontrolled localinterlinepenalty localleftbox localrightbox lpcode luabytecode luabytecodecall luacopyinputnodes luadef luaescapestring luafunction luafunctioncall luatexbanner luatexrevision luatexversion mathcontrolmode mathdelimitersmode mathdirection mathdisplayskipmode matheqnogapstep mathflattenmode mathfontcontrol mathitalicsmode mathnolimitsmode mathpenaltiesmode mathrulesfam mathrulesmode mathrulethicknessmode mathscale mathscriptboxmode mathscriptcharmode mathscriptsmode mathstyle mathsurroundmode mathsurroundskip mugluespecdef mutable noaligned noboundary nohrule norelax normalizelinemode nospaces novrule numericscale numexpression orelse orunless outputbox overloaded overloadmode pageboundary pagevsize parametercount parametermark parattribute pardirection permanent postexhyphenchar posthyphenchar prebinoppenalty predisplaygapfactor preexhyphenchar prehyphenchar prerelpenalty protrudechars protrusionboundary pxdimen quitvmode retokenized rightmarginkern rpcode savecatcodetable scantextokens semiexpanded semiprotected setfontid shapingpenaltiesmode shapingpenalty snapshotpar supmarkmode swapcsvalues tabsize textdirection thewithoutunit tokenized

toksapp tokspre tolerant tpack tracingalignments tracingexpressions tracingfonts tracinghyphenation tracinginserts tracingmarks tracingmath undent unletfrozen unletprotected untraced vpack wordboundary wrapuppar xdefcsname xtoksapp xtokspre

Note that luatex does not contain directlua, as that is considered to be a core primitive, along with all the TEX82 primitives, so it is part of the list that is returned from 'core'.

Running tex.extraprimitives will give you the complete list of primitives -ini startup. It is exactly equivalent to tex.extraprimitives("etex","luatex").

12.3.15.3 primitives

```
<table> t = tex.primitives()
```

This function returns a list of all primitives that LuaTEX knows about.

12.3.16 Core functionality interfaces

12.3.16.1 badness

```
<number> b = tex.badness(<number> t, <number> s)
```

This helper function is useful during linebreak calculations. t and s are scaled values; the function returns the badness for when total t is supposed to be made from amounts that sum to s. The returned number is a reasonable approximation of $100(t/s)^3$;

12.3.16.2 tex.resetparagraph

This function resets the parameters that TEX normally resets when a new paragraph is seen.

12.3.16.3 linebreak

```
local <node> nodelist, <table> info =
    tex.linebreak(<node> listhead, <table> parameters)
```

The understood parameters are as follows:

NAME	TYPE	EXPLANATION
pardir	string	
pretolerance	number	
tracingparagraphs	number	
tolerance	number	
looseness	number	
hyphenpenalty	number	
exhyphenpenalty	number	
pdfadjustspacing	number	
adjdemerits	number	
protrudechars	number	

linepenalty	number	
lastlinefit	number	
doublehyphendemerits	number	
finalhyphendemerits	number	
hangafter	number	
interlinepenalty	number or table	if a table, then it is an array like \interlinepenalties
clubpenalty	number or table	if a table, then it is an array like \clubpenalties
widowpenalty	number or table	if a table, then it is an array like \widowpenalties
brokenpenalty	number	
emergencystretch	number	in scaled points
hangindent	number	in scaled points
hsize	number	in scaled points
leftskip	glue_spec node	
rightskip	glue_spec node	
parshape	table	

Note that there is no interface for \displaywidowpenalties, you have to pass the right choice for widowpenalties yourself.

It is your own job to make sure that listhead is a proper paragraph list: this function does not add any nodes to it. To be exact, if you want to replace the core line breaking, you may have to do the following (when you are not actually working in the pre_linebreak_filter or linebreak_filter callbacks, or when the original list starting at listhead was generated in horizontal mode):

- add an 'indent box' and perhaps a par node at the start (only if you need them)
- replace any found final glue by an infinite penalty (or add such a penalty, if the last node is not a glue)
- add a glue node for the \parfillskip after that penalty node
- make sure all the prev pointers are OK

The result is a node list, it still needs to be vpacked if you want to assign it to a \vbox. The returned info table contains four values that are all numbers:

NAME	EXPLANATION
prevdepth	depth of the last line in the broken paragraph
prevgraf	number of lines in the broken paragraph
looseness	the actual looseness value in the broken paragraph
demerits	the total demerits of the chosen solution

Note there are a few things you cannot interface using this function: You cannot influence font expansion other than via pdfadjustspacing, because the settings for that take place elsewhere. The same is true for hbadness and hfuzz etc. All these are in the hpack routine, and that fetches its own variables via globals.

12.3.16.4 shipout

```
tex.shipout(<number> n)
```

Ships out box number `n` to the output file, and clears the box register.

12.3.16.5 getpagestate

This helper reports the current page state: `empty`, `box_there` or `inserts_only` as integer value.

12.3.16.6 getlocallevel

This integer reports the current level of the local loop. It's only useful for debugging and the (relative state) numbers can change with the implementation.

12.3.17 Functions related to synctex

The next helpers only make sense when you implement your own synctex logic. Keep in mind that the library used in editors assumes a certain logic and is geared for plain and LaTeX, so after a decade users expect a certain behaviour.

NAME	EXPLANATION
`setsynctexmode`	`0` is the default and used normal synctex logic, `1` uses the values set by the next helpers while `2` also sets these for glyph nodes; `3` sets glyphs and glue and `4` sets only glyphs
`setsynctextag`	set the current tag (file) value (obeys save stack)
`setsynctexline`	set the current line value (obeys save stack)
`setsynctexnofiles`	disable synctex file logging
`getsynctexmode`	returns the current mode (for values see above)
`getsynctextag`	get the currently set value of tag (file)
`getsynctexline`	get the currently set value of line
`forcesynctextag`	overload the tag (file) value (`0` resets)
`forcesynctexline`	overload the line value (`0` resets)

The last one is somewhat special. Due to the way files are registered in SyncTeX we need to explicitly disable that feature if we provide our own alternative if we want to avoid that overhead. Passing a value of 1 disables registering.

12.4 The texconfig table

This is a table that is created empty. A startup Lua script could fill this table with a number of settings that are read out by the executable after loading and executing the startup file. Watch out: some keys are different from LuaTeX, which is a side effect of a more granular and dynamic memory management.

KEY	TYPE	DEFAULT	COMMENT
`buffersize`	number/table	1000000	input buffer bytes

filesize	number/table	1000	max number of open files
fontsize	number/table	250	number of permitted fonts
hashsize	number/table	150000	number of hash entries
inputsize	number/table	10000	maximum input stack
languagesize	number/table	250	number of permitted languages
marksize	number/table	50	number of mark classes
nestsize	number/table	1000	max depth of nesting
nodesize	number/table	1000000	max node memory (various size)
parametersize	number/table	20000	max size of parameter stack
poolsize	number/table	10000000	max number of string bytes
savesize	number/table	100000	mas size of save stack
stringsize	number/table	150000	max number of strings
tokensize	number/table	1000000	max token memory
expandsize	number/table	10000	max expansion nesting
propertiessize	number	0	initial size of node properties table
functionsize	number	0	initial size of Lua functions table
errorlinesize	number	79	how much or an error is shown
halferrorlinesize	number	50	idem
formatname	string		
jobname	string		

If no format name or jobname is given on the command line, the related keys will be tested first instead of simply quitting. The statistics library has methods for tracking down how much memory is available and has been configured. The size parameters take a number (for the maximum allocated size) or a table with three possible keys: `size`, `plus` (for extra size) and step for the increment when more memory is needed. They all start out with a hard coded minimum and also have an hard coded maximum, the the configured size sits somewhere between these.

12.5 The texio library

This library takes care of the low-level I/O interface: writing to the log file and/or console.

12.5.1 write and writeselector

```
texio.write(<string> target, <string> s, ...)
texio.write(<string> s, ...)
texio.writeselector(<string> s, ...)
```

Without the `target` argument, writes all given strings to the same location(s) TEX writes messages to at this moment. If `\batchmode` is in effect, it writes only to the log, otherwise it writes to the log and the terminal. The optional `target` can be one of `terminal`, `logfile` or `terminal_and_logfile`.

Note: If several strings are given, and if the first of these strings is or might be one of the targets above, the `target` must be specified explicitly to prevent Lua from interpreting the first string as the target.

12.5.2 writenl and writeselectornl

```
texio.writenl(<string> target, <string> s, ...)
texio.writenl(<string> s, ...)
texio.writeselectornl(<string> target, ...)
```

This function behaves like `texio.write`, but makes sure that the given strings will appear at the beginning of a new line. You can pass a single empty string if you only want to move to the next line.

The selector variants always expect a selector, so there is no misunderstanding if `logfile` is a string or selector.

12.5.3 setescape

You can disable ^^ escaping of control characters by passing a value of zero.

12.5.4 closeinput

This function should be used with care. It acts as `\endinput` but at the Lua end. You can use it to (sort of) force a jump back to TEX. Normally a Lua call will just collect prints and at the end bump an input level and flush these prints. This function can help you stay at the current level but you need to know what you're doing (or more precise: what TEX is doing with input).

12.6 The token library

12.6.1 The scanner

The token library provides means to intercept the input and deal with it at the Lua level. The library provides a basic scanner infrastructure that can be used to write macros that accept a wide range of arguments. This interface is on purpose kept general and as performance is quite okay so one can build additional parsers without too much overhead. It's up to macro package writers to see how they can benefit from this as the main principle behind LuaTEX is to provide a minimal set of tools and no solutions. The scanner functions are probably the most intriguing.

FUNCTION	ARGUMENT	RESULT
`scankeyword`	string	returns true if the given keyword is gobbled; as with the regular TEX keyword scanner this is case insensitive (and ascii based)
`scankeywordcs`	string	returns true if the given keyword is gobbled; this variant is case sensitive and also suitable for utf8
`scanint`		returns an integer
`scanreal`		returns a number from e.g. `1`, `1.1`, `.1` with optional collapsed signs

`scanfloat`		returns a number from e.g. `1`, `1.1`, `.1`, `1.1E10`, , `.1e-10` with optional collapsed signs
`scandimen`	infinity, mu-units	returns a number representing a dimension or two numbers being the filler and order
`scanglue`	mu-units	returns a glue spec node
`scantoks`	definer, expand	returns a table of tokens
`scancode`	bitset	returns a character if its category is in the given bitset (representing catcodes)
`scanstring`		returns a string given between `{}`, as `\macro` or as sequence of characters with catcode 11 or 12
`scanargument`		this one is simular to `scanstring` but also accepts a `\cs` (which then get expanded)
`scanword`		returns a sequence of characters with catcode 11 or 12 as string
`scancsname`		returns `foo` after scanning `\foo`
`scanlist`		picks up a box specification and returns a `[h|v]list` node

The integer, dimension and glue scanners take an extra optional argument that signals that en optional equal is permitted.

The scanners can be considered stable apart from the one scanning for a token. The `scancode` function takes an optional number, the `scankeyword` function a normal Lua string. The `infinity` boolean signals that we also permit `fill` as dimension and the `mu-units` flags the scanner that we expect math units. When scanning tokens we can indicate that we are defining a macro, in which case the result will also provide information about what arguments are expected and in the result this is separated from the meaning by a separator token. The `expand` flag determines if the list will be expanded.

The `scanargument` function expands the given argument. When a braced argument is scanned, expansion can be prohibited by passing `false` (default is `true`). In case of a control sequence passing `false` will result in a one-level expansion (the meaning of the macro).

The string scanner scans for something between curly braces and expands on the way, or when it sees a control sequence it will return its meaning. Otherwise it will scan characters with catcode `letter` or `other`. So, given the following definition:

```
\def\oof{oof}
\def\foo{foo-\oof}
```

we get:

NAME	RESULT	
`\directlua{token.scanstring()}{foo}`	`foo`	full expansion
`\directlua{token.scanstring()}foo`	`foo`	letters and others
`\directlua{token.scanstring()}\foo`	`foo-oof`	meaning

The `\foo` case only gives the meaning, but one can pass an already expanded definition (`\edef`'d). In the case of the braced variant one can of course use the `\detokenize` and `\unexpanded` primitives since there we do expand.

The `scanword` scanner can be used to implement for instance a number scanner. An optional boolean argument can signal that a trailing space or `\relax` should be gobbled:

```
function token.scannumber(base)
    return tonumber(token.scanword(),base)
end
```

This scanner accepts any valid Lua number so it is a way to pick up floats in the input.

You can use the Lua interface as follows:

```
\directlua {
    function mymacro(n)
        ...
    end
}

\def\mymacro#1{%
    \directlua {
        mymacro(\number\dimexpr#1)
    }%
}

\mymacro{12pt}
\mymacro{\dimen0}
```

You can also do this:

```
\directlua {
    function mymacro()
        local d = token.scandimen()
        ...
    end
}

\def\mymacro{%
    \directlua {
        mymacro()
    }%
}

\mymacro 12pt
\mymacro \dimen0
```

It is quite clear from looking at the code what the first method needs as argument(s). For the second method you need to look at the Lua code to see what gets picked up. Instead of passing from TEX to Lua we let Lua fetch from the input stream.

In the first case the input is tokenized and then turned into a string, then it is passed to Lua

where it gets interpreted. In the second case only a function call gets interpreted but then the input is picked up by explicitly calling the scanner functions. These return proper Lua variables so no further conversion has to be done. This is more efficient but in practice (given what TEX has to do) this effect should not be overestimated. For numbers and dimensions it saves a bit but for passing strings conversion to and from tokens has to be done anyway (although we can probably speed up the process in later versions if needed).

12.6.2 Picking up one token

The scanners look for a sequence. When you want to pick up one token from the input you use `scannext`. This creates a token with the (low level) properties as discussed next. This token is just the next one. If you want to enforce expansion first you can use `scantoken` or the `_expanded` variants. Internally tokens are characterized by a number that packs a lot of information. In order to access the bits of information a token is wrapped in a userdata object.

The `expand` function will trigger expansion of the next token in the input. This can be quite unpredictable but when you call it you probably know enough about TEX not to be too worried about that. It basically is a call to the internal expand related function.

NAME	EXPLANATION
`scannext`	get the next token
`scannextexpanded`	get the next expanded token
`skipnext`	skip the next token
`skipnextexpanded`	skip the next expanded token
`peeknext`	get the next token and put it back in the input
`peeknextexpanded`	get the next expanded token and put it back in the input

The peek function accept a boolean argument that triggers skipping spaces and alike.

12.6.3 Creating tokens

The creator function can be used as follows:

```
local t = token.create("relax")
```

This gives back a token object that has the properties of the `\relax` primitive. The possible properties of tokens are:

NAME	EXPLANATION
`command`	a number representing the internal command number
`cmdname`	the type of the command (for instance the catcode in case of a character or the classifier that determines the internal treatment)
`csname`	the associated control sequence (if applicable)
`id`	the unique id of the token
`tok`	the full token number as stored in TEX
`active`	a boolean indicating the active state of the token
`expandable`	a boolean indicating if the token (macro) is expandable

protected	a boolean indicating if the token (macro) is protected
frozen	a boolean indicating if the token is a frozen command
user	a boolean indicating if the token is a user defined command
index	a number that indicated the subcommand; differs per command

Alternatively you can use a getter get<fieldname> to access a property of a token.

The numbers that represent a catcode are the same as in TEX itself, so using this information assumes that you know a bit about TEX's internals. The other numbers and names are used consistently but are not frozen. So, when you use them for comparing you can best query a known primitive or character first to see the values.

You can ask for a list of commands:

```
local t = token.commands()
```

The id of a token class can be queried as follows:

```
local id = token.command_id("math_shift")
```

If you really know what you're doing you can create character tokens by not passing a string but a number:

```
local letter_x = token.create(string.byte("x"))
local other_x  = token.create(string.byte("x"),12)
```

Passing weird numbers can give side effects so don't expect too much help with that. As said, you need to know what you're doing. The best way to explore the way these internals work is to just look at how primitives or macros or \chardef'd commands are tokenized. Just create a known one and inspect its fields. A variant that ignores the current catcode table is:

```
local whatever = token.new(123,12)
```

You can test if a control sequence is defined with is_defined, which accepts a string and returns a boolean:

```
local okay = token.is_defined("foo")
```

The largest character possible is returned by biggest_char, just in case you need to know that boundary condition.

12.6.4 Macros

The set_macro function can get upto 4 arguments:

```
set_macro("csname","content")
set_macro("csname","content","global")
set_macro("csname")
```

You can pass a catcodetable identifier as first argument:

```
set_macro(catcodetable,"csname","content")
set_macro(catcodetable,"csname","content","global")
set_macro(catcodetable,"csname")
```

The results are like:

```
 \def\csname{content}
\gdef\csname{content}
 \def\csname{}
```

The `getmacro` function can be used to get the content of a macro while the `getmeaning` function gives the meaning including the argument specification (as usual in TEX separated by `->`).

The `set_char` function can be used to do a `\chardef` at the Lua end, where invalid assignments are silently ignored:

```
set_char("csname",number)
set_char("csname",number,"global")
```

A special one is the following:

```
set_lua("mycode",id)
set_lua("mycode",id,"global","protected")
```

This creates a token that refers to a Lua function with an entry in the table that you can access with `lua.getfunctions_table`. It is the companion to `\luadef`. When the first (and only) argument is true the size will preset to the value of `texconfig.function_size`.

The `pushmacro` and `popmacro` function are very experimental and can be used to get and set an existing macro. The push call returns a user data object and the pop takes such a userdata object. These object have no accessors and are to be seen as abstractions.

12.6.5 Pushing back

There is a (for now) experimental putter:

```
local t1 = token.scannext()
local t2 = token.scannext()
local t3 = token.scannext()
local t4 = token.scannext()
-- watch out, we flush in sequence
token.putnext { t1, t2 }
-- but this one gets pushed in front
token.putnext ( t3, t4 )
```

When we scan `wxyz!` we get `yzwx!` back. The argument is either a table with tokens or a list of tokens. The `token.expand` function will trigger expansion but what happens really depends on what you're doing where.

This putter is actually a bit more flexible because the following input also works out okay:

```
\def\foo#1{[#1]}

\directlua {
    local list = { 101, 102, 103, token.create("foo"), "{abracadabra}" }
    token.putnext("(the)")
    token.putnext(list)
    token.putnext("(order)")
    token.putnext(unpack(list))
    token.putnext("(is reversed)")
}
```

We get this:

```
(is reversed)efg[abracadabra](order)efg[abracadabra](the)
```

So, strings get converted to individual tokens according to the current catcode regime and numbers become characters also according to this regime.

12.6.6 Nota bene

When scanning for the next token you need to keep in mind that we're not scanning like TEX does: expanding, changing modes and doing things as it goes. When we scan with Lua we just pick up tokens. Say that we have:

```
\oof
```

but `\oof` is undefined. Normally TEX will then issue an error message. However, when we have:

```
\def\foo{\oof}
```

We get no error, unless we expand `\foo` while `\oof` is still undefined. What happens is that as soon as TEX sees an undefined macro it will create a hash entry and when later it gets defined that entry will be reused. So, `\oof` really exists but can be in an undefined state.

oof : oof
foo : foo
myfirstoof :

This was entered as:

```
oof        : \directlua{tex.print(token.scancsname())}\oof
foo        : \directlua{tex.print(token.scancsname())}\foo
myfirstoof : \directlua{tex.print(token.scancsname())}\myfirstoof
```

The reason that you see `oof` reported and not `myfirstoof` is that `\oof` was already used in a previous paragraph.

If we now say:

```
\def\foo{}
```

we get:

oof : oof
foo : foo
myfirstoof :

And if we say

```
\def\foo{\oof}
```

we get:

oof : oof
foo : foo
myfirstoof :

When scanning from Lua we are not in a mode that defines (undefined) macros at all. There we just get the real primitive undefined macro token.

680183 537472695
678357 536969024
674849 536985953

This was generated with:

```
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\myfirstoof
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\mysecondoof
\directlua{local t = token.scannext() tex.print(t.id.." "..t.tok)}\mythirdoof
```

So, we do get a unique token because after all we need some kind of Lua object that can be used and garbage collected, but it is basically the same one, representing an undefined control sequence.

13 The MetaPost library mplib

13.1 Introduction

The library used in LuaMetaTEX differs from the one used in LuaTEX. There are for instance no backends and the binary number model is not available. There is also no textual output. There are scanners and injectors that make it possible to enhance the language and efficiently feed back into MetaPost. File handling is now completely delegated to Lua, so there are more callbacks.

Some functionality is experimental and therefore documentation is limited. Also, details are discussed in articles.

13.2 Process management

The MetaPost library interface registers itself in the table `mplib`. It is based on mplib version 3.11 (LuaTEX used version 2+). Not all functionality is described here. Once we're out of the experimental stage some more information will be added. Using the library boils down to initializing an instance, executing statements and picking up assembled figures in the form of Lua user data objects (and from there on Lua variables like tables).

13.2.1 new

To create a new MetaPost instance, call

```
<mpinstance> mp = mplib.new({...})
```

This creates the `mp` instance object. The argument is a hash table that can have a number of different fields, as follows:

NAME	TYPE	DESCRIPTION	DEFAULT
`error_line`	number	error line width	79
`print_line`	number	line length in ps output	100
`random_seed`	number	the initial random seed	variable
`math_mode`	string	the number system to use: `scaled`, `double` or `decimal`	`scaled`
`interaction`	string	the interaction mode: `batch`, `nonstop`, `scroll` or `errorstop`	`errorstop`
`job_name`	string	a compatibility value	
`utf8_mode`	boolean	permit characters in the range 128 upto 255 to be part of names	`false`
`text_mode`	boolean	permit characters 2 and 3 as fencing string literals	`false`
`tolerance`	number	the value used as criterium for straight lines	`131/65536`

extensions	boolean	enable all extensions	(might go)

The binary mode is no longer available in the LuaMetaTEX version of mplib. It offers no real advantage and brings a ton of extra libraries with platform specific properties that we can now avoid. We might introduce a high resolution scaled variant at some point but only when it pays of performance wise.

In addition to the above we need to provide functions that helps MetaPost communicate to the outside world.

NAME	TYPE	ARGUMENT(S)	RESULT
find_file	function	string, string, string	string
	function	string, string, number	string
open_file	function	string, string, string	table
	function	string, string, number	table
run_logger	function	number, string	
run_script	function	string	whatever [, boolean]
	function	number	whatever [, boolean]
make_text	function	string, number	string
run_internal	function	number, number, number, string	
run_overload	function	number, string, number	boolean
run_error	function	string, string, number	

The find_file and open_file functions should be of this form:

```
<string> found   = find_file (<string> name, <string> mode, <string> type)
<table>  actions = open_file (<string> name, <string> mode, <string> type)
```

where the mode is r or w and the type is mp, data, terminal or a number, The finder is supposed to return the full path name of the found file, or nil if the file cannot be found. The open_file is supposed to return a table with a close and read function. This is similar to the way we do it in TEX. The special name terminal is used for interactive input. A numeric type indicates a specific read or write channel.

The run_logger callback gets a target and a string. A target 1 means log, a value 2 means and 3 means both.

The run_script function gets either a number or a string. The string represents a script, the number can be used as reference to something stored. The return value can be a boolean, number, string or table. Booleans and numbers are injected directly, strings and concatenated tables are fed into scantokens. When the second argument is true, the strings are also injected directly and tables are injected as pairs, colors, paths, transforms, depending on how many elements there are.

The run_internal function triggers when internal MetaPost variables flagged with runscript are initialized, saved or restored. The first argument is an index, the second the action. When initialized a third and fourth argument are passed. This is an experimental feature.

The experimental run_overload callback kicks in when a variable (or macro) with a property other than zero is redefined. It gets a property, name and the value of overloadmode passed and when the function returns true redefinition is permitted.

The run_error callback gets the error message, help text and current interaction mode passed. Normally it's best to just quit and let the user fix the code.

When you are processing a snippet of text starting with btex or verbatimtex and ending with etex, the MetaPost texscriptmode parameter controls how spaces and newlines get honoured. The default value is 1. Possible values are:

NAME	MEANING
0	no newlines
1	newlines in verbatimtex
2	newlines in verbatimtex and etex
3	no leading and trailing strip in verbatimtex
4	no leading and trailing strip in verbatimtex and btex

That way the Lua handler (assigned to make_text) can do what it likes. An etex has to be followed by a space or ; or be at the end of a line and preceded by a space or at the beginning of a line. The make_text function can return a string that gets fed into scantokens.

13.2.2 getstatistics

You can request statistics with:

```
<table> stats = mp:getstatistics()
```

This function returns the vital statistics for an mplib instance. Some are useful, others make more sense when debugging.

FIELD	TYPE	EXPLANATION
memory	number	bytes of node memory
hash	number	size of the hash
parameters	number	allocated parameter stack
input	number	allocated input stack
tokens	number	number of token nodes
pairs	number	number of pair nodes
knots	number	number of knot nodes
nodes	number	number of value nodes
symbols	number	number of symbolic nodes
characters	number	number of string bytes
strings	number	number of strings
internals	number	number of internals

Note that in the new version of mplib, this is informational only. The objects are all allocated dynamically, so there is no chance of running out of space unless the available system memory is exhausted.

13.2.3 execute

You can ask the MetaPost interpreter to run a chunk of code by calling

```
<table> rettable = execute(mp,"metapost code")
```

for various bits of MetaPost language input. Be sure to check the `rettable.status` (see below) because when a fatal MetaPost error occurs the mplib instance will become unusable thereafter.

Generally speaking, it is best to keep your chunks small, but beware that all chunks have to obey proper syntax, like each of them is a small file. For instance, you cannot split a single statement over multiple chunks.

In contrast with the normal stand alone `mpost` command, there is *no* implied 'input' at the start of the first chunk. When no string is passed to the execute function, there will still be one triggered because it then expects input from the terminal and you can emulate that channel with the callback you provide.

13.2.4 finish

Once you create an instance it is likely that you will keep it open for successive processing, if only because you want to avoid loading a format each time. If for some reason you want to stop using an mplib instance while processing is not yet actually done, you can call `finish`.

```
<table> rettable = finish(mp)
```

Eventually, used memory will be freed and open files will be closed by the Lua garbage collector, but an explicit `finish` is the only way to capture the final part of the output streams.

13.2.5 settolerance and gettolerance

These two functions relate to the bend tolerance, a value that is used when the export determines if a path has straight lines (like a rectangle has).

13.2.6 Errors

In case of an error you can get the context where it happened with `showcontext`.

13.2.7 The scanner status

When processing a graphic an instance is in a specific state and again we have a getter for the (internal) values `mplib.getstates()`: 0: normal, 1: skipping, 2: flushing, 3: absorbing, 4: var_defining, 5: op_defining, 6: loop_defining. The current status can be queried with `getstatus`.

13.2.8 The hash

Macro names and variable names are stored in a hash table. You can get a list with entries with `gethashentries`, which takes an instance as first argument. When the second argument is `true` more details will be provided. With `gethashentry` you get info about the given macro or variable.

13.2.9 Callbacks

Some statistics about the number of calls to the callbacks can be queried with `getcallbackstate`, This function expects a valid instance.

13.3 The end result

13.3.1 The figure

The return value of `execute` and `finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

FIELD	TYPE	EXPLANATION
`status`	number	the return value: `0` = good, `1` = warning, `2` = errors, `3` = fatal error
`fig`	table	an array of generated figures (if any)

When `status` equals 3, you should stop using this mplib instance immediately, it is no longer capable of processing input.

If it is present, each of the entries in the `fig` array is a userdata representing a figure object, and each of those has a number of object methods you can call:

You can check if a figure uses stacking with the `stacking` function. When objects are fetched, memory gets freed so no information about stacking is available then. You can get the used bend tolerance of an object with `tolerance`.

FIELD	TYPE	EXPLANATION
`boundingbox`	function	returns the bounding box, as an array of 4 values
`objects`	function	returns the actual array of graphic objects in this `fig`
`filename`	function	the filename this `fig`'s PostScript output would have written to in stand alone mode
`width`	function	the `fontcharwd` value
`height`	function	the `fontcharht` value
`depth`	function	the `fontchardp` value
`italic`	function	the `fontcharit` value
`charcode`	function	the (rounded) `charcode` value
`stacking`	function	is there a non-zero stacking

Note: you can call `fig:objects()` only once for any one `fig` object! Some information, like

stacking, can only be queried when the complete figure is still present and calling up objects will free elements in the original once they are transferred.

When the boundingbox represents a 'negated rectangle', i.e. when the first set of coordinates is larger than the second set, the picture is empty.

Graphical objects come in various types: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `start_group` and `stop_group`. Each type has a different list of accessible values.

There is a helper function (`mplib.fields(obj)`) to get the list of accessible values for a particular object, but you can just as easily use the tables given below.

All graphical objects have a field `type` that gives the object type as a string value; it is not explicit mentioned in the following tables. In the following, `numbers` are PostScript points (base points in TEX speak) represented as a floating point number, unless stated otherwise. Field values that are of type `table` are explained in the next section.

13.3.2 fill

FIELD	TYPE	EXPLANATION
`path`	table	the list of knots
`htap`	table	the list of knots for the reversed trajectory
`pen`	table	knots of the pen
`color`	table	the object's color
`linejoin`	number	line join style (bare number)
`miterlimit`	number	miterlimit
`prescript`	string	the prescript text
`postscript`	string	the postscript text
`stacking`	number	the stacking (level)

The entries `htap` and `pen` are optional.

13.3.3 outline

FIELD	TYPE	EXPLANATION
`path`	table	the list of knots
`pen`	table	knots of the pen
`color`	table	the object's color
`linejoin`	number	line join style (bare number)
`miterlimit`	number	miterlimit
`linecap`	number	line cap style (bare number)
`dash`	table	representation of a dash list
`prescript`	string	the prescript text
`postscript`	string	the postscript text
`stacking`	number	the stacking (level)

The entry `dash` is optional.

13.3.4 start_bounds, start_clip, start_group

FIELD	TYPE	EXPLANATION
path	table	the list of knots
stacking	number	the stacking (level)

13.3.5 stop_bounds, stop_clip, stop_group

Here we have only one key:

FIELD	TYPE	EXPLANATION
stacking	number	the stacking (level)

13.4 Subsidiary table formats

13.4.1 Paths and pens

Paths and pens (that are really just a special type of paths as far as mplib is concerned) are represented by an array where each entry is a table that represents a knot.

FIELD	TYPE	EXPLANATION
left_type	string	when present: endpoint, but usually absent
right_type	string	like left_type
x_coord	number	X coordinate of this knot
y_coord	number	Y coordinate of this knot
left_x	number	X coordinate of the precontrol point of this knot
left_y	number	Y coordinate of the precontrol point of this knot
right_x	number	X coordinate of the postcontrol point of this knot
right_y	number	Y coordinate of the postcontrol point of this knot

There is one special case: pens that are (possibly transformed) ellipses have an extra key type with value elliptical besides the array part containing the knot list.

13.4.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

FIELD	TYPE	EXPLANATION
0	marking only	no values
1	greyscale	one value in the range $(0, 1)$, 'black' is 0
3	rgb	three values in the range $(0, 1)$, 'black' is $0, 0, 0$
4	cmyk	four values in the range $(0, 1)$, 'black' is $0, 0, 0, 1$

If the color model of the internal object was uninitialized, then it was initialized to the values

representing ‘black’ in the colorspace `defaultcolormodel` that was in effect at the time of the `shipout`.

13.4.3 Transforms

Each transform is a six-item array.

INDEX	TYPE	EXPLANATION
1	number	represents x
2	number	represents y
3	number	represents xx
4	number	represents yx
5	number	represents xy
6	number	represents yy

Note that the translation (index 1 and 2) comes first. This differs from the ordering in PostScript, where the translation comes last.

13.4.4 Dashes

Each `dash` is a hash with two items. We use the same model as PostScript for the representation of the dashlist. `dashes` is an array of ‘on’ and ‘off’, values, and `offset` is the phase of the pattern.

FIELD	TYPE	EXPLANATION
`dashes`	hash	an array of on-off numbers
`offset`	number	the starting offset value

13.4.5 Pens and `peninfo`

There is helper function (`peninfo(obj)`) that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

FIELD	TYPE	EXPLANATION
`width`	number	width of the pen
`sx`	number	x scale
`rx`	number	$x\,y$ multiplier
`ry`	number	$y\,x$ multiplier
`sy`	number	y scale
`tx`	number	x offset
`ty`	number	y offset

13.4.6 Character size information

These functions find the size of a glyph in a defined font. The `fontname` is the same name as the

argument to `infont`; the `char` is a glyph id in the range 0 to 255; the returned `w` is in AFM units.

```
<number> w = char_width(mp,<string> fontname, <number> char)
<number> h = char_height(mp,<string> fontname, <number> char)
<number> d = char_depth(mp,<string> fontname, <number> char)
```

13.5 Scanners

After a relative long period of testing the scanners are now part of the interface. That doesn't mean that there will be no changes: depending on the needs and experiences details might evolve. The summary below is there still preliminary and mostly provided as reminder.

SCANNER	ARGUMENT	RETURNS
`scannext`	instance, keep	token, mode, type
`scanexpression`	instance, keep	type
`scantoken`	instance, keep	token, mode, kind
`scansymbol`	instance, keep, expand	string
`scannumeric`	instance, type	number
`scaninteger`	instance, type	integer
`scanboolean`	instance, type	boolean
`scanstring`	instance, type	string
`scanpair`	instance, hashed, type	table or two numbers
`scancolor`	instance, hashed, type	table or three numbers
`scancmykcolor`	instance, hashed, type	table or four numbers
`scantransform`	instance, hashed, type	table or six numbers
`scanpath`	instance, hashed, type	table with hashes or arrays
`scanpen`	instance, hashed, type	table with hashes or arrays
`scanproperty`	*todo*	
`skiptoken`	*todo*	

The types and token codes are numbers but they actually depend on the implementation (although changes are unlikely). The types of data structures can be queried with `mplib.gettypes()`: 0: undefined, 1: vacuous, 2: boolean, 3: unknownboolean, 4: string, 5: unknownstring, 6: pen, 7: unknownpen, 8: path, 9: unknownpath, 10: picture, 11: unknownpicture, 12: transform, 13: color, 14: cmykcolor, 15: pair, 16: numeric, 17: known, 18: dependent, 19: protodependent, 20: independent, 21: tokenlist, 22: structured, 23: unsuffixedmacro, 24: suffixedmacro, and command codes with `mplib.getcodes()`: 0: undefined, 1: btex, 2: etex, 3: if, 4: fiorelse, 5: input, 6: iteration, 7: repeatloop, 8: exittest, 9: relax, 10: scantokens, 11: runscript, 12: maketext, 13: expandafter, 14: definedmacro, 15: save, 16: interim, 17: let, 18: newinternal, 19: macrodef, 20: shipout, 21: addto, 22: setbounds, 23: protection, 24: property, 25: show, 26: mode, 27: randomseed, 28: message, 29: everyjob, 30: delimiters, 31: write, 32: typename, 33: leftdelimiter, 34: begingroup, 35: nullary, 36: unary, 37: str, 38: void, 39: cycle, 40: ofbinary, 41: capsule, 42: string, 43: internal, 44: tag, 45: numeric, 46: plusorminus, 47: secondarydef, 48: tertiarybinary, 49: leftbrace, 50: pathjoin, 51: ampersand, 52: tertiarydef, 53: primarybinary, 54: equals, 55: and, 56: primarydef, 57: slash, 58: secondarybinary, 59: parametertype, 60: controls, 61: tension, 62: atleast, 63: curl, 64: macrospecial, 65: rightde-

limiter, 66: leftbracket, 67: rightbracket, 68: rightbrace, 69: with, 70: thingstoadd, 71: of, 72: to, 73: step, 74: until, 75: within, 76: assignment, 77: colon, 78: comma, 79: semicolon, 80: endgroup, 81: stop, 82: undefinedcs

Now, if you really want to use these, keep in mind that the internals of MetaPost are not trivial, especially because expression scanning can be complex. So you need to experiment a bit. In ConTEXt all is (and will be) hidden below an abstraction layer so users are not bothered by all these look-ahead and push-back issues that originate in the way MetaPost scans its input.

The supported color models are: `mplib.getcolormodels()`: 0: no, 1: grey, 2: rgb, 3: cmyk.

If you want the internal codes of the possible fields in a graphic object use `mplib.getobjecttypes()`: 0: , 1: fill, 2: outline, 3: start_clip, 4: start_group, 5: start_bounds, 6: stop_clip, 7: stop_group, 8: stop_bounds. You can query the id of a graphic object with the `gettype` function.

ID	OBJECT	FIELDS
1	`fill`	`type path htap pen color linejoin miterlimit prescript postscript stacking`
2	`outline`	`type path pen color linejoin miterlimit linecap dash prescript postscript stacking`
3	`start_clip`	`type path prescript postscript stacking`
4	`start_group`	`type path prescript postscript stacking`
5	`start_bounds`	`type path prescript postscript stacking`
6	`stop_clip`	`type stacking`
7	`stop_group`	`type stacking`
8	`stop_bounds`	`type stacking`

13.6 Injectors

It is important to know that piping code into the library is pretty fast and efficient. Most processing time relates to memory management, calculations and generation of output can not be neglected either. Out of curiousity I added some functions that directly push data into the library but the gain is not that large.[6]

SCANNER	ARGUMENT
`injectnumeric`	instance, number
`injectinteger`	instance, number
`injectboolean`	instance, boolean
`injectstring`	instance, string
`injectpair`	instance, (table with) two numbers
`injectcolor`	instance, (table with) three numbers
`injectcmykcolor`	instance, (table with) four numbers
`injecttransform`	instance, (table with) six numbers
`injectpath`	instance, table with hashes or arrays, cycle, variant
`injectwhatever`	instance, ont of the above depending on type and size

[6] The main motivation was checking of huge paths could be optimized. The other data structures were then added for completeness.

The path injector takes a table with subtables that are either hashed (like the path solver) or arrays with two, four or six entries. When the third argument has the value `true` the path is closed. When the fourth argument is `true` the path is constructed out of straight lines (as with `--`) by setting the `curl` values to 1 automatically.[7]

This is the simplest path definition:

```
{
    { x, y },
    ...,
    cycle = true
}
```

and this one also has the control points:

```
{
    { x0, y0, x1, y1, x2, y2 },
    ...,
    cycle = true
}
```

A very detailed specification is this but you have to make sure that the parameters make sense.

```
{
    {
        x_coord       = ...,
        y_coord       = ...,
        left_x        = ...,
        left_y        = ...,
        right_x       = ...,
        right_y       = ...,
        left_tension  = ...,
        right_tension = ...,
        left_curl     = ...,
        right_curl    = ...,
        direction_x   = ...,
        direction_y   = ...,
        left_type     = ...,
        right_type    = ...,
    },
    ...,
    cycle = true
}
```

Instead of the optional keyword `cycle` you can use `close`.

[7] This is all experimental so future versions might provide more control.

13.7 To be checked

```
% solvepath
% expandtex
```

14 The pdf related libraries

14.1 The pdfe library

14.1.1 Introduction

The pdfe library replaces the epdf library and provides an interface to pdf files. It uses the same code as is used for pdf image inclusion. The pplib library by Paweł Jackowski replaces the poppler (derived from xpdf) library.

A pdf file is basically a tree of objects and one descends into the tree via dictionaries (key/value) and arrays (index/value). There are a few topmost dictionaries that start at root that are accessed more directly.

Although everything in pdf is basically an object we only wrap a few in so called userdata Lua objects.

TYPE	MAPPING
pdf	**Lua**
null	nil
boolean	boolean
integer	integer
float	number
name	string
string	string
array	array userdatum
dictionary	dictionary userdatum
stream	stream userdatum (with related dictionary)
reference	reference userdatum

The regular getters return these Lua data types but one can also get more detailed information.

14.1.2 open, openfile, new, getstatus, close, unencrypt

A document is loaded from a file (by name or handle) or string:

```
<pdfe document> = pdfe.open(filename)
<pdfe document> = pdfe.openfile(filehandle)
<pdfe document> = pdfe.new(somestring,somelength)
```

Such a document is closed with:

```
pdfe.close(<pdfe document>)
```

You can check if a document opened well by:

```
pdfe.getstatus(<pdfe document>)
```

The returned codes are:

VALUE	EXPLANATION
-2	the document failed to open
-1	the document is (still) protected
0	the document is not encrypted
2	the document has been unencrypted

An encrypted document can be unencrypted by the next command where instead of either password you can give `nil`:

```
pdfe.unencrypt(<pdfe document>,userpassword,ownerpassword)
```

14.1.3 getsize, getversion, getnofobjects, getnofpages

A successfully opened document can provide some information:

```
bytes = getsize(<pdfe document>)
major, minor = getversion(<pdfe document>)
n = getnofobjects(<pdfe document>)
n = getnofpages(<pdfe document>)
bytes, waste = getnofpages(<pdfe document>)
```

14.1.4 get[catalog|trailer|info]

For accessing the document structure you start with the so called catalog, a dictionary:

```
<pdfe dictionary> = pdfe.getcatalog(<pdfe document>)
```

The other two root dictionaries are accessed with:

```
<pdfe dictionary> = pdfe.gettrailer(<pdfe document>)
<pdfe dictionary> = pdfe.getinfo(<pdfe document>)
```

14.1.5 getpage, getbox

A specific page can conveniently be reached with the next command, which returns a dictionary.

```
<pdfe dictionary> = pdfe.getpage(<pdfe document>,pagenumber)
```

Another convenience command gives you the (bounding) box of a (normally page) which can be inherited from the document itself. An example of a valid box name is `MediaBox`.

```
pages = pdfe.getbox(<pdfe dictionary>,boxname)
```

14.1.6 get[string|integer|number|boolean|name]

Common values in dictionaries and arrays are strings, integers, floats, booleans and names (which are also strings) and these are also normal Lua objects:

```
s = getstring (<pdfe array|dictionary>,index|key)
i = getinteger(<pdfe array|dictionary>,index|key)
n = getnumber (<pdfe array|dictionary>,index|key)
b = getboolean(<pdfe array|dictionary>,index|key)
n = getname   (<pdfe array|dictionary>,index|key)
```

The `getstring` function has two extra variants:

```
s, h = getstring (<pdfe array|dictionary>,index|key,false)
s    = getstring (<pdfe array|dictionary>,index|key,true)
```

The first call returns the original string plus a boolean indicating if the string is hex encoded. The second call returns the unencoded string.

14.1.7 get[dictionary|array|stream]

Normally you will use an index in an array and key in a dictionary but dictionaries also accept an index. The size of an array or dictionary is available with the usual # operator.

```
<pdfe dictionary>   = getdictionary(<pdfe array|dictionary>,index|key)
<pdfe array>        = getarray     (<pdfe array|dictionary>,index|key)
<pdfe stream>,
<pdfe dictionary>   = getstream    (<pdfe array|dictionary>,index|key)
```

These commands return dictionaries, arrays and streams, which are dictionaries with a blob of data attached.

Before we come to an alternative access mode, we mention that the objects provide access in a different way too, for instance this is valid:

```
print(pdfe.open("foo.pdf").Catalog.Type)
```

At the topmost level there are `Catalog`, `Info`, `Trailer` and `Pages`, so this is also okay:

```
print(pdfe.open("foo.pdf").Pages[1])
```

14.1.8 [open|close|readfrom|whole|]stream

Streams are sort of special. When your index or key hits a stream you get back a stream object and dictionary object. The dictionary you can access in the usual way and for the stream there are the following methods:

```
okay   = openstream(<pdfe stream>,[decode])
         closestream(<pdfe stream>)
```

```
str, n = readfromstream(<pdfe stream>)
str, n = readwholestream(<pdfe stream>,[decode])
```

You either read in chunks, or you ask for the whole. When reading in chunks, you need to open and close the stream yourself. The `n` value indicates the length read. The `decode` parameter controls if the stream data gets uncompressed.

As with dictionaries, you can access fields in a stream dictionary in the usual Lua way too. You get the content when you 'call' the stream. You can pass a boolean that indicates if the stream has to be decompressed.

14.1.9 getfrom[dictionary|array]

In addition to the interface described before, there is also a bit lower level interface available.

```
key, type, value, detail = getfromdictionary(<pdfe dictionary>,index)
type, value, detail = getfromarray(<pdfe array>,index)
```

TYPE	MEANING	VALUE	DETAIL
0	none	nil	
1	null	nil	
2	boolean	boolean	
3	integer	integer	
4	number	float	
5	name	string	
6	string	string	hex
7	array	arrayobject	size
8	dictionary	dictionaryobject	size
9	stream	streamobject	dictionary size
10	reference	integer	

A `hex` string is (in the pdf file) surrounded by <> while plain strings are bounded by <>.

14.1.10 [dictionary|array]totable

All entries in a dictionary or table can be fetched with the following commands where the return values are a hashed or indexed table.

```
hash = dictionarytotable(<pdfe dictionary>)
list = arraytotable(<pdfe array>)
```

You can get a list of pages with:

```
{ { <pdfe dictionary>, size, objnum }, ... } = pagestotable(<pdfe document>)
```

14.1.11 getfromreference

Because you can have unresolved references, a reference object can be resolved with:

```
type, <pdfe dictionary|array|stream>, detail = getfromreference(<pdfe reference>)
```

So, as second value you get back a new pdfe userdata object that you can query.

14.2 Memory streams

The pdfe.new function takes three arguments:

VALUE	EXPLANATION
stream	this is a (in low level Lua speak) light userdata object, i.e. a pointer to a sequence of bytes
length	this is the length of the stream in bytes (the stream can have embedded zeros)
name	optional, this is a unique identifier that is used for hashing the stream

The third argument is optional. When it is not given the function will return a pdfe document object as with a regular file, otherwise it will return a filename that can be used elsewhere (e.g. in the image library) to reference the stream as pseudo file.

Instead of a light userdata stream (which is actually fragile but handy when you come from a library) you can also pass a Lua string, in which case the given length is (at most) the string length.

The function returns a pdfe object and a string. The string can be used in the img library instead of a filename. You need to prevent garbage collection of the object when you use it as image (for instance by storing it somewhere).

Both the memory stream and it's use in the image library is experimental and can change. In case you wonder where this can be used: when you use the swiglib library for graphicmagick, it can return such a userdata object. This permits conversion in memory and passing the result directly to the backend. This might save some runtime in one-pass workflows. This feature is currently not meant for production and we might come up with a better implementation.

14.3 The pdfscanner library

This library is not available in LuaMetaTEX.

15 Extra libraries

15.1 Introduction

The libraries can be grouped in categories like fonts, languages, TEX, MetaPost, pdf, etc. There are however also some that are more general purpose and these are discussed here.

15.2 File and string readers: fio and type sio

This library provides a set of functions for reading numbers from a file and in addition to the regular io library functions. The following work on normal Lua file handles.

NAME	ARGUMENTS	RESULTS
readcardinal1	(f)	a 1 byte unsigned integer
readcardinal2	(f)	a 2 byte unsigned integer
readcardinal3	(f)	a 3 byte unsigned integer
readcardinal4	(f)	a 4 byte unsigned integer
readcardinaltable	(f,n,b)	n cardinals of b bytes
readinteger1	(f)	a 1 byte signed integer
readinteger2	(f)	a 2 byte signed integer
readinteger3	(f)	a 3 byte signed integer
readinteger4	(f)	a 4 byte signed integer
readintegertable	(f,n,b)	n integers of b bytes
readfixed2	(f)	a float made from a 2 byte fixed format
readfixed4	(f)	a float made from a 4 byte fixed format
read2dot14	(f)	a float made from a 2 byte in 2dot4 format
setposition	(f,p)	goto position p
getposition	(f)	get the current position
skipposition	(f,n)	skip n positions
readbytes	(f,n)	n bytes
readbytetable	(f,n)	n bytes

When relevant there are also variants that end with le that do it the little endian way. The fixed and dot floating points formats are found in font files and return Lua doubles.

A similar set of function as in the fio library is available in the sio library: sio.readcardinal1, sio.readcardinal2, sio.readcardinal3, sio.readcardinal4, sio.readcardinaltable, sio.readinteger1, sio.readinteger2, sio.readinteger3, sio.readinteger4, sio.readintegertable, sio.readfixed2, sio.readfixed4, sio.read2dot14, sio.setposition, sio.getposition, sio.skipposition, sio.readbytes and sio.readbytetable. Here the first argument is a string instead of a file handle.

15.3 md5

NAME	ARGUMENTS	RESULTS
sum		
hex		
HEX		

15.4 sha2

NAME	ARGUMENTS	RESULTS
digest256		
digest384		
digest512		

15.5 xzip

NAME	ARGUMENTS	RESULTS
compress		
decompress		
adler32		
crc32		

15.6 xmath

This library just opens up standard C math library and the main reason for it being there is that it permits advanced graphics in MetaPost (via the Lua interface). There are three constant values:

NAME	ARGUMENTS	RESULTS
inf	–	inf
nan	–	nan
pi	–	3.1415926535898

and a lot of functions:

NAME	ARGUMENTS	RESULTS
acos	(a)	
acosh	(a)	
asin	(a)	
asinh	(a)	
atan	(a[,b])	
atan2	(a[,b])	
atanh	(a)	
cbrt	(a)	

```
ceil                (a)
copysign            (a,b)
cos                 (a)
cosh                (a)
deg                 (a)
erf                 (a)
erfc                (a)
exp                 (a)
exp2                (a)
expm1               (a)
fabs                (a)
fdim                (a,b)
floor               (a)
fma                 (a,b,c)
fmax                (...)
fmin                (...)
fmod                (a,b)
frexp               (a,b)
gamma               (a)
hypot               (a,b)
isfinite            (a)
isinf               (a)
isnan               (a)
isnormal            (a)
j0                  (a)
j1                  (a)
jn                  (a,b)
ldexp               (a,b)
lgamma              (a)
l0                  (a)
l1                  (a)
ln                  (a,b)
log                 (a[,b])
log10               (a)
log1p               (a)
log2                (a)
logb                (a)
modf                (a,b)
nearbyint           (a)
nextafter           (a,b)
pow                 (a,b)
rad                 (a)
remainder           (a,b)
remquo              (a,b)
round               (a)
scalbn              (a,b)
```

sin	(a)
sinh	(a)
sqrt	(a)
tan	(a)
tanh	(a)
tgamma	(a)
trunc	(a)
y0	(a)
y1	(a)
yn	(a)

15.7 xcomplex

LuaMetaTEX also provides a complex library `xcomplex`. The complex number is a userdatum:

NAME	ARGUMENTS	RESULTS
new	(r,i)	a complex userdata type
tostring	(z)	a string representation
topair	(z)	two numbers

There is a bunch of functions that take a complex number:

NAME	ARGUMENTS	RESULTS
abs	(a)	
arg	(a)	
imag	(a)	
real	(a)	
onj	(a)	
proj	(a)	
exp"	(a)	
log	(a)	
sqrt	(a)	
pow	(a,b)	
sin	(a)	
cos	(a)	
tan	(a)	
asin	(a)	
acos	(a)	
atan	(a)	
sinh	(a)	
cosh	(a)	
tanh	(a)	
asinh	(a)	
acosh	(a)	
atanh	(a)	

These are accompanied by `libcerf` functions:

NAME	ARGUMENTS	RESULTS
erf	(a)	The complex error function erf(z)
erfc	(a)	The complex complementary error function erfc(z) = 1 - er
erfcx	(a)	The underflow-compensating function erfcx(z) = exp(z^2) e
erfi	(a)	The imaginary error function erfi(z) = -i erf(iz)
dawson	(a)	Dawson's integral D(z) = sqrt(pi)/2 * exp(-z^2) * erfi(z)
voigt	(a,b,c)	The convolution of a Gaussian and a Lorentzian
voigt_hwhm	(a,b)	The half width at half maximum of the Voigt profile

15.8 xdecimal

As an experiment LuaMetaTEX provides an interface to the `decNumber` library that we have on board for MetaPost anyway. Apart from the usual support for operators there are some functions.

NAME	ARGUMENTS	RESULTS
abs	(a)	
new	([n or s])	
copy	(a)	
trim	(a)	
tostring	(a)	
tonumber	(a)	
setprecision	(n)	
getprecision	()	
conj	(a)	
abs	(a)	
pow	(a,b)	
sqrt	(a)	
ln	(a)	
log	(a)	
exp	(a)	
bor	(a,b)	
bxor	(a,b)	
band	(a,b)	
shift	(a,b)	
rotate	(a,b)	
minus	(a)	
plus	(a)	
min	(a,b)	
max	(a,b)	

15.9 lfs

The original `lfs` module has been adapted a bit to our needs but for practical reasons we kept

the namespace. This module will probably evolve a bit over time.

NAME	ARGUMENTS	RESULTS
attributes	(name)	
chdir	(name)	
currentdir	()	
dir	(name)	name, mode, size and mtime
mkdir	(name)	
rmdir	(name)	
touch	(name)	
link	(name)	
symlinkattributes	(name)	
isdir	(name)	
isfile	(name)	
iswriteabledir	(name)	
iswriteablefile	(name)	
isreadabledir	(name)	
isreadablefile	(name)	

The `dir` function is a traverser which in addition to the name returns some more properties. Keep in mind that the traverser loops over a directory and that it doesn't run well when used nested. This is a side effect of the operating system. It is also the reason why we return some properties because querying them via `attributes` would interfere badly.

The following attributes are returned by `attributes`:

NAME	VALUE
mode	
size	
modification	
access	
change	
permissions	
nlink	

15.10 pngdecode

This module is experimental and used in image inclusion. It is not some general purpose module and is supposed to be used in a very controlled way. The interfaces might evolve.

NAME	ARGUMENTS	RESULTS
applyfilter	(str,nx,ny,slice)	string
splitmask	(str,nx,ny,bpp,bytes)	string
interlace	(str,nx,ny,slice,pass)	string
expand	(str,nx,ny,parts,xline,factor)	string

15.11 basexx

Some more experimental helpers:

NAME	ARGUMENTS	RESULTS
encode16	(str[,newline])	string
decode16	(str)	string
encode64	(str[,newline])	string
decode64	(str)	string
encode85	(str[,newline])	string
decode85	(str)	string
encodeRL	(str)	string
decodeRL	(str)	string
encodeLZW	(str[,defaults])	string
decodeLZW	(str[,defaults])	string

15.12 Multibyte `string` functions

The `string` library has a few extra functions, for example `string.explode`. This function takes upto two arguments: `string.explode(s[,m])` and returns an array containing the string argument `s` split into sub-strings based on the value of the string argument `m`. The second argument is a string that is either empty (this splits the string into characters), a single character (this splits on each occurrence of that character, possibly introducing empty strings), or a single character followed by the plus sign `+` (this special version does not create empty sub-strings). The default value for `m` is ‘ +’ (multiple spaces). Note: `m` is not hidden by surrounding braces as it would be if this function was written in TEX macros.

The `string` library also has six extra iterators that return strings piecemeal: `string.utfvalues`, `string.utfcharacters`, `string.characters`, `string.characterpairs`, `string.bytes` and `string.bytepairs`.

- `string.utfvalues(s)`: an integer value in the Unicode range
- `string.utfcharacters(s)`: a string with a single utf-8 token in it
- `string.characters(s)`: a string containing one byte
- `string.characterpairs(s)`: two strings each containing one byte or an empty second string if the string length was odd
- `string.bytes(s)`: a single byte value
- `string.bytepairs(s)`: two byte values or nil instead of a number as its second return value if the string length was odd

The `string.characterpairs()` and `string.bytepairs()` iterators are useful especially in the conversion of utf16 encoded data into utf8.

There is also a two-argument form of `string.dump()`. The second argument is a boolean which, if true, strips the symbols from the dumped data. This matches an extension made in `luajit`. This is typically a function that gets adapted as Lua itself progresses.

The `string` library functions `len`, `lower`, `sub` etc. are not Unicode-aware. For strings in the

utf8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the `slnunicode` library can be used, e.g., `unicode.utf8.len`, `unicode.utf8.lower` etc. The exceptions are `unicode.utf8.find`, that always returns byte positions in a string, and `unicode.utf8.match` and `unicode.utf8.gmatch`. While the latter two functions in general *are* Unicode-aware, they fall-back to non-Unicode-aware behavior when using the empty capture `()` but other captures work as expected. For the interpretation of character classes in `unicode.utf8` functions refer to the library sources at http://luaforge.net/projects/sln.

Version 5.3 of Lua provides some native utf8 support but we have added a few similar helpers too: `string.utfvalue`, `string.utfcharacter` and `string.utflength`.

- `string.utfvalue(s)`: returns the codepoints of the characters in the given string
- `string.utfcharacter(c,...)`: returns a string with the characters of the given code points
- `string.utflength(s)`: returns the length of the given string

These three functions are relative fast and don't do much checking. They can be used as building blocks for other helpers.

15.13 Extra os library functions

The `os` library has a few extra functions and variables: `os.selfdir`, `os.selfarg`, `os.setenv`, `os.env`, `os.gettimeofday`, `os.type`, `os.name` and `os.uname`, that we will discuss here. There are also some time related helpers in the `lua` namespace.

- `os.selfdir` is a variable that holds the directory path of the actual executable. For example: `\directlua{tex.sprint(os.selfdir)}`.
- `os.selfarg` is a table with the command line arguments.
- `os.setenv(key,value)` sets a variable in the environment. Passing `nil` instead of a value string will remove the variable.
- `os.env` is a hash table containing a dump of the variables and values in the process environment at the start of the run. It is writeable, but the actual environment is *not* updated automatically.
- `os.gettimeofday` returns the current 'Unix time', but as a float. Keep in mind that there might be platforms where this function is not available.
- `os.type` is a string that gives a global indication of the class of operating system. The possible values are currently `windows`, `unix`, and `msdos` (you are unlikely to find this value 'in the wild').
- `os.name` is a string that gives a more precise indication of the operating system. These possible values are not yet fixed, and for `os.type` values `windows` and `msdos`, the `os.name` values are simply `windows` and `msdos`
  The list for the type `unix` is more precise: `linux`, `freebsd`, `kfreebsd`, `cygwin`, `openbsd`, `solaris`, `sunos` (pre-solaris), `hpux`, `irix`, `macosx`, `gnu` (hurd), `bsd` (unknown, but bsd-like), `sysv`, `generic` (unknown). But ... we only provide LuaMetaTEX binaries for the mainstream variants.
  Officially we only support mainstream systems: MS Windows, linux, FreeBSD and os-x. Of course one can build LuaMetaTEX for other systems, in which case on has to check the above.
- `os.uname` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string values, and their names are: `sysname`, `machine`,

`release`, `version`, and `nodename`.

15.14 The `lua` library functions

The `lua` library provides some general helpers.

- The `newtable` and `newindex` functions can be used to create tables with space reserved beforehand for the given amount of entries.
- The `getstacktop` function returns a number that can be used for diagnostic purposes.
- The functions `getruntime`, `getcurrenttime`, `getpreciseticks` and `getpreciseseconds` return what their name suggests.
- On MS Windows the `getcodepage` function returns two numbers, one for the command handler and one for the graphical user interface.
- The name of the startup file is reported by `getstartupfile`.
- The Lua version is reported by `getversion`.
- The `lua.openfile` function can be used instead of `io.open`. On MS Windows it will convert the filename to a so called wide one which means that filenames in utf8 encoding will work ok. On the other hand, names given in the codepage won't.

Primitive codes

here follows a list with all primitives and their category is shown. When the engine starts up in ini mode all primitives get defined along with some properties that makes it possible to do a reverse lookup of a combination of command code and char code. But, a primitive, being also a regular command can be redefined later on. The table below shows the original pairs but in ConTEXt some of these primitives are redefined. However, any macro that fits a command and char pair is (reported as) a primitive in logs and error messages. In the end all tokens are such a combination, The first 16 command codes are reserved for characters (the whole Unicode range can be used as char code) with specific catcodes and not mentioned in the list.

PRIMITIVE	COMMAND NAME	CMD	CHR	ORIGIN
\	explicit_space	72	0	tex
\-	discretionary	55	1	tex
\/	italic_correction	52	0	tex
\UUskewed	math_fraction	60	7	luatex
\UUskewedwithdelims	math_fraction	60	15	luatex
\Uabove	math_fraction	60	4	luatex
\Uabovewithdelims	math_fraction	60	12	luatex
\Uatop	math_fraction	60	6	luatex
\Uatopwithdelims	math_fraction	60	14	luatex
\Uchar	convert	128	15	luatex
\Udelcode	define_char_code	100	9	luatex
\Udelcodenum	define_char_code	100	10	luatex
\Udelimiter	delimiter_number	24	1	luatex
\Udelimiterover	math_radical	74	6	luatex
\Udelimiterunder	math_radical	74	5	luatex
\Uhextensible	math_radical	74	7	luatex
\Uleft	math_fence	57	5	luatex
\Umathaccent	math_accent	54	1	luatex
\Umathaccentbaseheight	set_math_parameter	102	2	luatex
\Umathaccentvariant	set_math_parameter	102	131	luatex
\Umathadapttoleft	math_modifier	59	3	luatex
\Umathadapttoright	math_modifier	59	4	luatex
\Umathaxis	set_math_parameter	102	1	luatex
\Umathbinbinspacing	set_math_parameter	102	74	luatex
\Umathbinclosespacing	set_math_parameter	102	77	luatex
\Umathbininnerspacing	set_math_parameter	102	79	luatex
\Umathbinopenspacing	set_math_parameter	102	76	luatex
\Umathbinopspacing	set_math_parameter	102	73	luatex
\Umathbinordspacing	set_math_parameter	102	72	luatex
\Umathbinpunctspacing	set_math_parameter	102	78	luatex
\Umathbinrelspacing	set_math_parameter	102	75	luatex
\Umathbotaccentvariant	set_math_parameter	102	133	luatex
\Umathchar	math_char_number	26	1	luatex

\Umathcharclass	some_item	81	27	luatex
\Umathchardef	shorthand_def	114	2	luatex
\Umathcharfam	some_item	81	28	luatex
\Umathcharnum	math_char_number	26	2	luatex
\Umathcharnumdef	shorthand_def	114	3	luatex
\Umathcharslot	some_item	81	29	luatex
\Umathclass	math_char_number	26	3	luatex
\Umathclosebinspacing	set_math_parameter	102	98	luatex
\Umathcloseclosespacing	set_math_parameter	102	101	luatex
\Umathcloseinnerspacing	set_math_parameter	102	103	luatex
\Umathcloseopenspacing	set_math_parameter	102	100	luatex
\Umathcloseopspacing	set_math_parameter	102	97	luatex
\Umathcloseordspacing	set_math_parameter	102	96	luatex
\Umathclosepunctspacing	set_math_parameter	102	102	luatex
\Umathcloserelspacing	set_math_parameter	102	99	luatex
\Umathcode	define_char_code	100	6	luatex
\Umathcodenum	define_char_code	100	7	luatex
\Umathconnectoroverlapmin	set_math_parameter	102	51	luatex
\Umathdegreevariant	set_math_parameter	102	130	luatex
\Umathdelimiterovervariant	set_math_parameter	102	124	luatex
\Umathdelimiterundervariant	set_math_parameter	102	125	luatex
\Umathdenominatorvariant	set_math_parameter	102	136	luatex
\Umathextrasubpreshift	set_math_parameter	102	55	luatex
\Umathextrasubshift	set_math_parameter	102	53	luatex
\Umathextrasuppreshift	set_math_parameter	102	54	luatex
\Umathextrasupshift	set_math_parameter	102	52	luatex
\Umathfractiondelsize	set_math_parameter	102	25	luatex
\Umathfractiondenomdown	set_math_parameter	102	24	luatex
\Umathfractiondenomvgap	set_math_parameter	102	23	luatex
\Umathfractionnumup	set_math_parameter	102	22	luatex
\Umathfractionnumvgap	set_math_parameter	102	21	luatex
\Umathfractionrule	set_math_parameter	102	20	luatex
\Umathfractionvariant	set_math_parameter	102	128	luatex
\Umathhextensiblevariant	set_math_parameter	102	126	luatex
\Umathinnerbinspacing	set_math_parameter	102	114	luatex
\Umathinnerclosespacing	set_math_parameter	102	117	luatex
\Umathinnerinnerspacing	set_math_parameter	102	119	luatex
\Umathinneropenspacing	set_math_parameter	102	116	luatex
\Umathinneropspacing	set_math_parameter	102	113	luatex
\Umathinnerordspacing	set_math_parameter	102	112	luatex
\Umathinnerpunctspacing	set_math_parameter	102	118	luatex
\Umathinnerrelspacing	set_math_parameter	102	115	luatex
\Umathlimitabovebgap	set_math_parameter	102	29	luatex
\Umathlimitabovekern	set_math_parameter	102	30	luatex
\Umathlimitabovevgap	set_math_parameter	102	28	luatex
\Umathlimitbelowbgap	set_math_parameter	102	32	luatex

```
\Umathlimitbelowkern         set_math_parameter  102   33  luatex
\Umathlimitbelowvgap         set_math_parameter  102   31  luatex
\Umathlimits                 math_modifier        59    1  luatex
\Umathnoaxis                 math_modifier        59    6  luatex
\Umathnolimits               math_modifier        59    2  luatex
\Umathnolimitsubfactor       set_math_parameter  102   34  luatex
\Umathnolimitsupfactor       set_math_parameter  102   35  luatex
\Umathnumeratorvariant       set_math_parameter  102  135  luatex
\Umathopbinspacing           set_math_parameter  102   66  luatex
\Umathopclosespacing         set_math_parameter  102   69  luatex
\Umathopenbinspacing         set_math_parameter  102   90  luatex
\Umathopenclosespacing       set_math_parameter  102   93  luatex
\Umathopeninnerspacing       set_math_parameter  102   95  luatex
\Umathopenopenspacing        set_math_parameter  102   92  luatex
\Umathopenopspacing          set_math_parameter  102   89  luatex
\Umathopenordspacing         set_math_parameter  102   88  luatex
\Umathopenpunctspacing       set_math_parameter  102   94  luatex
\Umathopenrelspacing         set_math_parameter  102   91  luatex
\Umathopenupdepth            math_modifier        59   10  luatex
\Umathopenupheight           math_modifier        59    9  luatex
\Umathoperatorsize           set_math_parameter  102    4  luatex
\Umathopinnerspacing         set_math_parameter  102   71  luatex
\Umathopopenspacing          set_math_parameter  102   68  luatex
\Umathopopspacing            set_math_parameter  102   65  luatex
\Umathopordspacing           set_math_parameter  102   64  luatex
\Umathoppunctspacing         set_math_parameter  102   70  luatex
\Umathoprelspacing           set_math_parameter  102   67  luatex
\Umathordbinspacing          set_math_parameter  102   58  luatex
\Umathordclosespacing        set_math_parameter  102   61  luatex
\Umathordinnerspacing        set_math_parameter  102   63  luatex
\Umathordopenspacing         set_math_parameter  102   60  luatex
\Umathordopspacing           set_math_parameter  102   57  luatex
\Umathordordspacing          set_math_parameter  102   56  luatex
\Umathordpunctspacing        set_math_parameter  102   62  luatex
\Umathordrelspacing          set_math_parameter  102   59  luatex
\Umathoverbarkern            set_math_parameter  102    5  luatex
\Umathoverbarrule            set_math_parameter  102    6  luatex
\Umathoverbarvgap            set_math_parameter  102    7  luatex
\Umathoverdelimiterbgap      set_math_parameter  102   39  luatex
\Umathoverdelimitervariant   set_math_parameter  102  122  luatex
\Umathoverdelimitervgap      set_math_parameter  102   38  luatex
\Umathoverlayaccentvariant   set_math_parameter  102  134  luatex
\Umathoverlinevariant        set_math_parameter  102  120  luatex
\Umathphantom                math_modifier        59    7  luatex
\Umathpunctbinspacing        set_math_parameter  102  106  luatex
\Umathpunctclosespacing      set_math_parameter  102  109  luatex
```

```
\Umathpunctinnerspacing      set_math_parameter  102  111  luatex
\Umathpunctopenspacing       set_math_parameter  102  108  luatex
\Umathpunctopspacing         set_math_parameter  102  105  luatex
\Umathpunctordspacing        set_math_parameter  102  104  luatex
\Umathpunctpunctspacing      set_math_parameter  102  110  luatex
\Umathpunctrelspacing        set_math_parameter  102  107  luatex
\Umathquad                   set_math_parameter  102    0  luatex
\Umathradicaldegreeafter     set_math_parameter  102   15  luatex
\Umathradicaldegreebefore    set_math_parameter  102   14  luatex
\Umathradicaldegreeraise     set_math_parameter  102   16  luatex
\Umathradicalkern            set_math_parameter  102   11  luatex
\Umathradicalrule            set_math_parameter  102   12  luatex
\Umathradicalvariant         set_math_parameter  102  129  luatex
\Umathradicalvgap            set_math_parameter  102   13  luatex
\Umathrelbinspacing          set_math_parameter  102   82  luatex
\Umathrelclosespacing        set_math_parameter  102   85  luatex
\Umathrelinnerspacing        set_math_parameter  102   87  luatex
\Umathrelopenspacing         set_math_parameter  102   84  luatex
\Umathrelopspacing           set_math_parameter  102   81  luatex
\Umathrelordspacing          set_math_parameter  102   80  luatex
\Umathrelpunctspacing        set_math_parameter  102   86  luatex
\Umathrelrelspacing          set_math_parameter  102   83  luatex
\Umathskewedfractionhgap     set_math_parameter  102   26  luatex
\Umathskewedfractionvgap     set_math_parameter  102   27  luatex
\Umathspaceafterscript       set_math_parameter  102   50  luatex
\Umathspacebeforescript      set_math_parameter  102   49  luatex
\Umathspacingmode            set_math_parameter  102    3  luatex
\Umathstackdenomdown         set_math_parameter  102   19  luatex
\Umathstacknumup             set_math_parameter  102   18  luatex
\Umathstackvariant           set_math_parameter  102  139  luatex
\Umathstackvgap              set_math_parameter  102   17  luatex
\Umathsubscriptvariant       set_math_parameter  102  138  luatex
\Umathsubshiftdown           set_math_parameter  102   42  luatex
\Umathsubshiftdrop           set_math_parameter  102   40  luatex
\Umathsubsupshiftdown        set_math_parameter  102   43  luatex
\Umathsubsupvgap             set_math_parameter  102   48  luatex
\Umathsubtopmax              set_math_parameter  102   44  luatex
\Umathsupbottommin           set_math_parameter  102   46  luatex
\Umathsuperscriptvariant     set_math_parameter  102  137  luatex
\Umathsupshiftdrop           set_math_parameter  102   41  luatex
\Umathsupshiftup             set_math_parameter  102   45  luatex
\Umathsupsubbottommax        set_math_parameter  102   47  luatex
\Umathtopaccentvariant       set_math_parameter  102  132  luatex
\Umathunderbarkern           set_math_parameter  102    8  luatex
\Umathunderbarrule           set_math_parameter  102    9  luatex
\Umathunderbarvgap           set_math_parameter  102   10  luatex
```

\Umathunderdelimiterbgap	set_math_parameter	102	37	luatex
\Umathunderdelimitervariant	set_math_parameter	102	123	luatex
\Umathunderdelimitervgap	set_math_parameter	102	36	luatex
\Umathunderlinevariant	set_math_parameter	102	121	luatex
\Umathvextensiblevariant	set_math_parameter	102	127	luatex
\Umathvoid	math_modifier	59	8	luatex
\Umiddle	math_fence	57	6	luatex
\Unosubprescript	math_script	75	7	luatex
\Unosubscript	math_script	75	5	luatex
\Unosuperprescript	math_script	75	8	luatex
\Unosuperscript	math_script	75	6	luatex
\Uover	math_fraction	60	5	luatex
\Uoverdelimiter	math_radical	74	4	luatex
\Uoverwithdelims	math_fraction	60	13	luatex
\Uradical	math_radical	74	1	luatex
\Uright	math_fence	57	7	luatex
\Uroot	math_radical	74	2	luatex
\Uskewed	math_fraction	60	3	luatex
\Uskewedwithdelims	math_fraction	60	11	luatex
\Ustack	math_choice	62	1	luatex
\Ustartdisplaymath	math_shift_cs	76	2	luatex
\Ustartmath	math_shift_cs	76	0	luatex
\Ustopdisplaymath	math_shift_cs	76	3	luatex
\Ustopmath	math_shift_cs	76	1	luatex
\Ustyle	math_style	61	8	luatex
\Usubprescript	math_script	75	4	luatex
\Usubscript	math_script	75	1	luatex
\Usuperprescript	math_script	75	3	luatex
\Usuperscript	math_script	75	2	luatex
\Uunderdelimiter	math_radical	74	3	luatex
\Uvextensible	math_fence	57	4	luatex
\above	math_fraction	60	0	tex
\abovedisplayshortskip	internal_glue	90	5	tex
\abovedisplayskip	internal_glue	90	3	tex
\abovewithdelims	math_fraction	60	8	tex
\accent	accent	53	0	tex
\adjdemerits	internal_int	84	16	tex
\adjustspacing	internal_int	84	85	luatex
\adjustspacingshrink	internal_int	84	88	luatex
\adjustspacingstep	internal_int	84	86	luatex
\adjustspacingstretch	internal_int	84	87	luatex
\advance	arithmic	111	0	tex
\afterassigned	after_something	49	4	luatex
\afterassignment	after_something	49	1	tex
\aftergroup	after_something	49	0	tex
\aftergrouped	after_something	49	3	luatex

\aliased	prefix	112	11	luatex
\aligncontent	end_template	18	3	tex
\alignmark	parameter	6	0	luatex
\aligntab	alignment_tab	4	0	luatex
\atendofgroup	after_something	49	2	luatex
\atendofgrouped	after_something	49	5	luatex
\atop	math_fraction	60	2	tex
\atopwithdelims	math_fraction	60	10	tex
\attribute	register	109	1	luatex
\attributedef	shorthand_def	114	5	luatex
\automaticdiscretionary	discretionary	55	2	luatex
\automatichyphenpenalty	internal_int	84	114	luatex
\automigrationmode	internal_int	84	118	luatex
\autoparagraphmode	internal_int	84	126	luatex
\badness	some_item	81	6	tex
\baselineskip	internal_glue	90	1	tex
\batchmode	set_interaction	118	0	tex
\begincsname	cs_name	127	2	luatex
\begingroup	begin_group	70	0	tex
\beginlocalcontrol	begin_local	125	0	luatex
\beginsimplegroup	begin_group	70	1	tex
\belowdisplayshortskip	internal_glue	90	6	tex
\belowdisplayskip	internal_glue	90	4	tex
\binoppenalty	internal_int	84	9	tex
\botmark	get_mark	130	8	tex
\botmarks	get_mark	130	3	etex
\boundary	boundary	73	1	luatex
\box	make_box	30	0	tex
\boxattribute	set_box_property	98	10	luatex
\boxdirection	set_box_property	98	3	luatex
\boxmaxdepth	internal_dimen	88	7	tex
\boxorientation	set_box_property	98	4	luatex
\boxtotal	set_box_property	98	9	luatex
\boxxmove	set_box_property	98	7	luatex
\boxxoffset	set_box_property	98	5	luatex
\boxymove	set_box_property	98	8	luatex
\boxyoffset	set_box_property	98	6	luatex
\brokenpenalty	internal_int	84	8	tex
\catcode	define_char_code	100	0	tex
\catcodetable	internal_int	84	82	luatex
\char	char_number	25	0	tex
\chardef	shorthand_def	114	0	tex
\cleaders	special_box	41	4	tex
\clearmarks	set_mark	27	2	luatex
\clubpenalties	set_specification	99	0	etex
\clubpenalty	internal_int	84	5	tex

\copy	make_box	30	1	tex
\count	register	109	0	tex
\countdef	shorthand_def	114	4	tex
\cr	end_template	18	5	tex
\crampeddisplaystyle	math_style	61	1	luatex
\crampedscriptscriptstyle	math_style	61	7	luatex
\crampedscriptstyle	math_style	61	5	luatex
\crampedtextstyle	math_style	61	3	luatex
\crcr	end_template	18	6	tex
\csname	cs_name	127	0	tex
\csstring	convert	128	10	luatex
\currentgrouplevel	some_item	81	10	etex
\currentgrouptype	some_item	81	11	etex
\currentifbranch	some_item	81	14	etex
\currentiflevel	some_item	81	12	etex
\currentiftype	some_item	81	13	etex
\currentmarks	get_mark	130	0	luatex
\day	internal_int	84	20	tex
\deadcycles	set_page_property	97	9	tex
\def	def	115	1	tex
\defaulthyphenchar	internal_int	84	58	tex
\defaultskewchar	internal_int	84	59	tex
\defcsname	def	115	5	luatex
\delcode	define_char_code	100	8	tex
\delimiter	delimiter_number	24	0	tex
\delimiterfactor	internal_int	84	17	tex
\delimitershortfall	internal_dimen	88	10	tex
\detokenize	the	129	2	etex
\dimen	register	109	2	tex
\dimendef	shorthand_def	114	6	tex
\dimensiondef	shorthand_def	114	12	luatex
\dimexpr	some_item	81	43	etex
\dimexpression	some_item	81	47	luatex
\directlua	convert	128	4	luatex
\discretionary	discretionary	55	0	tex
\displayindent	internal_dimen	88	15	tex
\displaylimits	math_modifier	59	0	tex
\displaystyle	math_style	61	0	tex
\displaywidowpenalties	set_specification	99	0	etex
\displaywidowpenalty	internal_int	84	7	tex
\displaywidth	internal_dimen	88	14	tex
\divide	arithmic	111	2	tex
\doublehyphendemerits	internal_int	84	14	tex
\dp	set_box_property	98	2	tex
\dump	end_job	23	1	tex
\edef	def	115	0	tex

\edefcsname	def	115	4	luatex
\efcode	set_font_property	95	4	luatex
\else	if_test	126	3	tex
\emergencystretch	internal_dimen	88	18	tex
\end	end_job	23	0	tex
\endcsname	end_cs_name	77	0	tex
\endgroup	end_group	71	0	tex
\endinput	input	122	1	tex
\endlinechar	internal_int	84	60	tex
\endlocalcontrol	end_local	67	0	luatex
\endsimplegroup	end_group	71	1	tex
\enforced	prefix	112	14	luatex
\eqno	equation_number	56	1	tex
\errhelp	internal_toks	82	9	tex
\errmessage	message	65	1	tex
\errorcontextlines	internal_int	84	68	tex
\errorstopmode	set_interaction	118	3	tex
\escapechar	internal_int	84	57	tex
\etoksapp	combine_toks	110	1	luatex
\etokspre	combine_toks	110	3	luatex
\everybeforepar	internal_toks	82	10	luatex
\everycr	internal_toks	82	7	tex
\everydisplay	internal_toks	82	3	tex
\everyeof	internal_toks	82	11	etex
\everyhbox	internal_toks	82	4	tex
\everyjob	internal_toks	82	6	tex
\everymath	internal_toks	82	2	tex
\everypar	internal_toks	82	1	tex
\everytab	internal_toks	82	8	luatex
\everyvbox	internal_toks	82	5	tex
\exceptionpenalty	internal_int	84	116	luatex
\exhyphenchar	internal_int	84	84	tex
\exhyphenpenalty	internal_int	84	4	tex
\expand	expand_after	120	9	luatex
\expandafter	expand_after	120	0	tex
\expandafterpars	expand_after	120	6	luatex
\expandafterspaces	expand_after	120	5	luatex
\expandcstoken	expand_after	120	8	luatex
\expanded	convert	128	7	luatex
\expandtoken	expand_after	120	7	luatex
\explicitdiscretionary	discretionary	55	1	luatex
\explicithyphenpenalty	internal_int	84	115	luatex
\fam	internal_int	84	56	tex
\fi	if_test	126	2	tex
\finalhyphendemerits	internal_int	84	15	tex
\firstmark	get_mark	130	7	tex

\firstmarks	get_mark	130	2	etex
\firstvalidlanguage	internal_int	84	113	luatex
\floatingpenalty	internal_int	84	54	tex
\flushmarks	set_mark	27	3	luatex
\font	define_font	104	0	tex
\fontchardp	some_item	81	20	etex
\fontcharht	some_item	81	19	etex
\fontcharic	some_item	81	21	etex
\fontcharwd	some_item	81	18	etex
\fontdimen	set_font_property	95	5	tex
\fontid	some_item	81	17	luatex
\fontmathcontrol	some_item	81	23	luatex
\fontname	convert	128	17	tex
\fontspecifiedname	convert	128	18	tex
\fontspecifiedsize	some_item	81	22	luatex
\fonttextcontrol	some_item	81	24	luatex
\formatname	convert	128	20	luatex
\frozen	prefix	112	0	luatex
\futurecsname	cs_name	127	3	luatex
\futuredef	let	113	3	luatex
\futureexpand	expand_after	120	2	luatex
\futureexpandis	expand_after	120	3	luatex
\futureexpandisap	expand_after	120	4	luatex
\futurelet	let	113	2	tex
\gdef	def	115	3	tex
\gdefcsname	def	115	7	luatex
\gleaders	special_box	41	6	luatex
\glet	let	113	0	luatex
\gletcsname	let	113	11	luatex
\glettonothing	let	113	13	luatex
\global	prefix	112	7	tex
\globaldefs	internal_int	84	55	tex
\glueexpr	some_item	81	44	etex
\glueshrink	some_item	81	39	etex
\glueshrinkorder	some_item	81	16	etex
\gluespecdef	shorthand_def	114	13	luatex
\gluestretch	some_item	81	38	etex
\gluestretchorder	some_item	81	15	etex
\gluetomu	some_item	81	41	etex
\glyph	char_number	25	1	tex
\glyphdatafield	internal_int	84	75	luatex
\glyphoptions	internal_int	84	78	luatex
\glyphscale	internal_int	84	72	luatex
\glyphscriptfield	internal_int	84	77	luatex
\glyphscriptscale	internal_int	84	80	luatex
\glyphscriptscriptscale	internal_int	84	81	luatex

\glyphstatefield	internal_int	84	76	luatex
\glyphtextscale	internal_int	84	79	luatex
\glyphxoffset	internal_dimen	88	19	luatex
\glyphxscale	internal_int	84	73	luatex
\glyphyoffset	internal_dimen	88	20	luatex
\glyphyscale	internal_int	84	74	luatex
\gtoksapp	combine_toks	110	4	luatex
\gtokspre	combine_toks	110	6	luatex
\halign	halign	42	0	tex
\hangafter	internal_int	84	53	tex
\hangindent	internal_dimen	88	17	tex
\hbadness	internal_int	84	26	tex
\hbox	make_box	30	10	tex
\hccode	define_char_code	100	4	luatex
\hfil	hskip	36	0	tex
\hfill	hskip	36	1	tex
\hfilneg	hskip	36	3	tex
\hfuzz	internal_dimen	88	8	tex
\hjcode	hyphenation	117	7	luatex
\holdinginserts	internal_int	84	67	tex
\hpack	make_box	30	7	luatex
\hrule	hrule	45	0	tex
\hsize	internal_dimen	88	3	tex
\hskip	hskip	36	4	tex
\hss	hskip	36	2	tex
\ht	set_box_property	98	1	tex
\hyphenation	hyphenation	117	0	tex
\hyphenationmin	hyphenation	117	6	luatex
\hyphenationmode	internal_int	84	64	luatex
\hyphenchar	set_font_property	95	0	tex
\hyphenpenalty	internal_int	84	3	tex
\if	if_test	126	7	tex
\ifabsdim	if_test	126	11	luatex
\ifabsnum	if_test	126	9	luatex
\ifarguments	if_test	126	46	luatex
\ifboolean	if_test	126	41	luatex
\ifcase	if_test	126	32	tex
\ifcat	if_test	126	8	tex
\ifchkdim	if_test	126	29	luatex
\ifchknum	if_test	126	26	luatex
\ifcmpdim	if_test	126	31	luatex
\ifcmpnum	if_test	126	28	luatex
\ifcondition	if_test	126	37	luatex
\ifcsname	if_test	126	34	etex
\ifcstok	if_test	126	22	luatex
\ifdefined	if_test	126	33	etex

\ifdim	if_test	126	12	tex
\ifdimexpression	if_test	126	43	luatex
\ifdimval	if_test	126	30	luatex
\ifempty	if_test	126	39	luatex
\iffalse	if_test	126	25	tex
\ifflags	if_test	126	38	luatex
\iffontchar	if_test	126	36	etex
\ifhaschar	if_test	126	52	luatex
\ifhastok	if_test	126	49	luatex
\ifhastoks	if_test	126	50	luatex
\ifhasxtoks	if_test	126	51	luatex
\ifhbox	if_test	126	19	tex
\ifhmode	if_test	126	15	tex
\ifincsname	if_test	126	35	luatex
\ifinner	if_test	126	17	tex
\ifinsert	if_test	126	53	luatex
\ifmathparameter	if_test	126	44	luatex
\ifmathstyle	if_test	126	45	luatex
\ifmmode	if_test	126	16	tex
\ifnum	if_test	126	10	tex
\ifnumexpression	if_test	126	42	luatex
\ifnumval	if_test	126	27	luatex
\ifodd	if_test	126	13	tex
\ifparameter	if_test	126	48	luatex
\ifparameters	if_test	126	47	luatex
\ifrelax	if_test	126	40	luatex
\iftok	if_test	126	21	luatex
\iftrue	if_test	126	24	tex
\ifvbox	if_test	126	20	tex
\ifvmode	if_test	126	14	tex
\ifvoid	if_test	126	18	tex
\ifx	if_test	126	23	tex
\ignorearguments	ignore_something	48	2	luatex
\ignorepars	ignore_something	48	1	luatex
\ignorespaces	ignore_something	48	0	tex
\immediate	prefix	112	12	luatex
\immutable	prefix	112	2	luatex
\indent	begin_paragraph	51	1	tex
\initcatcodetable	catcode_table	66	1	luatex
\input	input	122	0	tex
\inputlineno	some_item	81	5	tex
\insert	insert	46	0	tex
\insertbox	make_box	30	11	luatex
\insertcopy	make_box	30	12	luatex
\insertdepth	set_page_property	97	20	luatex
\insertdistance	set_page_property	97	13	luatex

\insertheight	set_page_property	97	19	luatex
\insertheights	set_page_property	97	11	luatex
\insertlimit	set_page_property	97	15	luatex
\insertmaxdepth	set_page_property	97	18	luatex
\insertmode	set_auxiliary	96	4	luatex
\insertmultiplier	set_page_property	97	14	luatex
\insertpenalties	set_page_property	97	10	tex
\insertpenalty	set_page_property	97	17	luatex
\insertprogress	some_item	81	32	luatex
\insertstorage	set_page_property	97	16	luatex
\insertstoring	set_page_property	97	12	luatex
\insertunbox	un_vbox	34	11	luatex
\insertuncopy	un_vbox	34	12	luatex
\insertwidth	set_page_property	97	21	luatex
\instance	prefix	112	5	luatex
\integerdef	shorthand_def	114	11	luatex
\interactionmode	set_auxiliary	96	3	etex
\interlinepenalties	set_specification	99	0	etex
\interlinepenalty	internal_int	84	13	tex
\jobname	convert	128	19	tex
\kern	kern	39	0	tex
\language	internal_int	84	62	tex
\lastarguments	some_item	81	30	luatex
\lastbox	make_box	30	3	tex
\lastchkdim	some_item	81	49	luatex
\lastchknum	some_item	81	48	luatex
\lastkern	some_item	81	1	tex
\lastlinefit	internal_int	84	91	etex
\lastnamedcs	cs_name	127	1	luatex
\lastnodesubtype	some_item	81	4	luatex
\lastnodetype	some_item	81	3	etex
\lastparcontext	some_item	81	51	luatex
\lastpenalty	some_item	81	0	tex
\lastskip	some_item	81	2	tex
\lccode	define_char_code	100	1	tex
\leaders	special_box	41	3	tex
\left	math_fence	57	1	tex
\lefthyphenmin	internal_int	84	65	tex
\leftmarginkern	some_item	81	33	luatex
\leftskip	internal_glue	90	7	tex
\leqno	equation_number	56	0	tex
\let	let	113	1	tex
\letcharcode	let	113	4	luatex
\letcsname	let	113	10	luatex
\letfrozen	let	113	8	luatex
\letprotected	let	113	6	luatex

\lettonothing	let	113	12	luatex
\limits	math_modifier	59	1	tex
\linedirection	internal_int	84	124	luatex
\linepenalty	internal_int	84	2	tex
\lineskip	internal_glue	90	0	tex
\lineskiplimit	internal_dimen	88	2	tex
\localbrokenpenalty	internal_int	84	70	luatex
\localcontrol	begin_local	125	1	luatex
\localcontrolled	begin_local	125	2	luatex
\localinterlinepenalty	internal_int	84	69	luatex
\localleftbox	special_box	41	1	luatex
\localrightbox	special_box	41	2	luatex
\long	prefix	112	16	tex
\looseness	internal_int	84	18	tex
\lower	vmove	32	0	tex
\lowercase	case_shift	64	0	tex
\lpcode	set_font_property	95	2	luatex
\luabytecode	convert	128	6	luatex
\luabytecodecall	lua_function_call	68	1	luatex
\luacopyinputnodes	internal_int	84	117	luatex
\luadef	shorthand_def	114	10	luatex
\luaescapestring	convert	128	16	luatex
\luafunction	convert	128	5	luatex
\luafunctioncall	lua_function_call	68	0	luatex
\luatexbanner	convert	128	21	luatex
\luatexrevision	some_item	81	9	luatex
\luatexversion	some_item	81	8	luatex
\mark	set_mark	27	0	tex
\marks	set_mark	27	1	etex
\mathaccent	math_accent	54	0	tex
\mathbin	math_component	58	2	tex
\mathchar	math_char_number	26	0	tex
\mathchardef	shorthand_def	114	1	tex
\mathchoice	math_choice	62	0	tex
\mathclose	math_component	58	5	tex
\mathcode	define_char_code	100	5	tex
\mathcontrolmode	internal_int	84	108	luatex
\mathdelimitersmode	internal_int	84	104	luatex
\mathdirection	internal_int	84	123	luatex
\mathdisplayskipmode	internal_int	84	95	luatex
\matheqnogapstep	internal_int	84	94	luatex
\mathflattenmode	internal_int	84	106	luatex
\mathfontcontrol	internal_int	84	109	luatex
\mathinner	math_component	58	7	tex
\mathitalicsmode	internal_int	84	102	luatex
\mathnolimitsmode	internal_int	84	99	luatex

\mathop	math_component	58	1	tex
\mathopen	math_component	58	4	tex
\mathord	math_component	58	0	tex
\mathpenaltiesmode	internal_int	84	103	luatex
\mathpunct	math_component	58	6	tex
\mathrel	math_component	58	3	tex
\mathrulesfam	internal_int	84	101	luatex
\mathrulesmode	internal_int	84	100	luatex
\mathrulethicknessmode	internal_int	84	105	luatex
\mathscale	some_item	81	25	luatex
\mathscriptboxmode	internal_int	84	97	luatex
\mathscriptcharmode	internal_int	84	98	luatex
\mathscriptsmode	internal_int	84	96	luatex
\mathstyle	some_item	81	26	luatex
\mathsurround	internal_dimen	88	1	tex
\mathsurroundmode	internal_int	84	107	luatex
\mathsurroundskip	internal_glue	90	16	luatex
\maxdeadcycles	internal_int	84	52	tex
\maxdepth	internal_dimen	88	5	tex
\meaning	convert	128	12	tex
\meaningfull	convert	128	13	tex
\meaningless	convert	128	14	tex
\medmuskip	internal_mu_glue	92	2	tex
\message	message	65	0	tex
\middle	math_fence	57	2	tex
\mkern	mkern	40	0	tex
\month	internal_int	84	21	tex
\moveleft	hmove	31	1	tex
\moveright	hmove	31	0	tex
\mskip	mskip	38	0	tex
\muexpr	some_item	81	45	etex
\mugluespecdef	shorthand_def	114	14	luatex
\multiply	arithmic	111	1	tex
\muskip	register	109	4	tex
\muskipdef	shorthand_def	114	8	tex
\mutable	prefix	112	3	luatex
\mutoglue	some_item	81	40	etex
\newlinechar	internal_int	84	61	tex
\noalign	end_template	18	4	tex
\noaligned	prefix	112	4	luatex
\noboundary	boundary	73	0	luatex
\noexpand	no_expand	121	0	tex
\nohrule	hrule	45	1	luatex
\noindent	begin_paragraph	51	0	tex
\nolimits	math_modifier	59	2	tex
\nonscript	math_script	75	0	tex

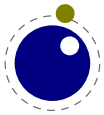
\nonstopmode	set_interaction	118	1	tex
\norelax	relax	16	1	luatex
\normalizelinemode	internal_int	84	119	luatex
\nospaces	internal_int	84	71	luatex
\novrule	vrule	44	1	luatex
\nulldelimiterspace	internal_dimen	88	11	tex
\nullfont	set_font	103	0	tex
\number	convert	128	0	tex
\numericscale	some_item	81	50	luatex
\numexpr	some_item	81	42	etex
\numexpression	some_item	81	46	luatex
\omit	end_template	18	2	tex
\or	if_test	126	4	tex
\orelse	if_test	126	5	luatex
\orunless	if_test	126	6	luatex
\outer	prefix	112	17	tex
\output	internal_toks	82	0	tex
\outputbox	internal_int	84	83	luatex
\outputpenalty	internal_int	84	51	tex
\over	math_fraction	60	1	tex
\overfullrule	internal_dimen	88	16	tex
\overline	math_component	58	9	tex
\overloaded	prefix	112	10	luatex
\overloadmode	internal_int	84	125	luatex
\overshoot	some_item	81	7	tex
\overwithdelims	math_fraction	60	9	tex
\pageboundary	boundary	73	4	luatex
\pagedepth	set_page_property	97	7	tex
\pagediscards	un_vbox	34	3	etex
\pagefilllstretch	set_page_property	97	5	tex
\pagefillstretch	set_page_property	97	4	tex
\pagefilstretch	set_page_property	97	3	tex
\pagegoal	set_page_property	97	0	tex
\pageshrink	set_page_property	97	6	tex
\pagestretch	set_page_property	97	2	tex
\pagetotal	set_page_property	97	1	tex
\pagevsize	set_page_property	97	8	luatex
\par	end_paragraph	22	0	tex
\parametercount	some_item	81	31	luatex
\parametermark	parameter	6	0	luatex
\parattribute	begin_paragraph	51	5	luatex
\pardirection	internal_int	84	121	luatex
\parfillleftskip	internal_glue	90	14	tex
\parfillskip	internal_glue	90	15	tex
\parindent	internal_dimen	88	0	tex
\parshape	set_specification	99	0	tex

\parshapedimen	some_item	81	37	etex
\parshapeindent	some_item	81	36	etex
\parshapelength	some_item	81	35	etex
\parskip	internal_glue	90	2	tex
\patterns	hyphenation	117	1	tex
\pausing	internal_int	84	28	tex
\penalty	penalty	50	0	tex
\permanent	prefix	112	1	luatex
\postdisplaypenalty	internal_int	84	12	tex
\postexhyphenchar	hyphenation	117	5	luatex
\posthyphenchar	hyphenation	117	3	luatex
\prebinoppenalty	internal_int	84	111	luatex
\predisplaydirection	internal_int	84	90	etex
\predisplaygapfactor	internal_int	84	110	luatex
\predisplaypenalty	internal_int	84	11	tex
\predisplaysize	internal_dimen	88	13	tex
\preexhyphenchar	hyphenation	117	4	luatex
\prehyphenchar	hyphenation	117	2	luatex
\prerelpenalty	internal_int	84	112	luatex
\pretolerance	internal_int	84	0	tex
\prevdepth	set_auxiliary	96	1	tex
\prevgraf	set_auxiliary	96	2	tex
\protected	prefix	112	9	etex
\protrudechars	internal_int	84	89	luatex
\protrusionboundary	boundary	73	2	luatex
\pxdimen	internal_dimen	88	21	luatex
\quitvmode	begin_paragraph	51	2	luatex
\radical	math_radical	74	0	tex
\raise	vmove	32	1	tex
\relax	relax	16	0	tex
\relpenalty	internal_int	84	10	tex
\retokenized	input	122	5	luatex
\right	math_fence	57	3	tex
\righthyphenmin	internal_int	84	66	tex
\rightmarginkern	some_item	81	34	luatex
\rightskip	internal_glue	90	8	tex
\romannumeral	convert	128	11	tex
\rpcode	set_font_property	95	3	luatex
\savecatcodetable	catcode_table	66	0	luatex
\savinghyphcodes	internal_int	84	93	etex
\savingvdiscards	internal_int	84	92	etex
\scaledfontdimen	set_font_property	95	6	tex
\scantextokens	input	122	3	luatex
\scantokens	input	122	2	etex
\scriptfont	define_family	101	1	tex
\scriptscriptfont	define_family	101	2	tex

\scriptscriptstyle	math_style	61	6	tex
\scriptspace	internal_dimen	88	12	tex
\scriptstyle	math_style	61	4	tex
\scrollmode	set_interaction	118	2	tex
\semiexpanded	convert	128	8	luatex
\semiprotected	prefix	112	13	luatex
\setbox	set_box	116	0	tex
\setfontid	internal_int	84	63	luatex
\setlanguage	internal_int	84	62	tex
\sfcode	define_char_code	100	3	tex
\shapingpenaltiesmode	internal_int	84	127	luatex
\shapingpenalty	internal_int	84	128	luatex
\shipout	special_box	41	0	tex
\show	xray	29	0	tex
\showbox	xray	29	1	tex
\showboxbreadth	internal_int	84	23	tex
\showboxdepth	internal_int	84	24	tex
\showgroups	xray	29	4	etex
\showifs	xray	29	6	etex
\showlists	xray	29	3	tex
\shownodedetails	internal_int	84	25	tex
\showthe	xray	29	2	tex
\showtokens	xray	29	5	etex
\skewchar	set_font_property	95	1	tex
\skip	register	109	3	tex
\skipdef	shorthand_def	114	7	tex
\snapshotpar	begin_paragraph	51	4	luatex
\spacefactor	set_auxiliary	96	0	tex
\spaceskip	internal_glue	90	12	tex
\span	end_template	18	1	tex
\splitbotmark	get_mark	130	10	tex
\splitbotmarks	get_mark	130	5	etex
\splitdiscards	un_vbox	34	4	etex
\splitfirstmark	get_mark	130	9	tex
\splitfirstmarks	get_mark	130	4	etex
\splitmaxdepth	internal_dimen	88	6	tex
\splittopskip	internal_glue	90	10	tex
\string	convert	128	9	tex
\supmarkmode	internal_int	84	120	luatex
\swapcsvalues	let	113	5	luatex
\tabsize	internal_dimen	88	0	luatex
\tabskip	internal_glue	90	11	tex
\textdirection	internal_int	84	122	luatex
\textfont	define_family	101	0	tex
\textstyle	math_style	61	2	tex
\the	the	129	0	tex

\thewithoutunit	the	129	1	luatex
\thickmuskip	internal_mu_glue	92	3	tex
\thinmuskip	internal_mu_glue	92	1	tex
\time	internal_int	84	19	tex
\todimension	convert	128	3	tex
\tointeger	convert	128	1	tex
\tokenized	input	122	4	luatex
\toks	register	109	5	tex
\toksapp	combine_toks	110	0	luatex
\toksdef	shorthand_def	114	9	tex
\tokspre	combine_toks	110	2	luatex
\tolerance	internal_int	84	1	tex
\tolerant	prefix	112	8	luatex
\topmark	get_mark	130	6	tex
\topmarks	get_mark	130	1	etex
\topskip	internal_glue	90	9	tex
\toscaled	convert	128	2	tex
\tpack	make_box	30	5	luatex
\tracingalignments	internal_int	84	45	luatex
\tracingassigns	internal_int	84	39	etex
\tracingcommands	internal_int	84	36	tex
\tracingexpressions	internal_int	84	49	luatex
\tracingfonts	internal_int	84	38	luatex
\tracinggroups	internal_int	84	40	etex
\tracinghyphenation	internal_int	84	48	luatex
\tracingifs	internal_int	84	41	etex
\tracinginserts	internal_int	84	46	luatex
\tracinglevels	internal_int	84	43	etex
\tracinglostchars	internal_int	84	35	tex
\tracingmacros	internal_int	84	30	tex
\tracingmarks	internal_int	84	47	luatex
\tracingmath	internal_int	84	42	luatex
\tracingnesting	internal_int	84	44	etex
\tracingonline	internal_int	84	29	tex
\tracingoutput	internal_int	84	34	tex
\tracingpages	internal_int	84	33	tex
\tracingparagraphs	internal_int	84	32	tex
\tracingrestores	internal_int	84	37	tex
\tracingstats	internal_int	84	31	tex
\uccode	define_char_code	100	2	tex
\uchyph	internal_int	84	50	tex
\undent	begin_paragraph	51	3	luatex
\underline	math_component	58	8	tex
\unexpanded	the	129	3	etex
\unhbox	un_hbox	33	0	tex
\unhcopy	un_hbox	33	1	tex

\unhpack	un_hbox	33	2	tex
\unkern	remove_item	35	0	tex
\unless	expand_after	120	1	etex
\unletfrozen	let	113	9	luatex
\unletprotected	let	113	7	luatex
\unpenalty	remove_item	35	1	tex
\unskip	remove_item	35	2	tex
\untraced	prefix	112	6	luatex
\unvbox	un_vbox	34	0	tex
\unvcopy	un_vbox	34	1	tex
\unvpack	un_vbox	34	2	tex
\uppercase	case_shift	64	1	tex
\vadjust	vadjust	47	0	tex
\valign	valign	43	0	tex
\vbadness	internal_int	84	27	tex
\vbox	make_box	30	9	tex
\vcenter	vcenter	63	0	tex
\vfil	vskip	37	0	tex
\vfill	vskip	37	1	tex
\vfilneg	vskip	37	3	tex
\vfuzz	internal_dimen	88	9	tex
\vpack	make_box	30	6	luatex
\vrule	vrule	44	0	tex
\vsize	internal_dimen	88	4	tex
\vskip	vskip	37	4	tex
\vsplit	make_box	30	4	tex
\vss	vskip	37	2	tex
\vtop	make_box	30	8	tex
\wd	set_box_property	98	0	tex
\widowpenalties	set_specification	99	0	etex
\widowpenalty	internal_int	84	6	tex
\wordboundary	boundary	73	3	luatex
\wrapuppar	begin_paragraph	51	6	luatex
\xdef	def	115	2	tex
\xdefcsname	def	115	6	luatex
\xleaders	special_box	41	5	tex
\xspaceskip	internal_glue	90	13	tex
\xtoksapp	combine_toks	110	5	luatex
\xtokspre	combine_toks	110	7	luatex
\year	internal_int	84	22	tex



Topics

a
Aleph *30, 66*
adjust *139*
attributes *45, 158, 200*

b
banner *40*
boundaries *84*
boundary *143*
boxes *13, 45, 203*
  split *204*
building *131*
bytecodes *187*

c
callbacks *175*
  building pages *177*
  contributions *177, 179*
  dump *183*
  errors *184*
  files *184*
  fonts *180, 185*
  format file *176*
  hyphenation *182*
  inserts *177*
  job run *183*
  jobname *176*
  kerning *182*
  ligature building *182*
  linebreaks *178, 179*
  log file *176*
  math *183*
  opening files *176*
  output *181*
  packing *180, 181*
  rules *181*
  warnings *184*
  whatsits *185*
  wrapping up *184*
catcodes *50*
characters *87*
  codes *202*
command line *35*
conditions *58*
  dimensions *56*
  numbers *56*
  tokens *57*
configuration *219*
convert commands *199*
csnames *31*

d
dimensions *56*
direct nodes *164*
directions *66, 144*
discretionaries *96, 98, 139*

e
ε-TEX *28*
engines *27*
errors *210*
escaping *48*
exceptions *94*
expansion *54*
  suppress *82*

f
files
  binary *31*
  names *65*
  writing *65*
fonts *66, 82*
  current *85*
  define *85*
  defining *212*
  extend *85*
  id *85*
  used *299*

g
getstartupfile *187*
getversion *187*
glue *140*
gluespec *141*
glyphs *87, 142*

h
hash *212*
helpers *209*
history *27*
hyphenation *63, 87, 92, 94*
  discretionaries *96*
  exceptions *94*
  how it works *96*
  patterns *94*
  tracing *92*

i
io *220*
images
  MetaPost *229*
  mplib *229*
initialization *35, 212*
insertions *138*
inserts *131*

k
kerning *97*
kerns *141*
  suppress *82*

l
Lua *13*
  extensions *37*
  interpreter *35*
  libraries *37*
  modules *37*
languages *63, 87*
  library *98*
last items *200*
leaders *63*
libraries
  `lua` *187*
  `status` *188*
  `tex` *197*
  `texconfig` *219*
  `texio` *220*
  `token` *221*
ligatures *97*
  suppress *82*
linebreaks *98, 217*
lists *137, 205*

m
MetaPost *229*
  mplib *229*
macros *225*
main loop *92*
marks *52, 131, 138*
math *66, 103*
  accents *120, 125*
  codes *125*
  cramped *106*
  delimiters *122, 125*
  extensibles *122*
  fences *119*
  flattening *127*
  fractions *123*
  italics *117*
  kerning *117*
  last line *126*
  limits *116*
  nodes *140, 144*
  parameters *108, 109, 204*
  penalties *119*
  radicals *121*
  scripts *117, 122, 127*
  spacing *106, 113, 114, 116*
  stacks *106*
  styles *105, 106, 126, 127*
  text *127*
  tracing *128*
  Unicode *103*
memory *31*

n
nesting *206, 219*
newline *32*
nodes *13, 44, 135*
  adjust *139*
  attributes *158*
  boundary *143*
  direct *164*
  direction *144*
  discretionaries *139*
  functions *151*
  glue *140, 141*
  glyph *142*
  insertions *138*

kerns *141*
lists *137*
marks *138*
math *140, 144*
paragraphs *143*
penalty *142*
properties *171*
rules *137*
text *136*
numbers *56*

o
Omega *66*
output *61*

p
pdf
analyze *241*
memory streams *245*
objects *241*
`pdfe` *241*
pdfTeX *29*
pages *131, 204, 219*
paragraphs *98, 131, 143*
reset *217*
parameters
internal *197*
math *204*
patterns *94*
penalty *142*
primitives *212*
printing *207*
properties *171*
protrusion *84*
suppress *82*

r
registers *200, 203*
bytecodes *187*
rules *61, 137*

s
shipout *219*
space *32*
spaces
suppress *84*
splitting *62*
synctex *219*

t
TeX *27*
tables *187*
testing *38*
text
math *127*
tokens *57, 221*
scanning *51*
tracing *64*

u
Unicode *41, 42*
math *103*

v
vcentering *45*
version *40*

w
web2c *30*

Primitives

This register contains the primitives that are mentioned in the manual. There are of course many more primitives. The ε-TEX and LuaTEX primitives are typeset in bold (some originate in pdfTEX).

\abovedisplayskip *116*
\abovewithdelims *123*
\accent *92*
\adjustspacing *29, 30, 77*
\adjustspacingshrink *29*
\adjustspacingstep *29*
\adjustspacingstretch *29*
\aftergrouped *54*
\aliased *72*
\aligncontent *53*
\alignmark *53*
\aligntab *53*
\atop *106, 108*
\atopwithdelims *106*
\attribute *200*
\attributedef *200*
\automatichyphenpenalty *92*
\automigrationmode *132*
\autoparagraphmode *131*
\batchmode *220*
\begincsname *52*
\begingroup *106*
\belowdisplayskip *116*
\botmark *132*
\boundary *64, 143*
\box *42*
\boxattribute *47*
\catcode *31, 39, 41, 202*
\catcodetable *50, 207*
\char *16, 41, 83, 92, 94, 142*
\chardef *41, 94, 225, 226*
\clearmarks *52, 132*
\clubpenalties *218*
\copy *42*
\count *37, 42, 45, 200*
\countdef *42, 200*
\crampedscriptstyle *107*
\csname *52, 54*
\csstring *52*
\currentiftype *65*
\currentmarks *52*
\defcsname *54*
\delcode *31, 103, 202, 203*
\delimiter *103*
\detokenize *222*
\dimen *37, 42, 200*
\dimendef *42, 200*
\dimensiondef *73*
\directlua *13, 39, 47, 48, 49, 207, 212, 213*
\discretionary *16, 92, 94, 96, 139*
\displaystyle *113*
\displaywidowpenalties *218*
\dp *42, 131*
\edef *49, 54, 222*
\edefcsname *54*
\efcode *29, 41, 76*
\endgroup *106*
\endinput *221*
\endlinechar *28, 51, 207, 209*
\enforced *73*
\errhelp *210*
\errmessage *210*
\etoksapp *51*
\etokspre *51*
\everybeforepar *131*
\everyeof *51*
\everyjob *36*
\exceptionpenalty *95*
\exhyphenchar *93*
\exhyphenpenalty *93, 96, 139*
\expandafter *54*
\expanded *29, 54*
\explicithyphenpenalty *92*
\firstmark *132*
\firstvalidlanguage *88*
\flushmarks *52, 132*
\fontdimen *83*

\fontid *82*
\fontmathcontrol *79*
\fonttextcontrol *79*
\formatname *213*
\frozen *72, 115*
\futureexpand *54*
\futureexpandis *54*
\futureexpandisap *54*
\gleaders *63*
\glet *53*
\glettonothing *53*
\global *72*
\glyph *83*
\glyphdatafield *86*
\glyphoptions *82*
\glyphscriptfield *86*
\glyphstatefield *86*
\glyphxoffset *83*
\glyphxscale *83*
\glyphyoffset *83*
\glyphyscale *83*
\gtoksapp *51*
\gtokspre *51*
\halign *53, 179*
\hbox *16, 45, 63, 117, 178, 179, 203*
\hjcode *31, 41, 88, 95*
\hpack *63*
\hrule *16, 61, 137*
\hskip *16, 140*
\ht *42, 131*
\hyphenation *94, 96, 97*
\hyphenationmin *63, 88*
\hyphenationmode *79, 90, 93, 98*
\hyphenchar *75, 93, 96*
\hyphenpenalty *96, 139*
\if *52*
\ifabsdim *29, 55*
\ifabsnum *29, 55*
\ifarguments *58*
\ifboolean *57*
\ifcase *57, 116*
\ifchkdim *56*
\ifchknum *56*
\ifcmpdim *56*
\ifcmpnum *56*
\ifcondition *58*
\ifcstok *57*
\ifdimval *56*
\ifempty *57*
\ifflags *60*
\ifincsname *29*
\ifmathparameter *57, 116*
\ifmathstyle *57*
\ifnumval *56*
\ifparameter *58*
\ifparameters *58*
\ifrelax *57*
\iftok *57*
\ignorepars *54*
\ignorespaces *54*
\immutable *71*
\initcatcodetable *50*
\insert *42, 138*
\insertdepth *131*
\insertdistance *131*
\insertheight *131*
\insertheights *131*
\insertlimit *131*
\insertmaxdepth *131*
\insertmode *131*
\insertmultiplier *131*
\insertpenalties *131*
\insertpenalty *131*
\insertstorage *131*
\insertstoring *131, 132*
\insertwidth *131*
\instance *72, 73*
\integerdef *73*
\interlinepenalties *218*
\jobname *36, 176*
\kern *16, 141*
\language *93, 94, 97, 99*
\lastchkdim *57*
\lastchknum *57*
\lastnamedcs *52*
\lastnodesubtype *65*
\lastnodetype *65*
\lccode *31, 41, 202*
\leaders *63*
\left *120*
\lefthyphenmin *63, 88*
\leftmarginkern *29*

\letcharcode *53*
\lettonothing *53*
\linedirection *68*
\localbrokenpenalty *143*
\localinterlinepenalty *143*
\localleftbox *144, 179*
\localrightbox *144, 178*
\lowercase *95*
\lpcode *29, 41, 76*
\luabytecode *49*
\luabytecodecall *49*
\luacopyinputnodes *208*
\luadef *49, 226*
\luaescapestring *48*
\luafunction *49*
\luafunctioncall *49*
\luatexbanner *40*
\luatexrevision *40, 41*
\luatexversion *40*
\mark *132, 138*
\marks *42, 52, 132, 162*
\mathaccent *103*
\mathchar *103, 127*
\mathchardef *103, 127*
\mathchoice *105*
\mathcode *31, 103, 202*
\mathcontrolmode *79*
\mathdelimitersmode *119*
\mathdirection *207*
\mathdisplayskipmode *116*
\matheqnogapstep *119*
\mathflattenmode *127, 128*
\mathitalicsmode *117, 119*
\mathnolimitsmode *116, 117*
\mathpenaltiesmode *119*
\mathscriptboxmode *117*
\mathscriptcharmode *118*
\mathscriptsmode *118*
\mathstyle *105, 106, 124, 207*
\mathsurround *113, 114, 140*
\mathsurroundmode *113*
\mathsurroundskip *113, 114*
\maxdepth *131, 180*
\meaning *72*
\meaningfull *72*
\meaningless *72*
\medmuskip *115*
\middle *206*
\muskip *42, 115, 200*
\muskipdef *42*
\mutable *71*
\newlinechar *28*
\noalign *53, 72*
\noaligned *72*
\noboundary *64, 93, 97, 143*
\noexpand *54*
\nohrule *61*
\normalizelinemode *68*
\nospaces *84*
\novrule *61*
\number *40, 41, 209*
\numericscale *70*
\orelse *59*
\orunless *59, 60*
\output *181*
\outputbox *61*
\over *106, 108, 206*
\overline *107*
\overloaded *73*
\overloadmode *71, 72, 197*
\overwithdelims *106*
\pageboundary *133*
\par *46, 54, 177*
\parattribute *47*
\parfillskip *178, 218*
\parindent *197*
\patterns *94, 96, 97*
\penalty *142*
\permanent *71*
\postexhyphenchar *96*
\posthyphenchar *96*
\predisplaygapfactor *126*
\preexhyphenchar *96*
\prehyphenchar *96*
\protrudechars *29, 30, 77*
\protrusionboundary *64, 84, 143*
\pxdimen *30*
\quitvmode *29*
\radical *103*
\relax *72, 94, 208, 213, 224*
\right *120*
\righthyphenmin *63, 88*

\rightmarginkern *29*
\romannumeral *105, 209*
\rpcode *29, 41, 76*
\savecatcodetable *51*
\savinghyphcodes *88, 89, 95, 101*
\scaledfontdimen *83*
\scantextokens *51*
\scantokens *48, 51*
\scriptfont *109*
\scriptscriptfont *109*
\scriptscriptstyle *121*
\scriptspace *113*
\scriptstyle *107*
\setbox *42*
\setfontid *82*
\setlanguage *88, 93, 97*
\sfcode *31, 41, 202*
\shapingpenaltiesmode *131*
\shapingpenaltymode *131*
\skewchar *75, 121*
\skip *42, 200, 201*
\skipdef *42, 200*
\snapshotpar *131*
\spaceskip *84*
\splitbotmark *132*
\splitfirstmark *132*
\string *52*
\tabsize *53*
\tabskip *53*
\textdirection *16, 67, 144, 207*
\textfont *109, 127*
\textstyle *105*
\the *40, 41, 45, 73, 197, 199, 201, 207*
\thickmuskip *115*
\thinmuskip *115*
\todimension *73*
\tointeger *73*
\tokenized *51*
\toks *42, 199, 200, 201, 207*
\toksapp *51*
\toksdef *42, 200*
\tokspre *51*
\tolerant *70, 72*
\topmark *132*
\toscaled *73*
\tpack *63*
\tracingalignments *53*
\tracingall *40*
\tracingassigns *29, 31*
\tracingcommands *93, 197*
\tracingfonts *30, 65*
\tracinghyphenation *93*
\tracinginserts *132*
\tracinglevels *64, 185*
\tracingmarks *52, 132*
\tracingnesting *211*
\tracingonline *64*
\tracingrestores *29, 31*
\Uabove *124*
\Uabovewithdelims *124*
\Uatop *124*
\Uatopwithdelims *124*
\Uchar *42*
\Udelcode *104, 203*
\Udelcodenum *104*
\Udelimiter *104*
\Udelimiterover *104, 122*
\Udelimiterunder *104, 122*
\Uhextensible *122, 123*
\Umathaccent *104, 120*
\Umathaxis *108*
\Umathbinbinspacing *114*
\Umathbinclosespacing *114*
\Umathbininnerspacing *114*
\Umathbinopenspacing *114*
\Umathbinopspacing *114*
\Umathbinordspacing *114*
\Umathbinpunctspacing *114*
\Umathbinrelspacing *114*
\Umathchar *104, 127*
\Umathcharclass *125*
\Umathchardef *104, 127*
\Umathcharfam *125*
\Umathcharnum *104*
\Umathcharnumdef *103, 104*
\Umathcharslot *125*
\Umathclosebinspacing *114*
\Umathcloseclosespacing *114*
\Umathcloseinnerspacing *114*
\Umathcloseopenspacing *114*
\Umathcloseopspacing *114*
\Umathcloseordspacing *114*

\Umathclosepunctspacing *114*
\Umathcloserelspacing *114*
\Umathcode *104, 125*
\Umathcodenum *104*
\Umathconnectoroverlapmin *109, 113*
\Umathfractiondelsize *108*
\Umathfractiondenomdown *108*
\Umathfractiondenomvgap *108*
\Umathfractionnumup *108*
\Umathfractionnumvgap *108*
\Umathfractionrule *108*
\Umathinnerbinspacing *115*
\Umathinnerclosespacing *115*
\Umathinnerinnerspacing *115*
\Umathinneropenspacing *115*
\Umathinneropspacing *115*
\Umathinnerordspacing *115*
\Umathinnerpunctspacing *115*
\Umathinnerrelspacing *115*
\Umathlimitabovebgap *109*
\Umathlimitabovekern *109, 112*
\Umathlimitabovevgap *108*
\Umathlimitbelowbgap *109*
\Umathlimitbelowkern *109, 112*
\Umathlimitbelowvgap *109*
\Umathnolimitsubfactor *116*
\Umathnolimitsupfactor *116*
\Umathopbinspacing *114*
\Umathopclosespacing *114*
\Umathopenbinspacing *114*
\Umathopenclosespacing *114*
\Umathopeninnerspacing *114*
\Umathopenopenspacing *114*
\Umathopenopspacing *114*
\Umathopenordspacing *114*
\Umathopenpunctspacing *114*
\Umathopenrelspacing *114*
\Umathoperatorsize *104, 108, 113*
\Umathopinnerspacing *114*
\Umathopopenspacing *114*
\Umathopopspacing *114*
\Umathopordspacing *114*
\Umathoppunctspacing *114*
\Umathoprelspacing *114*
\Umathordbinspacing *114*
\Umathordclosespacing *114*
\Umathordinnerspacing *114*
\Umathordopenspacing *114*
\Umathordopspacing *114*
\Umathordordspacing *114*
\Umathordpunctspacing *114*
\Umathordrelspacing *114*
\Umathoverbarkern *108*
\Umathoverbarrule *108*
\Umathoverbarvgap *108*
\Umathoverdelimiterbgap *109, 122*
\Umathoverdelimitervgap *109, 122*
\Umathpunctbinspacing *115*
\Umathpunctclosespacing *115*
\Umathpunctinnerspacing *115*
\Umathpunctopenspacing *115*
\Umathpunctopspacing *114*
\Umathpunctordspacing *114*
\Umathpunctpunctspacing *115*
\Umathpunctrelspacing *115*
\Umathquad *108, 112*
\Umathradicaldegreeafter *108, 113, 121*
\Umathradicaldegreebefore *108, 113, 121*
\Umathradicaldegreeraise *108, 113, 121*
\Umathradicalkern *108*
\Umathradicalrule *108, 112*
\Umathradicalvgap *108, 113*
\Umathrelbinspacing *114*
\Umathrelclosespacing *114*
\Umathrelinnerspacing *114*
\Umathrelopenspacing *114*
\Umathrelopspacing *114*
\Umathrelordspacing *114*
\Umathrelpunctspacing *114*
\Umathrelrelspacing *114*
\Umathskewedfractionhgap *123*
\Umathskewedfractionvgap *123*
\Umathspaceafterscript *109, 113*
\Umathspacebeforescript *109*
\Umathstackdenomdown *108*
\Umathstacknumup *108*
\Umathstackvgap *108*
\Umathsubshiftdown *109, 118*
\Umathsubshiftdrop *109*
\Umathsubsupshiftdown *109, 118*
\Umathsubsupvgap *109*
\Umathsubtopmax *109*

\Umathsupbottommin *109*
\Umathsupshiftdrop *109*
\Umathsupshiftup *109, 118*
\Umathsupsubbottommax *109*
\Umathunderbarkern *108*
\Umathunderbarrule *108*
\Umathunderbarvgap *108*
\Umathunderdelimiterbgap *109, 122*
\Umathunderdelimitervgap *109, 122*
\Umiddle *125*
\Unosubscript *127*
\Unosuperscript *127*
\Uover *124*
\Uoverdelimiter *104, 122*
\Uoverwithdelims *124*
\Uradical *104, 121*
\Uright *125*
\Uroot *104, 121, 147*
\Uskewed *123*
\Uskewedwithdelims *123*
\Ustack *106*
\Ustartdisplaymath *127*
\Ustartmath *127*
\Ustopdisplaymath *127*
\Ustopmath *127*
\Ustyle *124*
\Usubscript *126, 127*
\Usuperprescript *128, 129*
\Usuperscript *126, 127*
\UUskewed *124*
\UUskewedwithdelims *124*
\Uunderdelimiter *104, 122*
\uccode *31, 41, 202*
\uchyph *88, 92, 142*
\unexpanded *222*
\unhbox *42*
\unhcopy *42*
\unless *60*
\untraced *73*
\unvbox *42*
\unvcopy *42*
\uppercase *53, 95*
\vadjust *139, 177, 206*
\valign *53, 178*
\vbox *16, 45, 63, 179, 203, 218*
\vcenter *47, 63, 179*
\vpack *63*
\vrule *16, 61, 137*
\vskip *16, 140*
\vsplit *42, 62, 178, 204*
\vtop *16, 63, 179, 203*
\wd *42, 131*
\widowpenalties *218*
\wordboundary *64, 89, 143*
\wrapuppar *131*
\xdefcsname *54*
\xtoksapp *51*
\xtokspre *51*
\- *139*

Callbacks

b
buildpage_filter *177*
build_page_insert *177*

c
contribute_filter *177*

d
define_font *185*

f
find_format_file *176*
find_log_file *176*

g
glyph_run *180*

h
hpack_filter *178, 180*
hyphenate *182*

i
intercept_lua_error *184*
intercept_tex_error *184*

k
kerning *182*

l
ligaturing *182*
linebreak_filter *179, 218*

m
mlisttohlist *163*
mlist_to_hlist *119, 183*

o
open_data_file *176*

p
post_linebreak_filter *179*
pre_dump *183*
pre_linebreak_filter *178, 218*
process_jobname *176*
process_rule *181*

s
show_error_message *184*
show_warning_message *184*
show_whatsit *185*
start_file *184*
start_run *183*
stop_file *184*
stop_run *183*

v
vpack_filter *178, 180*

w
wrapup_run *184*

Nodes

This register contains the nodes that are known to LuaTEX. The primary nodes are in bold, whatsits that are determined by their subtype are normal.

a
accent *146*
adjust *90, 139*
attr *158*
attribute_list *158*

b
boundary *64, 90, 143*

c
choice *147*

d
delimiter *146*
delta *209*
dir *16, 90, 144*
disc *16, 44, 139, 155*

f
fence *148*
fraction *121, 148*

g
glue *16, 44, 90, 140, 155*
glue-spec *201*
glue_spec *140, 141, 197, 199, 201*
glyph *16, 44, 87, 88, 92, 142, 155*
glyphs *155*

h
hlist *16, 45, 46, 90, 137, 155*

i
insert *90, 138*

k
kern *16, 44, 90, 141*

m
mark *138*
math *140*
math_char *145*
math_text_char *145*

n
noad *146*

p
par *143, 218*
parameter *147*
penalty *90, 142*

r
radical *147*
rule *16, 90, 137*
rules *155*

s
style *146*
sub_box *145*
sub_mlist *145*

t
temp *136*

v
vlist *16, 45, 90, 137, 155*

w
whatsit *90*

290 Nodes

Libraries

This register contains the functions available in libraries. Not all functions are documented, for instance because they can be experimental or obsolete.

char_depth *236*
char_height *236*
char_width *236*
fields *233*
peninfo *236*
stacking *233*

callback
find *175*
known *175*
list *175*
register *175*

lang
clean *99*
clearhyphenation *99*
clearpatterns *99*
gethjcode *100*
hyphenate *100*
hyphenation *99*
hyphenationmin *100*
id *98*
new *98*
patterns *99*
postexhyphenchar *100*
posthyphenchar *100*
preexhyphenchar *100*
prehyphenchar *100*
sethjcode *100*

lua
bytecode *187*
callbytecode *187*
getbytecode *187*
getcurrenttime *188*
getpreciseseconds *188*
getpreciseticks *188*
getruntime *188*
getstacktop *188*
getstartupfile *187*
getversion *187*
newindex *187*
newtable *187*
setbytecode *187*

mplib
execute *232*
finish *232*
getcallbackstate *233*
gethashentries *233*
gethashentry *233*
getstates *232*
getstatus *232*
gettolerance *232*
new *229*
settolerance *232*
showcontext *232*
statistics *231*
version *229*

node
checkdiscretionaries *161*
checkdiscretionary *161*
copy *152, 165*
copylist *152, 165*
count *152*
currentattr *158*
currentattributes *166*
dimensions *162*
end_of_math *163*
fields *135, 150*
findattribute *159*
find_node *157*
firstglyph *160*
flattendiscretionaries *161*
flushlist *151, 166*
flushnode *151, 166*
free *151, 166*
getattribute *159, 166*
getfield *166*
getglue *157*
getpropertiestable *166, 171*
getproperty *167*
gettotal *167*
hasattribute *159, 167*

hasfield *150, 167*
hasglyph *160*
hpack *161*
id *150*
insertafter *153, 167*
insertbefore *153, 167*
isnode *167*
iszeroglue *157*
is_char *160*
is_glyph *160*
is_node *151*
kerning *160*
lastnode *153*
length *152*
ligaturing *160*
mlisttohlist *163*
new *151, 168*
prepend_prevdepth *162*
protectglyph *161*
protectglyphs *161*
protrusionskippable *161*
rangedimensions *162*
remove *153, 168*
setattribute *159, 168*
setfield *168*
setglue *157, 168*
setpropertiesmode *171*
setproperty *169*
slide *152*
subtypes *135*
tail *152, 169*
todirect *164*
tonode *164*
tostring *164, 169*
traverse *154, 169*
traverse_char *155, 169*
traverse_content *155, 169*
traverse_glyph *155, 169*
traverse_id *154, 169*
traverse_list *155, 169*
type *150, 169*
types *150*
unprotectglyph *160*
unprotectglyphs *160*
unsetattribute *159, 169*
values *135*
vpack *162*
write *152, 169*

node.direct
checkdiscretionaries *165*
checkdiscretionary *165*
copy *165*
copylist *165*
count *165*
currentattributes *166*
dimensions *166*
effectiveglue *166*
endofmath *166*
findattribute *166*
findattributerange *166*
findnode *166*
firstglyph *166*
flattendiscretionaries *166*
flushlist *166*
flushnode *166*
free *166*
getattribute *166*
getattributelist *166*
getattributes *166*
getboth *166*
getbox *166*
getchar *166*
getdata *166*
getdepth *166*
getdirection *166*
getdisc *166*
getexpansion *166*
getfam *166*
getfield *166*
getfont *166*
getglue *166*
getglyphdata *166*
getglyphdimensions *166*
getglyphscript *166*
getglyphstate *166*
getheight *166*
getid *166*
getindex *166*
getkern *166*
getkerndimension *166*
getlanguage *166*
getleader *166*

getlist *166*
getnext *166*
getnormalizedline *166*
getnucleus *166*
getoffsets *166*
getoptions *166*
getorientation *167*
getparstate *167*
getpenalty *167*
getpost *167*
getpre *167*
getprev *167*
getpropertiestable *166*
getproperty *167*
getreplace *167*
getscales *167*
getscript *167*
getshift *167*
getstate *167*
getsub *167*
getsubpre *167*
getsubtype *167*
getsup *167*
getsuppre *167*
getsynctexfields *166*
gettotal *167*
getwhd *167*
getwidth *167*
getxscale *167*
getxyscale *167*
getyscale *167*
hasattribute *167*
hasdimensions *167*
hasfield *167*
hasglyph *167*
hasglyphoption *167*
hpack *167*
hyphenating *167*
ignoremathskip *167*
insertafter *167*
insertbefore *167*
ischar *167*
isdirect *167*
isglyph *167*
isnextchar *167*
isnextglyph *167*
isnode *167*
isprevchar *167*
isprevglyph *167*
isvalid *167*
iszeroglue *167*
kerning *167*
lastnode *167*
length *167*
ligaturing *168*
makeextensible *168*
migrate *168*
mlisttohlist *168*
naturalwidth *168*
new *168*
protectglyph *168*
protectglyphs *168*
protrusionskippable *168*
rangedimensions *168*
remove *168*
setattribute *168*
setattributelist *168*
setattributes *168*
setboth *168*
setbox *168*
setchar *168*
setdata *168*
setdepth *168*
setdirection *168*
setdisc *168*
setexpansion *168*
setfam *168*
setfield *168*
setfont *168*
setglue *168*
setglyphdata *168*
setglyphscript *168*
setglyphstate *168*
setheight *168*
setindex *168*
setkern *168*
setlanguage *168*
setleader *168*
setlink *168*
setlist *168*
setnext *168*
setnucleus *168*

setoffsets *168*
setoptions *168*
setorientation *168*
setpenalty *168*
setpost *168*
setpre *168*
setprev *168*
setproperty *169*
setreplace *169*
setscales *169*
setscript *169*
setshift *169*
setsplit *169*
setstate *169*
setsub *169*
setsubpre *169*
setsubtype *169*
setsup *169*
setsuppre *169*
setsynctexfields *168*
setwhd *169*
setwidth *169*
slide *169*
startofpar *169*
tail *169*
todirect *169*
tonode *169*
total *169*
tovaliddirect *169*
traverse *169*
traverse_char *169*
traverse_content *169*
traverse_glyph *169*
traverse_id *169*
traverse_list *169*
unprotectglyph *169*
unprotectglyphs *169*
unsetattribute *169*
unsetattributes *169*
usedlist *169*
usesfont *169*
verticalbreak *169*
vpack *169*
write *169*

os

env *254*
gettimeofday *254*
name *254*
selfarg *254*
selfdir *254*
setenv *254*
type *254*
uname *254*

pdfe

arraytotable *244*
close *241*
closestream *243*
dictionarytotable *244*
getarray *243*
getboolean *243*
getbox *242*
getcatalog *242*
getdictionary *243*
getfromarray *243, 244*
getfromdictionary *243, 244*
getfromreference *245*
getfromstream *243*
getinfo *242*
getinteger *243*
getname *243*
getnofobjects *242*
getnofpages *242*
getnumber *243*
getpage *242*
getsize *242*
getstatus *241*
getstream *243*
getstring *243*
gettrailer *242*
getversion *242*
new *241, 245*
open *241*
openstream *243*
readfromstream *243*
readfromwholestream *243*
unencrypt *241*

sio

getposition *247*
readbytes *247*
readbytetable *247*
readcardinaltable *247*
readcardinal1 *247*

readcardinal2 *247*
readcardinal3 *247*
readcardinal4 *247*
readfixed2 *247*
readfixed4 *247*
readintegertable *247*
readinteger1 *247*
readinteger2 *247*
readinteger3 *247*
readinteger4 *247*
read2dot14 *247*
setposition *247*
skipposition *247*
status
list *188*
resetmessages *188*
setexitcode *188*
string
bytepairs *253*
bytes *253*
characterpairs *253*
characters *253*
explode *253*
utfcharacter *254*
utfcharacters *253*
utflength *254*
utfvalue *254*
utfvalues *253*
tex
attribute *200*
badness *217*
box *200, 203*
catcode *202*
count *200*
cprint *208*
definefont *212*
delcode *202*
dimen *200*
enableprimitives *212*
error *210*
extraprimitives *213*
fontidentifier *210*
fontname *210*
forcehmode *212*
forcesynctexline *219*
forcesynctextag *219*
get *197*
getattribute *200*
getbox *200, 203*
getcatcode *202*
getcount *200*
getdelcode *202*
getdelcodes *202*
getdimen *200*
getfamilyoffont *211*
getglue *200*
gethelptext *210*
getinteraction *211*
getlccode *202*
getlinenumber *210*
getlist *205*
getlocallevel *219*
getmark *200*
getmath *204*
getmathcode *202*
getmathcodes *202*
getmuglue *200*
getmuskip *200*
getnest *206*
getpagestate *219*
getsfcode *202*
getskip *200*
getsynctexline *219*
getsynctexmode *219*
getsynctextag *219*
gettoks *200*
getuccode *202*
glue *200*
hashtokens *212*
isattribute *200*
isbox *200*
iscount *200*
isdimen *200*
isglue *200*
ismuglue *200*
ismuskip *200*
isskip *200*
istoks *200*
lccode *202*
linebreak *217*
lists *205*
mathcode *202*

muglue *200*
muskip *200*
nest *206*
number *209*
primitives *217*
print *207*
ptr *206*
resetparagraph *217*
romannumeral *209*
round *209*
scale *209*
scantoks *200*
set *197*
setattribute *200*
setbox *200, 203*
setcatcode *202*
setcount *200*
setdelcode *202*
setdelcodes *202*
setdimen *200*
setglue *200*
setinteraction *211*
setlccode *202*
setlinenumber *210*
setlist *205*
setmath *204*
setmathcode *202*
setmathcodes *202*
setmuglue *200*
setmuskip *200*
setsfcode *202*
setskip *200*
setsynctexline *219*
setsynctexmode *219*
setsynctexnofiles *219*
setsynctextag *219*
settoks *200*
setuccode *202*
sfcode *202*
shipout *219*
show_context *210*
skip *200*
sp *210*
splitbox *204*
sprint *207*
toks *200*
tprint *208*
triggerbuildpage *204*
uccode *202*
write *209*
texio
closeinput *221*
setescape *221*
write *220*
writenl *221*
writeselector *220*
writeselectornl *221*
token
biggest_char *224*
commands *224*
command_id *224*
create *224*
expand *224*
getactive *224*
getcmdname *224*
getcommand *224*
getcsname *224*
getexpandable *224*
getfrozen *224*
getfunctionstable *225*
getid *224*
getindex *224*
getmacro *225*
getmeaning *225*
getmode *224*
getprotected *224*
gettok *224*
getuser *224*
is_defined *224*
is_token *224*
new *224*
peeknext *224*
peeknextexpanded *224*
popmacro *225*
pushmacro *225*
putnext *226*
scanargument *221*
scancode *221*
scancsname *221*
scandimen *221*
scanfloat *221*
scanglue *221*

scanint *221*
scankeyword *221*
scankeywordcs *221*
scanlist *221*
scannext *224, 226*
scannextexpanded *224*
scanreal *221*
scanstring *221*
scantoken *224*
scantoks *221*
scanword *221*
setchar *225*
setlua *225*
setmacro *225*
skipnext *224*
skipnextexpanded *224*

Statistics

The following fonts are used in this document:

used	filesize	version	filename
2	988.684	5.000	cambmath.ttf
1	927.280	5.020	cambria.ttf
1	163.452	1.802	LucidaBrightMathOT-Demi.otf
1	348.296	1.802	LucidaBrightMathOT.otf
1	73.284	1.801	LucidaBrightOT.otf
2	733.500	1.958	latinmodern-math.otf
1	64.684	2.004	lmmono10-regular.otf
1	64.160	2.004	lmmonoltcond10-regular.otf
1	111.536	2.004	lmroman10-regular.otf
3	525.008	1.106	texgyredejavu-math.otf
2	601.220	1.632	texgyrepagella-math.otf
1	218.100	2.501	texgyrepagella-regular.otf
1	693.876	2.340	DejaVuSans-Bold.ttf
1	741.536	2.340	DejaVuSans.ttf
1	318.392	2.340	DejaVuSansMono-Bold.ttf
1	245.948	2.340	DejaVuSansMono-Oblique.ttf
1	335.068	2.340	DejaVuSansMono.ttf
2	345.364	2.340	DejaVuSerif-Bold.ttf
1	336.884	2.340	DejaVuSerif-BoldItalic.ttf
1	343.388	2.340	DejaVuSerif-Italic.ttf
1	367.260	2.340	DejaVuSerif.ttf
27	**8.546.920**		**21 files loaded**

Some remarks

Here I collect remarks that I'd like to make but that don't fit into the manual. Consider in a notebook.

remark: LuaMetaTEX development is mostly done by Hans Hagen and in adapting the macros to the new features Wolfgang Schuster, who knows the code inside–out is a instrumental. In the initial phase Alan Braslau, who love playing with the three languages did extensive testing and compiled for several platforms. Later Mojca Miklavec make sure all compiles well on the buildbot infrastructure. After the first release more users got involved in testing. Many thanks for their patience! The development also triggered upgrading of the wiki support infrastructure where Taco Hoekwater and Paul Mazaitis have teamed up. So, progress all around.

remark: When there are non-intrusive features that also make sense in LuaTEX, these will be applied in the experimental branch first, so that there is no interference with the stable release. However, given that in the meantime the code bases differs a lot, it is unlikely that much will trickle back. This is no real problem as there's not much demand for that anyway.

remark: Most ConTEXt users seem always willing to keep up with the latest versions which means that LMTX is tested well. We can therefore safely claim that end of 2019 the code has become quite stable, although after that in some areas there were substantial additions. There are no complaints about performance (on my 2013 laptop this manual compiles at 24.5 pps with LMTX versus 20.7 pps for the LuaTEX manual with MkIV). After updating some of the ConTEXt code to use recently added features by the end of 2020 I could do more than 25.5 pps and in 2021 at some point to measured some 29.1 pps (probably also due to some performance improvements in the MetaFun code) but don't expect spectacular bumps in performance (I need a new machine for that to happen). Probably no one notices it, but memory consumption stepwise got reduced too. And ... the binary is still below 3 MegaBytes on all platforms.

remark: I tried to only add features that are sort of generic and much relates to controlling and opening up the engine. That also means that there are extensions that (at least not now) are used in ConTEXt, simply because there are already mechanisms in place that work well. So, it's also about trying to be complete in order not have to add more later, which makes it possible to shift to larger interval between updates. That way local experiments are also better isolated from stable versions.

In that perspective arguments like “This got added because ConTEXt needs it.” or “That got done because features creep.” as well as “Because of such features ConTEXt performs better.” are merely distractions from the fact that we are dealing with a project that just wants to upgrade the machinery while making that effort fun to do. There has not been much community drive and demand for substantial extensions over the last decades, so it has to be the fun factor, right? And the ConTEXt community being willing to join the experiment makes it even more fun. Just keep that in mind.

remark: It's is kind of strange to run into arguments for not using LuaTEX or for what it is worth LuaMetaTEX. No one forces anyone to use TEX in the first place, also because often word processors or web based editing provides plenty of benefits. And no one forces a TEX users to use a specific engine. I bet that for most users pdfTEX suits well, especially when you only need TEX for relative simple publications and reports in English, using default styles that put constraints

on the user. Often the math is what matters there. Also, using XeTeX is quite okay because it ships with built in font handling (of course that also has disadvantages, just consider the fact that it changed over time). When you want scripting LuaTeX is fine. When you need specific cjk support there are specialized engines for that. The same is true for ConTeXt. You don't have to dislike it: just ignore it and don't waste time on barking against a tree. But when you use ConTeXt the Lua enhanced engines are what you use.

remark: Yes there are bugs but I always consider the 'many' in "There are many bugs." to be an indication of frustration. Given the number of extensions and experiment one can expect bugs. But if someone can only mention a few, of which some fit into the category of engine limitations, it's probably more about ego. Abusing a mechanism for what it's not meant to, stretching it to the limits, running into a border case, those are not really bugs, more missing features. A crash is a bug indeed but we can count those in a few digits. The same is true for something missing in the manual: myabe it has a simple reason and explanation.

We have a fast cycle of resolving issues on the ConTeXt list where user also test new functionality so that it can get improved. Complaints are also kind of puzzling because when we talk new features we're also talking of something that could not be done before. No one forces anyone to use experimental features. Yes, trying out something that is not perfect is no fun, but I clearly remember working around many limitations which is not always fun but can also be interesting. Just choose a better program if you don't like it, and definitely stick to the robust older engines!

As a warning: the tone in an email of a complaint or remark nowadays determines how high it ends up on the to-be-dealt-with list: pretty low. There are always more interesting things on top.

remark: Some extensions involve the way macro arguments are dealt with. Combined with the possibility to parse the input stream using Lua one can come up with solution that are hard (or maybe even impossible) otherwise. For me it meant throwing away nice (but often complex) solutions that evolved over decades. That can hurt, especially when you consider the time spent on it. But all this doesn't change the concept of TeX the macro language. When pondering some criticism, just wonder first why TeX attracts users, some of which like to write code.

I'm always puzzled by folks who complain about TeX as a language (the other part being the typesetter). Why use it if you don't like it? A macro language has its own characteristics so live with it. After years of writing TeX code it's this language that intrigues me. It's also a reason why MetaPost and Lua are embedded: they are different languages and depending on the task they might suit better. When Alan, Aditya, I and others are playing with MetaPost extensions using the new scanners and interfaces resulting from that we do just that. We could invent a new language, with lots of fruitless debate, with limitations, but in the end there's nothing wrong with MetaPost (coming from MetaFont).