Contents

Introduction 13

1 Preamble 17

2 Basic \TeX\ enhancements 19

2.1 Introduction 19

2.1.1 Primitive behaviour 19

2.1.2 Version information 19

2.2 UNICODE text support 20

2.2.1 Extended ranges 20

2.2.2 \texttt{\Uchar} 21

2.2.3 Extended tables 21

2.3 Attributes 21

2.3.1 Nodes 21

2.3.2 Attribute registers 22

2.3.3 Box attributes 22

2.4 LUA related primitives 23

2.4.1 \texttt{\directlua} 23

2.4.2 \texttt{\latelua} and \texttt{\lateluafunc} 25

2.4.3 \texttt{\luaescapestring} 25

2.4.4 \texttt{\luafunc}, \texttt{\luafunct} and \texttt{\luadef} 25

2.4.5 \texttt{\luabytecode} and \texttt{\luabytecodecall} 26

2.5 Catcode tables 27

2.5.1 Catcodes 27

2.5.2 \texttt{\catcodetable} 27

2.5.3 \texttt{\initcatcodetable} 27

2.5.4 \texttt{\savecatcodetable} 27

2.6 Suppressing errors 28

2.6.1 \texttt{\suppressfontnotfounderror} 28

2.6.2 \texttt{\suppresslongerror} 28

2.6.3 \texttt{\suppressifcsnameerror} 28

2.6.4 \texttt{\suppressoutererror} 28

2.6.5 \texttt{\suppressmathparerror} 28

2.6.6 \texttt{\suppressprimitiveerror} 29

2.7 Fonts 29

2.7.1 Font syntax 29

2.7.2 \texttt{\fontid} and \texttt{\setfontid} 29

2.7.3 \texttt{\nolig}s and \texttt{\nokern}s 29

2.7.4 \texttt{\nospaces} 30

2.8 Tokens, commands and strings 30

2.8.1 \texttt{\scantexttokens} 30

2.8.2 \texttt{\toksapp}, \texttt{\tokspre}, \texttt{\etoksapp}, \texttt{\etokspre}, \texttt{\gtoksapp}, \texttt{\gtokspre}, \texttt{\gtokspre}, \texttt{\xtoksapp}, \texttt{\xtokspre} 30

2.8.3 \texttt{\csstring}, \texttt{\beginsname} and \texttt{\lastnamedcs} 31
2.8.4 \clearmarks
2.8.5 \alignmark and \aligntab
2.8.6 \letcharcode
2.8.7 \glet
2.8.8 \expanded, \immediateassignment and \immediateassigned
2.8.9 \ifcondition
2.9 Boxes, rules and leaders
2.9.1 \outputbox
2.9.2 \vpack, \hpack and \tpack
2.9.3 \vsplit
2.9.4 Images and reused box objects
2.9.5 \nohrule and \novrule
2.9.6 \gleaders
2.10 Languages
2.10.1 \hyphenationmin
2.10.2 \boundary, \noboundary, \protrusionboundary and \wordboundary
2.10.3 \glyphdimensionsmode
2.11 Control and debugging
2.11.1 Tracing
2.11.2 \outputmode
2.11.3 \draftmode
2.12 Files
2.12.1 File syntax
2.12.2 Writing to file
2.13 Math

3 Modifications
3.1 The merged engines
3.1.1 The need for change
3.1.2 Changes from \TeX 3.1415926
3.1.3 Changes from \e-\TeX 2.2
3.1.4 Changes from PDF\TeX 1.40
3.1.5 Changes from ALEPH RC4
3.1.6 Changes from anywhere
3.1.7 Changes from standard WEB2C
3.2 The backend primitives
3.2.1 Less primitives
3.2.2 \pdfextension, \pdfvariable and \pdffeedback
3.2.3 Defaults
3.2.4 Backward compatibility
3.3 Directions
3.3.1 Four directions
3.3.2 How it works
3.3.3 Controlling glue with \breakafterdirmode
3.3.4 Controlling parshapes with \shapemode
3.3.5 Symbols or numbers
3.4 Implementation notes
3.4.1 Memory allocation
3.4.2 Sparse arrays
3.4.3 Simple single-character csnames
3.4.4 The compressed format file
3.4.5 Binary file reading
3.4.6 Tabs and spaces
3.4.7 Hyperlinks

4 Using \LaTeX\TEX
4.1 Initialization
4.1.1 \LaTeX\TEX as a LUA interpreter
4.1.2 \LaTeX\TEX as a LUA byte compiler
4.1.3 Other commandline processing
4.2 LUA behaviour
4.2.1 The LUA version
4.2.2 Integration in the TDS ecosystem
4.2.3 Loading libraries
4.2.4 Executing programs
4.2.5 Multibyte string functions
4.2.6 Extra os library functions
4.2.7 Binary input from files with fio
4.2.8 Binary input from strings with sio
4.2.9 Hashes conform sha2
4.2.10 Locales
4.3 LUA modules
4.4 Testing

5 Languages, characters, fonts and glyphs
5.1 Introduction
5.2 Characters, glyphs and discretionaries
5.3 The main control loop
5.4 Loading patterns and exceptions
5.5 Applying hyphenation
5.6 Applying ligatures and kerning
5.7 Breaking paragraphs into lines
5.8 The lang library
5.8.1 new and id
5.8.2 hyphenation
5.8.3 clear_hyphenation and clean
5.8.4 patterns and clear_patterns
5.8.5 hyphenationmin
5.8.6 [pre|post][ex|]hyphenchar
5.8.7 hyphenate
5.8.8 [set|get]hjcode
6  Font structure  
6.1  The font tables  87  
6.2  Real fonts  
6.3  Virtual fonts  94  
6.3.1  The structure  94  
6.3.2  Artificial fonts  96  
6.3.3  Example virtual font  96  
6.4  The vf library  
6.5  The font library  97  
6.5.1  Loading a TFM file  97  
6.5.2  Loading a VF file  98  
6.5.3  The fonts array  98  
6.5.4  Checking a font’s status  99  
6.5.5  Defining a font directly  99  
6.5.6  Extending a font  99  
6.5.7  Projected next font id  99  
6.5.8  Font ids  100  
6.5.9  Iterating over all fonts  100  
6.5.10  \glyphdimensionsmode  100  
6.5.11  \discretionaryligaturemode  101  

7  Math  103  
7.1  Traditional alongside OPENTYPE  103  
7.2  Unicode math characters  103  
7.3  Math styles  105  
7.3.1  \mathstyle  105  
7.3.2  \Ustack  106  
7.3.3  Cramped math styles  106  
7.4  Math parameter settings  108  
7.4.1  Many new \Umath* primitives  108  
7.4.2  Font-based math parameters  109  
7.5  Math spacing  113  
7.5.1  Inline surrounding space  113  
7.5.2  Pairwise spacing  114  
7.5.3  Skips around display math  115  
7.5.4  Nolimit correction  115  
7.5.5  Math italic mess  116  
7.5.6  Script and kerning  116  
7.5.7  Fixed scripts  117  
7.5.8  Penalties: \math penaltiesmode  117  
7.5.9  Equation spacing: \matheqnogapstep  118  
7.6  Math constructs  118  
7.6.1  Unscaled fences  118  
7.6.2  Accent handling  119  
7.6.3  Radical extensions  120  
7.6.4  Super- and subscripts  120  
7.6.5  Scripts on extensibles  120
7.6.6 Fractions 122
7.6.7 Delimiters: \(\text{\textbackslash left}, \text{\textbackslash middle} \text{and} \text{\textbackslash right}\) 122

7.7 Extracting values 123
7.7.1 Codes 123
7.7.2 Last lines 123

7.8 Math mode 124
7.8.1 Verbose versions of single-character math commands 124
7.8.2 Script commands \Unosuperscript and \Unosubscript 124
7.8.3 Allowed math commands in non-math modes 125

7.9 Goodies 125
7.9.1 Flattening: \textbackslash mathflattenmode 125
7.9.2 Less Tracing 125
7.9.3 Math options with \textbackslash mathdefaults mode 126
7.9.4 Math options with \textbackslash mathoption 126

8 Nodes 127
8.1 LUA node representation 127
8.2 Main text nodes 127
8.2.1 hlist nodes 128
8.2.2 vlist nodes 128
8.2.3 rule nodes 128
8.2.4 ins nodes 129
8.2.5 mark nodes 129
8.2.6 adjust nodes 130
8.2.7 disc nodes 130
8.2.8 math nodes 130
8.2.9 glue nodes 131
8.2.10 kern nodes 132
8.2.11 penalty nodes 132
8.2.12 glyph nodes 132
8.2.13 boundary nodes 134
8.2.14 local_par nodes 134
8.2.15 dir nodes 134
8.2.16 marginkern nodes 135

8.3 Math noads 135
8.3.1 Math kernel subnodes 135
8.3.2 math_char and math_text_char subnodes 135
8.3.3 sub_box and sub_mlist subnodes 135
8.3.4 delim subnodes 136
8.3.5 Math core nodes 136
8.3.6 simple noad nodes 137
8.3.7 accent nodes 137
8.3.8 style nodes 137
8.3.9 choice nodes 137
8.3.10 radical nodes 138
8.3.11 fraction nodes 138
8.3.12 fence nodes 138
8.4 Front-end whatsits
  8.4.1 open
  8.4.2 write
  8.4.3 close
  8.4.4 user_defined
  8.4.5 save_pos
  8.4.6 late_lua

8.5 DVI backend whatsits
  8.5.1 special

8.6 PDF backend whatsits
  8.6.1 pdf_literal and pdf_late_literal
  8.6.2 pdf_refobj
  8.6.3 pdf_annot
  8.6.4 pdf_start_link
  8.6.5 pdf_end_link
  8.6.6 pdf_dest
  8.6.7 pdf_action
  8.6.8 pdf_thread
  8.6.9 pdf_start_thread
  8.6.10 pdf_end_thread
  8.6.11 pdf_colorstack
  8.6.12 pdf_setmatrix
  8.6.13 pdf_save
  8.6.14 pdf_restore

8.7 The node library
  8.7.1 Introduction
  8.7.2 is_node
  8.7.3 types and whatsits
  8.7.4 id
  8.7.5 type and subtype
  8.7.6 fields
  8.7.7 has_field
  8.7.8 new
  8.7.9 free, flush_node and flush_list
  8.7.10 copy and copy_list
  8.7.11 prev and next
  8.7.12 current_attr
  8.7.13 hpack
  8.7.14 vpack
  8.7.15 prepend_prevdepth
  8.7.16 dimensions and rangedimensions
  8.7.17 mlist_to_hlist
  8.7.18 slide
  8.7.19 tail
  8.7.20 length and type count
  8.7.21 is_char and is_glyph
8.7.22  traverse
8.7.23  traverse_id
8.7.24  traverse_char and traverse_glyph
8.7.25  traverse_list
8.7.26  has_glyph
8.7.27  end_of_math
8.7.28  remove
8.7.29  insert_before
8.7.30  insert_after
8.7.31  first_glyph
8.7.32  ligaturing
8.7.33  kerning
8.7.34  unprotect_glyph[s]
8.7.35  protect_glyph[s]
8.7.36  last_node
8.7.37  write
8.7.38  protrusion_skippable

8.8  Glue handling
8.8.1  setglue
8.8.2  getglue
8.8.3  is_zero_glue

8.9  Attribute handling
8.9.1  Attributes
8.9.2  attribute_list nodes
8.9.3  attr nodes
8.9.4  has_attribute
8.9.5  get_attribute
8.9.6  find_attribute
8.9.7  set_attribute
8.9.8  unset_attribute
8.9.9  slide
8.9.10  check_discretionary, check_discretionaries
8.9.11  flatten_discretionaries
8.9.12  family_font

8.10  Two access models
8.11  Properties

9  LUA callbacks
9.1  Registering callbacks
9.2  File discovery callbacks
9.2.1  find_read_file and find_write_file
9.2.2  find_font_file
9.2.3  find_output_file
9.2.4  find_format_file
9.2.5  find_vf_file
9.2.6  find_map_file
9.2.7  find_enc_file
<table>
<thead>
<tr>
<th>Section</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2.8</td>
<td>find_pk_file</td>
<td>171</td>
</tr>
<tr>
<td>9.2.9</td>
<td>find_data_file</td>
<td>171</td>
</tr>
<tr>
<td>9.2.10</td>
<td>find_opentype_file</td>
<td>171</td>
</tr>
<tr>
<td>9.2.11</td>
<td>find_truetype_file and find_type1_file</td>
<td>171</td>
</tr>
<tr>
<td>9.2.12</td>
<td>find_image_file</td>
<td>172</td>
</tr>
<tr>
<td>9.3</td>
<td>File reading callbacks</td>
<td>172</td>
</tr>
<tr>
<td>9.3.1</td>
<td>open_read_file</td>
<td>172</td>
</tr>
<tr>
<td>9.3.2</td>
<td>General file readers</td>
<td>173</td>
</tr>
<tr>
<td>9.4</td>
<td>Data processing callbacks</td>
<td>174</td>
</tr>
<tr>
<td>9.4.1</td>
<td>process_input_buffer</td>
<td>174</td>
</tr>
<tr>
<td>9.4.2</td>
<td>process_output_buffer</td>
<td>174</td>
</tr>
<tr>
<td>9.4.3</td>
<td>process_jobname</td>
<td>174</td>
</tr>
<tr>
<td>9.5</td>
<td>Node list processing callbacks</td>
<td>174</td>
</tr>
<tr>
<td>9.5.1</td>
<td>contribute_filter</td>
<td>174</td>
</tr>
<tr>
<td>9.5.2</td>
<td>buildpage_filter</td>
<td>175</td>
</tr>
<tr>
<td>9.5.3</td>
<td>build_page_insert</td>
<td>175</td>
</tr>
<tr>
<td>9.5.4</td>
<td>pre_linebreak_filter</td>
<td>176</td>
</tr>
<tr>
<td>9.5.5</td>
<td>linebreak_filter</td>
<td>177</td>
</tr>
<tr>
<td>9.5.6</td>
<td>append_to_vlist_filter</td>
<td>177</td>
</tr>
<tr>
<td>9.5.7</td>
<td>post_linebreak_filter</td>
<td>177</td>
</tr>
<tr>
<td>9.5.8</td>
<td>hpack_filter</td>
<td>177</td>
</tr>
<tr>
<td>9.5.9</td>
<td>vpack_filter</td>
<td>178</td>
</tr>
<tr>
<td>9.5.10</td>
<td>hpack_quality</td>
<td>178</td>
</tr>
<tr>
<td>9.5.11</td>
<td>vpack_quality</td>
<td>178</td>
</tr>
<tr>
<td>9.5.12</td>
<td>process_rule</td>
<td>179</td>
</tr>
<tr>
<td>9.5.13</td>
<td>pre_output_filter</td>
<td>179</td>
</tr>
<tr>
<td>9.5.14</td>
<td>hyphenate</td>
<td>179</td>
</tr>
<tr>
<td>9.5.15</td>
<td>ligaturing</td>
<td>179</td>
</tr>
<tr>
<td>9.5.16</td>
<td>kerning</td>
<td>180</td>
</tr>
<tr>
<td>9.5.17</td>
<td>insert_local_par</td>
<td>180</td>
</tr>
<tr>
<td>9.5.18</td>
<td>mlist_to_hlist</td>
<td>180</td>
</tr>
<tr>
<td>9.6</td>
<td>Information reporting callbacks</td>
<td>180</td>
</tr>
<tr>
<td>9.6.1</td>
<td>pre_dump</td>
<td>180</td>
</tr>
<tr>
<td>9.6.2</td>
<td>start_run</td>
<td>181</td>
</tr>
<tr>
<td>9.6.3</td>
<td>stop_run</td>
<td>181</td>
</tr>
<tr>
<td>9.6.4</td>
<td>start_page_number</td>
<td>181</td>
</tr>
<tr>
<td>9.6.5</td>
<td>stop_page_number</td>
<td>181</td>
</tr>
<tr>
<td>9.6.6</td>
<td>show_error_hook</td>
<td>181</td>
</tr>
<tr>
<td>9.6.7</td>
<td>show_error_message</td>
<td>182</td>
</tr>
<tr>
<td>9.6.8</td>
<td>show_lua_error_hook</td>
<td>182</td>
</tr>
<tr>
<td>9.6.9</td>
<td>start_file</td>
<td>182</td>
</tr>
<tr>
<td>9.6.10</td>
<td>stop_file</td>
<td>182</td>
</tr>
<tr>
<td>9.6.11</td>
<td>call_edit</td>
<td>182</td>
</tr>
<tr>
<td>9.6.12</td>
<td>finish_syncex</td>
<td>183</td>
</tr>
<tr>
<td>9.6.13</td>
<td>wrapup_run</td>
<td>183</td>
</tr>
</tbody>
</table>
9.7 PDF related callbacks
   9.7.1 finish_pdffile 183
   9.7.2 finish_pdfpage 183
   9.7.3 page_order_index 183
   9.7.4 process_pdf_image_content 184

9.8 Font-related callbacks
   9.8.1 define_font 184
   9.8.2 glyph_not_found and glyph_info 185

10 The \TeX related libraries 187
10.1 The lua library
   10.1.1 Version information 187
   10.1.2 Bytecode registers 187
   10.1.3 Chunk name registers 187
   10.1.4 Introspection 188
10.2 The status library 188
10.3 The \texttt{tex} library
   10.3.1 Introduction 190
   10.3.2 Internal parameter values, set and get 190
   10.3.3 Convert commands 193
   10.3.4 Last item commands 194
   10.3.5 Accessing registers: set*, get* and is* 194
   10.3.6 Character code registers: [get|set]*code[s] 196
   10.3.7 Box registers: [get|set]box 197
   10.3.8 Reusing boxes: [use|save]boxresource and getboxresourcedimensions 198
   10.3.9 triggerbuildpage 198
   10.3.10 splitbox 198
   10.3.11 Accessing math parameters: [get|set]math 198
   10.3.12 Special list heads: [get|set]list 200
   10.3.13 Semantic nest levels: getnest and ptr 200
   10.3.14 Print functions 201
   10.3.15 Helper functions 203
   10.3.16 Functions for dealing with primitives 206
   10.3.17 Core functionality interfaces 210
   10.3.18 Randomizers 212
   10.3.19 Functions related to synctex 212

10.4 The \texttt{texconfig} table 212
10.5 The \texttt{texio} library
   10.5.1 write 214
   10.5.2 write_nl 214
   10.5.3 setescape 215
   10.5.4 closeinput 215

10.6 The \texttt{token} library
   10.6.1 The scanner 215
   10.6.2 Picking up one token 217
   10.6.3 Creating tokens 218
   10.6.4 Macros 219
<table>
<thead>
<tr>
<th>10.6.5</th>
<th>Pushing back</th>
<th>219</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.6.6</td>
<td>Nota bene</td>
<td>220</td>
</tr>
</tbody>
</table>

10.7 The kpse library

| 10.7.1  | set_program_name and new | 221 |
| 10.7.2  | record_input_file and record_output_file | 222 |
| 10.7.3  | find_file | 222 |
| 10.7.4  | lookup | 223 |
| 10.7.5  | init_prog | 223 |
| 10.7.6  | readable_file | 223 |
| 10.7.7  | expand_path | 224 |
| 10.7.8  | expand_var | 224 |
| 10.7.9  | expand_braces | 224 |
| 10.7.10 | show_path | 224 |
| 10.7.11 | var_value | 224 |
| 10.7.12 | version | 224 |
| 10.7.13 | check_permission | 224 |

11 The graphic libraries

11.1 The img library

| 11.1.1  | new | 225 |
| 11.1.2  | fields | 225 |
| 11.1.3  | scan | 226 |
| 11.1.4  | copy | 227 |
| 11.1.5  | write, immediatewrite, immediatewriteobject | 228 |
| 11.1.6  | node | 229 |
| 11.1.7  | types | 229 |
| 11.1.8  | boxes | 229 |

11.2 The mplib library

| 11.2.1  | new | 230 |
| 11.2.2  | statistics | 230 |
| 11.2.3  | execute | 231 |
| 11.2.4  | finish | 231 |
| 11.2.5  | Result table | 232 |
| 11.2.6  | Subsidiary table formats | 234 |
| 11.2.7  | Pens and pen_info | 235 |
| 11.2.8  | Character size information | 236 |

12 The fontloader

12.1 Getting quick information on a font | 237 |
12.2 Loading an OPENTYPE or TRUETYPE file | 237 |
12.3 Applying a ‘feature file’ | 239 |
12.4 Applying an ‘AFM file’ | 239 |
12.5 Fontloader font tables | 239 |
12.6 Table types | 240 |
| 12.6.1  | The main table | 240 |
| 12.6.2  | glyphs | 242 |
| 12.6.3  | map | 245 |
12.6.4 private
12.6.5 cidinfo
12.6.6 pfminfo
12.6.7 names
12.6.8 anchor_classes
12.6.9 gpos
12.6.10 gsub
12.6.11 ttf_tables and ttf_tab_saved
12.6.12 mm
12.6.13 mark_classes
12.6.14 math
12.6.15 validation_state
12.6.16 horiz_base and vert_base
12.6.17 altuni
12.6.18 vert_variants and horiz_variants
12.6.19 mathkern
12.6.20 kerns
12.6.21 vkerns
12.6.22 texdata
12.6.23 lookups

13 The HarfBuzz libraries
13.1 The luahrfbuzz library
13.1.1 Example

14 The backend libraries
14.1 The pdf library
14.1.1 mapfile, mapline
14.1.2 [set|get][catalog|info|names|trailer]
14.1.3 [set|get][pageattributes|pageresources|pagesattributes]
14.1.4 [set|get][xformattributes|xformresources]
14.1.5 [set|get][major|minor]version
14.1.6 getcreationdate
14.1.7 [set|get]inclusionerrorlevel and [set|get]ignoreunknownimages
14.1.8 [set|get]suppressoptionalinfo, [set|get]trailerid,
14.1.9 [set|get][obj]compresslevel and [set|get]recompress
14.1.10 [set|get]gentounicode
14.1.11 [set|get]decimaldigits
14.1.12 [set|get]pkresolution
14.1.13 getlast[obj]link[annot] and getretval
14.1.14 getmaxobjnum and getobjtype, getfontname, getfontobjnum,
   getfontsize, getxformname
14.1.15 [set|get]origin
14.1.16 [set|get]imageresolution
14.1.17 [set|get][link|dest|thread|xform]margin
14.1.18 get[pos|hpos|vpos]
14.1.19 [has|get]matrix 281
14.1.20 print 282
14.1.21 immediateobj 282
14.1.22 obj 283
14.1.23 refobj 284
14.1.24 reserveobj 284
14.1.25 getpageref 284
14.1.26 registerannot 284
14.1.27 newcolorstack 284
14.1.28 setfontattributes 285

14.2 The pdfe library 285
14.2.1 Introduction 285
14.2.2 open, new, getstatus, close, unencrypt 285
14.2.3 getsize, getversion, getnofobjects, getnofpages, getmemoryusage 286
14.2.4 get[catalog|trailer|info] 286
14.2.5 getpage, getbox 286
14.2.6 get[string|integer|number|boolean|name], type 287
14.2.7 get[dictionary|array|stream] 287
14.2.8 [open|close|readfrom|readfromwhole]stream 287
14.2.9 getfrom[dictionary|array] 288
14.2.10 [dictionary|array]totable 288
14.2.11 getfromreference 288

14.3 Memory streams 289
14.4 The pdfscanner library 289

Topics 293

Primitives 297

Callbacks 305

Nodes 307

Libraries 309

Statistics 317
Introduction

This is the reference manual of \texttt{Lua\TeX}. We don’t claim it is complete and we assume that the reader knows about \TeX as described in “The \TeX Book”, the “\e\TeX manual”, the “pdf\TeX manual”, etc. Additional reference material is published in journals of user groups and Con\TeXt related documentation.

It took about a decade to reach stable version 1.0, but for good reason. Successive versions brought new functionality, more control, some cleanup of internals. Experimental features evolved into stable ones or were dropped. Already quite early \texttt{Lua\TeX} could be used for production and it was used on a daily basis by the authors. Successive versions sometimes demanded an adaption to the Lua interfacing, but the concepts were unchanged. The current version can be considered stable in functionality and there will be no fundamental changes. Of course we then can decide to move towards version 2.00 with different properties.

Don’t expect Lua\TeX to behave the same as pdf\TeX! Although the core functionality of that 8 bit engine was starting point, it has been combined with the directional support of Omega (Aleph). But, Lua\TeX can behave different due to its wide (32 bit) characters, many registers and large memory support. The pdf code produced differs from pdf\TeX but users will normally not notice that. There is native utf input, support for large (more than 8 bit) fonts, and the math machinery is tuned for OpenType math. There is support for directional typesetting too. The log output can differ from other engines and will likely differ more as we move forward. When you run plain \TeX for sure Lua\TeX runs slower than pdf\TeX but when you run for instance Con\TeXt MkIV in many cases it runs faster, especially when you have a bit more complex documents or input. Anyway, 32 bit all–over combined with more features has a price, but on a modern machine this is no real problem.

Testing is done with Con\TeXt, but Lua\TeX should work fine with other macro packages too. For that purpose we provide generic font handlers that are mostly the same as used in Con\TeXt. Discussing specific implementations is beyond this manual. Even when we keep Lua\TeX lean and mean, we already have enough to discuss here.

Lua\TeX consists of a number of interrelated but (still) distinguishable parts. The organization of the source code is adapted so that it can glue all these components together. We continue cleaning up side effects of the accumulated code in \TeX engines (especially code that is not needed any longer).

\begin{itemize}
  \item We started out with most of pdf\TeX version 1.40.9. The code base was converted to C and split in modules. Experimental features were removed and utility macros are not inherited because their functionality can be programmed in Lua. The number of backend interface commands has been reduced to a few. The so called extensions are separated from the core (which we try to keep close to the original \TeX core). Some mechanisms like expansion and protrusion can behave different from the original due to some cleanup and optimization. Some whatsthat based functionality (image support and reusable content) is now core functionality. We don’t stay in sync with pdf\TeX development.
  \item The direction model from Aleph RC4 (which is derived from Omega) is included. The related primitives are part of core Lua\TeX but at the node level directional support is no longer based
on so called whatsits but on real nodes with relevant properties. The number of directions is limited to the useful set and the backend has been made direction aware.

- Neither Aleph’s I/O translation processes, nor tcx files, nor encTeX are available. These encoding-related functions are superseded by a Lua-based solution (reader callbacks). In a similar fashion all file io can be intercepted.

- We currently use Lua 5.3.*. There are few Lua libraries that we consider part of the core Lua machinery, for instance lpeg. There are additional Lua libraries that interface to the internals of TeX. We also keep the Lua 5.2 bit32 library around.

- There are various TeX extensions but only those that cannot be done using the Lua interfaces. The math machinery often has two code paths: one traditional and the other more suitable for wide OpenType fonts. Here we follow the Microsoft specifications as much as possible. Some math functionality has been opened up a bit so that users have more control.

- The fontloader uses parts of FontForge 2008.11.17 combined with additional code specific for usage in a TeX engine. We try to minimize specific font support to what TeX needs: character references and dimensions and delegate everything else to Lua. That way we keep TeX open for extensions without touching the core. In order to minimize dependencies at some point we may decide to make this an optional library.

- The MetaPost library is integral part of LuaTeX. This gives TeX some graphical capabilities using a relative high speed graphical subsystem. Again Lua is used as glue between the frontend and backend. Further development of MetaPost is closely related to LuaTeX.

- The virtual font technology that comes with TeX has been integrated into the font machinery in a way that permits creating virtual fonts at runtime. Because LuaTeX can also act as a Lua interpreter this means that a complete TeX workflow can be built without the need for additional programs.

- The versions starting from 1.09 no longer use the poppler library for inclusion but a lightweight dedicated one. This removes a dependency but also makes the inclusion code of LuaTeX different from pdfTeX. In fact it was already much different due to the Lua image interfacing.

We try to keep upcoming versions compatible but intermediate releases can contain experimental features. A general rule is that versions that end up on TeXLive and/or are released around ConTeXt meetings are stable. Any version between the yearly TeXLive releases are to be considered beta and in the repository end up as trunk releases. We have an experimental branch that we use for development but there is no support for any of its experimental features. Intermediate releases (from trunk) are normally available via the ConTeXt distribution channels (the garden and so called minimals).

Version 1.10 is more or less an endpoint in development: this is what you get. Because not only ConTeXt, that we can adapt rather easily, uses LuaTeX, we cannot change fundamentals without unforeseen consequences. By now it has been proven that Lua can be used to extend the core functionality so there is no need to add more, and definitely no hard coded solutions for (not so) common problems. Of course there will be bug fixes, maybe some optimization, and there might
even be some additions or non-intrusive improvements, but only after testing outside the stable release. After all, the binary is already more than large enough and there is not that much to gain.

You might find Lua helpers that are not yet documented. These are considered experimental, although when you encounter them in a ConÎêXt version that has been around for a while you can assume that they will stay. Of course it can just be that we forgot to document them yet.

A manual like this is not really meant as tutorial, for that we refer to documents that ship with ConÎêXt, articles, etc. It is also never complete enough for all readers. We try to keep up but the reader needs to realize that it’s all volunteer work done in spare time. And for sure, complaining about a bad manual or crappy documentation will not really motivate us to spend more time on it. That being said, we hope that this document is useful.

Hans Hagen
Harmut Henkel
Taco Hoekwater
Luigi Scarso

Version : April 29, 2023
LuaÎêX : luatex 1.17 / 7581
ConÎêXt : MkIV 2023.03.10 12:15
**1 Preamble**

This is a reference manual, not a tutorial. This means that we discuss changes relative to traditional \TeX{} and also present new functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later.

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 \varepsilon-\TeX{} manual” and documentation about pdf\TeX{}.

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

\begin{verbatim}
\directlua{tex.print("Hi there")}
\end{verbatim}

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 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:

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

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 that we will discuss later. Each node points back to a previous node or next node, given that these exist.

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. We turn horizontal material into vertical. 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.

The Lua\TeX{} 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 mean these hooks.

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 demands, 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.

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

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

For now this should be enough to enable you to understand the next chapters.
2 Basic \TeX\ enhancements

2.1 Introduction

2.1.1 Primitive behaviour

From day one, LaTeX has offered extra features compared to the superset of pdf\TeX, which includes e-\TeX, and Aleph. This has not been limited to the possibility to execute Lua code via \directlua, but LaTeX also adds functionality via new \TeX-side primitives or extensions to existing ones.

When LaTeX starts up in ‘initex’ mode (\texttt{latex -ini}), it defines only the primitive commands known by \TeX82 and the one extra command \directlua. As is fitting, a Lua function has to be called to add the extra primitives to the user environment. The simplest method to get access to all of the new primitive commands is by adding this line to the format generation file:

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

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 10.3.16. For simplicity’s sake, this manual assumes that you have executed the \directlua command as given above.

The startup behaviour documented above is considered stable in the sense that there will not be backward-incompatible changes any more. We have promoted some rather generic pdf\TeX\ primitives to core LaTeX ones, and the few that we inherited from Aleph (Omega) are also promoted. Effectively this means that we now only have the \tex, \etex and \luatex\ sets left.

In Chapter 3 we discuss several primitives that are derived from pdf\TeX\ and Aleph (Omega). Here we stick to real new ones. In the chapters on fonts and math we discuss a few more new ones.

2.1.2 Version information

2.1.2.1 \texttt{\latexbanner, \texttt{\latexversion and \latexrevision}}

There are three new primitives to test the version of LaTeX:

<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\latexbanner</td>
<td>This is LaTeX, Version 1.17.0</td>
<td>the banner reported on the command line</td>
</tr>
</tbody>
</table>
The official LuaTeX version is defined as follows:

- The major version is the integer result of \texttt{\luatexversion} divided by 100. The primitive is an ‘internal variable’, so you may need to prefix its use with \texttt{\the} depending on the context.
- The minor version is the two-digit result of \texttt{\luatexversion} modulo 100.
- The revision is reported by \texttt{\luatexrevision}. This primitive expands to a positive integer.
- The full version number consists of the major version, minor version and revision, separated by dots.

\subsection{2.1.2.2 \formatname}

The \texttt{\formatname} syntax is identical to \texttt{\jobname}. In \LaTeX, the expansion is empty. Otherwise, the expansion is the value that \texttt{\jobname} had during the \LaTeX run that dumped the currently loaded format. You can use this token list to provide your own version info.

\section{2.2 UNICODE text support}

\subsection{2.2.1 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 LuaTeX 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, \texttt{\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: \texttt{\char}, \texttt{\lccode}, \texttt{\uccode}, \texttt{\hjcode}, \texttt{\catcode}, \texttt{\sfcode}, \texttt{\efcode}, \texttt{\lpcode}, \texttt{\rpcode}, \texttt{\chardef}.

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 \texttt{reader} callback. This will be explained in section 9.3. 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 \(c \geq 1,114,112\), LuaTeX will actually print the single byte corresponding to \(c \) minus 1,114,112.
Output to the terminal uses ^^ notation for the lower control range (c < 32), with the exception of ^^I, ^^J and ^^M. These are considered ‘safe’ and therefore printed as-is. You can disable escaping with texio.setescape(false) in which case you get the normal characters on the console.

### 2.2.2 \Uchar

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

### 2.2.3 Extended tables

All traditional \TeX and \e-\TeX registers can be 16-bit numbers. The affected commands are:

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

Because font memory management has been rewritten, character properties in fonts are no longer shared among font instances that originate from the same metric file. Of course we share fonts in the backend when possible so that the resulting pdf file is as efficient as possible, but for instance also expansion and protrusion no longer use copies as in pdf\TeX.

### 2.3 Attributes

#### 2.3.1 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 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).

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.

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 Lua\TeX 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 an node and this is where attributes kick in.

### 2.3.2 Attribute registers

Attributes are a completely new concept in Lua\TeX. 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 init\TeX.

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 8.

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 Lua\TeX.

### 2.3.3 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 Lua\TeX 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.

## 2.4 LUA related primitives

### 2.4.1 \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)
\directlua (16-bit number) (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 (16-bit number) designates a name of a Lua chunk and is taken from the lua.name array (see the documentation of the lua table further in this manual). When a chunk name starts with a @ it will be displayed as a file name. This is a side effect of the way Lua implements error handling.

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 places material on a pseudo-file to be immediately read by \TeX, as \TeX’s \scantokens. For a description of print functions look at section 10.3.14.
Because the \texttt{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 still pretty bad. 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 \TeX{} pretty bad. If you are not careful while working with the node list interface, you may even end up with assertion errors from within the \TeX{} portion of the executable.

### 2.4.2 \texttt{\latexlua} and \texttt{\latexluafn}

Contrary to \texttt{\directlua}, \texttt{\latexlua} stores Lua code in a whatsit that will be processed at the time of shipping out. Its intended use is a cross between pdf literals (often available as \texttt{\pdfliteral}) and the traditional \TeX{} extension \texttt{\write}. Within the Lua code you can print pdf statements directly to the pdf file via \texttt{pdf.print}, or you can write to other output streams via \texttt{texio.write} or simply using Lua io routines.

\begin{verbatim}
\latexlua (general text)
\latexlua (16-bit number)  \texttt{\ldots}  \latexlua (general text)
\end{verbatim}

Expansion of macros in the final \texttt{general text} is delayed until just before the whatsit is executed (like in \texttt{\write}). With regard to pdf output stream \texttt{\latexlua} behaves as pdf page literals. The name \texttt{\ldots} behave in the same way as they do for \texttt{\directlua}. The \texttt{\latexluafn} primitive takes a number and is similar to \texttt{\luafn} but gets delayed to shipout time. It’s just there for completeness.

### 2.4.3 \texttt{\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 \texttt{n} and \texttt{r} respectively. The token sequence is fully expanded.

\begin{verbatim}
\luaescapestring (general text)
\end{verbatim}

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 \texttt{dofile}:

\begin{verbatim}
\directlua { dofile('mysetups.lua') }
\end{verbatim}

### 2.4.4 \texttt{\luafn}, \texttt{\luafncall} and \texttt{\ludef}

The \texttt{\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: \texttt{\luafn}. 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 \LaTeX 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.

2.4.5 \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).

## 2.5 Catcode tables

### 2.5.1 Catcodes

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single statement. You can have a practically unlimited number of different tables. 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.

### 2.5.2 \catcodetable

\begin{verbatim}
\catcodetable ⟨15-bit number⟩
\end{verbatim}

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 ini\TeX.

### 2.5.3 \initcatcodetable

\begin{verbatim}
\initcatcodetable ⟨15-bit number⟩
\end{verbatim}

The primitive \initcatcodetable creates a new table with catcodes identical to those defined by ini\TeX. 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:

<table>
<thead>
<tr>
<th>CATCODE</th>
<th>CHARACTER</th>
<th>EQUIVALENT</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\</td>
<td>escape</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>^\text{M}</td>
<td>return</td>
<td>car_ret</td>
</tr>
<tr>
<td>9</td>
<td>^\text{@}</td>
<td>null</td>
<td>ignore</td>
</tr>
<tr>
<td>10</td>
<td>&lt;space&gt;</td>
<td>space</td>
<td>spacer</td>
</tr>
<tr>
<td>11</td>
<td>a–z</td>
<td>letter</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>A–Z</td>
<td>letter</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>everything else</td>
<td>other</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>%</td>
<td>comment</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>^\text{?}</td>
<td>delete</td>
<td>invalid_char</td>
</tr>
</tbody>
</table>

### 2.5.4 \savecatcodetable

\begin{verbatim}
\savecatcodetable ⟨15-bit number⟩
\end{verbatim}
\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. 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.

2.6 Suppressing errors

2.6.1 \suppressfontnotfounderror

If this integer parameter is non-zero, then Lua\TeX will not complain about font metrics that are not found. Instead it will silently skip the font assignment, making the requested csname for the font \texttt{\ifx} equal to \texttt{\nullfont}, so that it can be tested against that without bothering the user.

\verbatim
\suppressfontnotfounderror = 1
\endverbatim

2.6.2 \suppresslongerror

If this integer parameter is non-zero, then Lua\TeX will not complain about \texttt{\par} commands encountered in contexts where that is normally prohibited (most prominently in the arguments of macros not defined as \texttt{\long}).

\verbatim
\suppresslongerror = 1
\endverbatim

2.6.3 \suppressifcsnameerror

If this integer parameter is non-zero, then Lua\TeX will not complain about non-expandable commands appearing in the middle of a \texttt{\ifcsname} expansion. Instead, it will keep getting expanded tokens from the input until it encounters an \texttt{\endcsname} command. If the input expansion is unbalanced with respect to \texttt{\csname} ... \texttt{\endcsname} pairs, the Lua\TeX process may hang indefinitely.

\verbatim
\suppressifcsnameerror = 1
\endverbatim

2.6.4 \suppressoutererror

If this new integer parameter is non-zero, then Lua\TeX will not complain about \texttt{\outer} commands encountered in contexts where that is normally prohibited.

\verbatim
\suppressoutererror = 1
\endverbatim

2.6.5 \suppressmathparerror

The following setting will permit \texttt{\par} tokens in a math formula:

\verbatim
\suppressmathparerror = 1
\endverbatim
So, the next code is valid then:

\verbatim
$ x + 1 = $\endverbatim
2.6.6 \suppressprimitiveerror

When set to a non-zero value the following command will not issue an error:
\suppressprimitiveerror = 1
\primitive\notaprimitive

2.7 Fonts

2.7.1 Font syntax

Lua\TeX 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.

2.7.2 \fontid and \setfontid

\fontid\font

This primitive expands into a number. It is not a register so there is no need to prefix with \number (and using \the gives an error). The currently used font id is 29. Here are some more:

<table>
<thead>
<tr>
<th>STYLE</th>
<th>COMMAND</th>
<th>FONT ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>\tf</td>
<td>38</td>
</tr>
<tr>
<td>bold</td>
<td>\bf</td>
<td>38</td>
</tr>
<tr>
<td>italic</td>
<td>\it</td>
<td>50</td>
</tr>
<tr>
<td>bold italic</td>
<td>\bi</td>
<td>51</td>
</tr>
</tbody>
</table>

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 Con\TeX{}t 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.

2.7.3 \noligs and \nokerns

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by Lua\TeX{}’s main control loop. You can enable these primitives when you want to do node list processing of ‘characters’, where \TeX{}’s normal processing would get in the way.
\noligs\langle integer\rangle
\nokerns\langle integer\rangle

These primitives can also be implemented by overloading the ligature building and kerning functions, i.e. by assigning dummy functions to their associated callbacks. Keep in mind that when you define a font (using Lua) you can also omit the kern and ligature tables, which has the same effect as the above.

2.7.4 \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 2.1 we see the results for four characters separated by a space.

\begin{figure}[h]
\centering
\begin{tabular}{ccc}
\hline
\rule{5cm}{0.5mm} & \rule{5cm}{0.5mm} & \rule{5cm}{0.5mm} \\
0 / hsize 10mm & 1 / hsize 10mm & 2 / hsize 10mm \\
\hline
\end{tabular}
\end{figure}

\textbf{Figure 2.1} The \nospaces options.

2.8 Tokens, commands and strings

2.8.1 \scantextokens

The syntax of \scantextokens is identical to \scantokens. This primitive is a slightly adapted version of \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.

2.8.2 \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.

2.8.3 \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.

2.8.4 \clearmarks

This primitive complements the \v-TeX mark primitives and clears a mark class completely, resetting all three connected mark texts to empty. It is an immediate command.

\clearmarks ⟨16-bit number⟩

2.8.5 \alignmark and \aligntab

The primitive \alignmark duplicates the functionality of # inside alignment preambles, while \aligntab duplicates the functionality of &.

2.8.6 \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).
2.8.7 \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).

2.8.8 \expanded, \immediateassignment and \immediateassigned

The \expanded primitive takes a token list and expands it 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. But this kind of expansion it still doesn’t expand some primitive operations.
\newcount\NumberOfCalls
\def\TestMe{\advance\NumberOfCalls1 }
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\meaning\Tested

The result is a macro that has the not expanded code in its body:
macro:->\advance \NumberOfCalls 1 foo:0

Instead we can define \TestMe in a way that expands the assignment immediately. You need of course to be aware of preventing look ahead interference by using a space or \relax (often an expression works better as it doesn’t leave an \relax).
\def\TestMe{\immediateassignment\advance\NumberOfCalls1 }
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\meaning\Tested

This time the counter gets updates and we don’t see interference in the resulting \Tested macro:
macro:->foo:3

Here is a somewhat silly example of expanded comparison:
\def\expandeddoifelse#1#2#3#4% 
{\immediateassignment\edef\tempa{#1}\immediateassignment\edef\tempb{#2}
\ifx\tempa\tempb
  \immediateassignment\def\next{#3}%
\else
  \immediateassignment\def\next{#4}%
\fi
\next

\edef\Tested
  {{\expandeddoifelse\{\def\{\{\nop/\%
\expandeddoifelse\{\abc\{\yes\{\nop\)}}

\meaning\Tested

It gives:
macro:->(nop/yes)

A variant is:
\def\expandeddoifelse\#1\#2\#3\#4%
  {\immediateassigned{
    \edef\tempa{\#1}%
    \edef\tempb{\#2}%
  }%
  \ifx\tempa\tempb
    \immediateassignment\def\next{\#3}%
  \else
    \immediateassignment\def\next{\#4}%
  \fi
  \next}

The possible error messages are the same as using assignments in preambles of alignments and after the \accent command. The supported assignments are the so called prefixed commands (except box assignments).

2.8.9 \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}%
   \fi
   \next

\unexpanded\def\something\#1\#2%
  {\edef\tempa{\#1}%
   \edef\tempb{\#2}
   \ifx\tempa\tempb
     \ifcondition\something\{a\}{b}%
     \ifcondition\something\{a\}{a}%
   \fi
   \next

Basic \TeX enhancements  33
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.

## 2.9 Boxes, rules and leaders

### 2.9.1 \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

### 2.9.2 \vpack, \hpack and \tpack

These three primitives are like \vbox, \hbox and \vtop but don’t apply the related callbacks.

### 2.9.3 \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.

### 2.9.4 Images and reused box objects

These two concepts are now core concepts and no longer whatsits. They are in fact now implemented as rules with special properties. Normal rules have subtype 0, saved boxes have subtype 1 and images have subtype 2. This has the positive side effect that whenever we need to take content with dimensions into account, when we look at rule nodes, we automatically also deal with these two types.

The syntax of the \save...resource is the same as in pdf\TeX but you should consider them to be backend specific. This means that a macro package should treat them as such and check for the current output mode if applicable.
LuaTeX accepts 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
```

The box resources are of course implemented in the backend and therefore we do support the `attr` and `resources` keys that accept a token list. New is 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`.

### 2.9.5 `\nohrule` and `\nvrule`

Because introducing a new keyword can cause incompatibilities, two new primitives were introduced: `\nohrule` and `\nvrule`. These can be used to reserve space. This is often more efficient than creating an empty box with fake dimensions.

### 2.9.6 `\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.

### 2.10 Languages

#### 2.10.1 `\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.

#### 2.10.2 `\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:
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.

### 2.10.3 \glyphdimensionsmode

Already in the early days of Lua\TeX{} the decision was made to calculate the effective height and depth of glyphs in a way that reflected the applied vertical offset. The height got that offset added, the depth only when the offset was larger than zero. We can now control this in more detail with this mode parameter. An offset is added to the height and/or subtracted from the depth. The effective values are never negative. The zero mode is the default.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the old behaviour: add the offset to the height and only subtract the offset only from the depth when it is positive</td>
</tr>
<tr>
<td>1</td>
<td>add the offset to the height and subtract it from the depth</td>
</tr>
<tr>
<td>2</td>
<td>add the offset to the height and subtract it from the depth but keep the maxima of the current and previous results</td>
</tr>
<tr>
<td>3</td>
<td>use the height and depth of the glyph, so no offset is applied</td>
</tr>
</tbody>
</table>

### 2.11 Control and debugging

#### 2.11.1 Tracing

If \texttt{\tracingonline} is larger than 2, the node list display will also print the node number of the nodes.

#### 2.11.2 \outputmode

The \texttt{\outputmode} variable tells Lua\TeX{} what it has to produce:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dvi code</td>
</tr>
<tr>
<td>1</td>
<td>pdf code</td>
</tr>
</tbody>
</table>

#### 2.11.3 \draftmode

The value of the \texttt{\draftmode} counter signals the backend if it should output less. The pdf backend accepts a value of 1, while the dvi backend ignores the value. This is no critical feature so we can remove it in future versions when it can make the backend cleaner.
2.12 Files

2.12.1 File syntax

LuaTEX 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.

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.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>REPORTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\foo xyz</td>
</tr>
<tr>
<td>1</td>
<td>\foo (bar)</td>
</tr>
<tr>
<td>2</td>
<td>&lt;bar&gt; xyz</td>
</tr>
<tr>
<td>3</td>
<td>&lt;bar @ ..pt&gt; xyz</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id: bar&gt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id: bar @ ..pt&gt; xyz</td>
</tr>
</tbody>
</table>

2.12.2 Writing to file

You can now open up to 127 files with \openout. When no file is open writes will go to the console and log. As a consequence a system command is no longer possible but one can use os.execute to do the same.

2.13 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 T\TeX fonts and OpenType math.
3 Modifications

3.1 The merged engines

3.1.1 The need for change

The first version of Lua\TeX only had a few extra primitives and it was largely the same as pdf\TeX. Then we merged substantial parts of Aleph into the code and got more primitives. When we got more stable the decision was made to clean up the rather hybrid nature of the program. This means that some primitives have been promoted to core primitives, often with a different name, and that others were removed. This made it possible to start cleaning up the code base. In chapter 2 we discussed some new primitives, here we will cover most of the adapted ones.

Besides the expected changes caused by new functionality, there are a number of not-so-ex- pected 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.

3.1.2 Changes from \TeX 3.1415926

Of course it all starts with traditional \TeX. Even if we started with pdf\TeX, most still comes from the original. But we divert a bit.

- The current code base is written in C, not Pascal. We use cweb when possible. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like \texttt{tex}, \texttt{pdf}, \texttt{lang}, \texttt{font}, \texttt{lua}, etc. There are some artifacts of the conversion to C, but in due time we will clean up the source code and make sure that the documentation is done right. Many files are in the cweb format, but others, like those interfacing to Lua, are C files. Of course we want to stay as close as possible to the original so that the documentation of the fundamentals behind \TeX by Don Knuth still applies.
- See chapter 5 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 \texttt{of{}fice}) does not prevent ligature creation.
- There is no pool file, all strings are embedded during compilation.
- The specifier \texttt{plus 1 fillll} does not generate an error. The extra ‘\texttt{1}’ is simply typeset.
- The upper limit to \texttt{\endlinechar} and \texttt{\newlinechar} is 127.
- Magnification (\texttt{\mag}) is only supported in dvi output mode. You can set this parameter and it even works with \texttt{true} units till you switch to pdf output mode. When you use pdf output you can best not touch the \texttt{\mag} variable. This fuzzy behaviour is not much different from using pdf backend related functionality while eventually dvi output is required. After the output mode has been frozen (normally that happens when the first page is shipped out) or when pdf output is enabled, the \texttt{true} specification is ignored. When you preload a plain format adapted to Lua\TeX it can be that the \texttt{\mag} parameter already has been set.
3.1.3 Changes from $\varepsilon$-TeX 2.2

Being the de facto standard extension of course we provide the $\varepsilon$-TeX functionality, but with a few small adaptations.

- The $\varepsilon$-TeX functionality is always present and enabled so the prepended asterisk or -etex switch for init$\varepsilon$TeX is not needed.
- The TeXXeT extension is not present, so the primitives \TeXeTstate, \beginR, \beginL, \endR and \endL are missing. Instead we used the Omega/Aleph approach to directionality as starting point.
- Some of the tracing information that is output by $\varepsilon$-TeX's \tracingassigns and \tracingrestore is not there.
- Register management in LuaTeX 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 $\varepsilon$-TeX.
- When kpathsea is used to find files, LuaTeX uses the ofm file format to search for font metrics. In turn, this means that LuaTeX looks at the 0FMFONTS configuration variable (like Omega and Aleph) instead of TFMFONTS (like TeX and pdfTeX). Likewise for virtual fonts (LuaTeX uses the variable 0VFFONTS instead of VFFONTS).
- The primitives that report a stretch or shrink order report a value in a convenient range zero up to four. Because some macro packages can break on that we also provide \eTeXgluestretchorder and \eTeXglueshrinkorder which report values compatible with $\varepsilon$-TeX. The (new) fi value is reported as -1 (so when used in an \ifcase test that value makes one end up in the \else).

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.

For compatibility reasons we still refer to \pdf... commands but LuaTeX has a different backend interface. Instead of these primitives there are three interfacing primitives: \pdextension, \pdfvariable and \pdffeedback that take keywords and optional further arguments (below we will still use the \pdf prefix names as reference). This way we can extend the features when needed but don’t need to adapt the core engine. The front- and backend are decoupled as much as possible.

- The (experimental) support for snap nodes has been removed, because it is much more natural to build this functionality on top of node processing and attributes. The associated primitives that are gone are: \pdfsnaprefpoint, \pdfsnapy, and \pdfsnapycomp.
- The (experimental) support for specialized spacing around nodes has also been removed. The associated primitives that are gone are: \pdfadjustinterwordglue, \pdfprependkern, and \pdfappendkern, as well as the five supporting primitives \knbscode, \stbscode, \shbscode, \knbcode, and \knaccode.
- A number of ‘pdfTeX primitives’ have been removed as they can be implemented using Lua: \pdfelapsedtime, \pdfescapehex, \pdfescapename, \pdfescapestring, \pdffiledump, \pdffilemoddate, \pdffilesizes, \pdfforcepagebox, \pdfmatch, \pdffile...
The version related primitives \pdftexbanner, \pdftexversion and \pdftexrevision are no longer present as there is no longer a relationship with pdf\TeX{} development.

The experimental snapper mechanism has been removed and therefore also the primitives \pdfignoreddimen, \pdffirstlineheight, \pdfeachlineheight, \pdfeachlinedepth and \pdflastlinedepth.

The experimental primitives \primitive, \ifprimitive, \ifabsnum and \ifabsdim are promoted to core primitives. The \pdf* prefixed originals are not available.

Because Lua\TeX{} has a different subsystem for managing images, more diversion from its ancestor happened in the meantime. We don’t adapt to changes in pdf\TeX{}.

Two extra token lists are provided, \pdfxformresources and \pdfxformattr, as an alternative to \pdfxform keywords.

Image specifications also support visiblefilename, userpassword and ownerpassword. The password options are only relevant for encrypted pdf files.

The current version of Lua\TeX{} no longer replaces and/or merges fonts in embedded pdf files with fonts of the enveloping pdf document. This regression may be temporary, depending on how the rewritten font backend will look like.

The primitives \pdfpagewidth and \pdfpageheight have been removed because \pagewidth and \pageheight have that purpose.

The primitives \pdfnormaldeviate, \pdfuniformdeviate, \pdfsetrandomseed and \pdfrandomseed have been promoted to core primitives without pdf prefix so the original commands are no longer recognized.

The primitives \ifincsname, \expanded and \quitvmode are now core primitives.

As the hz 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 replace their prefixed with \pdf originals.

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 more efficient (pdf) code is generated.

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. With value of 1, font expansion is applied after \TeX{}’s normal paragraph breaking routines have broken the paragraph into lines. In this case, line breaks are identical to standard \TeX{} behavior (as with pdf\TeX{}).

The \tagcode primitive is promoted to core primitive.

The \letterspacefont feature is now part of the core but will not be changed (improved). We just provide it for legacy use.

The \pdfnoligatures primitive is now \ignoreligaturesinfont.

The \pdfcopyfont primitive is now \copyfont.

The \pdffontexpand primitive is now \expandglyphsinfont.

Because position tracking is also available in dvi mode the \savepos, \lastxpos and \lastypos commands now replace their pdf prefixed originals.

The introspective primitives \pdflastximagecolordepth and \pdfximagebbox have been removed. One can use external applications to determine these properties or use the built-in...
The initializers `\pdfoutput` has been replaced by `\outputmode` and `\pdfdraftmode` is now `\draftmode`.

The pixel multiplier dimension `\pdfpxdimen` lost its prefix and is now called `\pxdimen`.

An extra `\pdfimageaddfilename` option has been added that can be used to block writing the filename to the pdf file.

The primitive `\pdftracingfonts` is now `\tracingfonts` as it doesn’t relate to the backend.

The experimental primitive `\pdfinserttht` is kept as `\inserttht`.

There is some more control over what metadata goes into the pdf file.

The promotion of primitives to core primitives as well as the separation of font- and backend means that the initialization namespace `pdftex` is gone.

One change involves the so called xforms and ximages. In pdfTeX these are implemented as so called whatsis. But contrary to other whatsis they have dimensions that need to be taken into account when for instance calculating optimal line breaks. In LuaTeX these are now promoted to a special type of rule nodes, which simplifies code that needs those dimensions.

Another reason for promotion is that these are useful concepts. Backends can provide the ability to use content that has been rendered in several places, and images are also common. As already mentioned in section 2.9.4, we now have:

```
\texttt{LUATEX} \hspace{1cm} \texttt{PDFTEX}
\begin{tabular}{ll}
\texttt{\saveboxresource} & \texttt{\pdfxform} \\
\texttt{\saveimageresource} & \texttt{\pdfximage} \\
\texttt{\useboxresource} & \texttt{\pdfrefxform} \\
\texttt{\useimageresource} & \texttt{\pdfrefximage} \\
\texttt{\lastsavedboxresourceindex} & \texttt{\pdflastxform} \\
\texttt{\lastsavedimageresourceindex} & \texttt{\pdflastximage} \\
\texttt{\lastsavedimageresourcepages} & \texttt{\pdflastximagepages} \\
\end{tabular}
```

There are a few `\pdffeedback` features that relate to this but these are typical backend specific ones. The index that gets returned is to be considered as ‘just a number’ and although it still has the same meaning (object related) as before, you should not depend on that.

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?»

### 3.1.5 Changes from ALEPH RC4

Because we wanted proper directional typesetting the Aleph mechanisms looked most attractive. These are rather close to the ones provided by Omega, so what we say next applies to both these programs.

- The extended 16-bit math primitives (\omathcode etc.) have been removed.
- The OCP processing has been removed completely and as a consequence, the following primitives have been removed: \ocp, \externalocp, \ocplist, \pushocplist, \popocplist, \clearocplists, \addbeforeocplist, \addafterocplist, \removebeforeocplist, \removeafterocplist and \ocptracelevel.
- LuaTeX only understands 4 of the 16 direction specifiers of Aleph: TLT (latin), TRT (arabic), RTT (cjk), LTL (mongolian). All other direction specifiers generate an error. In addition to a keyword driven model we also provide an integer driven one.
- The input translations from Aleph are not implemented, the related primitives are not available: \DefaultInputMode, \noDefaultInputMode, \noInputMode, \InputMode, \DefaultOutputMode, \noDefaultOutputMode, \noOutputMode, \OutputMode, \DefaultInputTranslation, \noDefaultInputTranslation, \InputTranslation, \DefaultOutputTranslation, \noDefaultOutputTranslation, \noOutputTranslation and \OutputTranslation.
- Several bugs have been fixed and confusing implementation details have been sorted out.
- The scanner for direction specifications now allows an optional space after the direction is completely parsed.
- 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.
- Several mechanisms that need to be right-to-left aware have been improved. For instance placement of formula numbers.
- The page dimension related primitives \pagewidth and \pageheight have been promoted to core primitives. The \hoffset and \voffset primitives have been fixed.
- The primitives \charwd, \charht, \chardp and \charit have been removed as we have the \TeX variants \fontchar*.
- The two dimension registers \pagerightoffset and \pagebottomoffset are now core primitives.
- The direction related primitives \pagedir, \bodydir, \pardir, \textdir, \mathdir and \boxdir are now core primitives.
- The promotion of primitives to core primitives as well as removing of all others means that the initialization namespace aleph that early versions of LuaTeX provided is gone.
The above let’s itself summarize as: we took the 32 bit aspects and much of the directional mechanisms and merged it into the pdf\TeX\ code base as starting point for further development. Then we simplified directionality, fixed it and opened it up.

### 3.1.6 Changes from anywhere

The \partokenname and \partokencontext primitives are taken from the pdf\TeX change file posted on the implementers list. They are explained in the pdf\TeX manual and are classified as $\varepsilon$-\TeX extensions.

### 3.1.7 Changes from standard WEB2C

The compilation framework is web2c and we keep using that but without the Pascal to C step. This framework also provides some common features that deal with reading bytes from files and locating files in tds. This is what we do different:

- There is no mltex support.
- There is no enctex support.
- The following encoding related command line switches are silently ignored, even in non-Lua mode: -8bit, -translate-file, -mltex, -enc and -etex.
- The \openout what sits are not written to the log file.
- Some of the so-called web2c extensions are hard to set up in non-kpse mode because \texmf.cnf is not read: shell-escape is off (but that is not a problem because of Lua’s os.execute), and the paranoia checks on openin and openout do not happen. However, it is easy for a Lua script to do this itself by overloading io.open and alike.
- The ‘E’ option does not do anything useful.

### 3.2 The backend primitives

#### 3.2.1 Less primitives

In a previous section we mentioned that some pdf\TeX primitives were removed and others promoted to core Lua\TeX primitives. That is only part of the story. In order to separate the backend specific primitives in de code these commands are now replaced by only a few. In traditional \TeX we only had the dvi backend but now we have two: dvi and pdf. Additional functionality is implemented as ‘extensions’ in \TeX speak. By separating more strickly we are able to keep the core (frontend) clean and stable and isolate these extensions. If for some reason an extra backend option is needed, it can be implemented without touching the core. The three pdf backend related primitives are:

```
\pdfextension command [specification]
\pdfvariable name
\pdffeedback name
```

An extension triggers further parsing, depending on the command given. A variable is a (kind of) register and can be read and written, while a feedback is reporting something (as it comes from the backend it’s normally a sequence of tokens).
3.2.2 \texttt{pdftexversion}, \texttt{pdfvariable} and \texttt{pdffeedback}

In order for Lua\TeX{} to be more than just T\Ex{} you need to enable primitives. That has already been the case right from the start. If you want the traditional \texttt{pdfTEx} primitives (for as far their functionality is still around) you now can do this:

\begin{verbatim}
\protected\def\pdfliteral \{\pdffeedback literal\}
\protected\def\pdflateliteral \{\pdffeedback lateliteral\}
\protected\def\pdfcolorstack \{\pdffeedback colorstack\}
\protected\def\pdfsetmatrix \{\pdffeedback setmatrix\}
\protected\def\pdfsave \{\pdffeedback save\relax\}
\protected\def\pdfrestore \{\pdffeedback restore\relax\}
\protected\def\pdfobj \{\pdffeedback obj \}
\protected\def\pdfrefobj \{\pdffeedback refobj \}
\protected\def\pdfannot \{\pdffeedback annot \}
\protected\def\pdfstartlink \{\pdffeedback startlink \}
\protected\def\pdfendlink \{\pdffeedback endlink\relax\}
\protected\def\pdfoutline \{\pdffeedback outline \}
\protected\def\pdfdest \{\pdffeedback dest \}
\protected\def\pdfthread \{\pdffeedback thread \}
\protected\def\pdfstartthread \{\pdffeedback startthread \}
\protected\def\pdfendthread \{\pdffeedback endthread\relax\}
\protected\def\pdfinfo \{\pdffeedback info \}
\protected\def\pdfcatalog \{\pdffeedback catalog \}
\protected\def\pdfnames \{\pdffeedback names \}
\protected\def\pdfincludechars \{\pdffeedback includechars \}
\protected\def\pdffontattr \{\pdffeedback fontattr \}
\protected\def\pdfmapfile \{\pdffeedback mapfile \}
\protected\def\pdfmapline \{\pdffeedback mapline \}
\protected\def\pdftrailer \{\pdffeedback trailer \}
\protected\def\pdfglyphtounicode \{\pdffeedback glyphtounicode \}
\protected\def\pdfrunninglinkoff \{\pdffeedback linkstate 1 \}
\protected\def\pdfrunninglinkon \{\pdffeedback linkstate 0 \}
\end{verbatim}

The introspective primitives can be defined as:

\begin{verbatim}
\def\pdftexversion \{\numexpr\pdffeedback version\relax\}
\def\pdftexrevision \{\pdffeedback revision\}
\def\pdflastlink \{\numexpr\pdffeedback lastlink\relax\}
\def\pdfretval \{\numexpr\pdffeedback retval\relax\}
\def\pdflastobj \{\numexpr\pdffeedback lastobj\relax\}
\def\pdflastannot \{\numexpr\pdffeedback lastannot\relax\}
\def\pdfxformname \{\numexpr\pdffeedback xformname\relax\}
\def\pdfcreationdate \{\pdffeedback creationdate\}
\def\pdffontname \{\numexpr\pdffeedback fontname\relax\}
\def\pdffontbnum \{\numexpr\pdffeedback fontbnum\relax\}
\def\pdffontsize \{\dimexpr\pdffeedback fontsize\relax\}
\def\pdfpageref \{\numexpr\pdffeedback pageref\relax\}
\end{verbatim}
The configuration related registers have become:

\edef\pdfcompresslevel \{\pdfvariable compresslevel\}
\edef\pdfobjcompresslevel \{\pdfvariable objcompresslevel\}
\edef\pdfrecompress \{\pdfvariable recompress\}
\edef\pdfdecimaldigits \{\pdfvariable decimaldigits\}
\edef\pdfgamma \{\pdfvariable gamma\}
\edef\pdfimageresolution \{\pdfvariable imageresolution\}
\edef\pdfimageapplygamma \{\pdfvariable imageapplygamma\}
\edef\pdfimagegamma \{\pdfvariable imagegamma\}
\edef\pdfimagehicolor \{\pdfvariable imagehicolor\}
\edef\pdfimageaddfilename \{\pdfvariable imageaddfilename\}
\edef\pdfpkresolution \{\pdfvariable pkresolution\}
\edef\pdfpkfixeddpi \{\pdfvariable pkfixeddpi\}
\edef\pdfinclusioncopyfonts \{\pdfvariable inclusioncopyfonts\}
\edef\pdfinclusionerrorlevel \{\pdfvariable inclusionerrorlevel\}
\edef\pdfignoreunknownimages \{\pdfvariable ignoreunknownimages\}
\edef\pdfgentounicode \{\pdfvariable gentounicode\}
\edef\pdfomitcidset \{\pdfvariable omitcidset\}
\edef\pdfomitcharset \{\pdfvariable omitcharset\}
\edef\pdfomitinfodict \{\pdfvariable omitinfodict\}
\edef\pdfomitmediabox \{\pdfvariable omitmediabox\}
\edef\pdfpagebox \{\pdfvariable pagebox\}
\edef\pdfminorversion \{\pdfvariable minorversion\}
\edef\pdfuniqueresname \{\pdfvariable uniqueresname\}
\edef\pdfhorigin \{\pdfvariable horigin\}
\edef\pdfvorigin \{\pdfvariable vorigin\}
\edef\pdflinkmargin \{\pdfvariable linkmargin\}
\edef\pdfdestmargin \{\pdfvariable destmargin\}
\edef\pdfthreadmargin \{\pdfvariable threadmargin\}
\edef\pdfxformmargin \{\pdfvariable xformmargin\}
\edef\pdfpagesattr \{\pdfvariable pagesattr\}
\edef\pdfpageattr \{\pdfvariable pageattr\}
\edef\pdfpageresources \{\pdfvariable pageresources\}
\edef\pdfxformattr \{\pdfvariable xformattr\}
\edef\pdfxformresources \{\pdfvariable xformresources\}
\edef\pdfpkmode \{\pdfvariable pkmode\}
\edef\pdfsuppressoptionalinfo \{\pdfvariable suppressoptionalinfo\}
\edef\pdftrailerid \{\pdfvariable trailerid\}

The variables are internal ones, so they are anonymous. When you ask for the meaning of a few previously defined ones:
\meaning\pdfhorigin
\meaning\pdfcompresslevel
\meaning\pdfpageattr

you will get:

macro:->[internal backend dimension]
macro:->[internal backend integer]
macro:->[internal backend tokenlist]

The \edef can also be a \def but it’s a bit more efficient to expand the lookup related register beforehand.

The backend is derived from pdf\TeX\ so the same syntax applies. However, the \texttt{outline} command accepts a \texttt{objnum} followed by a number. No checking takes place so when this is used it had better be a valid (flushed) object.

In order to be (more or less) compatible with pdf\TeX\ we also support the option to suppress some info but we do so via a bitset:

\pdfvariable suppressoptionalinfo \numexpr
  0 + 1 \% PTEX.FullBanner + 2 \% PTEX.FileName + 4 \% PTEX.PageNumber + 8 \% PTEX.InfoDict + 16 \% Creator + 32 \% CreationDate + 64 \% ModDate + 128 \% Producer + 256 \% Trapped + 512 \% ID \relax

In addition you can overload the trailer id, but we don’t do any checking on validity, so you have to pass a valid array. The following is like the ones normally generated by the engine. You even need to include the brackets here!

\pdfvariable trailerid {
  <FA052949448907805BA83C1E78896398>
  <FA052949448907805BA83C1E78896398>
}

Although we started from a merge of pdf\TeX\ and Aleph, by now the code base as well as functionality has diverted from those parents. Here we show the options that can be passed to the extensions. The \texttt{shipout} option is a compatibility feature. Instead one can use the \texttt{deferred} prefix.

\pdfextension literal
[shipout] [ direct | page | raw ] { tokens }
\pdfeextensio dest
num integer | name { tokens }!crlf
[ fitbh | fitbv | fitb | fith | fitv | fit |
fitr <rule spec> | xyz [ zoom <integer> ]

\pdfeextensio annot
reserveobjnum | useobjnum <integer>
{ tokens }

\pdfeextensio save
\pdfeextensio restore
\pdfeextensio setmatrix
{ tokens }
[
\immediate \pdfeextensio obj
reserveobjnum
[
\immediate \pdfeextensio obj
[ useobjnum <integer> ]
[ uncompressed ]
[ stream [ attr { tokens } ] ]
[ file ]
{ tokens }

\pdfeextensio refobj
<integer>

\pdfeextensio colorstack
<integer>
set { tokens } | push { tokens } | pop | current

\pdfeextensio startlink
[ attr { tokens } ]
user { tokens } | goto | thread
[ file { tokens } ]
[ page <integer> { tokens } | name { tokens } | num integer ]
[ newwindow | nonewwindow ]

\pdfeextensio endlink

\pdfeextensio startthread
num <integer> | name { tokens }

\pdfeextensio endthread

\pdfeextensio thread
num <integer> | name { tokens }

\pdfeextensio outline
3.2.3 Defaults

The engine sets the following defaults.

\pdfcompresslevel 9
\pdfobjcompresslevel 1 \% used: (0,9)
\pdfrecompress 0 \% mostly for debugging
\pdfdecimaldigits 4 \% used: (3,6)
\pdfgamma 1000
\pdfimageresolution 71
3.2.4 Backward compatibility

If you also want some backward compatibility, you can add:

\let\pdfpagewidth \pagewidth
\let\pdfpageheight \pageheight
\let\pdfadjustspacing \adjustspacing
\let\pdfprotrudechars \protrudechars
\let\pdfnoligatures \ignoreligaturesinfont
\let\pdffontexpand \expandglyphsinfont
\let\pdfcopyfont \copyfont
\let\pdfxform \saveboxresource
\let\pdflastxform \lastsavedboxresourceindex
\let\pdfrefxform \useboxresource
\let\pdfximage \saveimageresource
\let\pdflastximage \lastsavedimageresourceindex
\let\pdflastximagepages \lastsavedimageresourcepages
\let\pdfrefximage \useimageresource
And even:

\newdimen\pdfeachlineheight
\newdimen\pdfeachlinddepth
\newdimen\pdflastlinedepth
\newdimen\pdffirstlineheight
\newdimen\pdfignoreddimen

3.3 Directions

3.3.1 Four directions

The directional model in LuaTeX is inherited from Omega/Aleph but we tried to improve it a bit. At some point we played with recovery of modes but that was disabled later on when we found that it interfered with nested directions. That itself had as side effect that the node list was no longer balanced with respect to directional nodes which in turn can give side effects when a series of dir changes happens without grouping.

When extending the pdf backend to support directions some inconsistencies were found and as a result we decided to support only the four models that make sense TLT (latin), TRT (arabic), RTT (cjk) and LTL (mongolian).

3.3.2 How it works

The approach is that we again make the list balanced but 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 proper 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 dir
injection.

This is what happens:

\textdir TRT nur {\textdir TLT run \textdir TRT NUR} nur

This becomes stepwise:

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

And this:

\textdir TRT nur {nur \textdir TLT run \textdir TRT 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 {RUNrun } run

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

\textdir TRT nur {{\textdir TLT run} {\textdir TRT NUR}} nur

This becomes:

- run RUN run run

Compare this to:

\textdir TRT nur {{{\textdir TLT run }{\textdir TRT NUR}}} nur

Which renders as:

- run RUNrun run

So how do we deal with the next?

\def\ltr{\textdir TLT}\relax
\def\rtl{\textdir TRT}\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 lose the subtype information that for some applications can be handy to be kept as-is. This is why we now have a variant of \textdir which injects the balanced node before the skip. Instead of the previous definition we can use:

\def\ltr{\linedir TLT\relax}
\def\rtl{\linedir TRT\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 \linedir 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 handled as paragraphs with content.

### 3.3.3 Controlling glue with \breakafterdirmode

Glue after a dir node is ignored in the linebreak decision but you can bypass that by setting \breakafterdirmode to 1. The following table shows the difference. Watch your spaces.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre {\textdir TLT xxx} post</td>
<td>pre</td>
<td>pre</td>
</tr>
<tr>
<td></td>
<td>xxx post</td>
<td>xxx post</td>
</tr>
<tr>
<td>pre {\textdir TLT xxx }post</td>
<td>pre</td>
<td>pre</td>
</tr>
<tr>
<td></td>
<td>xxx</td>
<td>xxx</td>
</tr>
<tr>
<td></td>
<td>post</td>
<td>post</td>
</tr>
<tr>
<td>pre{ \textdir TLT xxx} post</td>
<td>pre</td>
<td>pre</td>
</tr>
<tr>
<td></td>
<td>xxx post</td>
<td>xxx post</td>
</tr>
<tr>
<td>pre{ \textdir TLT xxx }post</td>
<td>pre</td>
<td>pre</td>
</tr>
<tr>
<td></td>
<td>xxx</td>
<td>xxx</td>
</tr>
<tr>
<td></td>
<td>post</td>
<td>post</td>
</tr>
</tbody>
</table>
3.3.4 Controlling parshapes with \textbackslash{shapemode}

Another adaptation to the Aleph directional model is control over shapes driven by \textbackslash{hangindent} and \textbackslash{parshape}. This is controlled by a new parameter \textbackslash{shapemode}:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>\textbackslash{HANGINDENT}</th>
<th>\textbackslash{PARSHAPE}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal</td>
<td>normal</td>
</tr>
<tr>
<td>1</td>
<td>mirrored</td>
<td>normal</td>
</tr>
<tr>
<td>2</td>
<td>normal</td>
<td>mirrored</td>
</tr>
<tr>
<td>3</td>
<td>mirrored</td>
<td>mirrored</td>
</tr>
</tbody>
</table>

The value is reset to zero (like \textbackslash{hangindent} and \textbackslash{parshape}) after the paragraph is done with. You can use negative values to prevent this. In figure 3.1 a few examples are given.

3.3.5 Symbols or numbers

Internally the implementation is different from Aleph. First of all we use no whatssits but dedicated nodes, but also we have only 4 directions that are mapped onto 4 numbers. A text direction node can mark the start or end of a sequence of nodes, and therefore has two states. At the T\TeX{} end we don’t see these states because T\TeX{} itself will add proper end state nodes if needed.

The symbolic names TLT, TRT, etc. originate in Omega. In LuaT\TeX{} we also have a number based model which sometimes makes more sense.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EQUIVALENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TLT</td>
</tr>
<tr>
<td>1</td>
<td>TRT</td>
</tr>
<tr>
<td>2</td>
<td>LTL</td>
</tr>
<tr>
<td>3</td>
<td>RTT</td>
</tr>
</tbody>
</table>

We support the Omega primitives \textbackslash{textdir}, \textbackslash{pardir}, \textbackslash{pagedir}, \textbackslash{pardir} and \textbackslash{mathdir}. These accept three character keywords. The primitives that set the direction by number are: \textbackslash{textdirection}, \textbackslash{pardirection}, \textbackslash{pagedirection} and \textbackslash{bodydirection} and \textbackslash{mathdirection}. When specifying a direction for a box you can use bdir instead of dir.
The effect of shapemode.

### 3.4 Implementation notes

#### 3.4.1 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.

The `texmf.cnf` settings related to main memory are no longer used (these are: `main_memory`, `mem_bot`, `extra_mem_top` and `extra_mem_bot`). ‘Out of main memory’ errors can still occur, but the limiting factor is now the amount of RAM in your system, not a predefined limit.

Also, the memory (de)allocation routines for nodes are completely rewritten. The relevant code now lives in the C file `texnode.c`, and basically uses a dozen or so ‘avail’ lists instead of a doubly-linked model. An extra function layer is added so that the code can ask for nodes by type instead of directly requisitioning a certain amount of memory words.

Because of the split into two arrays and the resulting differences in the data structures, some of the macros have been duplicated. For instance, there are now `vlink` and `vinfo` as well as `token_link` and `token_info`. All access to the variable memory array is now hidden behind a macro called `vmem`. We mention this because using the \TeX{} as reference is still quite valid.
but not for memory related details. Another significant detail is that we have double linked node lists and that most nodes carry more data.

The input line buffer and pool size are now also reallocated when needed, and the `texmf.cnf` settings `buf_size` and `pool_size` are silently ignored.

### 3.4.2 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 `\ETeX` tracing routines `\tracingassigns` and `\tracingrestores` but we don’t see that as a real limitation.

A side-effect of the current implementation is that `\global` is now more expensive in terms of processing than non-global assignments but not many users will notice that.

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.

### 3.4.3 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 csnames.

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.

Active characters are internally implemented as a special type of multi-letter control sequences that uses a prefix that is otherwise impossible to obtain.

### 3.4.4 The compressed format file

The format is passed through zlib, allowing it to shrink to roughly half of the size it would have had in uncompressed form. This takes a bit more cpu cycles but much less disk io, so it should still be faster. We use a level 3 compression which we found to be the optimal trade-off between filesize and decompression speed.

### 3.4.5 Binary file reading

All of the internal code is changed in such a way that if one of the `read_xxx_file` callbacks is not set, then the file is read by a C function using basically the same convention as the callback: a single read into a buffer big enough to hold the entire file contents. While this uses more memory than the previous code (that mostly used `getc` calls), it can be quite a bit faster (depending on your io subsystem).
3.4.6 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 \TeX{}Live 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 Lua\TeX{} 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 and the fact that in kpse mode a common file io layer is used can have a side effect of breaking compatibility. We still stick to our view that at the log level we can (and might be) more incompatible. We already expose some more details.

3.4.7 Hyperlinks

There is an experimental feature that makes multi-line hyper links behave a little better, fixing some side effects that showed up in r2l typesetting but also can surface in l2r. Because this got unnoticed till 2023, and because it depends bit on how macro packages deal with hyper links, the fix is currently under parameter control:

\texttt{pdfvariable linking = 1}

That way (we hope) legacy documents come out as expected, whatever those expectations are. One of the aspects dealt with concerns (unusual) left and right skips.
4 Using \text{LUATEX}

4.1 Initialization

4.1.1 \text{LUATEX} as a LUA interpreter

There are some situations that make Lua\TeX{} behave like a standalone Lua interpreter:

- if a \texttt{--luaonly} option is given on the command line, or
- if the executable is named \texttt{texlua} or \texttt{luatexlua}, or
- if the only non-option argument (file) on the command line has the extension \texttt{lua} or \texttt{luc}.

In this mode, it will set Lua’s \texttt{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.

Lua\TeX{} 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.

4.1.2 \text{LUATEX} as a LUA byte compiler

There are two situations that make Lua\TeX{} behave like the Lua byte compiler:

- if a \texttt{--luaonly} option is given on the command line, or
- if the executable is named \texttt{texluac}

In this mode, Lua\TeX{} is exactly like \texttt{luac} from the stand alone Lua distribution, except that it does not have the \texttt{-l} switch, and that it accepts (but ignores) the \texttt{--luaonly} switch. The current version of Lua can dump bytecode using \texttt{string.dump} so we might decide to drop this version of Lua\TeX{}.

4.1.3 Other commandline processing

When the Lua\TeX{} executable starts, it looks for the \texttt{--lua} command line option. If there is no \texttt{--lua} option, the command line is interpreted in a similar fashion as the other \TeX{} engines. Some options are accepted but have no consequence. The following command-line options are understood:

<table>
<thead>
<tr>
<th>COMMANDLINE ARGUMENT</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{--credits}</td>
<td>display credits and exit</td>
</tr>
<tr>
<td>\texttt{--debug-format}</td>
<td>enable format debugging</td>
</tr>
<tr>
<td>\texttt{--draftmode}</td>
<td>switch on draft mode i.e. generate no output in pdf mode</td>
</tr>
<tr>
<td>\texttt{--[no-]file-line-error}</td>
<td>disable/enable file:line:.error style messages</td>
</tr>
<tr>
<td>\texttt{--[no-]file-line-error-style}</td>
<td>aliases of \texttt{--[no-]file-line-error}</td>
</tr>
<tr>
<td>\texttt{--fmt=FORMAT}</td>
<td>load the format file \texttt{FORMAT}</td>
</tr>
</tbody>
</table>
--halt-on-error  stop processing at the first error
--help          display help and exit
--ini           be initlualatex, for dumping formats
--interaction=STRING set interaction mode: batchmode, nonstopmode, scrollmode or errorstopmode
--jobname=STRING set the job name to STRING
--kpathsea-debug=NUMBER set path searching debugging flags according to the bits of NUMBER
--lua=FILE      load and execute a Lua initialization script
--[no-]mktex=FMT disable/enable mktexFMT generation with FMT is tex or tfm
--nosocket      disable the Lua socket library
--no-socket     disable the Lua socket library
--socket        enable the Lua socket library
--output-comment=STRING use STRING for dvi file comment instead of date (no effect for pdf)
--output-directory=DIR use DIR as the directory to write files to
--output-format=FORMAT use FORMAT for job output; FORMAT is dvi or pdf
--prognam=STRING set the program name to STRING
--recorder      enable filename recorder
--safer         disable easily exploitable Lua commands
--[no-]shell-escape disable/enable system calls
--shell-restricted restrict system calls to a list of commands given in texmf.cnf
--synctex=NUMBER enable synctex
--utc           use utc times when applicable
--version       display version and exit

We don’t support \write 18 because os.execute can do the same. It simplifies the code and makes more write targets possible.

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.

The file names for output files that are generated automatically are created by attaching the proper extension (log, pdf, etc.) to the found \jobname. These files are created in the directory pointed to by --output-directory, or in the current directory, if that switch is not present.

Without the --lua option, command line processing works like it does in any other web2c-based typesetting engine, except that LuaTeX has a few extra switches and lacks some others. Also, if the --lua option is present, LuaTeX will enter an alternative mode of command line processing in comparison to the standard web2c programs. In this mode, a small series of actions is taken in the following order:
1. First, it will parse the command line as usual, but it will only interpret a small subset of the options immediately: --safer, --nosocket, --no-socket, --socket, --[no-]shell-escape, --enable-write18, --disable-write18, --shell-restricted, --help, --version, and --credits.

2. Next LuaTeX searches for the requested Lua initialization script. If it cannot be found using the actual name given on the command line, a second attempt is made by prepending the value of the environment variable LUATEXDIR, if that variable is defined in the environment.

3. Then it checks the various safety switches. You can use those to disable some Lua commands that can easily be abused by a malicious document. At the moment, --safer nils the following functions:

<table>
<thead>
<tr>
<th>LIBRARY</th>
<th>FUNCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>os</td>
<td>execute, exec, kpseopen, spawn, setenv, rename, remove, tmpdir</td>
</tr>
<tr>
<td>io</td>
<td>popen, output, tmpfile</td>
</tr>
<tr>
<td>lfs</td>
<td>rmdir, mkdir, chdir, lock, touch</td>
</tr>
</tbody>
</table>

Furthermore, it disables loading of compiled Lua libraries and it makes \texttt{io.open()} fail on files that are opened for anything besides reading.

Finally, it disables the socket library unconditionally (but not the \texttt{mime} library which is always available).

4. When LuaTeX starts it sets the locale to a neutral value. If for some reason you use \texttt{os.locale}, you need to make sure you \texttt{nil} it afterwards because otherwise it can interfere with code that for instance generates dates. You can ignore the locale with:

\texttt{os.setlocale(nil,nil)}

The --nosocket or --no-socket option makes the socket library unavailable, so that Lua cannot use networking; --socket option makes the socket library available.

The switches --[no-]shell-escape, --[enable|disable]-write18, and --shell-restricted have the same effects as in \texttt{pdfTeX}, and additionally make \texttt{io.popen()}, \texttt{os.execute}, \texttt{os.exec}, \texttt{os.kpseopen} and \texttt{os.spawn} adhere to the requested option.

By default the socket library is not enabled: one can enable it with with --socket or with --shell-escape (but without --shell-restricted) and disable it with --nosocket (or --no-socket) or unconditionally with --safer.

In case of conflictual options, the most restrictive wins.

The \texttt{mime} library is always available.

5. Next the initialization script is loaded and executed. From within the script, the entire command line is available in the Lua table \texttt{arg}, beginning with \texttt{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. For that reason, the tables that deal with typesetting, like \texttt{tex}, \texttt{token}, \texttt{node} and \texttt{pdf}, are off-limits during the execution of the startup file (they are nil’d). Special care is taken that \texttt{texio.write} and \texttt{texio.write_nl} function properly, so that you can at least report your actions to the log file when (and if) it eventually becomes opened (note that \TeX does not even know its \texttt{\jobname} yet at this point).

Everything you do in the Lua initialization script will remain visible during the rest of the run, with the exception of the \TeX specific libraries like \texttt{tex}, \texttt{token}, \texttt{node} and \texttt{pdf} tables. These will
be initialized to their documented state after the execution of the script. You should not store anything in variables or within tables with these four global names, as they will be overwritten completely.

We recommend you use the startup file only for your own \TeX-independent initializations (if you need any), to parse the command line, set values in the texconfig table, and register the callbacks you need.

\LaTeX 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).

Unless the texconfig table tells \LaTeX not to initialize kpathsea at all (set texconfig.kpse_init to false for that), \LaTeX acts on some more command line options after the initialization script is finished: in order to initialize the built-in kpathsea library properly, \LaTeX needs to know the correct program name to use, and for that it needs to check --progname, or --ini and --fmt, if --progname is missing.

### 4.2 LUA behaviour

#### 4.2.1 The LUA version

We currently use Lua 5.3 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. Also, Luajit\LaTeX lags behind in the sense that Luajit is not in sync with regular Lua development. Here is an example of one aspect.

Lua's 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 \texttt{\dimen} or \texttt{\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.

#### 4.2.2 Integration in the TDS ecosystem

The main \TeX distributions follow the \TeX directory structure (tds). \LaTeX is able to use the kpathsea library to find require()d modules. For this purpose, package.searchers[2] is replaced by a different loader function, that decides at runtime whether to use kpathsea or the built-in core Lua function. It uses kpathsea when that is already initialized at that point in time, otherwise it reverts to using the normal package.path loader.

Initialization of kpathsea can happen either implicitly (when \LaTeX starts up and the startup script has not set texconfig.kpse_init to false), or explicitly by calling the Lua function kpse.set_program_name().

#### 4.2.3 Loading libraries

\LaTeX is able to use dynamically loadable Lua libraries, unless \texttt{--safer} was given as an option on the command line. For this purpose, package.searchers[3] is replaced by a different loader...
function, that decides at runtime whether to use kpathsea or the built-in core Lua function. It uses kpathsea when that is already initialized at that point in time, otherwise it reverts to using the normal package.cpath loader.

This functionality required an extension to kpathsea. There is a new kpathsea file format: kpse_clua_format that searches for files with extension .dll and .so. The texmf.cnf setting for this variable is CLUAINPUTS, and by default it has this value:

CLUAINPUTS=.$SELFAUTOLOC/lib/{$proname,$engine,}/lua/

This path is imperfect (it requires a tds subtree below the binaries directory), but the architecture has to be in the path somewhere, and the currently simplest way to do that is to search below the binaries directory only. Of course it no big deal to write an alternative loader and use that in a macro package. One level up (a lib directory parallel to bin) would have been nicer, but that is not doable because TeXLive uses a bin/<arch> structure.

Loading dynamic Lua libraries will fail if there are two Lua libraries loaded at the same time (which will typically happen on win32, because there is one Lua 5.3 inside LuaTeX, and another will likely be linked to the dll file of the module itself).

4.2.4 Executing programs

In keeping with the other TeX-like programs in TeXLive, the Lua functions os.execute, os.kpsepopen and io.popen, as well as the two new functions os.exec and os.spawn that are explained below, take the value of shell_escape and/or shell_escape_commands in account. Whenever LuaTeX is run with the assumed intention to typeset a document (and by that we mean that it is called as luatex, as opposed to texlua, and that the command line option --luaonly was not given), it will only run the four functions above if the matching texmf.cnf variable(s) or their texconfig (see section 10.4) counterparts allow execution of the requested system command. In 'script interpreter' runs of LuaTeX, these settings have no effect, and all four functions have their original meaning.

Some libraries have a few more functions, either coded in C or in Lua. For instance, when we started with LuaTeX we added some helpers to the luafilesystem namespace lfs. The two boolean functions lfs.isdir and lfs.isfile were speedy and better variants of what could be done with lfs.attributes. The additional function lfs.shortname takes a file name and returns its short name on win32 platforms. Finally, for non-win32 platforms only, we provided lfs.readlink that takes an existing symbolic link as argument and returns its name. However, the lfs library evolved so we have dropped these in favour of pure Lua variants. The shortname helper is obsolete and now just returns the name.

4.2.5 Multibyte string functions

The string library has a few extra functions, for example string.explode. This function takes up to 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.

- \texttt{string.utfvalues(s)}: an integer value in the Unicode range
- \texttt{string.utfcharacters(s)}: a string with a single utf-8 token in it
- \texttt{string.characters(s)}: a string containing one byte
- \texttt{string.characterpairs(s)}: two strings each containing one byte or an empty second string if the string length was odd
- \texttt{string.bytes(s)}: a single byte value
- \texttt{string.bytepairs(s)}: two byte values or nil instead of a number as its second return value if the string length was odd

The \texttt{string.characterpairs()} and \texttt{string.bytepairs()} iterators are useful especially in the conversion of utf16 encoded data into utf8.

There is also a two-argument form of \texttt{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 \texttt{len}, \texttt{lower}, \texttt{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 slunicode library can be used, e.g., \texttt{unicode.utf8.len}, \texttt{unicode.utf8.lower} etc. The exceptions are \texttt{unicode.utf8.find}, that always returns byte positions in a string, and \texttt{unicode.utf8.match} and \texttt{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 \texttt{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: \texttt{string.utfvalue}, \texttt{string.utfcharacter} and \texttt{string.utflength}.

- \texttt{string.utfvalue(s)}: returns the codepoints of the characters in the given string
- \texttt{string.utfcharacter(c,...)}: returns a string with the characters of the given code points
- \texttt{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.

### 4.2.6 Extra os library functions

The os library has a few extra functions and variables: \texttt{os.selfdir}, \texttt{os.exec}, \texttt{os.kpsopen}, \texttt{os.socketgettime}, \texttt{os.socketsleep}, \texttt{os.spawn}, \texttt{os.setenv}, \texttt{os.env}, \texttt{os_gettimeofday}, \texttt{os.times}, \texttt{os.sleep}, \texttt{os.tmpdir}, \texttt{os.type}, \texttt{os.name} and \texttt{os.uname},\texttt{os uname}, that we will discuss here.
\begin{itemize}
\item \texttt{os.selfdir} is a variable that holds the directory path of the actual executable. For example: \texttt{\directlua{tex.sprint(os.selfdir)}}.
\item \texttt{os.exec(commandline)} is a variation on \texttt{os.execute}. Here \texttt{commandline} can be either a single string or a single table.
  \begin{itemize}
  \item If the argument is a table Lua\TeX\ first checks if there is a value at integer index zero. If there is, this is the command to be executed. Otherwise, it will use the value at integer index one. If neither are present, nothing at all happens.
  \item The set of consecutive values starting at integer 1 in the table are the arguments that are passed on to the command (the value at index 1 becomes \texttt{arg[0]}). The command is searched for in the execution path, so there is normally no need to pass on a fully qualified path name.
  \item If the argument is a string, then it is automatically converted into a table by splitting on whitespace. In this case, it is impossible for the command and first argument to differ from each other.
  \item In the string argument format, whitespace can be protected by putting (part of) an argument inside single or double quotes. One layer of quotes is interpreted by Lua\TeX, and all occurrences of "", \' or \" within the quoted text are unescaped. In the table format, there is no string handling taking place.
  \end{itemize}
\item This function normally does not return control back to the Lua script: the command will replace the current process. However, it will return the two values \texttt{nil} and \texttt{error} if there was a problem while attempting to execute the command.
\item On MS Windows, the current process is actually kept in memory until after the execution of the command has finished. This prevents crashes in situations where \TeX\Lua\ scripts are run inside integrated \TeX\ environments.
\item The original reason for this command is that it cleans out the current process before starting the new one, making it especially useful for use in \TeX\Lua.
\item \texttt{os.kpsepopen(commandline,[opt])} is similar to \texttt{io.popen} but with a preliminary check of the command line; if the check is ok then the return value is the same as in \texttt{io.popen}; Otherwise it will return the two values \texttt{nil} and \texttt{error}.
\item \texttt{os.socketgettime} and \texttt{ossocketsleep} are the same as for \texttt{socket.gettime} and \texttt{socket.sleep} but they are always available.
\item \texttt{os.spawn(commandline)} is a returning version of \texttt{os.exec}, with otherwise identical calling conventions.
\item If the command ran ok, then the return value is the exit status of the command. Otherwise, it will return the two values \texttt{nil} and \texttt{error}.
\item \texttt{os.setenv(key,value)} sets a variable in the environment. Passing \texttt{nil} instead of a value string will remove the variable.
\item \texttt{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 \texttt{not} updated automatically.
\item \texttt{os.gettimeofday()} returns the current ‘Unix time’, but as a float. This function is not available on the SunOS platforms, so do not use this function for portable documents.
\item \texttt{os.times()} returns the current process times according to the Unix C library function ‘times’. This function is not available on the MS Windows and SunOS platforms, so do not use this function for portable documents.
\item \texttt{os.sleep(interval[, unit])} suspends the execution of the current run for a given number
of seconds. If the optional argument unit is present, the function waits interval / units seconds. \texttt{os.sleep(1, 1000)} for example pauses the program for one millisecond.

- \texttt{os.tmpdir([template])} creates a directory in the ‘current directory’ with the name \texttt{lualatex.XXXXXX} where the X-es are replaced by a unique string. The function also returns this string, so you can \texttt{lfs.chdir()} into it, or \texttt{nil} if it failed to create the directory. The user is responsible for cleaning up at the end of the run, it does not happen automatically. You can also use your own template for the name of the temporary folder. However, the passed string must end with six capital X-es. For example, the template \texttt{tmp.XXXXXX} could result in the folder name \texttt{tmp.vX3gPo}.

- \texttt{os.type} is a string that gives a global indication of the class of operating system. The possible values are currently \texttt{windows}, \texttt{unix}, and \texttt{msdos} (you are unlikely to find this value ‘in the wild’).

- \texttt{os.name} is a string that gives a more precise indication of the operating system. These possible values are not yet fixed, and for \texttt{os.type} values \texttt{windows} and \texttt{msdos}, the \texttt{os.name} values are simply \texttt{windows} and \texttt{msdos}

The list for the type \texttt{unix} is more precise: \texttt{linux}, \texttt{freebsd}, \texttt{kfreebsd}, \texttt{cygwin}, \texttt{openbsd}, \texttt{solaris}, \texttt{sunos} (pre-solaris), \texttt{hpux}, \texttt{irix}, \texttt{macosx}, \texttt{gnu} (hurd), \texttt{bsd} (unknown, but \texttt{bsd}-like), \texttt{sysv} (unknown, but \texttt{sysv}-like), \texttt{generic} (unknown).

- \texttt{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: \texttt{sysname}, \texttt{machine}, \texttt{release}, \texttt{version}, and \texttt{nodename}.

4.2.7 Binary input from files with \texttt{fio}

There is a whole set of helpers for reading numbers and strings from a file: \texttt{fio.readcardinal1}, \texttt{fio.readcardinal2}, \texttt{fio.readcardinal3}, \texttt{fio.readcardinal4}, \texttt{fio.readcardinaltable}, \texttt{fio.readinteger1}, \texttt{fio.readinteger2}, \texttt{fio.readinteger3}, \texttt{fio.readinteger4}, \texttt{fio.readintegeritable}, \texttt{fio.readfixed2}, \texttt{fio.readfixed4}, \texttt{fio.read2dot14}, \texttt{fio.setposition}, \texttt{fio.getposition}, \texttt{fio.skipposition}, \texttt{fio.readbytes}, \texttt{fio.readbytetable}. They work on normal Lua file handles.

This library provides a set of functions for reading numbers from a file and in addition to the regular \texttt{io} library functions.

\begin{verbatim}
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
readintegeritable(f,n,b) n integers of b bytes
readfixed2(f)         a 2 byte float (used in font files)
readfixed4(f)         a 4 byte float (used in font files)
read2dot14(f)         a 2 byte float (used in font files)
setposition(f,p)      goto position p
\end{verbatim}
getposition(f) get the current position
skipposition(f,n) skip n positions
readbytes(f,n) n bytes
readbytetable(f,n) n bytes


4.2.8 Binary input from strings with sio

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.readinteger4, sio.readinteger4, sio.readintertable, 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. More details can be found in the previous section.

4.2.9 Hashes conform sha2

This library is a side effect of the pdfe library that needs such helpers. The sha2.digest256, sha2.digest384 and sha2.digest512 functions accept a string and return a string with the hash.

4.2.10 Locales

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

LC_CTYPE=C
LC_COLLATE=C
LC_NUMERIC=C

4.3 LUA modules

Some modules that are normally external to Lua are statically linked in with LuaTeX, because they offer useful functionality:

- lpeg, by Roberto Ierusalimschy, http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html. 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 encoded in more than two bytes, 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('aâ') results in the message bad argument #1 to 'R' (range must have two characters), since to lpeg, å 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.

- slunicode, from the selene libraries, http://luaforge.net/projects/sln. This library has been slightly extended so that the unicode.utf8.* functions also accept the first 256 values of plane 18. This is the range LuaTeX uses for raw binary output, as explained above. We have no plans to provide more like this because you can basically do all that you want in Lua.
- luafilesystem, also from the kepler project, http://www.keplerproject.org/luafilesystem/.
- luasocket, by Diego Nehab http://w3.impa.br/~diego/software/luasocket/. The .lua support modules from luasocket are also preloaded inside the executable, there are no external file dependencies.

### 4.4 Testing

For development reasons you can influence the used startup date and time. This can be done in two ways.

1. By setting the environment variable SOURCE_DATE_EPOCH. This will influence the \TeX\ parameters time and date, the random seed, the pdf timestamp and the pdf id that is derived from the time as well. This variable is consulted when the kpse library is enabled. Resolving is delegated to this library.
2. By setting the `start_time` variable in the `texconfig` table; as with other variables we use the internal name there. For compatibility reasons we also honour a SOURCE_DATE_EPOCH entry. It should be noted that there are no such variables in other engines and this method is only relevant in case the while setup happens in Lua.

When Universal Time is needed, you can pass the flag `utc` to the engine. This property also works when the date and time are set by Lua\TeX\ itself. It has a complementary entry `use_utc_time` in the `texconfig` table.

There is some control possible, for instance prevent filename to be written to the pdf file. This is discussed elsewhere. In ConTeXt we provide the command line argument `--nodates` that does a bit more disabling of dates.
5 Languages, characters, fonts and glyphs

5.1 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.

5.2 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 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 8.2.12, 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.
- ghost, for so-called ‘ghost objects’ bit 2 is set.
- left, for ligatures created from a left word boundary and for ghosts created from \leftghost bit 3 gets set.
- right, for ligatures created from a right word boundary and for ghosts created from \rightghost bit 4 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 it 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 one character. 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
   discrete—discrete

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, ins, and adjust node indicate a start or end. You can omit the last set from the test by setting \hyphenationbounds to a non-zero value:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>BEHAVIOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>not strict</td>
</tr>
<tr>
<td>1</td>
<td>strict start</td>
</tr>
<tr>
<td>2</td>
<td>strict end</td>
</tr>
<tr>
<td>3</td>
<td>strict start and strict end</td>
</tr>
</tbody>
</table>

The word start is determined as follows:

<table>
<thead>
<tr>
<th>NODE</th>
<th>BEHAVIOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary</td>
<td>yes when wordboundary</td>
</tr>
<tr>
<td>hlist</td>
<td>when hyphenationbounds 1 or 3</td>
</tr>
<tr>
<td>vlist</td>
<td>when hyphenationbounds 1 or 3</td>
</tr>
<tr>
<td>rule</td>
<td>when hyphenationbounds 1 or 3</td>
</tr>
<tr>
<td>dir</td>
<td>when hyphenationbounds 1 or 3</td>
</tr>
<tr>
<td>whatsit</td>
<td>when hyphenationbounds 1 or 3</td>
</tr>
<tr>
<td>glue</td>
<td>yes</td>
</tr>
<tr>
<td>math</td>
<td>skipped</td>
</tr>
<tr>
<td>glyph</td>
<td>exhyphenchar (one only) : yes (so no –—)</td>
</tr>
<tr>
<td>otherwise</td>
<td>yes</td>
</tr>
</tbody>
</table>

The word end is determined as follows:

<table>
<thead>
<tr>
<th>NODE</th>
<th>BEHAVIOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundary</td>
<td>yes</td>
</tr>
<tr>
<td>glyph</td>
<td>yes when different language</td>
</tr>
<tr>
<td>glue</td>
<td>yes</td>
</tr>
<tr>
<td>penalty</td>
<td>yes</td>
</tr>
<tr>
<td>kern</td>
<td>yes when not italic (for some historic reason)</td>
</tr>
<tr>
<td>hlist</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>vlist</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>rule</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>dir</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>whatsit</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>ins</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
<tr>
<td>adjust</td>
<td>when hyphenationbounds 2 or 3</td>
</tr>
</tbody>
</table>

Figures 5.1 upto 5.5 show some examples. In all cases we set the min values to 1 and make sure that the words hyphenate at each character.
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).

When `\texttt{automatichyphenmode}` has a value of 0, a hyphen will be turned into an automatic discretionary. The snippets before and after it will not be hyphenated. A side effect is that a leading hyphen can lead to a split but one will seldom run into that situation. Setting a pre and post character makes this more prominent. A value of 1 will prevent this side effect and a value of 2 will not turn the hyphen into a discretionary. Experiments with other options, like permitting hyphenation of the words on both sides were discarded.

In figure 5.6 and 5.7 we show what happens with three samples:

Input A:

before-after \par
before--after \par
Figure 5.5 \null one \null two \null three

Figure 5.6 The automatic modes 0 (default), 1 and 2, with a \hsize of 6em and 2pt (which triggers a linebreak).

before--after \par
Input B:

-before \par
after- \par
--before \par
after-- \par
---before \par
after--- \par

74 Languages, characters, fonts and glyphs
Input C:

before-after \par
before--after \par
before---after \par

As with primitive companions of other single character commands, the \- command has a more verbose primitive version in \explicitdiscretionary and the normally intercepted in the hyphenator character - (or whatever is configured) is available as \automaticdiscretionary.

### 5.3 The main control loop

In Lua\TeX'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 Lua\TeX 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 Lua\TeX 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. The Aleph-derived commands \leftghost and \rightghost create nodes of a third subtype: ‘ghost’. These nodes are ignored completely by all further processing until the stage where inter-glyph kerning is added.

4. 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.\footnote{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.}

In Lua\TeX, it works like this: if Lua\TeX 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=`\-

5. The \setlanguage command no longer creates whatsit. 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.

6. 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.

7. 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 Lua\TeX, 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 \texttt{tracing commands} 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:

\texttt{\hyphenpenaltymode}  
\texttt{\automatichyphenpenalty}  
\texttt{\explicithyphenpenalty}

The first parameter has the following consequences for automatic discs (the ones resulting from 
an \texttt{\exhyphenchar}):

<table>
<thead>
<tr>
<th>MODE</th>
<th>AUTOMATIC DISC</th>
<th>EXPLICIT DISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\exhyphenpenalty</td>
<td>\exhyphenpenalty</td>
</tr>
<tr>
<td>1</td>
<td>\hyphenpenalty</td>
<td>\hyphenpenalty</td>
</tr>
<tr>
<td>2</td>
<td>\exhyphenpenalty</td>
<td>\hyphenpenalty</td>
</tr>
<tr>
<td>3</td>
<td>\hyphenpenalty</td>
<td>\exhyphenpenalty</td>
</tr>
<tr>
<td>4</td>
<td>\automatichyphenpenalty</td>
<td>\explicithyphenpenalty</td>
</tr>
<tr>
<td>5</td>
<td>\exhyphenpenalty</td>
<td>\explicithyphenpenalty</td>
</tr>
<tr>
<td>6</td>
<td>\hyphenpenalty</td>
<td>\explicithyphenpenalty</td>
</tr>
<tr>
<td>7</td>
<td>\automatichyphenpenalty</td>
<td>\exhyphenpenalty</td>
</tr>
<tr>
<td>8</td>
<td>\automatichyphenpenalty</td>
<td>\hyphenpenalty</td>
</tr>
</tbody>
</table>

other values do what we always did in Lua\TeX: insert \texttt{\exhyphenpenalty}.

### 5.4 Loading patterns and exceptions

Although we keep the traditional approach towards hyphenation (which is still superior) the 
implementation of the hyphenation algorithm in Lua\TeX{} is quite different from the one in \TeX{}82.

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

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

<table>
<thead>
<tr>
<th>VALUE</th>
<th>IMPLIED KEY (INPUT)</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta-ble</td>
<td>table</td>
<td>ta-ble (= ta\discretionary{-}{}{}ble)</td>
</tr>
<tr>
<td>ba{\k-}{}{c}ken</td>
<td>backen</td>
<td>ba\discretionary{\k-}{}{}c\ken</td>
</tr>
</tbody>
</table>

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 Lua\TeX
using one of the functions in the Lua \lang 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 \textepsilon-\TeX extension \savinghyphcodes was that hyphenation heavily de-
depended on font encodings. This is no longer true in Lua\TeX, and the corresponding primitive is
basically ignored. Because we now have \hjcode, the case relate 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{}{}{}xbar 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 |
|------------|------------|------------|------------|
| 123 xxxxxxx 123 | 123       | 123        | 123        |
| xxa-        | xxa-       | xxa-       | xxa-       |
| -bxas-      | -bxas-     | -bxas-     | -bxas-     |
| -bxx        | -bxx       | -bxx       | -bxx       |
| 123         | 123        | 123        | 123        |

123 xxxxxxx        123             123 xxxxxxx
xxxxxxx xxxa-       xxxxxx xxxa-
-bxxxx xxxa-        -bxxxx xxxa-
bxxxxx 123          -bxxxxx 123
5.5 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 \texttt{\patterns} (and its Lua counterpart \texttt{lang.patterns}) can be used at any time, not only in \texttt{init\TeX}.
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.

Lua\TeX{} uses the language-specific variables \prehyphenchar and \posthyphenchar in the creation of implicit discretionaries, instead of \TeX{}82’s \hyphenchar, and the values of the language-specific variables \preexhyphenchar and \postexhyphenchar for explicit discretionaries (instead of \TeX{}82’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 \texttt{\lang.hyphenate} function the current values are used.

The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the \texttt{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:

\begin{verbatim}
\hsizelmm
1:foo{\hyphenpenalty 10000\discretionary{}{}{}}bar\par
2:foo\discretionary penalty 10000 {}{}{}bar\par
3:foo\discretionary{}{}{}bar\par
\end{verbatim}

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 \texttt{\setlanguage} command forces a word boundary).

All languages start out with \prehyphenchar=\textbackslash-, \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 \texttt{\language}, this behaviour is compatible with \patterns and \hyphenation.

Lua\TeX{} also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in \TeX{}82). 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 \TeX{}82, but there the behaviour cannot be controlled).
If you are using the Lua function `lang.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.

## 5.6 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, LuaT\(\TeX\) 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 word 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 LuaT\(\TeX\) 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 ConT\(\TeX\)t) 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.

Let’s look at an example. Take the word `office`, hyphenated `of-fice`, using a ‘normal’ font with all the f-f and f-i type ligatures:

\[
\begin{align*}
\text{initial} & \quad \{o\}\{f\}\{f\}\{i\}\{c\}\{e\} \\
\text{after hyphenation} & \quad \{o\}\{f\}\{\{-\},\{\},\{\}\}\{f\}\{i\}\{c\}\{e\} \\
\text{first ligature stage} & \quad \{o\}\{\{-\},\{f\},\{<ff>\}\}\{i\}\{c\}\{e\} \\
\text{final result} & \quad \{o\}\{\{-\},\{<fi>\},\{<ffi>\}\}\{c\}\{e\}
\end{align*}
\]

That’s bad enough, but let us assume that there is also a hyphenation point between the f and the i, to create of-\(f\)-ice. Then the final result should be:

\[
\begin{align*}
\{o\}\{\{-\},\{<fi>\}\} \\
\quad \{<ffi>\}\} & \quad \{c\}\{e\}
\end{align*}
\]

with discretionaries in the post-break text as well as in the replacement text of the top-level discretionary that resulted from the first hyphenation point.
Here is that nested solution again, in a different representation:

<table>
<thead>
<tr>
<th>PRE</th>
<th>POST</th>
<th>REPLACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>topdisc</td>
<td>f-  (1)</td>
<td>sub 1</td>
</tr>
<tr>
<td>sub 1</td>
<td>f-  (2)</td>
<td>i     (3)</td>
</tr>
<tr>
<td>sub 2</td>
<td>&lt;ff&gt;- (5)</td>
<td>i     (6)</td>
</tr>
</tbody>
</table>

When line breaking is choosing its breakpoints, the following fields will eventually be selected:

- off-ice f-  (1)
- f-  (2)
- i     (3)
- off-ice f-  (1)
- <fi> (4)
- off-ice <ff>- (5)
- i     (6)
- office <ffi> (7)

The current solution in LuaTeX is not able to handle nested discretionaries, but it is in fact smart enough to handle this fictional off-ice example. It does so by combining two sequential discretionary nodes as if they were a single object (where the second discretionary node is treated as an extension of the first node).

One can observe that the off-ice and off-ice cases both end with the same actual post replacement list (i), and that this would be the case even if i was the first item of a potential following ligature like ic. This allows LuaTeX to do away with one of the fields, and thus make the whole stuff fit into just two discretionary nodes.

The mapping of the seven list fields to the six fields in this discretionary node pair is as follows:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc1.pre</td>
<td>f-  (1)</td>
</tr>
<tr>
<td>disc1.post</td>
<td>&lt;fi&gt; (4)</td>
</tr>
<tr>
<td>disc1.replace</td>
<td>&lt;ffi&gt; (7)</td>
</tr>
<tr>
<td>disc2.pre</td>
<td>f-  (2)</td>
</tr>
<tr>
<td>disc2.post</td>
<td>i     (3,6)</td>
</tr>
<tr>
<td>disc2.replace</td>
<td>&lt;ff&gt;- (5)</td>
</tr>
</tbody>
</table>

What is actually generated after ligaturing has been applied is therefore:

{{f-},
 {<fi>},
 {<ffi>}}

{{f-},
 {i},
 {<ff>-}}{c}{e}

The two discretionaries have different subtypes from a discretionary appearing on its own: the first has subtype 4, and the second has subtype 5. The need for these special subtypes stems
from the fact that not all of the fields appear in their ‘normal’ location. The second discretionary especially looks odd, with things like the `<ff>`- appearing in `disc2.replace`. The fact that some of the fields have different meanings (and different processing code internally) is what makes it necessary to have different subtypes: this enables LuaTeX to distinguish this sequence of two joined discretionary nodes from the case of two standalone discretionaries appearing in a row.

Of course there is still that relationship with fonts: ligatures can be implemented by mapping a sequence of glyphs onto one glyph, but also by selective replacement and kerning. This means that the above examples are just representing the traditional approach.

### 5.7 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 it does not expect there to be 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.

### 5.8 The lang library

#### 5.8.1 new and id

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

```
<language> l = lang.new()
<language> l = lang.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 lang 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 = lang.id(<language> l)
```
The number returned is the internal \language id number this object refers to.

### 5.8.2 hyphenation

You can hyphenate a string directly with:

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

### 5.8.3 clear_hyphenation 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 5.4.

```latex
lang.clear_hyphenation(<language> l)
```

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

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

This function creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 5.4. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

### 5.8.4 patterns and clear_patterns

```latex
<string> n = lang.patterns(<language> l)
lang.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 5.4.

```latex
lang.clear_patterns(<language> l)
```

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

### 5.8.5 hyphenationmin

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

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

### 5.8.6 [pre|post][ex|]hyphenchar

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

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

<number> n = lang.postexhyphenchar(<language> l)
lang.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).

### 5.8.7 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 = lang.hyphenate(<node> head)
<boolean> success = lang.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 5.2 for more details.

### 5.8.8 [set|get]hjcode

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

lang.sethjcode(<language> l, <number> char, <number> usedchar)
<number> usedchar = lang.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.
6 Font structure

6.1 The font tables

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 returned by the `define_font` callback, or if they result from the normal tfm/vf reading routines if there is no `define_font` callback defined.

The column ‘vf’ means that this key will be created by the `font.read_vf()` routine, ‘tfm’ means that the key will be created by the `font.read_tfm()` routine, and ‘used’ means whether or not the Lua\TeX\ engine itself will do something with the key. The top-level keys in the table are as follows:

<table>
<thead>
<tr>
<th>KEY</th>
<th>VF</th>
<th>TFM</th>
<th>USED</th>
<th>VALUE TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>string</td>
<td>metric (file) name</td>
</tr>
<tr>
<td>area</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>string</td>
<td>(directory) location, typically empty</td>
</tr>
<tr>
<td>used</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>boolean</td>
<td>indicates usage (initial: false)</td>
</tr>
<tr>
<td>characters</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>table</td>
<td>the defined glyphs of this font</td>
</tr>
<tr>
<td>checksum</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>number</td>
<td>default: 0</td>
</tr>
<tr>
<td>designsize</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>expected size (default: 655360 == 10pt)</td>
</tr>
<tr>
<td>direction</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>default: 0</td>
</tr>
<tr>
<td>encodingbytes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: depends on format</td>
</tr>
<tr>
<td>encodingname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>encoding name</td>
</tr>
<tr>
<td>fonts</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>table</td>
<td>locally used fonts</td>
</tr>
<tr>
<td>psname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>This is the PostScript fontname in the incoming font source, and it's used as font-name identifier in the pdf output. This has to be a valid string, e.g. no spaces and such, as the backend will not do a cleanup. This gives complete control to the loader.</td>
</tr>
<tr>
<td>fullname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>output font name, used as a fallback in the pdf output if the psname is not set</td>
</tr>
<tr>
<td>subfont</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: 0, index in (ttc) font with multiple fonts</td>
</tr>
<tr>
<td>header</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>string</td>
<td>header comments, if any</td>
</tr>
<tr>
<td>hyphenchar</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: \TeX's <code>\hyphenchar</code></td>
</tr>
<tr>
<td>parameters</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>hash</td>
<td>default: 7 parameters, all zero</td>
</tr>
<tr>
<td>size</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>the required scaling (by default the same as designsize)</td>
</tr>
<tr>
<td>skewchar</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: \TeX's <code>\skewchar</code></td>
</tr>
<tr>
<td>type</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>basic type of this font</td>
</tr>
<tr>
<td>format</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>disk format type</td>
</tr>
<tr>
<td>embedding</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>pdf inclusion</td>
</tr>
<tr>
<td>filename</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>the name of the font on disk</td>
</tr>
</tbody>
</table>
tounicode | no | yes | yes | number | When this is set to 1 LuaTeX assumes per-
glyph tounicode entries are present in the
font.
stretch | no | no | yes | number | the ‘stretch’ value from \expandglyphsin-
font
shrink | no | no | yes | number | the ‘shrink’ value from \expandglyphsin-
font
step | no | no | yes | number | the ‘step’ value from \expandglyphsinfont
expansion_factor | no | no | no | number | the actual expansion factor of an expanded
font
attributes | no | no | yes | string | the \pdffontattr

| cache | no | no | yes | string | This key controls caching of the Lua ta-
ble on the \TeX{} end where yes means: use
a reference to the table that is passed to
LuaTeX{} (this is the default), and no means:
don’t store the table reference, don’t cache
any Lua data for this font while renew
means: don’t store the table reference, but
save a reference to the table that is created
at the first access to one of its fields in the
font.

nomath | no | no | yes | 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 | no | no | yes | boolean | This key flags a font as representing an
old school \TeX{} math font and disables the
OpenType code path.

| slant | no | no | yes | number | This parameter will tilt the font and does
the same as SlantFont in the map file for
Type1 fonts.

| extend | no | no | yes | number | This parameter will scale the font horizon-
tally and does the same as ExtendFont in
the map file for Type1 fonts.

| squeeze | no | no | yes | number | This parameter will scale the font vertically
and has no equivalent in the map file.

| width | no | no | yes | number | The backend will inject pdf operators that
set the penwidth. The value is (as usual
in \TeX{}) divided by 1000. It works with the
mode file.

| mode | no | no | yes | number | The backend will inject pdf operators that
relate to the drawing mode with 0 being a
fill, 1 being an outline, 2 both draw and fill
and 3 no painting at all.

The saved reference in the cache option is thread-local, so be careful when you are using corou-
tines: an error will be thrown if the table has been cached in one thread, but you reference it from another thread.

The key name is always required. The keys stretch, shrink, step only have meaning when used together: they can be used to replace a post-loading \expandglyphsinfont command. The auto_expand option is not supported in LuaTEX. In fact, the primitives that create expanded or protruding copies are probably only useful when used with traditional fonts because all these extra OpenType properties are kept out of the picture. The expansion_factor is value that can be present inside a font in font.fonts. It is the actual expansion factor (a value between -shrink and stretch, with step step) of a font that was automatically generated by the font expansion algorithm.

The subfont parameter can be used to specify the subfont in a ttc font. When given, it is used instead of the psname and fullname combination. The first subfont has number 1. A zero value signals using the names as lookup.

Because we store the actual state of expansion with each glyph and don’t have special font instances, we can change some font related parameters before lines are constructed, like:

```lua
font.setexpansion(font.current(),100,100,20)
```

This is mostly meant for experiments (or an optimizing routing written in Lua) so there is no primitive.

The key attributes can be used to set font attributes in the pdf file. The key used is set by the engine when a font is actively in use, this makes sure that the font’s definition is written to the output file (dvi or pdf). The tfm reader sets it to false. The direction is a number signalling the ‘normal’ direction for this font. There are sixteen possibilities:

<table>
<thead>
<tr>
<th>#</th>
<th>DIR</th>
<th>#</th>
<th>DIR</th>
<th>#</th>
<th>DIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>4</td>
<td>RT</td>
<td>8</td>
<td>TT</td>
</tr>
<tr>
<td>1</td>
<td>LL</td>
<td>5</td>
<td>RL</td>
<td>9</td>
<td>TL</td>
</tr>
<tr>
<td>2</td>
<td>LB</td>
<td>6</td>
<td>RB</td>
<td>10</td>
<td>TB</td>
</tr>
<tr>
<td>3</td>
<td>LR</td>
<td>7</td>
<td>RR</td>
<td>11</td>
<td>TR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These are Omega-style direction abbreviations: the first character indicates the ‘first’ edge of the character glyphs (the edge that is seen first in the writing direction), the second the ‘top’ side. Keep in mind that LuaTEX has a bit different directional model so these values are not used for anything.

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:

<table>
<thead>
<tr>
<th>NAME</th>
<th>REMAPPING</th>
</tr>
</thead>
<tbody>
<tr>
<td>slant</td>
<td>1</td>
</tr>
<tr>
<td>space</td>
<td>2</td>
</tr>
<tr>
<td>space_stretch</td>
<td>3</td>
</tr>
<tr>
<td>space_shrink</td>
<td>4</td>
</tr>
</tbody>
</table>
The keys type, format, embedding, fullname and filename are used to embed OpenType fonts in the result pdf.

The characters table is a list of character hashes indexed by an integer number. The number is the ‘internal code’ TpX knows this character by.

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).

Each character hash itself is a hash. 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 }
    }
}
```

The following top-level keys can be present inside a character hash:

<table>
<thead>
<tr>
<th>KEY</th>
<th>VF</th>
<th>TFM</th>
<th>USED</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s width, in sp (default 0)</td>
</tr>
<tr>
<td>height</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s height, in sp (default 0)</td>
</tr>
<tr>
<td>depth</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s depth, in sp (default 0)</td>
</tr>
<tr>
<td>italic</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s italic correction, in sp (default zero)</td>
</tr>
<tr>
<td>top_accent</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s top accent alignment place, in sp (default zero)</td>
</tr>
<tr>
<td>bot_accent</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s bottom accent alignment place, in sp (default zero)</td>
</tr>
<tr>
<td>left_protruding</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s \lpcode</td>
</tr>
<tr>
<td>right_protruding</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s \rpcode</td>
</tr>
</tbody>
</table>
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`.

The usage of `tounicode` is this: if this font specifies a `tounicode=1` at the top level, then \LaTeX{} will construct a `/ToUnicode` entry for the pdf font (or font subset) based on the character-level `tounicode` strings, where they are available. If a character does not have a sensible Unicode equivalent, do not provide a string either (no empty strings).

If the font level `tounicode` is not set, then \LaTeX{} will build up `/ToUnicode` based on the TeX code points you used, and any character-level `tounicode`s will be ignored. The string format is exactly the format that is expected by Adobe CMap files (utf-16BE in hexadecimal encoding), minus the enclosing angle brackets. For instance the `tounicode` for a fi ligature would be `00660069`. When you pass a number the conversion will be done for you.

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:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>number</td>
<td>top character index</td>
</tr>
<tr>
<td>mid</td>
<td>number</td>
<td>middle character index</td>
</tr>
<tr>
<td>bot</td>
<td>number</td>
<td>bottom character index</td>
</tr>
<tr>
<td>rep</td>
<td>number</td>
<td>repeatable character index</td>
</tr>
</tbody>
</table>

The `horiz_variants` and `vert_variants` are arrays of components. Each of those components is itself a hash of up to five keys:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>glyph</td>
<td>number</td>
<td>The character index. Note that this is an encoding number, not a name.</td>
</tr>
<tr>
<td>extender</td>
<td>number</td>
<td>One (1) if this part is repeatable, zero (0) otherwise.</td>
</tr>
<tr>
<td>start</td>
<td>number</td>
<td>The maximum overlap at the starting side (in scaled points).</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>number</td>
<td>the type of this ligature command, default 0</td>
</tr>
<tr>
<td>char</td>
<td>number</td>
<td>the character index of the resultant ligature</td>
</tr>
</tbody>
</table>

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’.

<table>
<thead>
<tr>
<th>TEXTUAL (KNUTH)</th>
<th>NUMBER</th>
<th>STRING</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>l + r :=: n</td>
<td>0</td>
<td>:=</td>
<td></td>
</tr>
<tr>
<td>l + r =:=</td>
<td>n</td>
<td>1</td>
<td>=:=</td>
</tr>
<tr>
<td>l + r</td>
<td>:=: n</td>
<td>2</td>
<td>:=:</td>
</tr>
<tr>
<td>l + r</td>
<td>:=:=</td>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>l + r</td>
<td>:=&gt;</td>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>l + r</td>
<td>:=&gt;</td>
<td>n</td>
<td>6</td>
</tr>
<tr>
<td>l + r</td>
<td>:=&gt;</td>
<td>n</td>
<td>7</td>
</tr>
<tr>
<td>l + r</td>
<td>:=&gt;</td>
<td>&gt;&gt; n</td>
<td>11</td>
</tr>
</tbody>
</table>

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 commands array is explained below.

### 6.2 Real fonts

Whether or not a \TeX font is a ‘real’ font that should be written to the pdf document is decided by the type value in the top-level font structure. If the value is real, then this is a proper font, and the inclusion mechanism will attempt to add the needed font object definitions to the pdf.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>this is a base font</td>
</tr>
<tr>
<td>virtual</td>
<td>this is a virtual font</td>
</tr>
</tbody>
</table>
The actions to be taken depend on a number of different variables:

- Whether the used font fits in an 8-bit encoding scheme or not. This is true for traditional \TeX fonts that communicate via tfm files.
- The type of the disk font file, for instance a bitmap file or an outline Type1, TrueType or OpenType font.
- The level of embedding requested, although in most cases a subset of characters is embedded. The times when nothing got embedded are (in our opinion at least) basically gone.

A font that uses anything other than an 8-bit encoding vector has to be written to the pdf in a different way. When the font table has encodingbytes set to 2, then it is a wide font, in all other cases it isn’t. The value 2 is the default for OpenType and TrueType fonts loaded via Lua. For Type1 fonts, you have to set encodingbytes to 2 explicitly. For pk bitmap fonts, wide font encoding is not supported at all.

If no special care is needed, Lua\TeX falls back to the mapfile-based solution used by pdf\TeX and dvips, so that legacy fonts are supported transparently. If a ‘wide’ font is used, the new subsystem kicks in, and some extra fields have to be present in the font structure. In this case, Lua\TeX does not use a map file at all. These extra fields are: format, embedding, fullname, cidinfo (as explained above), filename, and the index key in the separate characters.

The format variable can have the following values. type3 fonts are provided for backward compatibility only, and do not support the new wide encoding options.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>this is a PostScript Type1 font</td>
</tr>
<tr>
<td>type3</td>
<td>this is a bitmapped (pk) font</td>
</tr>
<tr>
<td>trueype</td>
<td>this is a TrueType or TrueType-based OpenType font</td>
</tr>
<tr>
<td>openype</td>
<td>this is a PostScript-based OpenType font</td>
</tr>
</tbody>
</table>

Valid values for the embedding variable are:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>don’t embed the font at all</td>
</tr>
<tr>
<td>subset</td>
<td>include and attempt to subset the font</td>
</tr>
<tr>
<td>full</td>
<td>include this font in its entirety</td>
</tr>
</tbody>
</table>

The other fields are used as follows. The fullname will be the PostScript/pdf font name. The cidinfo will be used as the character set: the CID /Ordering and /Registry keys. The filename points to the actual font file. If you include the full path in the filename or if the file is in the local directory, Lua\TeX will run a little bit more efficient because it will not have to re-run the find_*_file callback in that case.

Be careful: when mixing old and new fonts in one document, it is possible to create PostScript name clashes that can result in printing errors. When this happens, you have to change the fullname of the font to a more unique one.

Typeset strings are written out in a wide format using 2 bytes per glyph, using the index key in the character information as value. The overall effect is like having an encoding based on numbers instead of traditional (PostScript) name-based reencoding. One way to get the correct
index numbers for Type1 fonts is by loading the font via fontloader.open and use the table indices as index fields.

In order to make sure that cut and paste of the final document works okay you can best make sure that there is a tounicode vector enforced. Not all pdf viewers handle this right so take Acrobat as reference.

6.3 Virtual fonts

6.3.1 The structure

You have to take the following steps if you want Lua\TeX{} to treat the returned table from define_font as a virtual font:

- Set the top-level key type to virtual. In most cases it’s optional because we look at the commands entry anyway.
- Make sure there is at least one valid entry in fonts (see below), although recent versions of Lua\TeX{} add a default entry when this table is missing.
- Add a commands array to those characters that matter. 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).

The presence of the toplevel type key with the specific value virtual will trigger handling of the rest of the special virtual font fields in the table, but the mere existence of 'type' is enough to prevent Lua\TeX{} from looking for a virtual font on its own. This also works 'in reverse': if you are absolutely certain that a font is not a virtual font, assigning the value real to type will inhibit Lua\TeX{} from looking for a virtual font file, thereby saving you a disk search. This only matters when we load a tfm file.

The fonts is an (indexed) Lua 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, you can use the font.nextid() function which returns the index of the next to be defined font which is probably the currently defined one. So, a table looks like this:

```lua
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 Lua\TeX{} 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 allowed commands and their arguments are:
<table>
<thead>
<tr>
<th>COMMAND</th>
<th>ARGUMENTS</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>font</td>
<td>1 number</td>
<td>number</td>
<td>select a new font from the local fonts table</td>
</tr>
<tr>
<td>char</td>
<td>1 number</td>
<td>number</td>
<td>typeset this character number from the current font, and move right by the character's width</td>
</tr>
<tr>
<td>node</td>
<td>1 node</td>
<td>node</td>
<td>output this node (list), and move right by the width of this list</td>
</tr>
<tr>
<td>slot</td>
<td>2 2 numbers</td>
<td>node</td>
<td>a shortcut for the combination of a font and char command</td>
</tr>
<tr>
<td>push</td>
<td>0</td>
<td>node</td>
<td>save current position</td>
</tr>
<tr>
<td>nop</td>
<td>0</td>
<td>node</td>
<td>do nothing</td>
</tr>
<tr>
<td>pop</td>
<td>0</td>
<td>node</td>
<td>pop position</td>
</tr>
<tr>
<td>rule</td>
<td>2 2 numbers</td>
<td>number</td>
<td>output a rule $ht \times wd$, and move right.</td>
</tr>
<tr>
<td>down</td>
<td>1 number</td>
<td>number</td>
<td>move down on the page</td>
</tr>
<tr>
<td>right</td>
<td>1 number</td>
<td>number</td>
<td>move right on the page</td>
</tr>
<tr>
<td>special</td>
<td>1 string</td>
<td>string</td>
<td>output a \special command</td>
</tr>
<tr>
<td>pdf</td>
<td>2 2 strings</td>
<td>string</td>
<td>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</td>
</tr>
<tr>
<td>lua</td>
<td>1 string, function</td>
<td>string, function</td>
<td>execute a Lua script when the glyph is embedded; in case of a function it gets the font id and character code passed</td>
</tr>
<tr>
<td>image</td>
<td>1 image</td>
<td>image</td>
<td>output an image (the argument can be either an &lt;image&gt; variable or an image_spec table)</td>
</tr>
<tr>
<td>comment</td>
<td>any any</td>
<td>any</td>
<td>the arguments of this command are ignored</td>
</tr>
</tbody>
</table>

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. Normally one could use font.nextid but when more complex fonts are built in the meantime other instances could have been loaded.

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.

<table>
<thead>
<tr>
<th>MODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>enter page mode and set the position</td>
</tr>
<tr>
<td>page</td>
<td>enter page mode</td>
</tr>
<tr>
<td>text</td>
<td>enter text mode</td>
</tr>
<tr>
<td>font</td>
<td>enter font mode (kind of text mode, only in virtual fonts)</td>
</tr>
<tr>
<td>always</td>
<td>finish the current string and force a transform if needed</td>
</tr>
<tr>
<td>raw</td>
<td>finish the current string</td>
</tr>
</tbody>
</table>

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.

### 6.3.2 Artificial fonts

Even in a ‘real’ font, there can be virtual characters. When Lua\TeX{} encounters a commands field inside a character when it becomes time to typeset the character, it will interpret the commands, just like for a true virtual character. In this case, if you have created no ‘fonts’ array, then the default (and only) ‘base’ font is taken to be the current font itself. In practice, this means that you can create virtual duplicates of existing characters which is useful if you want to create composite characters.

Note: this feature does not work the other way around. There cannot be ‘real’ characters in a virtual font! You cannot use this technique for font re-encoding either; you need a truly virtual font for that (because characters that are already present cannot be altered).

### 6.3.3 Example virtual font

Finally, here is a plain \TeX{} input file with a virtual font demonstration:

\directlua {
    callback.register('define_font',

96 Font structure
function (name, size)
    if name == 'cmr10-red' then
        local f = font.read_tfm('cmr10', size)
        f.name = 'cmr10-red'
        f.type = 'virtual'
        f.fonts = {
            { name = 'cmr10', size = size }
        }
        for i, v in pairs(f.characters) do
            if string.char(i):find('[tacohanshartmut]') then
                v.commands = {
                    { "special", "pdf: 1 0 0 rg" },
                    { "char", i },
                    { "special", "pdf: 0 g" },
                }
            end
        end
        return f
    else
        return font.read_tfm(name, size)
    end
end

\font\myfont = cmr10-red at 10pt \myfont  This is a line of text \par
\font\myfontx = cmr10  at 10pt \myfontx Here is another line of text \par

6.4 The vf library

The vf library can be used when Lua code, as defined in the commands of the font, is executed. The functions provided are similar as the commands: char, down, fontid, image, node, nop, pop, push, right, rule, special and pdf. This library has been present for a while but not been advertised and tested much, if only because it’s easy to define an invalid font (or mess up the pdf stream). Keep in mind that the Lua snippets are executed each time when a character is output.

6.5 The font library

The font library provides the interface into the internals of the font system, and it also contains helper functions to load traditional TeX font metrics formats. Other font loading functionality is provided by the fontloader library that will be discussed in the next section.

6.5.1 Loading a TFM file

The behaviour documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.
The number 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 designsize of the font.

### 6.5.2 Loading a VF file

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

```lua
default fnt =
    font.read_tfm(<string> name, <number> s)
```

The meaning of the number \( s \) and the format of the returned table are similar to the ones in the `read_tfm` function.

### 6.5.3 The fonts array

The whole table of \( \TeX \) fonts is accessible from Lua using a virtual array.

```lua
font.fonts[n] = { ... }
default f = font.fonts[n]
```

Because this is a virtual array, you cannot call `pairs` on it, but see below for the `font.each` iterator.

The two metatable functions implementing the virtual array are:

```lua
default f = font.getfont(<number> n)
font.setfont(<number> n, <table> f)
```

Note that at the moment, each access to the `font.fonts` or call to `font.getfont` creates a Lua table for the whole font unless you cached it. If you want a copy of the internal data you can use `font.getcopy`:

```lua
default f = font.getcopy(<number> n)
```

This one will return a table of the parameters as known to \( \TeX \). These can be different from the ones in the cached table:

```lua
default p = font.getparameters(<number> n)
```

Also note the following: assignments can only be made to fonts that have already been defined in \( \TeX \), but have not been accessed at all since that definition. This limits the usability of the write access to `font.fonts` quite a lot, a less stringent ruleset will likely be implemented later.
6.5.4 Checking a font’s status

You can test for the status of a font by calling this function:

```plaintext
<boolean> f =
    font.frozen(<number> n)
```

The return value is one of true (unassignable), false (can be changed) or nil (not a valid font at all).

6.5.5 Defining a font directly

You can define your own font into font.fonts by calling this function:

```plaintext
<number> i =
    font.define(<table> f)
```

The return value is the internal id number of the defined font (the index into font.fonts). If the font creation fails, an error is raised. The table is a font structure. An alternative call is:

```plaintext
<number> i =
    font.define(<number> n, <table> f)
```

Where the first argument is a reserved font id (see below).

6.5.6 Extending a font

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.

```plaintext
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.)

6.5.7 Projected next font id

```plaintext
<number> i =
    font.nextid()
```

This returns the font id number that would be returned by a font.define call if it was executed at this spot in the code flow. This is useful for virtual fonts that need to reference themselves. If you pass true as argument, the id gets reserved and you can pass to font.define as first argument. This can be handy when you create complex virtual fonts.

```plaintext
<number> i =
```
font.nextid(true)

### 6.5.8 Font ids

<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 in font.fonts.

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

This gets or sets the currently used font number.

### 6.5.9 Iterating over all fonts

for i,v in font.each() do
    ...
end

This is an iterator over each of the defined \TeX fonts. The first returned value is the index in font.fonts, the second the font itself, as a Lua table. The indices are listed incrementally, but they do not always form an array of consecutive numbers: in some cases there can be holes in the sequence.

### 6.5.10 \glyphdimensionsmode

Already in the early days of Lua\TeX the decision was made to calculate the effective height and depth of glyphs in a way that reflected the applied vertical offset. The height got that offset added, the depth only when the offset was larger than zero. We can now control this in more detail with this mode parameter. An offset is added to the height and/or subtracted from the depth. The effective values are never negative. The zero mode is the default.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the old behavior: add the offset to the height and only subtract the offset only from the depth when it is positive</td>
</tr>
<tr>
<td>1</td>
<td>add the offset to the height and subtract it from the depth</td>
</tr>
<tr>
<td>2</td>
<td>add the offset to the height and subtract it from the depth but keep the maxima of the current and previous results</td>
</tr>
<tr>
<td>3</td>
<td>use the height and depth of the glyph, so no offset is applied</td>
</tr>
</tbody>
</table>
6.5.11 \texttt{\textbackslash discretionaryligaturemode}

This parameter controls how complex ligatures interact with discretionaries (as injected by the hyphenator). The non-zero values prevent the construction of so called init and select discretionaries.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx-</td>
<td>xx-</td>
<td>xx-</td>
</tr>
<tr>
<td>f-</td>
<td>f-</td>
<td>ff-</td>
</tr>
<tr>
<td>fi-</td>
<td>i-</td>
<td></td>
</tr>
<tr>
<td>xx</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>xx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7 Math

7.1 Traditional alongside OPENTYPE

The handling of mathematics in LuaTeX differs quite a bit from how \TeX\ (and therefore pdf\TeX) handles math. First, LuaTeX adds primitives and extends some others so that Unicode input can be used easily. Second, all of \TeX\’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.

7.2 Unicode math characters

Character handling is now extended up to the full Unicode range (the \texttt{\U} prefix), which is compatible with X\TeX.

The math primitives from \TeX are kept as they are, except for the ones that convert from input to math commands: \texttt{mathcode}, and \texttt{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:

\begin{verbatim}
\def\overbrace{\Umathaccent 0 1 "23DE }
\end{verbatim}

The altered \TeX\ primitives are:

\begin{tabular}{|l|l|l|l|l|}
\hline
PRIMITIVE & MIN & MAX & MIN & MAX \\
\hline
\texttt{\mathcode} & 0 & 10FFFF = 0 & 8000 \\
\texttt{\delcode} & 0 & 10FFFF = 0 & FFFFFF \\
\hline
\end{tabular}

The unaltered ones are:

\begin{tabular}{|l|l|l|}
\hline
PRIMITIVE & MIN & MAX \\
\hline
\texttt{\mathchardef} & 0 & 8000 \\
\texttt{\mathchar} & 0 & 7FFF \\
\texttt{\mathaccent} & 0 & 7FFF \\
\texttt{\delimiter} & 0 & 7FFFFFF \\
\texttt{\radical} & 0 & 7FFFFFF \\
\hline
\end{tabular}

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

The following new primitives are compatible with X\TeX:

\begin{tabular}{|l|l|l|l|l|}
\hline
PRIMITIVE & MIN & MAX & MIN & MAX \\
\hline
\texttt{\Umathchardef} & 0+0+0 & 7+FF+10FFFF \\
\hline
\end{tabular}
Specifications typically look like:

\[\text{\textbackslash mathchardef} xx = "1"0"456\]
\[\text{\textbackslash mathcode} 123 = "1"0"789\]

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 \textbackslash mathoperatorsize parameter.

For some of these primitives, all information is packed into a single signed integer. For the first two (\textbackslash mathcharnum and \textbackslash mathcodenum), 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 \textbackslash delcodenum 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 \textbackslash mathaccent command accepts optional keywords to control various details regarding math accents. See section 7.6.2 below for details.

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

<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>VALUE RANGE (IN HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbackslash root</td>
<td>0 + 0–FF + 10FFFF</td>
</tr>
<tr>
<td>\textbackslash overdelimiter</td>
<td>0 + 0–FF + 10FFFF</td>
</tr>
<tr>
<td>\textbackslash underdelimiter</td>
<td>0 + 0–FF + 10FFFF</td>
</tr>
<tr>
<td>\textbackslash delimeterover</td>
<td>0 + 0–FF + 10FFFF</td>
</tr>
<tr>
<td>\textbackslash delimiterunder</td>
<td>0 + 0–FF + 10FFFF</td>
</tr>
</tbody>
</table>

Instead of the pseudo class variable (7) you can use a family number as signal for using the current family. This permits classifying characters with a class and still let the family adapt. The trigger family is set with \textbackslash variablefam. So:

\[\textbackslash variablefam}"24\]
\[\text{\textbackslash mathchardef} \text{foo} "3 }"24 123\]
\[\text{\textbackslash foo} \text{\textbackslash fam9}\]

Results in a curly left brace taken from family 9 with class ‘relation’ and spacing around it will be accordingly.
7.3 Math styles

7.3.1 \textstyle

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, \LaTeX adds the new primitive: \textstyle. This is a ‘convert command’ like e.g. \textnumero: its value can only be read, not set.

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:

\begin{verbatim}
\ifnum\mathstyle=\textstyle
  \message{normal text style}
\else \ifnum\mathstyle=\crampedtextstyle
  \message{cramped text style}
\fi \fi
\fi
\end{verbatim}

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 \textchoice 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 \textstyle 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.

\begin{verbatim}
[a:\textstyle]\quad
\bgroup
\textchoice
\lbrace\bf \textstyle (x:d :\textstyle)\rbrace
\lbrace\bf \textscriptstyle (x:t :\textstyle)\rbrace
\lbrace\bf \textscriptscriptstyle (x:s :\textstyle)\rbrace
\lbrace\bf \textscriptscriptscriptstyle (x:ss:\textstyle)\rbrace
\egroup
\quad[b:\textstyle]\quad
\textchoice
\lbrace\bf \textstyle (y:d :\textstyle)\rbrace
\lbrace\bf \textscriptstyle (y:t :\textstyle)\rbrace
\end{verbatim}
This gives:

\[
\begin{array}{c}
[a:0] & \begin{array}{c}
(x:d:4) & [b:0] & (y:d:4) & [c:0] & (z:s:6) & [d:0]
\end{array} \\
[a:2] & \begin{array}{c}
\end{array}
\end{array}
\]

Using \begingroup ... \endgroup instead gives:

\[
\begin{array}{c}
[a:0] & \begin{array}{c}
(x:d:4) & [b:0] & (y:s:6) & [c:0] & (z:ss:6) & [d:0]
\end{array} \\
[a:2] & \begin{array}{c}
\end{array}
\end{array}
\]

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.

### 7.3.2 \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, Lua\TeX 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.

### 7.3.3 Cramped math styles

Lua\TeX 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 Eijkhout’s “\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 \LaTeX 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:

<table>
<thead>
<tr>
<th>STYLE</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>script</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>crampedscript</td>
<td>(b^x=x)</td>
</tr>
</tbody>
</table>

Now we set the following parameters

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

This gives a different result:

<table>
<thead>
<tr>
<th>STYLE</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>script</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>crampedscript</td>
<td>(b^x=x)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>STYLE</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>script</td>
<td>(b^x=x)</td>
</tr>
<tr>
<td>crampedscript</td>
<td>(b^x=x)</td>
</tr>
</tbody>
</table>
# 7.4 Math parameter settings

## 7.4.1 Many new \math* primitives

In Lua\TeX, 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 many more parameters than were not accessible before.

<table>
<thead>
<tr>
<th>PRIMITIVE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mathquad</td>
<td>the width of 18 mu’s</td>
</tr>
<tr>
<td>\mathaxis</td>
<td>height of the vertical center axis of the math formula above the baseline</td>
</tr>
<tr>
<td>\mathoperatorsize</td>
<td>minimum size of large operators in display mode</td>
</tr>
<tr>
<td>\mathoverbarkern</td>
<td>vertical clearance above the rule</td>
</tr>
<tr>
<td>\mathoverbarrule</td>
<td>the width of the rule</td>
</tr>
<tr>
<td>\mathoverbarvgap</td>
<td>vertical clearance below the rule</td>
</tr>
<tr>
<td>\mathunderbarkern</td>
<td>vertical clearance below the rule</td>
</tr>
<tr>
<td>\mathunderbarrule</td>
<td>the width of the rule</td>
</tr>
<tr>
<td>\mathunderbarvgap</td>
<td>vertical clearance above the rule</td>
</tr>
<tr>
<td>\mathradicalkern</td>
<td>vertical clearance above the rule</td>
</tr>
<tr>
<td>\mathradicalrule</td>
<td>the width of the rule</td>
</tr>
<tr>
<td>\mathradicalvgap</td>
<td>vertical clearance below the rule</td>
</tr>
<tr>
<td>\mathradicaldegreebefore</td>
<td>the forward kern that takes place before placement of the radical degree</td>
</tr>
<tr>
<td>\mathradicaldegreeafter</td>
<td>the backward kern that takes place after placement of the radical degree</td>
</tr>
<tr>
<td>\mathradicaldegreeraise</td>
<td>this is the percentage of the total height and depth of the radical sign</td>
</tr>
<tr>
<td></td>
<td>that the degree is raised by; it is expressed in percents, so 60% is</td>
</tr>
<tr>
<td></td>
<td>expressed as the integer 60</td>
</tr>
<tr>
<td>\mathstackvgap</td>
<td>vertical clearance between the two elements in a \atop stack</td>
</tr>
<tr>
<td>\mathstacknumup</td>
<td>numerator shift upward in \atop stack</td>
</tr>
<tr>
<td>\mathstackdenomdown</td>
<td>denominator shift downward in \atop stack</td>
</tr>
<tr>
<td>\mathfractionrule</td>
<td>the width of the rule in a \over</td>
</tr>
<tr>
<td>\mathfractionnumvgap</td>
<td>vertical clearance between the numerator and the rule</td>
</tr>
<tr>
<td>\mathfractionnumup</td>
<td>numerator shift upward in \over</td>
</tr>
<tr>
<td>\mathfractiondenomvgap</td>
<td>vertical clearance between the denominator and the rule</td>
</tr>
<tr>
<td>\mathfractiondenomdown</td>
<td>denominator shift downward in \over</td>
</tr>
<tr>
<td>\mathfractiondelsize</td>
<td>minimum delimiter size for \ldelimms</td>
</tr>
<tr>
<td>\mathlimitabovevgap</td>
<td>vertical clearance for limits above operators</td>
</tr>
<tr>
<td>\mathlimitabovevbgap</td>
<td>vertical baseline clearance for limits above operators</td>
</tr>
<tr>
<td>\mathlimitabovemakern</td>
<td>space reserved at the top of the limit</td>
</tr>
<tr>
<td>\mathlimitbelowvgap</td>
<td>vertical clearance for limits below operators</td>
</tr>
<tr>
<td>\mathlimitbelowvbgap</td>
<td>vertical baseline clearance for limits below operators</td>
</tr>
<tr>
<td>\mathlimitbelowmakern</td>
<td>space reserved at the bottom of the limit</td>
</tr>
<tr>
<td>\mathoverdelimitervgap</td>
<td>vertical clearance for limits above delimiters</td>
</tr>
<tr>
<td>\mathoverdelimiterbgap</td>
<td>vertical baseline clearance for limits above delimiters</td>
</tr>
</tbody>
</table>
Each of the parameters in this section can be set by a command like this:
\textbackslash{mathquad}\textbackslash{displaystyle}=1em

they obey grouping, and you can use \textbackslash{the}\textbackslash{mathquad}\textbackslash{displaystyle} if needed.

### 7.4.2 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, Lua\TeX{} 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 \texttt{cmsy} and \texttt{cmex}), or based on the named values in a potential \texttt{MathConstants} table when the font is loaded via Lua. If there is a \texttt{MathConstants} table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the \texttt{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 \TeX{} book. Assignments to \texttt{\textfont} set the values for the cramped and uncramped display and text styles, \texttt{\scriptfont} sets the script styles, and \texttt{\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, Lua\TeX{} also looks at the ‘space’ font dimension parameter. For math fonts, this should be set to zero.

<table>
<thead>
<tr>
<th>\texttt{Variable / Style}</th>
<th>TFM / Opentype</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\mathaxis}</td>
<td>axis_height</td>
</tr>
<tr>
<td>\texttt{\mathaxis}</td>
<td>AxisHeight</td>
</tr>
<tr>
<td>\texttt{\mathoperatorsize}</td>
<td>–</td>
</tr>
</tbody>
</table>
D, D' DisplayOperatorMinHeight
\Umathfractiondelsize delim1 FractionDelimiterDisplayStyleSize
D, D' \Umathfractiondelsize delim2 FractionDelimiterDisplayStyleSize
T, T', S, S', SS, SS' \Umathfractiondelsize denom1 FractionDenominatorDisplayStyleSize
D, D' \Umathfractiondelsize denom2 FractionDenominatorDisplayStyleSize
T, T', S, S', SS, SS' \Umathfractiondelsize denom1 FractionDenominatorDisplayStyleShiftDown
D, D' \Umathfractiondelsize denom2 FractionDenominatorDisplayStyleShiftDown
T, T', S, S', SS, SS' \Umathfractiondelsize denom1 FractionDenominatorDisplayStyleGapMin
D, D' \Umathfractiondelsize denom2 FractionDenominatorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleShiftUp
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleShiftUp
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleGapMin
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleGapMin
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleGapMin
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleGapMin
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize num1 FractionNumeratorDisplayStyleGapMin
D, D' \Umathfractiondelsize num2 FractionNumeratorDisplayStyleGapMin
T, T', S, S', SS, SS' \Umathfractiondelsize rule default_rule_thickness FractionRuleThickness
\Umathskewedfractionhgap math_quad/2 SkewedFractionHorizontalGap
\Umathskewedfractionvgap math_x_height SkewedFractionVerticalGap
\Umathlimitabovebgap big_op_spacing3 UpperLimitBaselineRiseMin
\Umathlimitabovekern big_op_spacing5 0
\Umathlimitabovevgap big_op_spacing1 UpperLimitGapMin
\Umathlimitbelowbgap big_op_spacing4 LowerLimitBaselineDropMin
\Umathlimitbelowkern big_op_spacing5 0
\Umathlimitbelowvgap big_op_spacing2 LowerLimitGapMin
\Umathoverdelimitervgap big_op_spacing1 StretchStackGapBelowMin
\Umathoverdelimiterbgap big_op_spacing3
\texttt{\textbackslash \text{SubscriptShiftDown}} \quad \texttt{sub\_drop} \\
\texttt{\textbackslash \text{SubscriptBaselineDropMin}}

8 \texttt{\textbackslash \text{SubscriptShiftDownWithSuperscript}} \\
\texttt{\textbackslash \text{SubscriptTopMax}} \\
\texttt{\textbackslash \text{SubSuperscriptGapMin}} \\
\texttt{\textbackslash \text{SuperscriptBottomMin}} \\
\texttt{\textbackslash \text{SuperscriptBaselineDropMax}} \\
\texttt{\textbackslash \text{SuperscriptShiftUp}} \quad \texttt{sup\_drop} \\
\texttt{\textbackslash \text{SuperscriptBaselineDropMax}} \\
\texttt{\textbackslash \text{SuperscriptShiftUp}} \quad \texttt{D} \\
\texttt{\textbackslash \text{SuperscriptShiftUp}} \quad \texttt{sup\_2} \\
\texttt{\textbackslash \text{SuperscriptShiftUp}} \quad \texttt{T, S, SS, D', T', S', SS'} \\
\texttt{\textbackslash \text{SuperscriptShiftUpCrammed}} \\
\texttt{\textbackslash \text{SuperscriptBottomMaxWithSubscript}} \\
\texttt{\textbackslash \text{SuperscriptBottomMaxWithSubscript}} \\
\texttt{\textbackslash \text{UnderbarExtraDescender}} \\
\texttt{\textbackslash \text{UnderbarRuleThickness}} \\
\texttt{\textbackslash \text{UnderbarVerticalGap}}

5 \texttt{\textbackslash \text{MinConnectorOverlap}} \quad \texttt{0}

\texttt{\textbackslash \text{UnderbarExtraDescender}} \\
\texttt{\textbackslash \text{UnderbarRuleThickness}} \\
\texttt{\textbackslash \text{UnderbarVerticalGap}}

Note 1: OpenType fonts set \texttt{\textbackslash \text{LimitAboveKern}} and \texttt{\textbackslash \text{LimitBelowKern}} to zero and set \texttt{\textbackslash \text{Quad}} to the font size of the used font, because these are not supported in the MATH table.

Note 2: Traditional \texttt{\textbackslash \text{tfm}} fonts do not set \texttt{\textbackslash \text{RadicalRule}} because \TeX{} uses the height of the radical instead. When this parameter is indeed not set when \texttt{\textbackslash \text{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 \texttt{\textbackslash \text{RadicalDegreeBefore}}, \texttt{\textbackslash \text{RadicalDegreeAfter}}, and \texttt{\textbackslash \text{RadicalDegreeRaise}}. These are then automatically initialized to 5/18 quad, −10/18 quad, and 60.

Note 3: If \texttt{\textbackslash \text{tfm}} fonts are used, then the \texttt{\textbackslash \text{RadicalVGap}} is not set until the first time \texttt{\textbackslash \text{LuaTeX}} has to typeset a formula because this needs parameters from both family 2 and family 3. This provides a partial backward compatibility with \TeX{}82, but that compatibility is only partial: once the \texttt{\textbackslash \text{RadicalVGap}} is set, it will not be recalculated any more.

Note 4: When \texttt{\textbackslash \text{tfm}} fonts are used a similar situation arises with respect to \texttt{\textbackslash \text{SpaceAfterScript}}: it is not set until the first time \texttt{\textbackslash \text{LuaTeX}} has to typeset a formula. This provides some
backward compatibility with \TeX82. But once the \texttt{\umathspaceafterscript} is set, \texttt{\scriptspace} will never be looked at again.

Note 5: Traditional tfm fonts set \texttt{\umathconnectoroverlapmin} to zero because \TeX82 always stacks extensibles without any overlap.

Note 6: The \texttt{\umathoperatorsize} is only used in \texttt{\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 \texttt{\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: Subscript\texttt{\textX} does not actually exist in the ‘standard’ OpenType math font Cambria, but it is useful enough to be added.

Note 9: \texttt{\textX} and \texttt{\textX} do not actually exist in the ‘standard’ OpenType math font Cambria, but were useful enough to be added.

7.5 Math spacing

7.5.1 Inline surrounding space

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

\begin{verbatim}
\mathsurround 10pt \\
\mathsurroundskip20pt
\end{verbatim}

Method six omits the surround glue when there is (x)spacing glue present while method seven does the opposite, the glue is only applied when there is (x)space glue present too. 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).

### 7.5.2 Pairwise spacing

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 $\TeX$book). The primitive names are a simple matter of combining two math atom types, but for completeness’ sake, here is the whole list:

\begin{itemize}
  \item \texttt{\textbackslash Umathordordspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenordspacing}
  \item \texttt{\textbackslash Umathordopspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenopspacing}
  \item \texttt{\textbackslash Umathordbinspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenbinspacing}
  \item \texttt{\textbackslash Umathordrelspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenrelspacing}
  \item \texttt{\textbackslash Umathordopenspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenopenspacing}
  \item \texttt{\textbackslash Umathordclosespacing}  \hspace{1cm} \texttt{\textbackslash Umathopenclosespacing}
  \item \texttt{\textbackslash Umathordpunctspacing}  \hspace{1cm} \texttt{\textbackslash Umathopenpunctspacing}
  \item \texttt{\textbackslash Umathordinnerspacing}  \hspace{1cm} \texttt{\textbackslash Umathopeninnerspacing}
  \item \texttt{\textbackslash Umathopordspacing}  \hspace{1cm} \texttt{\textbackslash Umathcloseordspacing}
  \item \texttt{\textbackslash Umathopopspacing}  \hspace{1cm} \texttt{\textbackslash Umathcloseopspacing}
  \item \texttt{\textbackslash Umathopbinspacing}  \hspace{1cm} \texttt{\textbackslash Umathclosebinspacing}
  \item \texttt{\textbackslash Umathoprelspacing}  \hspace{1cm} \texttt{\textbackslash Umathcloserelspacing}
  \item \texttt{\textbackslash Umathopopenspacing}  \hspace{1cm} \texttt{\textbackslash Umathcloseopenspacing}
  \item \texttt{\textbackslash Umathopclosespacing}  \hspace{1cm} \texttt{\textbackslash Umathcloseclosespacing}
  \item \texttt{\textbackslash Umathoppunctspacing}  \hspace{1cm} \texttt{\textbackslash Umathclosepunctspacing}
  \item \texttt{\textbackslash Umathopinnerspacing}  \hspace{1cm} \texttt{\textbackslash Umathcloseinnerspacing}
  \item \texttt{\textbackslash Umathbinordspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctordspacing}
  \item \texttt{\textbackslash Umathbinopspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctopspacing}
  \item \texttt{\textbackslash Umathbinbinspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctbinspacing}
  \item \texttt{\textbackslash Umathbinrelspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctrelspacing}
  \item \texttt{\textbackslash Umathbinopenspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctopenspacing}
  \item \texttt{\textbackslash Umathbinrelclosespacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctclosespacing}
  \item \texttt{\textbackslash Umathbinrelpunctspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctpunctspacing}
  \item \texttt{\textbackslash Umathbinrelinnerspacing}  \hspace{1cm} \texttt{\textbackslash Umathpunctinnerspacing}
  \item \texttt{\textbackslash Umathrelordspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerordspacing}
  \item \texttt{\textbackslash Umathrelordspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerordspacing}
  \item \texttt{\textbackslash Umathrelbinspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerbinspacing}
  \item \texttt{\textbackslash Umathrelrelspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerrelspacing}
  \item \texttt{\textbackslash Umathrelclosespacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerclosespacing}
  \item \texttt{\textbackslash Umathrelpunctspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerpunctspacing}
  \item \texttt{\textbackslash Umathrelinnerspacing}  \hspace{1cm} \texttt{\textbackslash Umathinnerinnerspacing}
\end{itemize}

These parameters are of type $\\textbackslash \text{muskip}$, so setting a parameter can be done like this:
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.

### 7.5.3 Skips around display math

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 \LaTeX\ comes with a new directive: \mathdisplayskipmode. The following values apply:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal \TeX\ behaviour</td>
</tr>
<tr>
<td>1</td>
<td>always (same as 0)</td>
</tr>
<tr>
<td>2</td>
<td>only when not zero</td>
</tr>
<tr>
<td>3</td>
<td>never, not even when not zero</td>
</tr>
</tbody>
</table>

By default the short skip detection is not adapted to r2l typesetting and that hasn’t been the case since the start of the project. Changing it could break hacks that users came up with but when you set \matheqdirmode to a positive value direction will be taken into account.

### 7.5.4 Nolimit correction

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.

\[
\begin{array}{cccccc}
0 & 0 & 0 & +ic/2 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & +ic/2 & +ic/2 & 8000ic/1000 \\
\end{array}
\]
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.

### 7.5.5 Math italic mess

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). A value of 2 will enforce the old school font code path for all italics. We show a Cambria example.

\mathitalicsmode = 0 \[ \frac{1}{T} \frac{|T|}{T + 1} \frac{1}{2} \frac{T}{\sqrt{T}} \]
\mathitalicsmode = 1 \[ \frac{1}{T} \frac{|T|}{T + 1} \frac{1}{2} \frac{T}{\sqrt{T}} \]

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

### 7.5.6 Script and kerning

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 a \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.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>forget about kerning</td>
</tr>
<tr>
<td>1</td>
<td>kern math sub lists with a valid glyph</td>
</tr>
<tr>
<td>2</td>
<td>also kern math sub boxes that have a valid glyph</td>
</tr>
<tr>
<td>3</td>
<td>only kern math sub boxes with a boundary node present</td>
</tr>
</tbody>
</table>

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 korns.

<table>
<thead>
<tr>
<th>\ST_{\text{fluff}}</th>
<th>\ST_{\text{fluff}}</th>
<th>\ST_{\text{fluff}}</th>
<th>\ST_{\text{fluff}}</th>
<th>\ST_{\text{fluff}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>modern</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
</tr>
<tr>
<td>lucidaot</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
</tr>
<tr>
<td>pagella</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
</tr>
<tr>
<td>cambria</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
</tr>
<tr>
<td>dejavu</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
<td>\T_{\text{fluff}}</td>
</tr>
</tbody>
</table>

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 bounding box, 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.

<table>
<thead>
<tr>
<th>Normal Modern</th>
<th>Pagella</th>
<th>Cambria</th>
<th>Lucidaot</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( f )</td>
<td>( \gamma )</td>
<td>( \epsilon )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bold Modern</th>
<th>Pagella</th>
<th>Cambria</th>
<th>Lucidaot</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( f )</td>
<td>( \gamma )</td>
<td>( \epsilon )</td>
</tr>
</tbody>
</table>

### 7.5.7 Fixed scripts

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d )</td>
<td>\texttt{\textbackslash Umathsubshiftdown}</td>
</tr>
<tr>
<td>( u )</td>
<td>\texttt{\textbackslash Umathsupshiftup}</td>
</tr>
<tr>
<td>( s )</td>
<td>\texttt{\textbackslash Umathsubsupshiftdown}</td>
</tr>
</tbody>
</table>

When we set \texttt{\textbackslash 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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Down</th>
<th>Up</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dynamic</td>
<td>dynamic</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
<tr>
<td>1</td>
<td>( d )</td>
<td>( u )</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
<tr>
<td>2</td>
<td>( s )</td>
<td>( u )</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
<tr>
<td>3</td>
<td>( s )</td>
<td>( u + s - d )</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
<tr>
<td>4</td>
<td>( d + (s - d)/2 )</td>
<td>( u + (s - d)/2 )</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
<tr>
<td>5</td>
<td>( d )</td>
<td>( u + s - d )</td>
<td>( \text{CH}_2 + \text{CH}_2^+ + \text{CH}_2^- )</td>
</tr>
</tbody>
</table>

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

### 7.5.8 Penalties: \texttt{\textbackslash mathpenaltiesmode}

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

\texttt{\textbackslash mathpenaltiesmode = 1}
This primitive is not really needed in \LaTeX{} because you can use the callback \texttt{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:

\begin{verbatim}
\prebinoppenalty = -100 \% example value
\prerelpenalty = 900 \% example value
\end{verbatim}

They default to infinite 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.

### 7.5.9 Equation spacing: \texttt{\matheqnogapstep}

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

\begin{verbatim}
\matheqnogapstep = 1000
\end{verbatim}

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 mimic floating point factors in \TeX{}.

### 7.6 Math constructs

#### 7.6.1 Unscaled fences

The \texttt{\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 \texttt{mathitalicsmode} enabled).

\begin{verbatim}
\mathdelimitersmode = 0 \{[\}] \{[\}]
\mathdelimitersmode = 1 \{[\}] \{[\}]
\mathdelimitersmode = 2 \{[\}] \{[\}]
\mathdelimitersmode = 3 \{[\}] \{[\}]
\mathdelimitersmode = 4 \{[\}] \{[\}]
\mathdelimitersmode = 5 \{[\}] \{[\}]
\mathdelimitersmode = 6 \{[\}] \{[\}]
\mathdelimitersmode = 7 \{[\}] \{[\}]
\end{verbatim}

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 \texttt{\left} and \texttt{\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:
<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;01&quot;</td>
<td>don’t apply the usual shift</td>
</tr>
<tr>
<td>&quot;02&quot;</td>
<td>apply italic correction when possible</td>
</tr>
<tr>
<td>&quot;04&quot;</td>
<td>force an ordinary subformula</td>
</tr>
<tr>
<td>&quot;08&quot;</td>
<td>no shift when a base character</td>
</tr>
<tr>
<td>&quot;10&quot;</td>
<td>only shift when an extensible</td>
</tr>
</tbody>
</table>

The effect can depend on the font (and for Cambria one can use for instance "16").

### 7.6.2 Accent handling

Lua\TeX{} 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:

\begin{verbatim}
\Umathaccent both fixed 0 0 "20D7 fixed 0 0 "20D7 {example}
\end{verbatim}

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 \TeX{}X82.

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 \texttt{x.height} of the font of the accentee (as explained in the \TeX{}Xbook), 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.
7.6.3 Radical extensions

The new primitive \texttt{\textbackslash\texttt{Uroot}} allows the construction of a radical node including a degree field. Its syntax is an extension of \texttt{\textbackslash\texttt{Urada}}l:

\texttt{\textbackslash\texttt{Urada}}l \texttt{<fam integer> <char integer> <radicand>}
\texttt{\textbackslash\texttt{Uroot} <fam integer> <char integer> <degree> <radicand>}

The placement of the degree is controlled by the math parameters \texttt{\textbackslash\texttt{Mmathradicaldegreebefore}}, \texttt{\textbackslash\texttt{Mmathradicaldegreeafter}}, and \texttt{\textbackslash\texttt{Mmathradicaldegreeraise}. The degree will be typeset in }\texttt{\textbackslash\texttt{scriptscriptstyle}.}

7.6.4 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, Lua\TeX\ 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 Lua\TeX\ 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).

7.6.5 Scripts on extensibles

The primitives \texttt{\textbackslash\texttt{Uunderdelimiter}} and \texttt{\textbackslash\texttt{Uoverdelimiter}} allow the placement of a subscript or superscript on an automatically extensible item and \texttt{\textbackslash\texttt{Udelimiterunder}} and \texttt{\textbackslash\texttt{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
-----
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 accept a 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 \U/extensible is available that can be used like this:

\[ \U/extensible \ \text{width} \ 10cm \ \ 0 \ 2194 \]

This will render this:

-----

Here you can also pass options, like:

\[ \U/extensible \ \text{width} \ 1pt \ \text{middle} \ \ 0 \ 2194 \]

This gives:

-----

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.

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. In the following example the all-zero specification is the trigger to make a fake box with the last delimiter dimensions and shift. It’s an ugly hack but its relative simple and not intrusive implementation has no side effects. Any other heuristic solution would not satisfy possible demands anyway.

Here is a rather low level example:

\startformula
\Uleft \ \Udelimiter \ 5 \ 0 \ 222B \\
\frac{\frac{a}{b}}{\frac{c}{d}}
\endformula
The last line, by passing zero values, results in a fake operator that has the dimensions of the previous delimiter. We can then backtrack over the (presumed) width and the two numbers become limit operators. As said, it’s not pretty but it works.

### 7.6.6 Fractions

The \texttt{\above\withdelims} command accepts a keyword \texttt{exact}. When issued the extra space relative to the rule thickness is not added. One can of course use the \texttt{\Umath\fraction..gap} commands to influence the spacing. Also the rule is still positioned around the math axis.

\begin{equation}
\{ \{a\} \above\withdelims() \texttt{exact} 4\text{pt} \{b\} \}
\end{equation}

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 \texttt{\Umathskewed\fraction\hgap} and \texttt{\Umathskewed\fraction\vgap}.

The syntax used here is:

\begin{verbatim}
{ \{1\} \Uskewed / \texttt{<options>} \{2\} }
{ \{1\} \Uskewed\withdelims / () \texttt{<options>} \{2\} }
\end{verbatim}

where the options can be \texttt{noaxis} and \texttt{exact}. By default we zero the width of the middle character. For Latin Modern the result looks as follows:

\begin{verbatim}
\texttt{x + a/b + x} x + \frac{1}{2} + x x + \frac{a/b}{x} x + \frac{1/2}{x} + x
\texttt{exact} x + \frac{a/b}{x} x + \frac{1/2}{x} + x x + \frac{a/b}{x} x + \frac{1/2}{x} + x
\texttt{noaxis} x + a/b + x x + \frac{1}{2} + x x + a/b + x x + \frac{1/2}{x} + x
\texttt{exact noaxis} x + a/b + x x + \frac{1/2}{x} + x x + a/b + x x + \frac{1/2}{x} + x
\end{verbatim}

The keyword \texttt{norule} will hide the rule with the above variants while keeping the rule related spacing.

### 7.6.7 Delimiters: \texttt{\Uleft}, \texttt{\Umiddle} and \texttt{\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 if the used characters in the font have depth or height. Commands like (plain \TeX) \texttt{\big} need use this feature. In Lua\TeX{} we provide a bit more control by three variants that support optional parameters height, depth and axis. The following example uses this:

\begin{verbatim}
\Uleft height 30pt depth 10pt \Udelimiter "0 "0 "000028 \quad x\quad
\end{verbatim}
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.

### 7.7 Extracting values

#### 7.7.1 Codes

You can extract the components of a math character. Say that we have defined:

\[ \text{\textbackslash mathcode 1 2 3 4} \]

then

\[ [\text{\textbackslash mathcharclass1}] [\text{\textbackslash mathcharfam1}] [\text{\textbackslash mathcharslot1}] \]

will return:

\[ [2] [3] [4] \]

These commands are provides as convenience. Before they come available you could do the following:

\[ \text{\texttt{\textbackslash def\textbackslash mathcharclass}} \{\texttt{\textbackslash directlua{\textbackslash tex.print(\textbackslash tex.getmathcode(token.scan_int()))[1]}}} \}\]

\[ \text{\texttt{\textbackslash def\textbackslash mathcharfam}} \{\texttt{\textbackslash directlua{\textbackslash tex.print(\textbackslash tex.getmathcode(token.scan_int()))[2]}}} \}\]

\[ \text{\texttt{\textbackslash def\textbackslash mathcharslot}} \{\texttt{\textbackslash directlua{\textbackslash tex.print(\textbackslash tex.getmathcode(token.scan_int()))[3]}}} \}\]

#### 7.7.2 Last lines

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:
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 ( \textwidth - (nodes.dimensions(node.tail(tex.lists.page_head).list)) \textwidth - sp) \relax}

7.8 Math mode

7.8.1 Verbose versions of single-character math commands

LuaTEX defines six new primitives that have the same function as \^, \_, $, and $$:

<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Usuperscript</td>
<td>duplicates the functionality of ^</td>
</tr>
<tr>
<td>\Usubscript</td>
<td>duplicates the functionality of _</td>
</tr>
<tr>
<td>\Ustartmath</td>
<td>duplicates the functionality of $, when used in non-math mode.</td>
</tr>
<tr>
<td>\Ustopmath</td>
<td>duplicates the functionality of $, when used in inline math mode.</td>
</tr>
<tr>
<td>\Ustartdisplaymath</td>
<td>duplicates the functionality of $$, when used in non-math mode.</td>
</tr>
<tr>
<td>\Ustopdisplaymath</td>
<td>duplicates the functionality of $$, when used in display math mode.</td>
</tr>
</tbody>
</table>

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).

7.8.2 Script commands \Unosuperscript and \Unosubscript

These two commands result in super- and subscripts but with the current style (at the time of rendering). So,

\$
\begin{align*}
  x\Usuperscript {1}\Usubscript {2} &= x\Unosuperscript{1}\Unosubscript{2} \\
  x\Ustartmath\Unosuperscript{1}\Usubscript{2} &= x\Usuperscript {1}\Unosubscript{2} \\
  x\Ustopmath\Unosuperscript{1}\Usubscript{2} &= x\Unosuperscript{1}\Usubscript{2}
\end{align*}
$

results in $x_2^1 = x_{1/2} = x_{1/2} = x_{1/2}$.  

124 Math
7.8.3 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.

7.9 Goodies

7.9.1 Flattening: \mathflattenmode

The \TeX math engine collapses ord noads without sub- and superscripts and a character as nucleus. and 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:

\[ V v V \]
\[ V v V \]

When we set \mathflattenmode to 31 we get:

\[ V v V \]
\[ V v V \]

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.

<table>
<thead>
<tr>
<th>MODE</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ord</td>
</tr>
<tr>
<td>2</td>
<td>bin</td>
</tr>
<tr>
<td>4</td>
<td>rel</td>
</tr>
<tr>
<td>8</td>
<td>punct</td>
</tr>
<tr>
<td>16</td>
<td>inner</td>
</tr>
</tbody>
</table>

7.9.2 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.
7.9.3 Math options with \mathdefaultsmode

This option has been introduced because \LaTeX developers wanted some of the defaults to be different from the ones that were set in stone when we froze \LaTeX. The default values are:

<table>
<thead>
<tr>
<th></th>
<th>SCANNING</th>
<th>RENDERING</th>
</tr>
</thead>
<tbody>
<tr>
<td>radical/root</td>
<td>cramped</td>
<td>cramped</td>
</tr>
<tr>
<td>under delimiter</td>
<td>cramped</td>
<td>supstyle</td>
</tr>
<tr>
<td>over delimiter</td>
<td>cramped</td>
<td>substyle</td>
</tr>
<tr>
<td>delimiter under</td>
<td>cramped</td>
<td>current</td>
</tr>
<tr>
<td>delimiter over</td>
<td>cramped</td>
<td>current</td>
</tr>
</tbody>
</table>

When \mathdefaultsmode is larger than zero, we have:

<table>
<thead>
<tr>
<th></th>
<th>SCANNING</th>
<th>RENDERING</th>
</tr>
</thead>
<tbody>
<tr>
<td>radical/root</td>
<td>cramped</td>
<td>cramped</td>
</tr>
<tr>
<td>under delimiter</td>
<td>substyle</td>
<td>substyle</td>
</tr>
<tr>
<td>over delimiter</td>
<td>supstyle</td>
<td>supstyle</td>
</tr>
<tr>
<td>delimiter under</td>
<td>current</td>
<td>current</td>
</tr>
<tr>
<td>delimiter over</td>
<td>cramped</td>
<td>cramped</td>
</tr>
</tbody>
</table>

It is outside the scope of this manual to discuss the rationale behind these defaults. The zero values date back from the early times. If needed you can explicitly set the style in the content argument.

7.9.4 Math options with \mathoption

The logic in the math engine is rather complex and there are often no universal solutions (read: what works out well for one font, fails for another). Therefore some variations in the implementation are driven by parameters (modes). In addition there is a new primitive \mathoption which will be used for testing. Don’t rely on any option to be there in a production version as they are meant for development.

This option was introduced for testing purposes when the math engine got split code paths and it forces the engine to treat new fonts as old ones with respect to italic correction etc. There are no guarantees given with respect to the final result and unexpected side effects are not seen as bugs as they relate to font properties. There is currently only one option:

The oldmath boolean flag in the Lua font table is the official way to force old treatment as it’s bound to fonts. Like with all options we may temporarily introduce with this command this feature is not meant for production.
8 Nodes

8.1 LUA node representation

\TeX's nodes are represented in Lua as userdata objects with a variable set of fields. In the following syntax tables, such as the type of such a userdata object is represented as ⟨node⟩.

The current return value of node.types() is: hlist (0), vlist (1), rule (2), ins (3), mark (4), adjust (5), boundary (6), disc (7), whatsit (8), local_par (9), dir (10), math (11), glue (12), kern (13), penalty (14), unset (15), style (16), choice (17), noad (18), radical (19), fraction (20), accent (21), fence (22), math_char (23), sub_box (24), sub_mlist (25), math_text_char (26), delim (27), margin_kern (28), glyph (29), align_record (30), pseudo_file (31), pseudo_line (32), page_insert (33), split_insert (34), expr_stack (35), nested_list (36), span (37), attribute (38), glue_spec (39), attribute_list (40), temp (41), align_stack (42), movement_stack (43), if_stack (44), unhyphenated (45), hyphenated (46), delta (47), passive (48), shape (49).

The \lastnodetype primitive is \e-\TeX compliant. The valid range is still [−1, 15] and glyph nodes (formerly known as char nodes) have number 0 while ligature nodes are mapped to 7. That way macro packages can use the same symbolic names as in traditional \e-\TeX. Keep in mind that these \e-\TeX node numbers are different from the real internal ones and that there are more \e-\TeX node types than 15.

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 dir, direction, glue, pdf_literal, pdf_action, pdf_window and color_stack. 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 page state values are also reported with this helper.

8.2 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:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>node</td>
<td>the next node in a list, or nil</td>
</tr>
<tr>
<td>id</td>
<td>number</td>
<td>the node's type (id) number</td>
</tr>
<tr>
<td>subtype</td>
<td>number</td>
<td>the node subtype identifier</td>
</tr>
</tbody>
</table>

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.

### 8.2.1 hlist nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = unknown, 1 = line, 2 = box, 3 = indent, 4 = alignment, 5 = cell, 6 = equation, 7 = equationnumber, 8 = math, 9 = mathchar, 10 = hex-tensible, 11 = vextensible, 12 = hdelimiter, 13 = vdelimiter, 14 = overdelimiter, 15 = underdelimiter, 16 = numerator, 17 = denominator, 18 = limits, 19 = fraction, 20 = nucleus, 21 = sup, 22 = sub, 23 = degree, 24 = scripts, 25 = over, 26 = under, 27 = accent, 28 = radical</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width of the box</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height of the box</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth of the box</td>
</tr>
<tr>
<td>shift</td>
<td>number</td>
<td>a displacement perpendicular to the character progression direction</td>
</tr>
<tr>
<td>glue_order</td>
<td>number</td>
<td>a number in the range [0, 4], indicating the glue order</td>
</tr>
<tr>
<td>glue_set</td>
<td>number</td>
<td>the calculated glue ratio</td>
</tr>
<tr>
<td>glue_sign</td>
<td>number</td>
<td>0 = normal, 1 = stretching, 2 = shrinking</td>
</tr>
<tr>
<td>head/list</td>
<td>node</td>
<td>the first node of the body of this list</td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction of this box, see 8.2.15</td>
</tr>
</tbody>
</table>

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.

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.

### 8.2.2 vlist nodes

This node is similar to hlist, except that ‘shift’ is a displacement perpendicular to the line progression direction, and ‘subtype’ only has the values 0, 4, and 5.

### 8.2.3 rule nodes

Contrary to traditional TeX, LuaTeX has more \rule subtypes because we also use rules to store reuseable objects and images. User nodes are invisible and can be intercepted by a callback.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = normal, 1 = box, 2 = image, 3 = empty, 4 = user, 5 = over, 6 = under, 7 = fraction, 8 = radical, 9 = outline</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width of the rule where the special value \texttt{-1073741824} is used for ‘running’ glue dimensions</td>
</tr>
</tbody>
</table>
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 8.2.15
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.

8.2.4 ins nodes

This node relates to the \insert primitive.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>the insertion class</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>cost</td>
<td>number</td>
<td>the penalty associated with this insert</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>height of the insert</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>depth of the insert</td>
</tr>
<tr>
<td>head/list</td>
<td>node</td>
<td>the first node of the body of this insert</td>
</tr>
</tbody>
</table>

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).

8.2.5 mark nodes

This one relates to the \mark primitive.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>unused</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>class</td>
<td>number</td>
<td>the mark class</td>
</tr>
<tr>
<td>mark</td>
<td>table</td>
<td>a table representing a token list</td>
</tr>
</tbody>
</table>
8.2.6 adjust nodes

This node comes from \adjust primitive.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = normal, 1 = pre</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>head/list</td>
<td>node</td>
<td>adjusted material</td>
</tr>
</tbody>
</table>

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.

8.2.7 disc nodes

The \discretionary and \-, the - character but also the hyphenation mechanism produces these nodes.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = discretionary, 1 = explicit, 2 = automatic, 3 = regular, 4 = first, 5 = second</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>pre</td>
<td>node</td>
<td>pointer to the pre-break text</td>
</tr>
<tr>
<td>post</td>
<td>node</td>
<td>pointer to the post-break text</td>
</tr>
<tr>
<td>replace</td>
<td>node</td>
<td>pointer to the no-break text</td>
</tr>
<tr>
<td>penalty</td>
<td>number</td>
<td>the penalty associated with the break, normally \hyphenpenalty or \exhyphenpenalty</td>
</tr>
</tbody>
</table>

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:

```lua
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 LuaTeX 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 LuaTeX).

8.2.8 math nodes

Math nodes represent the boundaries of a math formula, normally wrapped into $ signs.
There is a set of extra fields that concern the associated glue: width, stretch, stretch_order, shrink and shrink_order. These are all numbers.

### 8.2.9 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 is called a glue_spec, and it has the following accessible fields:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>number</td>
<td>the horizontal or vertical displacement</td>
</tr>
<tr>
<td>stretch</td>
<td>number</td>
<td>extra (positive) displacement or stretch amount</td>
</tr>
<tr>
<td>stretch_order</td>
<td>number</td>
<td>factor applied to stretch amount</td>
</tr>
<tr>
<td>shrink</td>
<td>number</td>
<td>extra (negative) displacement or shrink amount</td>
</tr>
<tr>
<td>shrink_order</td>
<td>number</td>
<td>factor applied to shrink amount</td>
</tr>
</tbody>
</table>

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 \texttt{effective_glue} function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item. When you pass \texttt{true} as third argument the value will be rounded.

A glue_spec node is a special kind of node that is used for storing a set of glue values in registers. Originally they were also used to store properties of glue nodes (using a system of reference counts) but we now keep these properties in the glue nodes themselves, which gives a cleaner interface to Lua.

The indirect spec approach was in fact an optimization in the original \TeX code. First of all it can save quite some memory because all these spaces that become glue now share the same specification (only the reference count is incremented), and zero testing is also a bit faster because only the pointer has to be checked (this is no longer true for engines that implement for instance protrusion where we really need to ensure that zero is zero when we test for bounds). Another side effect is that glue specifications are read-only, so in the end copies need to be made when they are used from Lua (each assignment to a field can result in a new copy). So in the end the advantages of sharing are not that high (and nowadays memory is less an issue, also given that a glue node is only a few memory words larger than a spec).
In addition there are the width, stretch stretch_order, shrink, and shrink_order fields. 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.

A regular word space also results in a spaceskip subtype (this used to be a userskip with subtype zero).

### 8.2.10 kern nodes

The \texttt{kern} command creates such nodes but for instance the font and math machinery can also add them.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = fontkern, 1 = userkern, 2 = accentkern, 3 = italiccorrection</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>kern</td>
<td>number</td>
<td>fixed horizontal or vertical advance</td>
</tr>
</tbody>
</table>

### 8.2.11 penalty nodes

The \texttt{penalty} command is one that generates these nodes.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = userpenalty, 1 = linebreakpenalty, 2 = linepenalty, 3 = wordpenalty, 4 = finalpenalty, 5 = noadpenalty, 6 = beforedisplaypenalty, 7 = afterdisplaypenalty, 8 = equationnumberpenalty</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>penalty</td>
<td>number</td>
<td>the penalty value</td>
</tr>
</tbody>
</table>

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.

### 8.2.12 glyph nodes

These are probably the mostly used nodes and although you can push them in the current list with for instance \texttt{\char} \TeX will normally do it for you when it considers some input to be text.
attr node list of attributes
char number the character index in the font
font number the font identifier
lang number the language identifier
left number the frozen \lefthyphenmin value
right number the frozen \righthyphenmin value
uchyph boolean the frozen \uchyph value
components node pointer to ligature components
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.

A warning: never assign a node list to the components field unless you are sure its internal link structure is correct, otherwise an error may be result. Valid bits for the subtype field are:

```
BIT MEANING
0 character
1 ligature
2 ghost
3 left
4 right
```

See section 5.2 for a detailed description of the subtype field.

The expansion_factor has been introduced as part of the separation between font- 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.
### 8.2.13 boundary nodes

This node relates to the \noboundary, \boundary, \protrusionboundary and \wordboundary primitives.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = cancel, 1 = user, 2 = protrusion, 3 = word</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>value</td>
<td>number</td>
<td>values 0-255 are reserved</td>
</tr>
</tbody>
</table>

### 8.2.14 local_par nodes

This node is inserted at the start of a paragraph. You should not mess too much with this one.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>pen_inter</td>
<td>number</td>
<td>local interline penalty (from \localinterlinepenalty)</td>
</tr>
<tr>
<td>pen_broken</td>
<td>number</td>
<td>local broken penalty (from \localbrokenpenalty)</td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction of this par. see 8.2.15</td>
</tr>
<tr>
<td>box_left</td>
<td>node</td>
<td>the \localleftbox</td>
</tr>
<tr>
<td>box_left_width</td>
<td>number</td>
<td>width of the \localleftbox</td>
</tr>
<tr>
<td>box_right</td>
<td>node</td>
<td>the \localrightbox</td>
</tr>
<tr>
<td>box_right_width</td>
<td>number</td>
<td>width of the \localrightbox</td>
</tr>
</tbody>
</table>

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.

### 8.2.15 dir nodes

Direction nodes mark parts of the running text that need a change of direction and the \textdir command generates them.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction (but see below)</td>
</tr>
<tr>
<td>level</td>
<td>number</td>
<td>nesting level of this direction whatsis</td>
</tr>
</tbody>
</table>

Direction specifiers are three-letter combinations of T, B, R, and L. These are built up out of three separate items:

- the first is the direction of the ‘top’ of paragraphs
- the second is the direction of the ‘start’ of lines
- the third is the direction of the ‘top’ of glyphs

However, only four combinations are accepted: TLT, TRT, RTT, and LTL. Inside actual dir nodes, the representation of dir is not a three-letter but a combination of numbers. When printed the
8.2.16 marginkern nodes

Margin kerns result from protrusion.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = left, 1 = right</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the advance of the kern</td>
</tr>
<tr>
<td>glyph</td>
<td>node</td>
<td>the glyph to be used</td>
</tr>
</tbody>
</table>

8.3 Math noads

These are the so-called ‘noad’s and the nodes that are specifically associated with math processing. Most of these nodes contain subnodes so that the list of possible fields is actually quite small. First, the subnodes:

8.3.1 Math kernel subnodes

Many object fields in math mode are either simple characters in a specific family or math lists or node lists. There are four associated subnodes that represent these cases (in the following node descriptions these are indicated by the word <kernel>).

The next and prev fields for these subnodes are unused.

8.3.2 math_char and math_text_char subnodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>char</td>
<td>number</td>
<td>the character index</td>
</tr>
<tr>
<td>fam</td>
<td>number</td>
<td>the family number</td>
</tr>
</tbody>
</table>

The math_char is the simplest subnode field, it contains the character and family for a single glyph object. 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).

8.3.3 sub_box and sub_mlist subnodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>head/list</td>
<td>node</td>
<td>list of nodes</td>
</tr>
</tbody>
</table>
These two subnode types are used for subsidiary list items. For sub_box, the head points to a 'normal' vbox or hbox. For sub_mlist, the head points to a math list that is yet to be converted. A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error is triggered.

**8.3.4 delim subnodes**

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the next and prev fields are unused.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>small_char</td>
<td>number</td>
<td>character index of base character</td>
</tr>
<tr>
<td>small_fam</td>
<td>number</td>
<td>family number of base character</td>
</tr>
<tr>
<td>large_char</td>
<td>number</td>
<td>character index of next larger character</td>
</tr>
<tr>
<td>large_fam</td>
<td>number</td>
<td>family number of next larger character</td>
</tr>
</tbody>
</table>

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.

**8.3.5 Math core nodes**

First, there are the objects (the \TeX{}book calls them ‘atoms’) that are associated with the simple math objects: ord, op, bin, rel, open, close, punct, inner, over, under, vcent. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation.

Some noads have an option field. The values in this bitset are common:

<table>
<thead>
<tr>
<th>MEANING</th>
<th>BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>0x08</td>
</tr>
<tr>
<td>internal</td>
<td>0x00 + 0x08</td>
</tr>
<tr>
<td>internal</td>
<td>0x01 + 0x08</td>
</tr>
<tr>
<td>axis</td>
<td>0x02 + 0x08</td>
</tr>
<tr>
<td>no axis</td>
<td>0x04 + 0x08</td>
</tr>
<tr>
<td>exact</td>
<td>0x10 + 0x08</td>
</tr>
<tr>
<td>left</td>
<td>0x11 + 0x08</td>
</tr>
<tr>
<td>middle</td>
<td>0x12 + 0x08</td>
</tr>
<tr>
<td>right</td>
<td>0x14 + 0x08</td>
</tr>
<tr>
<td>no sub script</td>
<td>0x21 + 0x08</td>
</tr>
<tr>
<td>no super script</td>
<td>0x22 + 0x08</td>
</tr>
<tr>
<td>no script</td>
<td>0x23 + 0x08</td>
</tr>
</tbody>
</table>
8.3.6 Simple noad nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = ord, 1 = opdisplaylimits, 2 = oplimits, 3 = opnolimits, 4 = bin, 5 = rel, 6 = open, 7 = close, 8 = punct, 9 = inner, 10 = under, 11 = over, 12 = vcenter</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>nucleus</td>
<td>kernel node</td>
<td>base</td>
</tr>
<tr>
<td>sub</td>
<td>kernel node</td>
<td>subscript</td>
</tr>
<tr>
<td>sup</td>
<td>kernel node</td>
<td>superscript</td>
</tr>
<tr>
<td>options</td>
<td>number</td>
<td>bitset of rendering options</td>
</tr>
</tbody>
</table>

8.3.7 Accent nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = bothflexible, 1 = fixedtop, 2 = fixedbottom, 3 = fixedboth</td>
</tr>
<tr>
<td>nucleus</td>
<td>kernel node</td>
<td>base</td>
</tr>
<tr>
<td>sub</td>
<td>kernel node</td>
<td>subscript</td>
</tr>
<tr>
<td>sup</td>
<td>kernel node</td>
<td>superscript</td>
</tr>
<tr>
<td>accent</td>
<td>kernel node</td>
<td>top accent</td>
</tr>
<tr>
<td>bot_accent</td>
<td>kernel node</td>
<td>bottom accent</td>
</tr>
<tr>
<td>fraction</td>
<td>number</td>
<td>larger step criterium (divided by 1000)</td>
</tr>
</tbody>
</table>

8.3.8 Style nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>string</td>
<td>contains the style</td>
</tr>
</tbody>
</table>

There are eight possibilities for the string value: one of display, text, script, or scriptscript. Each of these can have be prefixed by cramped.

8.3.9 Choice nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>display</td>
<td>node</td>
<td>list of display size alternatives</td>
</tr>
<tr>
<td>text</td>
<td>node</td>
<td>list of text size alternatives</td>
</tr>
<tr>
<td>script</td>
<td>node</td>
<td>list of scriptsizes alternatives</td>
</tr>
<tr>
<td>scriptscript</td>
<td>node</td>
<td>list of scriptscript sizes alternatives</td>
</tr>
</tbody>
</table>

Warning: never assign a node list to the display, text, script, or scriptscript field unless you are sure its internal link structure is correct, otherwise an error can occur.
8.3.10 radical nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = radical, 1 = uradical, 2 = uroot, 3 = uunderdelimiter, 4 = uoverdelimiter, 5 = udelimiterunder, 6 = udelimiterover</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>nucleus</td>
<td>kernel node</td>
<td>base</td>
</tr>
<tr>
<td>sub</td>
<td>kernel node</td>
<td>subscript</td>
</tr>
<tr>
<td>sup</td>
<td>kernel node</td>
<td>superscript</td>
</tr>
<tr>
<td>left</td>
<td>delimiter node</td>
<td></td>
</tr>
<tr>
<td>degree</td>
<td>kernel node</td>
<td>only set by \Uroot</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>required width</td>
</tr>
<tr>
<td>options</td>
<td>number</td>
<td>bitset of rendering options</td>
</tr>
</tbody>
</table>

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.

8.3.11 fraction nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>(optional) width of the fraction</td>
</tr>
<tr>
<td>num</td>
<td>kernel node</td>
<td>numerator</td>
</tr>
<tr>
<td>denom</td>
<td>kernel node</td>
<td>denominator</td>
</tr>
<tr>
<td>left</td>
<td>delimiter node</td>
<td>left side symbol</td>
</tr>
<tr>
<td>right</td>
<td>delimiter node</td>
<td>right side symbol</td>
</tr>
<tr>
<td>middle</td>
<td>delimiter node</td>
<td>middle symbol</td>
</tr>
<tr>
<td>options</td>
<td>number</td>
<td>bitset of rendering options</td>
</tr>
</tbody>
</table>

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.

8.3.12 fence nodes

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = unset, 1 = left, 2 = middle, 3 = right, 4 = no</td>
</tr>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>delim</td>
<td>delimiter node</td>
<td>delimiter specification</td>
</tr>
<tr>
<td>italic</td>
<td>number</td>
<td>italic correction</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>required height</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>required depth</td>
</tr>
<tr>
<td>options</td>
<td>number</td>
<td>bitset of rendering options</td>
</tr>
<tr>
<td>class</td>
<td>number</td>
<td>spacing related class</td>
</tr>
</tbody>
</table>

Warning: some of these fields are used by the renderer and might get adapted in the process.
### 8.4 Front-end whatsits

Whatsit nodes come in many subtypes that you can ask for them by running `node.whatsits`: open (0), write (1), close (2), special (3), late_special (4), save_pos (7), late_lua (8), user_defined (9), pdf_literal (16), pdf_late_literal (17), pdf_refobj (18), pdf_annot (19), pdf_start_link (20), pdf_end_link (21), pdf_dest (22), pdf_action (23), pdf_thread (24), pdf_start_thread (25), pdf_end_thread (26), pdf_thread_data (27), pdf_link_data (28), pdf_colorstack (29), pdf_setmatrix (30), pdf_save (31), pdf_restore (32), pdf_link_state (33).

Some of them are generic and independent of the output mode and others are specific to the chosen backend: dvi or pdf. Here we discuss the generic font-end nodes nodes.

#### 8.4.1 open

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>stream</td>
<td>number</td>
<td>\TeX’s stream id number</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>file name</td>
</tr>
<tr>
<td>ext</td>
<td>string</td>
<td>file extension</td>
</tr>
<tr>
<td>area</td>
<td>string</td>
<td>file area (this may become obsolete)</td>
</tr>
</tbody>
</table>

#### 8.4.2 write

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>stream</td>
<td>number</td>
<td>\TeX’s stream id number</td>
</tr>
<tr>
<td>data</td>
<td>table</td>
<td>a table representing the token list to be written</td>
</tr>
</tbody>
</table>

#### 8.4.3 close

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>stream</td>
<td>number</td>
<td>\TeX’s stream id number</td>
</tr>
</tbody>
</table>

#### 8.4.4 user_defined

User-defined whatsit nodes can only be created and handled from Lua code. In effect, they are an extension to the extension mechanism. The Lua\TeX{} engine will simply step over such whatsits without ever looking at the contents.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>user_id</td>
<td>number</td>
<td>id number</td>
</tr>
</tbody>
</table>
### 8.4.5 save_pos

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
</tbody>
</table>

### 8.4.6 late_lua

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>data</td>
<td>string or function</td>
<td>the to be written information stored as Lua value</td>
</tr>
<tr>
<td>token</td>
<td>string</td>
<td>the to be written information stored as token list</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>the name to use for Lua error reporting</td>
</tr>
</tbody>
</table>

The difference between data and string is that on assignment, the data field is converted to a token list, cf. use as `\lateLua`. The string version is treated as a literal string.

When a function is used, it gets called with as first argument the node that triggers the call.

### 8.5 DVI backend whatsits

#### 8.5.1 special

There is only one dvi backend whatsit, and it just flushes its content to the output file.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>the \special information</td>
</tr>
</tbody>
</table>
8.6 PDF backend whatsits

8.6.1 pdf_literal and pdf_late_literal

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>mode</td>
<td>number</td>
<td>the ‘mode’ setting of this literal</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>the to be written information stored as Lua string</td>
</tr>
<tr>
<td>token</td>
<td>string</td>
<td>the to be written information stored as token list</td>
</tr>
</tbody>
</table>

Possible mode values are:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>KEYWORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>origin</td>
</tr>
<tr>
<td>1</td>
<td>page</td>
</tr>
<tr>
<td>2</td>
<td>direct</td>
</tr>
<tr>
<td>3</td>
<td>raw</td>
</tr>
<tr>
<td>4</td>
<td>text</td>
</tr>
</tbody>
</table>

The higher the number, the less checking and the more you can run into trouble. Especially the raw variant can produce bad pdf so you can best check what you generate.

8.6.2 pdf_refobj

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the referenced pdf object number</td>
</tr>
</tbody>
</table>

8.6.3 pdf_annot

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width (not used in calculations)</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height (not used in calculations)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth (not used in calculations)</td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the referenced pdf object number</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>the annotation data</td>
</tr>
</tbody>
</table>

8.6.4 pdf_start_link

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width (not used in calculations)</td>
</tr>
</tbody>
</table>
8.6.5 pdf_end_link

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td></td>
</tr>
</tbody>
</table>

8.6.6 pdf_dest

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width (not used in calculations)</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height (not used in calculations)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth (not used in calculations)</td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td>is the dest_id a string value?</td>
</tr>
<tr>
<td>dest_id</td>
<td>number</td>
<td>the destination id</td>
</tr>
<tr>
<td>string</td>
<td>the destination name</td>
<td></td>
</tr>
<tr>
<td>dest_type</td>
<td>number</td>
<td>type of destination</td>
</tr>
<tr>
<td>xyz_zoom</td>
<td>number</td>
<td>the zoom factor (times 1000)</td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the pdf object number; for structure references the pdf object number of the linked structure element</td>
</tr>
</tbody>
</table>

8.6.7 pdf_action

These are a special kind of items that only appear inside pdf start link objects.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>action_type</td>
<td>number</td>
<td>the kind of action involved</td>
</tr>
<tr>
<td>action_id</td>
<td>number or string</td>
<td>token list reference or string</td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td>are dest_id and struct_id string values?</td>
</tr>
<tr>
<td>file</td>
<td>string</td>
<td>the target filename</td>
</tr>
<tr>
<td>new_window</td>
<td>number</td>
<td>the window state of the target</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>the name of the destination</td>
</tr>
<tr>
<td>struct_id</td>
<td>nil</td>
<td>the action does not reference a structure destination</td>
</tr>
<tr>
<td>number</td>
<td>id of the referenced structure destination</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>name of the referenced structure destination</td>
<td></td>
</tr>
</tbody>
</table>

Valid action types are:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>page</td>
</tr>
</tbody>
</table>
Valid window types are:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>notset</td>
</tr>
<tr>
<td>1</td>
<td>new</td>
</tr>
<tr>
<td>2</td>
<td>nonew</td>
</tr>
</tbody>
</table>

### 8.6.8 pdf_thread

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width (not used in calculations)</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height (not used in calculations)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth (not used in calculations)</td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td>is thread_id a string value?</td>
</tr>
<tr>
<td>thread_id</td>
<td>number</td>
<td>the thread id</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>the thread name</td>
</tr>
<tr>
<td>thread_attr</td>
<td>number</td>
<td>extra thread information</td>
</tr>
</tbody>
</table>

### 8.6.9 pdf_start_thread

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width (not used in calculations)</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height (not used in calculations)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth (not used in calculations)</td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td>is thread_id a string value?</td>
</tr>
<tr>
<td>thread_id</td>
<td>number</td>
<td>the thread id</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>the thread name</td>
</tr>
<tr>
<td>thread_attr</td>
<td>number</td>
<td>extra thread information</td>
</tr>
</tbody>
</table>

### 8.6.10 pdf_end_thread

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td></td>
</tr>
</tbody>
</table>

### 8.6.11 pdf_colorstack

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
</tbody>
</table>
8.6.12 **pdf_setmatrix**

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>data</td>
</tr>
</tbody>
</table>

8.6.13 **pdf_save**

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
</tbody>
</table>

8.6.14 **pdf_restore**

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>node</td>
<td>list of attributes</td>
</tr>
</tbody>
</table>

8.7 **The node library**

8.7.1 **Introduction**

The node library contains functions that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, copy, delete, and insert Lua\TeX\ node objects, the core objects within the typesetter.

Lua\TeX\ nodes are represented in Lua as userdata with the metadata type luatex.node. 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 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, but it is most important when dealing with ‘whatsits’, because they are differentiated solely based on their subtype.

The other available fields depend on the id (and for ‘whatsits’, the subtype) of the node.

Support for unset (alignment) nodes is partial: they can be queried and modified from Lua code, but not created.
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.

At the moment, memory management of nodes should still 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.

There are statistics available with regards to the allocated node memory, which can be handy for tracing.

8.7.2 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.

8.7.3 types and whatsits

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()

TEX’s ‘whatsits’ all have the same id. The various subtypes are defined by their subtype fields. The function is much like types, except that it provides an array of subtype mappings.

<table> t =
    node.whatsits()

8.7.4 id

This converts a single type name to its internal numeric representation.

<number> id =
    node.id(<string> type)

8.7.5 type and subtype

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)

This next one converts a single whatsit name to its internal numeric representation (subtype).

<number> subtype =
    node.subtype(<string> type)

**8.7.6 fields**

This function returns an array of valid field names for a particular type of node. If you want to get the valid fields for a ‘whatsit’, you have to supply the second argument also. In other cases, any given second argument will be silently ignored.

<table> t =
    node.fields(<number> id)
<table> t =
    node.fields(<number> id, <number> subtype)

The function accepts string id and subtype values as well.

**8.7.7 has_field**

This function returns a boolean that is only true if n is actually a node, and it has the field.

<boolean> t =
    node.has_field(<node> n, <string> field)

**8.7.8 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 create a new whatsit node the second argument is required. As with all node functions, this function creates a node at the TeX level.

<node> n =
    node.new(<number> id)
<node> n =
    node.new(<number> id, <number> subtype)

**8.7.9 free, flush_node and flush_list**

The next one removes the 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.

<node> next =
node.free(<node> n)
flush_node(<node> n)

The free function returns the next field of the freed node, while the flush_node 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.flush_list(<node> n)

### 8.7.10 copy and copy_list

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.copy_list(<node> n)
<node> m =
    node.copy_list(<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.

### 8.7.11 prev and next

These returns the node preceding or following the given node, or nil if there is no such node.

```
<node> m =
    node.next(<node> n)
<node> m =
    node.prev(<node> n)
```

### 8.7.12 current_attr

This returns the currently active list of attributes, if there is one.

```
<node> m =
    node.current_attr()
```

The intended usage of current_attr is as follows:
local x1 = node.new("glyph")
x1.attr = node.current_attr()
local x2 = node.new("glyph")
x2.attr = node.current_attr()

or:
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.current_attr()
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 \textsetttemperature) or when control has been passed back to \TeX.

Note: this function is somewhat experimental, and it returns the actual attribute list, not a copy thereof. Therefore, changing any of the attributes in the list will change these values for all nodes that have the current attribute list assigned to them.

### 8.7.13 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.

\[
\langle \text{node} \rangle \ h, \ \langle \text{number} \rangle \ b = \\
\text{node.hpack}(\langle \text{node} \rangle \ n)
\]

\[
\langle \text{node} \rangle \ h, \ \langle \text{number} \rangle \ b = \\
\text{node.hpack}(\langle \text{node} \rangle \ n, \ \langle \text{number} \rangle \ w, \ \langle \text{string} \rangle \ info)
\]

\[
\langle \text{node} \rangle \ h, \ \langle \text{number} \rangle \ b = \\
\text{node.hpack}(\langle \text{node} \rangle \ n, \ \langle \text{number} \rangle \ w, \ \langle \text{string} \rangle \ info, \ \langle \text{string} \rangle \ dir)
\]

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!

### 8.7.14 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.

\[
\langle \text{node} \rangle \ h, \ \langle \text{number} \rangle \ b = 
\]
```haskell
node.vpack(<node> n)

h, b =
node.vpack(<node> n, <number> w, <string> info)

h, 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.

### 8.7.15 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.

```haskell
n, delta =
node.prepend_prevdepth(n, prevdepth)
```

### 8.7.16 dimensions and rangedimensions

```haskell
w, h, d =
node.dimensions(n)

w, h, d =
node.dimensions(n, dir)

w, h, d =
node.dimensions(n, t)

w, h, d =
node.dimensions(n, t, dir)
```

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:

```haskell
w, h, d =
node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, n)

w, h, d =
node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, n, dir)

w, h, d =
node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, n, t)

w, h, d =
node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, n, t, dir)
```
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)

8.7.17 mlist_to_hlist

<node> h =
    node.mlist_to_hlist(<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 mlist_to_hlist.

8.7.18 slide

<node> m =
    node.slide(<node> n)

Returns the last node of the node list that starts at n. As a side-effect, it also creates a reverse chain of prev pointers between nodes.

8.7.19 tail

<node> m =
    node.tail(<node> n)

Returns the last node of the node list that starts at n.
8.7.20 length and type 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 \textit{n}. If \textit{m} is also supplied it stops at \textit{m} instead of at the end of the list. The node \textit{m} is not counted.

<number> i =
    node.count(<number> \textit{id}, <node> n)
<number> i =
    node.count(<number> \textit{id}, <node> n, <node> m)

Returns the number of nodes contained in the node list that starts at \textit{n} that have a matching \textit{id} field. If \textit{m} is also supplied, counting stops at \textit{m} instead of at the end of the list. The node \textit{m} is not counted. This function also accept string \textit{id}'s.

8.7.21 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> \textit{b} =
    node.is_char(<node> \textit{n})
<boolean> \textit{b} =
    node.is_glyph(<node> \textit{n})

8.7.22 traverse

<node> \textit{t}, \textit{id}, subtype =
    node.traverse(<node> \textit{n})

This is a Lua iterator that loops over the node list that starts at \textit{n}. Typically code looks like this:

\begin{verbatim}
for \textit{n} in node.traverse(head) do
    ...
end
\end{verbatim}

is functionally equivalent to:

\begin{verbatim}
do
    local \textit{n}
    local function \textit{f} (head,\textit{var})
        local \textit{t}
        if \textit{var} == nil then
            \textit{t} = head
        \end}
\end{verbatim}
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.

8.7.23 \textit{traverse\_id}

\begin{verbatim}
<node> t, subtype =
node.traverse_id(<number> id, <node> n)
\end{verbatim}

This is an iterator that loops over all the nodes in the list that starts at \( n \) that have a matching \textit{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 \( id \):

\begin{verbatim}
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
\end{verbatim}

8.7.24 \textit{traverse\_char} and \textit{traverse\_glyph}

The \textit{traverse\_char} iterator loops over the glyph nodes in a list. Only nodes with a subtype less than 256 are seen.

\begin{verbatim}
<node> n, char, font =
\end{verbatim}
The `traverse_glyph` iterator loops over a list and returns the list and filters all glyphs:

```plaintext
<n> char, font = node.traverse_glyph(<node> n)
```

### 8.7.25 `traverse_list`

This iterator loops over the hlist and vlist nodes in a list.

```plaintext
<n> id, subtype, list = node.traverse_list(<node> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. So consider it a (side effect of experimental) convenience.

### 8.7.26 `has_glyph`

This function returns the first glyph or disc node in the given list:

```plaintext
<n> n = node.has_glyph(<node> n)
```

### 8.7.27 `end_of_math`

```plaintext
<n> 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.

### 8.7.28 `remove`

```plaintext
<n> head, current = node.remove(<node> head, <node> current)
```

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.

### 8.7.29 `insert_before`

```plaintext
<n> head, new =
```
node.insert_before(<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.

8.7.30 insert_after

<node> head, new =
    node.insert_after(<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.

8.7.31 first_glyph

<node> n =
    node.first_glyph(<node> n)
<node> n =
    node.first_glyph(<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.

8.7.32 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).

8.7.33 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).
8.7.34 unprotect_glyph[s]

node.unprotect_glyph(<node> n)
node.unprotect_glyphs(<node> n,[<node> ,m])

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.

8.7.35 protect_glyph[s]

node.protect_glyph(<node> n)
node.protect_glyphs(<node> n,[<node> m])

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.

8.7.36 last_node

<n= node.last_node()

This function pops the last node from \TeX’s ‘current list’. It returns that node, or nil if the current list is empty.

8.7.37 write

node.write(<node> n)

This function that 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.

8.7.38 protrusion_skippable

<boolean> skippable =
   node.protrusion_skippable(<node> n)

Returns true if, for the purpose of line boundary discovery when character protrusion is active, this node can be skipped.

8.8 Glue handling

8.8.1 setglue

You can set the five properties of a glue in one go. Non-numeric values are equivalent to zero and reset a property.
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.

8.8.2 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.

8.8.3 is_zero_glue

This function returns true when the width, stretch and shrink properties are zero.

<boolean> isglue =
node.is_zero_glue(<node> n)

8.9 Attribute handling

8.9.1 Attributes

The newly introduced attribute registers are non-trivial, because the value that is attached to a
node is essentially a 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, but for completeness,
here is the low-level interface.

Attributes appear as linked list of userdata objects in the attr field of individual nodes. They can
be handled individually, but it is much safer and more efficient to use the dedicated functions
associated with them.

8.9.2 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:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>node</td>
<td>pointer to the first attribute</td>
</tr>
</tbody>
</table>
8.9.3 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.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>node</td>
<td>pointer to the next attribute</td>
</tr>
<tr>
<td>number</td>
<td>number</td>
<td>the attribute type id</td>
</tr>
<tr>
<td>value</td>
<td>number</td>
<td>the attribute value</td>
</tr>
</tbody>
</table>

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.

8.9.4 has_attribute

\[
<\text{number}> v = \\
\quad \text{node}.\text{has_attribute}(<\text{node}> n, <\text{number}> id) \\
<\text{number}> v = \\
\quad \text{node}.\text{has_attribute}(<\text{node}> n, <\text{number}> id, <\text{number}> val)
\]

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.

8.9.5 get_attribute

\[
<\text{number}> v = \\
\quad \text{node}.\text{get_attribute}(<\text{node}> n, <\text{number}> id)
\]

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.

8.9.6 find_attribute

\[
<\text{number}> v, <\text{node}> n = \\
\quad \text{node}.\text{find_attribute}(<\text{node}> n, <\text{number}> id)
\]

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.

8.9.7 set_attribute

\[
\text{node}.\text{set_attribute}(<\text{node}> n, <\text{number}> id, <\text{number}> val)
\]

Sets the attribute with number id to the value val. Duplicate assignments are ignored.
8.9.8 unset_attribute

<number> v =
  node.unset_attribute(<node> n, <number> id)
<number> v =
  node.unset_attribute(<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.

8.9.9 slide

This helper makes sure that the node lists is double linked and returns the found tail node.

<number> tail =
  node.slide(<node> n)

After some callbacks automatic sliding takes place. This feature can be turned off with
node.fix_node_lists(false) but you better make sure then that you don’t mess up lists. In
most cases \TeX itself only uses next pointers but your other callbacks might expect proper prev
pointers too. Future versions of Lua\TeX can add more checking but this will not influence usage.

8.9.10 check_discretionary, check_discretionaries

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.check_discretionary(<node> n)
node.check_discretionaries(<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).

8.9.11 flatten_discretionaries

This function will remove the discretionaries in the list and inject the replace field when set.

<number> head, count = node.flatten_discretionaries(<node> n)

8.9.12 family_font

When you pass a proper family identifier the next helper will return the font currently associated
with it. You can normally also access the font with the normal font field or getter because it will
resolve the family automatically for noads.
8.10 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:

\begin{verbatim}
<integer> d = node.direct.todirect(<node> n))
<integer> id = node.family_font(<integer> fam)

<node> n = node.direct.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:

\begin{verbatim}
nodeobject.char
getfield(nodenumber,"char")
\end{verbatim}

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 Lua\TeX{} 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:

\begin{verbatim}
local next = current.next
if next then
    -- do something
end
\end{verbatim}

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).

```lua
local next = node.next(current)
if next then
    -- do something
end
```

Some accessors are used frequently and for these we provide more efficient helpers:

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>getnext</td>
<td>parsing nodeList always involves this one</td>
</tr>
<tr>
<td>getprev</td>
<td>used less but a logical companion to getnext</td>
</tr>
<tr>
<td>getboth</td>
<td>returns the next and prev pointer of a node</td>
</tr>
<tr>
<td>getid</td>
<td>consulted a lot</td>
</tr>
<tr>
<td>getsubtype</td>
<td>consulted less but also a topper</td>
</tr>
<tr>
<td>getfont</td>
<td>used a lot in OpenType handling (glyph nodes are consulted a lot)</td>
</tr>
<tr>
<td>getchar</td>
<td>idem and also in other places</td>
</tr>
<tr>
<td>getwhd</td>
<td>returns the width, height and depth of a list, rule or (unexpanded) glyph as well as glue (its spec is looked at) and unset nodes</td>
</tr>
<tr>
<td>getdisc</td>
<td>returns the pre, post and replace fields and optionally when true is passed also the tail fields</td>
</tr>
<tr>
<td>getlist</td>
<td>we often parse nested lists so this is a convenient one too</td>
</tr>
<tr>
<td>getleader</td>
<td>comparable to list, seldom used in TeX (but needs frequent consulting like lists; leaders could have been made a dedicated node type)</td>
</tr>
<tr>
<td>getfield</td>
<td>generic getter, sufficient for the rest (other field names are often shared so a specific getter makes no sense then)</td>
</tr>
<tr>
<td>getbox</td>
<td>gets the given box (a list node)</td>
</tr>
<tr>
<td>getoffsets</td>
<td>gets the xoffset and yoffset of a glyph or left and right values of a rule</td>
</tr>
</tbody>
</table>

In the direct namespace there are more such 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.²

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.

² 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.
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 `get-field` and `set-field` with little overhead. When the second argument of `setattributelist` is true the current attribute list is assumed.

### Function Table

<table>
<thead>
<tr>
<th>Function</th>
<th>Node</th>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_discretionaries</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>check_discretionary</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>copy_list</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>copy</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>count</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>current_attr</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>dimensions</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>effective_glue</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>end_of_math</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>family_font</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>fields</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>find_attribute</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>first_glyph</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>fix_node_lists</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>flatten_discretionaries</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>flush_list</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>flush_node</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>flush_properties_table</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>free</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>get_attribute</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>get_properties_table</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>get_synctex_fields</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getattributelist</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getboth</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getbox</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getchar</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getcomponents</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getdata</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getdepth</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getdirection</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getdir</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getdisc</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getexpansion</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getfam</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getfield</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getfont</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getglue</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getheight</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getid</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Function</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>getkern</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>getlang</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>getleader</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getlist</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getnext</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getnucleus</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getoffsets</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getpenalty</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getprev</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getproperty</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getshift</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getsubtype</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getsub</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getsup</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>getwhd</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getwidth</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>has_attribute</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>has_field</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>has_glyph</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>hpack</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>hyphenating</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>id</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>insert_after</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>insert_before</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>is_char</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>is_direct</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>is_glyph</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>is_node</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>is_zero_glue</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>kerning</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>last_node</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>length</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ligaturing</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>make_extensible</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>mlist_to_hlist</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>new</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>next</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>prepend_prevdepth</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>prev</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>protect_glyphs</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>protect_glyph</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>protrusion_skippable</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>rangedimensions</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>remove</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>set_attribute</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>set_properties_mode</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**162 Nodes**
set_synctex_fields  -  +
setattributelist   -  +
setboth            -  +
setbox             -  +
setchar            -  +
setcomponents      -  +
setdata            -  +
setdepth           -  +
setdirection       -  +
setdir             -  +
setdisc            -  +
setexpansion       -  +
setfam             -  +
setfield           +  +
setfont            -  +
setglue            +  +
setheight          -  +
setkern            -  +
setlang            -  +
setleader          -  +
setlink            -  +
setlist            -  +
setnext            -  +
setnucleus         -  +
setoffsets         -  +
setpenalty         -  +
setprev            -  +
setproperty        +  +
setshift           -  +
setsplit           -  +
setsubtype         -  +
setsup             -  +
setwhd             -  +
setwidth           -  +
slide              +  +
subtypes           +  -
subtype            +  -
tail               +  +
todirect           -  +
tonode             -  +
tostring           +  +
traverse_char      +  +
traverse_glyph     +  +
traverse_id        +  +
traverse_list      -  +
The \texttt{node.next} and \texttt{node.prev} functions will stay but for consistency there are variants called \texttt{getnext} and \texttt{getprev}. We had to use \texttt{get} because \texttt{node.id} and \texttt{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 \texttt{kern} and \texttt{width}, or \texttt{data} and \texttt{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, \texttt{getfont} and \texttt{.font} can be used on them. The same is true for the \texttt{width}, \texttt{height} and \texttt{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 \LaTeX{} can do a bit of extra checking for valid node lists and you can enable that with:

\begin{verbatim}
node.fix_node_lists(<boolean> b)
\end{verbatim}

You can set and query the \texttt{Sync\LaTeX} fields, a file number aka tag and a line number, for a glue, \texttt{kern}, \texttt{hlist}, \texttt{vlist}, \texttt{rule} and math nodes as well as glyph nodes (although this last one is not used in native \texttt{Sync\LaTeX}).

\begin{verbatim}
node.direct.set_synctex_fields(<integer> n, <integer> f, <integer> l)
<integer> f, <integer> l = node.direct.get_synctex_fields(<integer> n)
\end{verbatim}

Of course you need to know what you’re doing as no checking on sane values takes place. Also, the \texttt{synctex} interpreter used in editors is rather peculiar and has some assumptions (heuristics).

### 8.11 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 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"
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: `flush_properties_table` will wipe the table (normally a bad idea), `get_properties_table` 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.

Managing properties in the node (de)allocator functions is disabled by default and is enabled by:

```lua
node.set_properties_mode(true)
```

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 kind 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.
9 LUA callbacks

9.1 Registering callbacks

This library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kind of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In mostly 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:

```lua
id, error =
    callback.register(<string> callback_name, <function> func)
```

```lua
id, error =
    callback.register(<string> callback_name, nil)
```

```lua
id, error =
    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. In the latter case, `error` contains an error message, otherwise it is nil.

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!

```lua
<table>
    info =
        callback.list()
</table>
```

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).

```lua
<function>
    f = callback.find(callback_name)
</function>
```

If the callback is not set, find returns nil.

9.2 File discovery callbacks

The behaviour documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.
9.2.1 find_read_file and find_write_file

Your callback function should have the following conventions:

```lua
<string> actual_name =
    function (<number> id_number, <string> asked_name)
```

**Arguments:**

- `id_number`
  - This number is zero for the log or \input files. For \TeX's \read or \write the number is incremented by one, so \read0 becomes 1.
- `asked_name`
  - This is the user-supplied filename, as found by \input, \openin or \openout.

**Return value:**

- `actual_name`
  - This is the filename used. For the very first file that is read in by \TeX, you have to make sure you return an `actual_name` that has an extension and that is suitable for use as jobname. If you don’t, you will have to manually fix the name of the log file and output file after Lua\TeX is finished, and an eventual format filename will become mangled. That is because these file names depend on the jobname.
  - You have to return `nil` if the file cannot be found.

9.2.2 find_font_file

Your callback function should have the following conventions:

```lua
<string> actual_name =
    function (<string> asked_name)
```

The `asked_name` is an otf or tfm font metrics file.

Return `nil` if the file cannot be found.

9.2.3 find_output_file

Your callback function should have the following conventions:

```lua
<string> actual_name =
    function (<string> asked_name)
```

The `asked_name` is the pdf or dvi file for writing.

9.2.4 find_format_file

Your callback function should have the following conventions:

```lua
<string> actual_name =
```
function (<string> asked_name)

The asked_name is a format file for reading (the format file for writing is always opened in the current directory).

9.2.5 find_vf_file

Like find_font_file, but for virtual fonts. This applies to both Aleph’s ovf files and traditional Knuthian vf files.

9.2.6 find_map_file

Like find_font_file, but for map files.

9.2.7 find_enc_file

Like find_font_file, but for enc files.

9.2.8 find_pk_file

Like find_font_file, but for pk bitmap files. This callback takes two arguments: name and dpi. In your callback you can decide to look for:

<base res>dpi/<fontname>.<actual res>pk

but other strategies are possible. It is up to you to find a ‘reasonable’ bitmap file to go with that specification.

9.2.9 find_data_file

Like find_font_file, but for embedded files (\pdfobj file '...').

9.2.10 find_opentype_file

Like find_font_file, but for OpenType font files.

9.2.11 find_truetype_file and find_type1_file

Your callback function should have the following conventions:

<string> actual_name =
    function (<string> asked_name)

The asked_name is a font file. This callback is called while LuaTEX is building its internal list of needed font files, so the actual timing may surprise you. Your return value is later fed back into the matching read_file callback.
Strangely enough, find_type1_file is also used for OpenType (otf) fonts.

9.2.12 find_image_file

Your callback function should have the following conventions:

<string> actual_name =
    function (<string> asked_name)

The asked_name is an image file. Your return value is used to open a file from the hard disk, so make sure you return something that is considered the name of a valid file by your operating system.

9.3 File reading callbacks

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

9.3.1 open_read_file

Your callback function should have the following conventions:

<table> env =
    function (<string> file_name)

Argument:

file_name
    The filename returned by a previous find_read_file or the return value of kpse.find_file() if there was no such callback defined.

Return value:

env
    This is a table containing at least one required and one optional callback function for this file. The required field is reader and the associated function will be called once for each new line to be read, the optional one is close that will be called once when 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.

9.3.1.1 reader

LuaTeX will run this function whenever it needs a new input line from the file.

function(<table> env)
    return <string> line
Your function should return either a string or nil. The value nil signals that the end of file has occurred, and will make \TeX{} call the optional close function next.

### 9.3.1.2 close

Lua\TeX{} will run this optional function when it decides to close the file.

```
function(<table> env)
end
```

Your function should not return any value.

### 9.3.2 General file readers

There is a set of callbacks for the loading of binary data files. These all use the same interface:

```
function(<string> name)
    return <boolean> success, <string> data, <number> data_size
end
```

The name will normally be a full path name as it is returned by either one of the file discovery callbacks or the internal version of kpse.find_file().

- **success**
  
  Return false when a fatal error occurred (e.g. when the file cannot be found, after all).

- **data**
  
  The bytes comprising the file.

- **data_size**
  
  The length of the data, in bytes.

Return an empty string and zero if the file was found but there was a reading problem.

The list of functions is:

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_font_file</td>
<td>ofm or tfm files</td>
</tr>
<tr>
<td>read_vf_file</td>
<td>virtual fonts</td>
</tr>
<tr>
<td>read_map_file</td>
<td>map files</td>
</tr>
<tr>
<td>read_enc_file</td>
<td>encoding files</td>
</tr>
<tr>
<td>read_pk_file</td>
<td>pk bitmap files</td>
</tr>
<tr>
<td>read_data_file</td>
<td>embedded files (as is possible with pdf objects)</td>
</tr>
<tr>
<td>read_truetype_file</td>
<td>TrueType font files</td>
</tr>
<tr>
<td>read_type1_file</td>
<td>Type1 font files</td>
</tr>
<tr>
<td>read_opentype_file</td>
<td>OpenType font files</td>
</tr>
</tbody>
</table>
9.4 Data processing callbacks

9.4.1 process_input_buffer

This callback allows you to change the contents of the line input buffer just before LuaTeX actually starts looking at it.

```lua
function(<string> buffer)
    return <string> adjusted_buffer
end
```

If you return `nil`, LuaTeX will pretend like your callback never happened. You can gain a small amount of processing time from that. This callback does not replace any internal code.

9.4.2 process_output_buffer

This callback allows you to change the contents of the line output buffer just before LuaTeX actually starts writing it to a file as the result of a `\write` command. It is only called for output to an actual file (that is, excluding the log, the terminal, and so called `\write` 18 calls).

```lua
function(<string> buffer)
    return <string> adjusted_buffer
end
```

If you return `nil`, LuaTeX will pretend like your callback never happened. You can gain a small amount of processing time from that. This callback does not replace any internal code.

9.4.3 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.

```lua
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.

9.5 Node list processing callbacks

The description of nodes and node lists is in chapter 8.

9.5.1 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.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_box</td>
<td>interline material is being added</td>
</tr>
<tr>
<td>pre_adjust</td>
<td>\vadjust material is being added</td>
</tr>
<tr>
<td>box</td>
<td>a typeset box is being added (always called)</td>
</tr>
<tr>
<td>adjust</td>
<td>\vadjust material is being added</td>
</tr>
</tbody>
</table>

### 9.5.2 buildpage_filter

This callback is called whenever Lua\TeX{} 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:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>a (partial) alignment is being added</td>
</tr>
<tr>
<td>after_output</td>
<td>an output routine has just finished</td>
</tr>
<tr>
<td>new_graf</td>
<td>the beginning of a new paragraph</td>
</tr>
<tr>
<td>vmode_par</td>
<td>\par was found in vertical mode</td>
</tr>
<tr>
<td>hmode_par</td>
<td>\par was found in horizontal mode</td>
</tr>
<tr>
<td>insert</td>
<td>an insert is added</td>
</tr>
<tr>
<td>penalty</td>
<td>a penalty (in vertical mode)</td>
</tr>
<tr>
<td>before_display</td>
<td>immediately before a display starts</td>
</tr>
<tr>
<td>after_display</td>
<td>a display is finished</td>
</tr>
<tr>
<td>end</td>
<td>Lua\TeX{} is terminating (it’s all over)</td>
</tr>
</tbody>
</table>

### 9.5.3 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
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.

### 9.5.4 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.

```lUA
function(<node> head, <string> groupcode)
    return true | false | <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.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>main vertical list</td>
</tr>
<tr>
<td>hbox</td>
<td>\hbox in horizontal mode</td>
</tr>
<tr>
<td>adjusted hbox</td>
<td>\hbox in vertical mode</td>
</tr>
<tr>
<td>vbox</td>
<td>\vbox</td>
</tr>
<tr>
<td>vtop</td>
<td>\vtop</td>
</tr>
<tr>
<td>align</td>
<td>\halign or \valign</td>
</tr>
<tr>
<td>disc</td>
<td>discretionaries</td>
</tr>
<tr>
<td>insert</td>
<td>packaging an insert</td>
</tr>
<tr>
<td>vcenter</td>
<td>\vcenter</td>
</tr>
<tr>
<td>local_box</td>
<td>\localleftbox or \localrightbox</td>
</tr>
<tr>
<td>split_off</td>
<td>top of a \vsplit</td>
</tr>
<tr>
<td>split_keep</td>
<td>remainder of a \vsplit</td>
</tr>
<tr>
<td>align_set</td>
<td>alignment cell</td>
</tr>
<tr>
<td>fin_row</td>
<td>alignment row</td>
</tr>
</tbody>
</table>

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.
9.5.5 linebreak_filter

This callback replaces Lua\TeX's line breaking algorithm.

```latex
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>, Lua\TeX 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 a 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’.

9.5.6 append_to_vlist_filter

This callback is called whenever Lua\TeX adds a box to a vertical list:

```latex
function(<node> box, <string> locationcode, <number prevdepth>,
    <boolean> mirrored)
    return list, prevdepth
end
```

It is ok to return nothing 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. You can pass nil instead of a node.

9.5.7 post_linebreak_filter

This callback is called just after Lua\TeX has converted a list of nodes into a stack of \hboxes.

```latex
function(<node> head, <string> groupcode)
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

9.5.8 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.

```latex
function(<node> head, <string> groupcode, <number> size,
    <string> packtype [, <string> direction] [, <node> attributelist])
```
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.

The direction is either one of the three-letter direction specifier strings, or nil.

This callback does not replace any internal code.

9.5.9 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 [, <string> direction] [, <node> attributelist])
    return true | false | <node> newhead
end

This callback does not replace any internal code.

9.5.10 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).

9.5.11 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)
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.

**9.5.12 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.

**9.5.13 pre_output_filter**

This callback is called when \TeX{} is ready to start boxing the box 255 for `\output`.

```lua
function(<node> head, <string> groupcode, <number> size, <string> packtype, 
    <number> maxdepth [, <string> direction])
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

**9.5.14 hyphenate**

```lua
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.

**9.5.15 ligaturing**

```lua
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 Lua\TeX into panic mode.

### 9.5.16 kerning

```latex
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 Lua\TeX into panic mode.

### 9.5.17 insert_local_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:

```latex
function(<node> local_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.

### 9.5.18 mlist_to_hlist

This callback replaces Lua\TeX’s math list to node list conversion algorithm.

```latex
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.

### 9.6 Information reporting callbacks

#### 9.6.1 pre_dump

```latex
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.

### 9.6.2 start_run

```lua
function()
end
```

This callback replaces the code that prints Lua\TeX{}’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.

### 9.6.3 stop_run

```lua
function()
end
```

This callback replaces the code that prints Lua\TeX{}’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.

### 9.6.4 start_page_number

```lua
function()
end
```

Replaces the code that prints the [ and the page number at the begin of \texttt{\shipout}. This callback will also override the printing of box information that normally takes place when \texttt{\tracingoutput} is positive.

### 9.6.5 stop_page_number

```lua
function()
end
```

Replaces the code that prints the ] at the end of \texttt{\shipout}.

### 9.6.6 show_error_hook

```lua
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 \texttt{status} table useful. This callback does not replace any internal code.
9.6.7 show_error_message

function()
end

This callback replaces the code that prints the error message. The usual interaction after the message is not affected.

9.6.8 show_lua_error_hook

function()
end

This callback replaces the code that prints the extra Lua error message.

9.6.9 start_file

function(category, filename)
end

This callback replaces the code that prints Lua\TeX’s when a file is opened like \texttt{(filename} for regular files. The category is a number:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a normal data file, like a \TeX source</td>
</tr>
<tr>
<td>2</td>
<td>a font map coupling font names to resources</td>
</tr>
<tr>
<td>3</td>
<td>an image file (png, pdf, etc)</td>
</tr>
<tr>
<td>4</td>
<td>an embedded font subset</td>
</tr>
<tr>
<td>5</td>
<td>a fully embedded font</td>
</tr>
</tbody>
</table>

9.6.10 stop_file

function(category)
end

This callback replaces the code that prints Lua\TeX’s when a file is closed like the \texttt{)} for regular files.

9.6.11 call_edit

function(filename, linenumber)
end

This callback replaces the call to an external editor when ‘E’ is pressed in reply to an error message. Processing will end immediately after the callback returns control to the main program.
9.6.12 finish_synctex

This callback can be used to wrap up alternative synctex methods. It kicks in after the normal synctex finalizer (that happens to remove the synctex files after a run when native synctex is not enabled).

9.6.13 wrapup_run

This callback is called after the pdf and log files are closed. Use it at your own risk.

9.7 PDF related callbacks

9.7.1 finish_pdffile

```lua
function()
end
```

This callback is called when all document pages are already written to the pdf file and LuaTeX is about to finalize the output document structure. Its intended use is final update of pdf dictionaries such as /Catalog or /Info. The callback does not replace any code. There are neither arguments nor return values.

9.7.2 finish_pdfpage

```lua
function(shippingout)
end
```

This callback is called after the pdf page stream has been assembled and before the page object gets finalized.

9.7.3 page_order_index

This is one that experts can use to juggle the page tree, a data structure that determines the order in a pdf file:

```lua
function(pagenumber)
    return pagenumber
end
```

Say that we have 12 pages, then we can do this:

```lua
callback.register("page_order_index",function(page)
    if page == 1 then return 12
    elseif page == 2 then return 11
    elseif page == 11 then return 2
    elseif page == 12 then return 1
end
```
This will swap the first two and last two pages. You need to know the number of pages which is a side effect of the implementation. When you mess things up … don’t complain.

9.7.4 process_pdf_image_content

When a page from a pdf file is embedded its page stream as well as related objects are copied to the target file. However, it can be that the page stream has operators that assume additional resources, for instance marked text. You can decide to filter that for which LuaTeX provides a callback. Here is a simple demonstration of use:

```lua
pdf.setrecompress(1)

callback.register("process_pdf_image_content",function(s)
    print(s)
    return s
end)
```

You need to explicitly enable recompression because otherwise the content stream gets just passed on in its original compressed form.

9.8 Font-related callbacks

9.8.1 define_font

```lua
function(<string> name, <number> size, <number> id)
    return <table> font | <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 designsize of the font.

The id is the internal number assigned to the font.

The internal structure of the font table that is to be returned 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.
9.8.2 glyph_not_found and glyph_info

The glyph_not_found callback, when set, kicks in when the backend cannot insert a glyph. When no callback is defined a message is written to the log.

```lua
function(<number> id, <number> char)
    -- do something with font id and char code
end
```

The glyph_info callback can be set to report a useful representation of a glyph.

```lua
function(<node> g)
    -- return a string or nil
end
```

When nil is returned the character code is printed, otherwise the returned string is used. By default the utf representation is shown which is not always that useful, especially when there is no real representation. Keep in mind that setting this callback can change the log in an incompatible way.
10 The \TeX related libraries

10.1 The \texttt{lua} library

10.1.1 Version information

This library contains one read-only item:

\begin{verbatim}
<string> s = lua.version
\end{verbatim}

This returns the Lua version identifier string. The value is currently Lua 5.3.

10.1.2 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.

\begin{verbatim}
lua.bytecode[<number> n] = <function> f
lua.bytecode[<number> n]()
\end{verbatim}

The contents of the \texttt{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:

\begin{verbatim}
<function> f = lua.getbytecode(<number> n)
lua.setbytecode(<number> n, <function> f)
\end{verbatim}

Note: Since a Lua file loaded using \texttt{loadfile(filename)} is essentially an anonymous function, a complete file can be stored in a bytecode register like this:

\begin{verbatim}
lua.bytecode[n] = loadfile(filename)
\end{verbatim}

Now all definitions (functions, variables) contained in the file can be created by executing this bytecode register:

\begin{verbatim}
lua.bytecode[n]()
\end{verbatim}

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 \texttt{ini\TeX}. 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 \texttt{ini\TeX}.

10.1.3 Chunk name registers

There is an array of 65536 (0–65535) potential chunk names for use with the \texttt{\directlua} and \texttt{\latelua} primitives.
lua.name[<number> n] = <string> s
<string> s = lua.name[<number> n]

If you want to unset a Lua name, you can assign nil to it. The function accessors are:

lua.setluaname(<number> n, <string> s)
<string> s = lua.getluaname(<number> n)

## 10.1.4 Introspection

The getstacktop and getcalllevel functions return numbers indicating how much nesting is going on. They are only of use as breakpoints when checking some mechanism going haywire.

## 10.2 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>
<table>
<thead>
<tr>
<th>info = status.list()</th>
</tr>
</thead>
</table>

The keys in the table are the known items, the value is the current value. Almost all of the values in status are fetched through a metatable at run-time whenever they are accessed, so you cannot use pairs on status, but you can use pairs on info, of course. If you do not need the full list, you can also ask for a single item by using its name as an index into status. The current list is:

<table>
<thead>
<tr>
<th>KEY</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>banner</td>
<td>terminal display banner</td>
</tr>
<tr>
<td>best_page_break</td>
<td>the current best break (a node)</td>
</tr>
<tr>
<td>buf_size</td>
<td>current allocated size of the line buffer</td>
</tr>
<tr>
<td>callbacks</td>
<td>total number of executed callbacks so far</td>
</tr>
<tr>
<td>cs_count</td>
<td>number of control sequences</td>
</tr>
<tr>
<td>dest_names_size</td>
<td>pdf destination table size</td>
</tr>
<tr>
<td>dvi_gone</td>
<td>written dvi bytes</td>
</tr>
<tr>
<td>dvi_ptr</td>
<td>not yet written dvi bytes</td>
</tr>
<tr>
<td>dyn_used</td>
<td>token (multi-word) memory in use</td>
</tr>
<tr>
<td>filename</td>
<td>name of the current input file</td>
</tr>
<tr>
<td>fix_mem_end</td>
<td>maximum number of used tokens</td>
</tr>
<tr>
<td>fix_mem_min</td>
<td>minimum number of allocated words for tokens</td>
</tr>
<tr>
<td>fix_mem_max</td>
<td>maximum number of allocated words for tokens</td>
</tr>
<tr>
<td>font_ptr</td>
<td>number of active fonts</td>
</tr>
<tr>
<td>hash_extra</td>
<td>extra allowed hash</td>
</tr>
<tr>
<td>hash_size</td>
<td>size of hash</td>
</tr>
<tr>
<td>indirect_callbacks</td>
<td>number of those that were themselves a result of other callbacks (e.g. file readers)</td>
</tr>
<tr>
<td>ini_version</td>
<td>true if this is an ini\TeX run</td>
</tr>
<tr>
<td>init_pool_ptr</td>
<td>ini\TeX string pool index</td>
</tr>
</tbody>
</table>
init_str_ptr number of init\TeX{} strings
input_ptr the level of input we’re at
inputid numeric id of the current input
largest_used_mark max referenced marks class
lasterrorcontext last error context string (with newlines)
lasterrorstring last \TeX{} error string
lastluaerrorstring last Lua error string
lastwarningstring last warning tag, normally an indication of in what part
lastwarningtag last warning string
linenumber location in the current input file
log_name name of the log file
luabytecode_bytes number of bytes in Lua bytecode registers
luabytecodes number of active Lua bytecode registers
luastate_bytes number of bytes in use by Lua interpreters
luatex_engine the Lua\TeX{} engine identifier
luatex_hashchars length to which Lua hashes strings ($2^n$)
luatex_hashtype the hash method used (in Luajit\TeX{})
luatex_version the Lua\TeX{} version number
luatex_revision the Lua\TeX{} revision string
max_buf_stack max used buffer position
max_in_stack max used input stack entries
max_nest_stack max used nesting stack entries
max_param_stack max used parameter stack entries
max_save_stack max used save stack entries
max_strings maximum allowed strings
nest_size nesting stack size
node_mem_usage a string giving insight into currently used nodes
obj_ptr max pdf object pointer
obj_tab_size pdf object table size
output_active true if the \texttt{\output} routine is active
output_file_name name of the pdf or dvi file
param_size parameter stack size
pdf_dest_names_ptr max pdf destination pointer
pdf_gone written pdf bytes
pdf_mem_ptr max pdf memory used
pdf_mem_size pdf memory size
pdf_os_cntr max pdf object stream pointer
pdf_os_objidx pdf object stream index
pdf_ptr not yet written pdf bytes
pool_ptr string pool index
pool_size current size allocated for string characters
save_size save stack size
shell_escape 0 means disabled, 1 means anything is permitted, and 2 is restricted
safer_option 1 means safer is enforced
kpse_used 1 means that kpse is used
stack_size input stack size
str_ptr  number of strings
total_pages  number of written pages
var_mem_max  number of allocated words for nodes
var_used  variable (one-word) memory in use
lc_collate  the value of LC_COLLATE at startup time (becomes C at startup)
lc_ctype  the value of LC_CTYPE at startup time (becomes C at startup)
lc_numeric  the value of LC_NUMERIC at startup time

The error and warning messages can be wiped with the resetmessages function. A return value can be set with setexitcode.

10.3 The \texttt{tex} library

10.3.1 Introduction

The \texttt{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 \texttt{pairs} and \#) do not work on such items.

At the moment, it is possible to access almost every parameter that you can use after \texttt{\the}, is a single tokens or is sort of special in \TeX. This excludes parameters that need extra arguments, like \texttt{\the\scriptfont}. The subset comprising simple integer and dimension registers are writable as well as readable (like \texttt{\tracingcommands} and \texttt{\parindent}).

10.3.2 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 \texttt{tex} table, or by using one of the functions \texttt{tex.get} and \texttt{tex.set}.

The exact parameters and return values differ depending on the actual parameter, and so does whether \texttt{tex.set} has any effect. For the parameters that can be set, it is possible to use \texttt{global} as the first argument to \texttt{tex.set}; this makes the assignment global instead of local.

\begin{verbatim}
tex.set ("global",] <string> n, ...) ...
... = tex.get (<string> n)
\end{verbatim}

Glue is kind of special because there are five values involved. The return value is a glue_spec node but when you pass \texttt{false} as last argument to \texttt{tex.get} you get the width of the glue and when you pass \texttt{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. If you pass \texttt{true} to get you get 5 values returned for a glue and when you pass \texttt{false} you only get the width returned.

10.3.2.1 Integer parameters

The integer parameters accept and return Lua numbers. These are read-write:
10.3.2.2 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:
\begin{verbatim}
 These are read-only:

 tex.pagedepth        tex.pagefillstretched        tex.pagestretch
 tex.pagefillstretched tex.pagegoal          tex.pagetotal
 tex.pagefillstretch  tex.pageshrink

 Beware: as with all Lua tables you can add values to them. 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 \texttt{tex.nest} table:

 \begin{verbatim}
 tex.nest[tex.nest.ptr].prevdepth = p
 tex.nest[tex.nest.ptr].spacefactor = s
 \end{verbatim}

 However, the following also works:

 \begin{verbatim}
 tex.prevdepth = p
 tex.spacefactor = s
 \end{verbatim}

 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 Lua\TeX{} can guess your intentions. By using the accessor in the \texttt{tex} tables, you get and set the values at the top of the nesting stack.

 10.3.2.3 Direction parameters

 The direction parameters are read-only and return a Lua string.

 tex.bodydir                    tex.pagedir                    tex.textdir
 tex.mathdir                    tex.pardir
\end{verbatim}

192 The \TeX{} related libraries
10.3.2.4 Glue parameters

The glue parameters accept and return a userdata object that represents a glue_spec node.

- \texttt{tex.abovedisplayshortskip}
- \texttt{tex.abovedisplayskip}
- \texttt{tex.baselineskip}
- \texttt{tex.belowdisplayshortskip}
- \texttt{tex.belowdisplayskip}
- \texttt{tex.leftskip}
- \texttt{tex.lineskip}
- \texttt{tex.parfillskip}
- \texttt{tex.parskip}
- \texttt{tex.rightskip}
- \texttt{tex.spaceskip}
- \texttt{tex.splittopskip}
- \texttt{tex.tabskip}
- \texttt{tex.topskip}
- \texttt{tex.xspaceskip}

10.3.2.5 Muglue parameters

All muglue parameters are to be used read-only and return a Lua string.

- \texttt{tex.medmuskip}
- \texttt{tex.thickmuskip}
- \texttt{tex.thinmuskip}

10.3.2.6 Tokenlist parameters

The tokenlist parameters accept and return Lua strings. Lua strings are converted to and from token lists using \texttt{\the\toks} style expansion: all category codes are either space (10) or other (12). It follows that assigning to some of these, like \texttt{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.

- \texttt{tex.errhelp}
- \texttt{tex.everycr}
- \texttt{tex.everydisplay}
- \texttt{tex.everyeof}
- \texttt{tex.everyhbox}
- \texttt{tex.everyjob}
- \texttt{tex.everymath}
- \texttt{tex.everyeof}
- \texttt{tex.everypar}

10.3.3 Convert commands

All ‘convert’ commands are read-only and return a Lua string. The supported commands at this moment are:

- \texttt{tex.eTeXVersion}
- \texttt{tex.eTeXrevision}
- \texttt{tex.formatname}
- \texttt{tex.jobname}
- \texttt{tex.luatexbanner}
- \texttt{tex.luatexrevision}
- \texttt{tex.fontname(number)}
- \texttt{tex.uniformdeviate(number)}
- \texttt{tex.number(number)}
- \texttt{tex.romannumeral(number)}
- \texttt{tex.fontidentifier(number)}

If you are wondering why this list looks haphazard; these are all the cases of the ‘convert’ internal command that do not require an argument, as well as the ones that require only a simple numeric value. The special (Lua-only) case of \texttt{tex.fontidentifier} returns the \texttt{csname} string that matches a font id number (if there is one).
10.3.4 Last item commands

All ‘last item’ commands are read-only and return a number. The supported commands at this moment are:

- `tex.lastpenalty`
- `tex.lastkern`
- `tex.lastskip`
- `tex.lastnodetype`
- `tex.inputlineno`
- `tex.lastxpos`
- `tex.randomseed`
- `tex.luatexversion`
- `tex.eTeXminorversion`
- `tex.eTeXversion`
- `tex.currentgrouplevel`
- `tex.currentgrouptype`
- `tex.currentiflevel`
- `tex.currentiftype`
- `tex.currentifbranch`

10.3.5 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 up to 5 arguments: width, stretch, shrink, stretch order and shrink order. If you pass no values or if a value is not a number the corresponding property will become a zero. The getglue function reports all 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("bar") then
        tex.setdimen("bar",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 up to five 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,"$\int\limits^1_2$")

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.

10.3.6 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 \text table

\begin{verbatim}
tex.lccode tex.uccode tex.sfcode tex.catcode tex.mathcode tex.delcode
\end{verbatim}

The function call interfaces are roughly as above, but there are a few twists. \textcodes are the simple ones:

\begin{verbatim}
tex.setsfcode ("global",] <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
\end{verbatim}

The function call interface for \lccode and \uccode additionally allows you to set the associated sibling at the same time:

\begin{verbatim}
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)
\end{verbatim}

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):

\begin{verbatim}
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)
\end{verbatim}

The interfaces for \delcode and \mathcode use small array tables to set and retrieve values:

\begin{verbatim}
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)
\end{verbatim}

Where the table for \mathcode is an array of 3 numbers, like this:

\begin{verbatim}
{
 <number> class,
 <number> family,
 <number> character
}\end{verbatim}
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 \delcode 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.

### 10.3.7 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])

10.3.8 Reusing boxes: [use|save]boxresource and getboxresourcedimensions

The following function will register a box for reuse (this is modelled after so called xforms in pdf). You can (re)use the box with \useboxresource or by creating a rule node with subtype 2.

local index = tex.saveboxresource(n,attributes,resources,immediate,type,margin)

The optional second and third arguments are strings, the fourth is a boolean. The fifth argument is a type. When set to non-zero the /Type entry is omitted. A value of 1 or 3 still writes a /BoundingBox, while 2 or 3 will write a /Matrix. The sixth argument is the (virtual) margin that extends beyond the effective bounding box as seen by \TeX. Instead of a box number one can also pass a [h|v]list node.

You can generate the reference (a rule type) with:

local reused = tex.useboxresource(n,wd,ht,dp)

The dimensions are optional and the final ones are returned as extra values. The following is just a bonus (no dimensions returned means that the resource is unknown):

local w, h, d, m = tex.getboxresourcedimensions(n)

This returns the width, height, depth and margin of the resource.

10.3.9 triggerbuildpage

You should not expect too 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.

10.3.10 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.

10.3.11 Accessing math parameters: [get|set]math

It is possible to set and query the internal math parameters using:
\texttt{tex.setmath(["global"], \texttt{<string> n, <string> t, <number> n})
<number> n = tex.getmath(\texttt{<string> n, <string> t})

As before an optional first parameter \texttt{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:

\begin{verbatim}
quad axis operatorsize 
overbarkern overbarrule overbarvgap 
underbarkern underbarrule underbarvgap 
radicalkern radicalrule radicalvgap 
radicaldegreebefore radicaldegreeafter radicaldegreeraise 
stackvgap stacknumup stackdenomdown 
fractionrule fractionnumvgap fractionnumup 
fractiondenomvagap 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 ordinnerspacing 
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 openopspacing 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
\end{verbatim}

The values for the style parameter are:

\begin{verbatim}
display crampeddisplay 
text crampedtext 
script crampedscript 
scriptscript crampedscriptscript
\end{verbatim}
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.

10.3.12 Special list heads: [get|set]list

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>page_ins_head</td>
<td>circular list of pending insertions</td>
</tr>
<tr>
<td>contrib_head</td>
<td>the recent contributions</td>
</tr>
<tr>
<td>page_head</td>
<td>the current page content</td>
</tr>
<tr>
<td>hold_head</td>
<td>used for held-over items for next page</td>
</tr>
<tr>
<td>adjust_head</td>
<td>head of the current <code>\vadjust</code> list</td>
</tr>
<tr>
<td>pre_adjust_head</td>
<td>head of the current <code>\vadjust pre</code> list</td>
</tr>
<tr>
<td>page_discards_head</td>
<td>head of the discarded items of a page break</td>
</tr>
<tr>
<td>split_discards_head</td>
<td>head of the discarded items in a vsplit</td>
</tr>
</tbody>
</table>

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.

10.3.13 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:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>MODES</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>number</td>
<td>all</td>
<td>the meaning of these numbers depends on the engine and sometimes even the version; you can use <code>tex.getmodevalues()</code> to get the mapping: positive values signal vertical, horizontal and math mode, while negative values indicate inner and inline variants</td>
</tr>
<tr>
<td>modeline</td>
<td>number</td>
<td>all</td>
<td>source input line where this mode was entered in, negative inside the output routine</td>
</tr>
<tr>
<td>head</td>
<td>node</td>
<td>all</td>
<td>the head of the current list</td>
</tr>
<tr>
<td>tail</td>
<td>node</td>
<td>all</td>
<td>the tail of the current list</td>
</tr>
<tr>
<td>prevgraf</td>
<td>number</td>
<td>vmode</td>
<td>number of lines in the previous paragraph</td>
</tr>
</tbody>
</table>
\begin{itemize}
\item prevdepth \hspace{1cm} number \hspace{1cm} vmode \hspace{1cm} depth of the previous paragraph
\item spacefactor \hspace{1cm} number \hspace{1cm} hmode \hspace{1cm} the current space factor
\item dirs \hspace{1cm} node \hspace{1cm} hmode \hspace{1cm} used for temporary storage by the line break algorithm
\item noad \hspace{1cm} node \hspace{1cm} mmode \hspace{1cm} used for temporary storage of a pending fraction numerator, for \texttt{\LaTeX} etc.
\item delimptr \hspace{1cm} node \hspace{1cm} mmode \hspace{1cm} used for temporary storage of the previous math delimiter, for \texttt{\LaTeX}
\item mathdir \hspace{1cm} boolean \hspace{1cm} mmode \hspace{1cm} true when during math processing the \textbackslash mathdir is not the same as the surrounding \textbackslash textdir
\item mathstyle \hspace{1cm} number \hspace{1cm} mmode \hspace{1cm} the current \texttt{\LaTeX} style
\end{itemize}

## 10.3.14 Print functions

The \texttt{tex} table also contains the three print functions that are the major interface from Lua scripting to \LaTeX. The arguments to these three functions are all stored in an in-memory virtual file that is fed to the \LaTeX scanner as the result of the expansion of \texttt{\LaTeX}. The total amount of returnable text from a \texttt{\LaTeX} command is only limited by available system ram. However, each separate printed string has to fit completely in \LaTeX’s input buffer. The result of using these functions from inside callbacks is undefined at the moment.

### 10.3.14.1 print

\begin{verbatim}
tex.print(<string> s, ...)
tex.print(<number> n, <string> s, ...)
tex.print(<table> t)
tex.print(<number> n, <table> t)
\end{verbatim}

Each string argument is treated by \LaTeX 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 \texttt{\catcode{n}}. If \texttt{n} is \texttt{-1}, the currently active catcode regime is used. If \texttt{n} is \texttt{-2}, the resulting catcodes are the result of \texttt{\toks}: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, if \texttt{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 \texttt{tex.print} command in a \texttt{\LaTeX} will not have the \texttt{\endlinechar} appended, all others do.

### 10.3.14.2 sprint

\begin{verbatim}
tex.sprint(<string> s, ...)
tex.sprint(<number> n, <string> s, ...)
tex.sprint(<table> t)
tex.sprint(<number> n, <table> t)
\end{verbatim}

\FloatBarrier
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 \texttt{\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

\begin{verbatim}
before\directlua{tex.sprint("\\relax")tex.sprint(" inbetween")}after
\end{verbatim}

the space before \texttt{in} between will be gobbled as a result of the ‘normal’ scanning of \texttt{\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 \texttt{\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 \texttt{<token userdata>} already is tokenized and effectively is a token list now. A \texttt{<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 \texttt{\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.

### \texttt{\tex.tprint} \subsection*{10.3.14.3 \texttt{tprint}}

\texttt{tex.tprint({<number> n, <string> s, ...}, {...})}

This function is basically a shortcut for repeated calls to \texttt{\tex.sprint(<number> n, <string> s, ...)}, once for each of the supplied argument tables.
10.3.14.4 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.

```latex
\begin{verbatim}
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
\end{verbatim}
```

10.3.14.5 write

```latex
\begin{verbatim}
tex.write(<string> s, ...) tex.write(<table> t)
\end{verbatim}
```

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).

10.3.15 Helper functions

10.3.15.1 round

```latex
<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.

10.3.15.2 scale

```latex
<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 Lua\TeX{}’s internal, 100% portable scale function will be added at a later date.
10.3.15.3 number and romannumeral

These are the companions to the primitives \number and \romannumeral. They can be used like:

tex.print(tex.romannumeral(123))

10.3.15.4 fontidentifier and fontname

The first one returns the name only, the second one reports the size too.

tex.print(tex.fontidentifier(1))
tex.print(tex.fontname(1))

10.3.15.5 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 Lua\TeX{} 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)

10.3.15.6 error and show_context

tex.error(<string> s)
tex.error(<string> s, <table> help)

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).

10.3.15.7 run, finish

These two functions start the interpretations and force its end. A runs normally boils down to \TeX{} entering the so called main loop. A token is fetched and depending on it current meaning some actions takes place. Sometimes that actions comes immediately, sometimes more scanning is needed. Quite often tokens get pushed back into the input. This all means that the \TeX{} scanner is constantly pushing and popping input states, but in the end after all the action is done returns to the main loop.
10.3.15.8 runtoks

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>)
```

When the \tracingnesting parameter is set to a value larger than 2 some information is reported about the state of the local loop.

This function has two optional arguments in case a token register is passed:

```
tex.runtoks(<token register>,force,grouped)
```

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.

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!).

10.3.15.9 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.

10.3.15.10 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.

10.3.15.11 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).

### 10.3.16 Functions for dealing with primitives

#### 10.3.16.1 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 \texttt{\textbackslash{}LuaTeX\textbackslash{}formatname} with the same intrinsic meaning as the documented primitive \texttt{\textbackslash{}formatname}, provided that the control sequences \texttt{\textbackslash{}\template{\textbackslash{}LuaTeX\textbackslash{}formatname}} is currently undefined.

When \texttt{\textbackslash{}\template{\textbackslash{}Lua\textbackslash{}TeX}} is run with \texttt{--ini} only the \texttt{\template{\textbackslash{}TeX82}} primitives and \texttt{\textbackslash{}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 \texttt{\template{\textbackslash{}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.

#### 10.3.16.2 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.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUES</th>
</tr>
</thead>
</table>
| tex   | Uleft Umiddle Uright above abovedisplayshortskip abovedisplayskip abovewithdelims accent adjdemerits advance afterassignment aftergroup atop atopwithdelims badness baselineskip batchmode begingroup belowdisplayshortskip belowdisplayskip binoppenalty botmark boundary box boxmaxdepth brokenpenalty catcode char chardef cleaders closein closeout clubpenalty copy count countdef cr crrc csname day deadcycles def defaulthyphenchar defaultskewchar deferred delcode delimiter delimiterfactor delimitershortfall dimen dimendef discretionary displayindent displaylimits displaystyle displaywidowpenalty displaywidth divide doublehyphenmerits dp dump edef else emergencystretch end endcsname endgroup endinput endlinechar eqno errhelp errmessage errorcontextlines errortopmode escapechar everycr everydisplay every hbox everyjob everymath everypar everyxybox exhyphenchar exhyphenpenalty expandafter fam fi finalhyphenmerits firstmark firstvalidlanguage floatingpenalty fontdimen fontname futurelet gdef glet global globaldefs halign hangafter hangindent hbadness hbox hfil hfilneg h fuzz hoffset holdinginserts hpack hrule hsize hskip hss ht hyphenation hyphenchar hyphenpenalty if ifcase ifcat ifdim ifelse iffalse if hbox ifmode ifinner ifnum ifodd iftrue if vbox if void ifx ignorespaces immediate indent input inputlineno insertinsertpenalties interlinepenalty jobname kern language lastbox last kern lastpenalty lastskip lccode leaders left lefthyphenmin leftskip leftmark limits linepenalty lineskip lineskiplimit long looseness lower lowercase magmark m athaccent mathbin mathchar mathchardef mathchoice mathcode mathinner mathop mathopen mathord mathrel mathsurround maxdeadcycles maxdepth meaning medmuskip message middle mkern month moveleft movergroup m skip multiply muskip muskipdef newlinechar noalign noboundary noexpand noident nolimits nonscript nonstopmode null delimiterspace nullfont number omit openin openout or outer output outputpenalty overfullrule overline overwithdelims pagedepth pagefillstretch pagefillstret ch pagegoal pageshrink pagestretch pagetotal par parfillskip parindent parshape parskip patterns pausing penalty postdisplaypenalty predisplaypenalty pre-displaysize pretolerance prevdepth prevgraf protrusionboundary radical raise read relax relpenalty right righthyphenmin rightskip roman numeral scriptfont scriptscript font scriptscriptstyle scripts pace scriptstyle scrollmode setbox setlanguage sf code shipout show showbox showboxbreadth showboxdepth showlists showstream showthe skewchar skip skipdef spacefactor space span special splitbotmark splitfirstmark splitmaxdepth splittopskip string tabskip text font text style the thick muskip thinmuskip time toks tokskdef tolerance topmark topskip tpack tracingcommands tracing lostchars tracing macros tracingonline tracingoutput tracingparagraphs tracing restores tracing stats uccode uch yph underline unhbox unh copy unkern unpenalty unskip unvbox unvcopy uppercase vadjust valign vbadness vbox vcenter vfil vfill vfil-
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").
10.3.16.3 primitives

<table> t = tex.primitives()

This function returns a list of all primitives that LuaTEX knows about.

10.3.17 Core functionality interfaces

10.3.17.1 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 $100(t/s)^3$.

10.3.17.2 tex.resetparagraph

This function resets the parameters that TeX normally resets when a new paragraph is seen.

10.3.17.3 linebreak

local <node> nodelist, <table> info =
   tex.linebreak(<node> listhead, <table> parameters)

The understood parameters are as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>pardir</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>pretolerance</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tracingparagraphs</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tolerance</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>looseness</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>exhyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pdfadjustspacing</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>adjdemerits</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pdfprotrudechars</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>linepenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastlinefit</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>doublehyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>finalhyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hangafter</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>interlinepenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \interlinepenalties</td>
</tr>
<tr>
<td>clubpenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \clubpenalties</td>
</tr>
</tbody>
</table>
widowpenalty \quad number\ or\ table\quad if\ a\ table,\ then\ it\ is\ an\ array\ like  \textbackslash widowpenalties

brokenpenalty \quad number

emergencystretch \quad number \quad in\ scaled\ points

hangindent \quad number \quad in\ scaled\ points

hsize \quad number \quad in\ scaled\ points

leftskip \quad glue\_spec\ node

rightskip \quad glue\_spec\ node

parshape \quad table

Note that there is no interface for \textbackslash 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 \texttt{pre\_linebreak\_filter} or \texttt{linebreak\_filter} callbacks, or when the original list starting at listhead was generated in horizontal mode):

- add an ‘indent box’ and perhaps a \texttt{local\_par} node at the start (only if you need them)
- replace any found final glue by an infinite \texttt{par} node (or add such a penalty, if the last node is not a glue)
- add a glue node for the \texttt{parfillskip} after that penalty node
- make sure all the prev pointers are OK

The result is a node list, it still needs to be \texttt{vpacked} if you want to assign it to a \texttt{vbox}. The returned \texttt{info} table contains four values that are all numbers:

<table>
<thead>
<tr>
<th>NAME</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevdepth</td>
<td>depth of the last line in the broken paragraph</td>
</tr>
<tr>
<td>prevgraf</td>
<td>number of lines in the broken paragraph</td>
</tr>
<tr>
<td>looseness</td>
<td>the actual looseness value in the broken paragraph</td>
</tr>
<tr>
<td>demerits</td>
<td>the total demerits of the chosen solution</td>
</tr>
</tbody>
</table>

Note there are a few things you cannot interface using this function: You cannot influence font expansion other than via \texttt{pdfadjustspacing}, because the settings for that take place elsewhere. The same is true for \texttt{hbadness} and \texttt{hfuzz} etc. All these are in the \texttt{hpack} routine, and that fetches its own variables via globals.

10.3.17.4 shipout

tex.shipout(<number> n)

Ships out box number n to the output file, and clears the box register.

10.3.17.5 getpagestate

This helper reports the current page state: empty, box\_there or inserts\_only as integer value.
10.3.17.6 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.

10.3.18 Randomizers

For practical reasons Lua\TeX{} has its own random number generator. The original Lua random function is available as \texttt{tex.lua\_math\_random}. You can initialize with a new seed with \texttt{init\_rand} (\texttt{lua\_math\_randomseed} is equivalent to this one).

There are three generators: \texttt{normal\_rand} (no argument is used), \texttt{uniform\_rand} (takes a number that will get rounded before being used) and \texttt{uniformdeviate} which behaves like the primitive and expects a scaled integer, so

\begin{verbatim}
tex.print(tex.uniformdeviate(65536)/65536)
\end{verbatim}

will give a random number between zero and one.

10.3.19 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.

<table>
<thead>
<tr>
<th>NAME</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_synctex_mode</td>
<td>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</td>
</tr>
<tr>
<td>set_synctex_tag</td>
<td>set the current tag (file) value (obeys save stack)</td>
</tr>
<tr>
<td>set_synctex_line</td>
<td>set the current line value (obeys save stack)</td>
</tr>
<tr>
<td>set_synctex_no_files</td>
<td>disable synctex file logging</td>
</tr>
<tr>
<td>get_synctex_mode</td>
<td>returns the current mode (for values see above)</td>
</tr>
<tr>
<td>get_synctex_tag</td>
<td>get the currently set value of tag (file)</td>
</tr>
<tr>
<td>get_synctex_line</td>
<td>get the currently set value of line</td>
</tr>
<tr>
<td>force_synctex_tag</td>
<td>overload the tag (file) value (0 resets)</td>
</tr>
<tr>
<td>force_synctex_line</td>
<td>overload the line value (0 resets)</td>
</tr>
</tbody>
</table>

The last one is somewhat special. Due to the way files are registered in Sync\TeX{} 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.

10.4 The \texttt{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.
<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>DEFAULT</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>kpse_init</td>
<td>boolean</td>
<td>true</td>
<td>false totally disables kpathsea initialisation, and enables interpretation of the following numeric key–value pairs. (only ever unset this if you implement all file find callbacks!)</td>
</tr>
<tr>
<td>shell_escape</td>
<td>string</td>
<td>'f'</td>
<td>Use 'y' or 't' or '1' to enable \write 18 unconditionally, 'p' to enable the commands that are listed in shell_escape_commands</td>
</tr>
<tr>
<td>shell_escape_commands</td>
<td>string</td>
<td></td>
<td>Comma-separated list of command names that may be executed by \write 18 even if shell_escape is set to 'p'. Do not use spaces around commas, separate any required command arguments by using a space, and use the ascii double quote (&quot;) for any needed argument or path quoting</td>
</tr>
<tr>
<td>string_vacancies</td>
<td>number</td>
<td>75000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>pool_free</td>
<td>number</td>
<td>5000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_strings</td>
<td>number</td>
<td>15000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>strings_free</td>
<td>number</td>
<td>100</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>nest_size</td>
<td>number</td>
<td>50</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_in_open</td>
<td>number</td>
<td>15</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>param_size</td>
<td>number</td>
<td>60</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>save_size</td>
<td>number</td>
<td>4000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>stack_size</td>
<td>number</td>
<td>300</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>dvi_buf_size</td>
<td>number</td>
<td>16384</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>error_line</td>
<td>number</td>
<td>79</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>half_error_line</td>
<td>number</td>
<td>50</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_print_line</td>
<td>number</td>
<td>79</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>hash_extra</td>
<td>number</td>
<td>0</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>pk_dpi</td>
<td>number</td>
<td>72</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>trace_file_names</td>
<td>boolean</td>
<td>true</td>
<td>false disables \TeX's normal file open-close feedback (the assumption is that callbacks will take care of that)</td>
</tr>
</tbody>
</table>
Note: the numeric values that match web2c parameters are only used if kpse_init is explicitly set to false. In all other cases, the normal values from texmf.cnf are used.

You can kick in your own nesting level visualizer, for instance:

callback.register("input_level_string", function(n)
    if tex.tracingmacros > 0 and tex.count.tracingstacklevels > 0 then
        if tex.tracingmacros > 1 then
            return "! " .. string.rep(">",n) .. " "
        end
    end
end)

Or, in sync with other engines (not checked):

\directlua{
    callback.register("input_level_string", function(n)
        if tex.tracingmacros > 0 then
            local l = tex.count.tracingstacklevels
            if l > 0 then
                return string.rep("~",l) .. string.rep( ",",n-l)
            end
        end
        end)
    }

10.5 The texio library

This library takes care of the low-level I/O interface: writing to the log file and/or console.

10.5.1 write

texio.write(<string> target, <string> s, ...)
texio.write(<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 three possibilities: term, log or term and log.

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.

10.5.2 write_nl

texio.write_nl(<string> target, <string> s, ...)
texio.write_nl(<string> s, ...)

214 The \TeX related libraries
This function behaves like texio.write, but make 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.

### 10.5.3 setescape

You can disable ^^ escaping of control characters by passing a value of zero.

### 10.5.4 closeinput

This function that 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 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).

### 10.6 The token library

#### 10.6.1 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 ok. 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 Lua\TeX is to provide a minimal set of tools and no solutions. The scanner functions are probably the most intriguing.

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>ARGUMENT</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_keyword</td>
<td>string</td>
<td>returns true if the given keyword is gobbled; as with the regular \TeX keyword scanner this is case insensitive (and ascii based)</td>
</tr>
<tr>
<td>scan_keywordcs</td>
<td>string</td>
<td>returns true if the given keyword is gobbled; this variant is case sensitive and also suitable for utf8</td>
</tr>
<tr>
<td>scan_int</td>
<td></td>
<td>returns a number from e.g. 1, 1.1, .1 with optional collapsed signs</td>
</tr>
<tr>
<td>scan_real</td>
<td></td>
<td>returns a number from e.g. 1, 1.1, .1, 1.1E10, .1e-10 with optional collapsed signs</td>
</tr>
<tr>
<td>scan_float</td>
<td>infinity, mu-units</td>
<td>returns a number representing a dimension and or two numbers being the filler and order</td>
</tr>
<tr>
<td>scan_glue</td>
<td>mu-units</td>
<td>returns a glue spec node</td>
</tr>
<tr>
<td>scan_toks</td>
<td>definer, expand</td>
<td>returns a table of tokens tokens</td>
</tr>
<tr>
<td>scan_code</td>
<td>bitset</td>
<td>returns a character if its category is in the given bitset (representing catcodes)</td>
</tr>
<tr>
<td>scan_string</td>
<td></td>
<td>returns a string given between {}, as \macro or as sequence of characters with catcode 11 or 12</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| `scan_argument` | boolean  
this one is similar to `scanstring` but also accepts a \cs  
returns a sequence of characters with catcode 11 or 12 as string |
| `scan_word`    | returns foo after scanning \foo                                           |
| `scan_csname`  | picks up a box specification and returns a [h|v]list node                  |
| `scan_list`    |                                                                             |

The scanners can be considered stable apart from the one scanning for a token. The `scan_code` function takes an optional number, the `keyword` 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 `scan_argument` 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:

```latex
\def\bar{bar}
\def\foo{foo-\bar}
```

we get:

<table>
<thead>
<tr>
<th>NAME</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\directlua{token.scan_string()}{foo}</code></td>
<td>foo full expansion</td>
</tr>
<tr>
<td><code>\directlua{token.scan_string()}{foo}</code></td>
<td>foo letters and others</td>
</tr>
<tr>
<td><code>\directlua{token.scan_string()}{\foo}</code></td>
<td><code>foo-bar</code> meaning</td>
</tr>
</tbody>
</table>

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 `scan_word` scanner can be used to implement for instance a number scanner:

```lua
function token.scan_number(base)  
   return tonumber(token.scan_word(),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:

```latex
\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.scan_dimen()
        ...
    end
}\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).

\subsection{Picking up one token}

The scanners look for a sequence. When you want to pick up one token from the input you use get_next. 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 scan_token. 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.
10.6.3 Creating tokens

The creator function can be used as follows:

```lua
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:

<table>
<thead>
<tr>
<th>NAME</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>a number representing the internal command number</td>
</tr>
<tr>
<td>cmdname</td>
<td>the type of the command (for instance the catcode in case of a character or the classifier that determines the internal treatment</td>
</tr>
<tr>
<td>csname</td>
<td>the associated control sequence (if applicable)</td>
</tr>
<tr>
<td>id</td>
<td>the unique id of the token</td>
</tr>
<tr>
<td>tok</td>
<td>the full token number as stored in \TeX</td>
</tr>
<tr>
<td>active</td>
<td>a boolean indicating the active state of the token</td>
</tr>
<tr>
<td>expandable</td>
<td>a boolean indicating if the token (macro) is expandable</td>
</tr>
<tr>
<td>protected</td>
<td>a boolean indicating if the token (macro) is protected</td>
</tr>
<tr>
<td>mode</td>
<td>a number either representing a character or another entity</td>
</tr>
<tr>
<td>index</td>
<td>a number running from 0x0000 upto 0xFFFF indicating a \TeX register index</td>
</tr>
</tbody>
</table>

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:

```lua
local t = token.commands()
```

The id of a token class can be queried as follows:

```lua
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:

```lua
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:

```lua
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")

When a second argument to \is\_defined is true the check is for an undefined control sequence (only), otherwise any undefined command gives true.

The largest character possible is returned by \biggest\_char, just in case you need to know that boundary condition.

### 10.6.4 Macros

The \set\_macro function can get up to 4 arguments:

```lua
set_macro("csname","content")
set_macro("csname","content","global")
set_macro("csname")
```

You can pass a catcode table identifier as first argument:

```lua
set_macro(catcodetable,"csname","content")
set_macro(catcodetable,"csname","content","global")
set_macro(catcodetable,"csname")
```

The results are like:

```latex
\def\csname{content}
\gdef\csname{content}
\def\csname{}
```

The \get\_macro function can be used to get the content of a macro while the \get\_meaning function gives the meaning including the argument specification (as usual in \TeX separated by \texttt{->}).

The \set\_char function can be used to do a \texttt{\chardef} at the Lua end, where invalid assignments are silently ignored:

```lua
set_char("csname",number)
set_char("csname",number,"global")
```

A special one is the following:

```lua
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 \texttt{lua.get\_functions\_table}. It is the companion to \texttt{\luadef}.

### 10.6.5 Pushing back

There is a (for now) experimental putter:

```lua
local t1 = token.get\_next()
```
local t2 = token.get_next()
local t3 = token.get_next()
local t4 = token.get_next()
-- watch out, we flush in sequence
token.put_next { t1, t2 }
-- but this one gets pushed in front
token.put_next ( 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.

### 10.6.6 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:

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

but \bar is undefined. Normally \TeX\ will then issue an error message. However, when we have:

\begin{verbatim}
\def\foo{\bar}
\end{verbatim}

We get no error, unless we expand \foo while \bar 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, \bar really exists but can be in an undefined state.

\begin{verbatim}
bar : bar
foo : foo
myfirstbar :
\end{verbatim}

This was entered as:

\begin{verbatim}
bar  : \directlua{tex.print(token.scan_csname())}\bar
foo  : \directlua{tex.print(token.scan_csname())}\foo
myfirstbar : \directlua{tex.print(token.scan_csname())}\myfirstbar
\end{verbatim}

The reason that you see bar reported and not myfirstbar is that \bar was already used in a previous paragraph.

If we now say:

\begin{verbatim}
\def\foo{}
\end{verbatim}

we get:

\begin{verbatim}
bar : bar
foo : foo
myfirstbar :
\end{verbatim}
And if we say
\def\foo{\bar}
we get:
bar : bar
foo : foo
myfirstbar :

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.

This was generated with:
\directlua{local t = token.get_next() tex.print(t.id.." ..t.tok)}\myfirstbar
\directlua{local t = token.get_next() tex.print(t.id.." ..t.tok)}\mysecondbar
\directlua{local t = token.get_next() tex.print(t.id.." ..t.tok)}\mythirdbar

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.

10.7 The kpse library

This library provides two separate, but nearly identical interfaces to the kpathsea file search functionality: there is a ‘normal’ procedural interface that shares its kpathsea instance with Lua\TeX itself, and an object oriented interface that is completely on its own.

10.7.1 set_program_name and new

The way the library looks up variables is driven by the texmf.cmf file where the currently set program name acts as filter. You can check what file is used by with default_texmf.cnf.

Before the search library can be used at all, its database has to be initialized. There are three possibilities, two of which belong to the procedural interface.

First, when Lua\TeX is used to typeset documents, this initialization happens automatically and the kpathsea executable and program names are set to luatex (that is, unless explicitly prohibited by the user’s startup script. See section 4.1 for more details).

Second, in \TeX Lua mode, the initialization has to be done explicitly via the kpse.set_program_name function, which sets the kpathsea executable (and optionally program) name.

kpse.set_program_name(<string> name)
kpse.set_program_name(<string> name, <string> progranme)
The second argument controls the use of the ‘dotted’ values in the `texmf.cnf` configuration file, and defaults to the first argument.

Third, if you prefer the object oriented interface, you have to call a different function. It has the same arguments, but it returns a userdata variable.

```lua
local kpathsea = kpse.new(<string> name)
local kpathsea = kpse.new(<string> name, <string> progname)
```

Apart from these two functions, the calling conventions of the interfaces are identical. Depending on the chosen interface, you either call `kpse.find_file` or `kpathsea:find_file`, with identical arguments and return values.

### 10.7.2 record_input_file and record_output_file

These two function can be used to register used files. Because callbacks can load files themselves you might need these helpers (if you use recording at all).

```lua
kpse.record_input_file(<string> name)
kpse.record_output_file(<string> name)
```

### 10.7.3 find_file

The most often used function in the library is `find_file`:

```lua
<string> f = kpse.find_file(<string> filename)
<string> f = kpse.find_file(<string> filename, <string> ftype)
<string> f = kpse.find_file(<string> filename, <boolean> mustexist)
<string> f = kpse.find_file(<string> filename, <string> ftype, <boolean> mustexist)
<string> f = kpse.find_file(<string> filename, <string> ftype, <number> dpi)
```

Arguments:

- **filename**
  - the name of the file you want to find, with or without extension.
- **ftype**
  - maps to the `-format` argument of `kpsewhich`. The supported `ftype` values are the same as the ones supported by the standalone `kpsewhich` program: MetaPost support, PostScript header, TeX system documentation, TeX system sources, Troff fonts, afm, base, bib, bitmap font, bst, cid maps, clua, cmap files, cnf, cweb, dvips config, enc files, fmt, font feature files, gf, graphic/figure, ist, lig files, ls-R, lua, map, mem, mf, mfpool, mft, misc fonts, mlbib, mlbst, mp, mpool, ocp, ofm, opentype fonts, opf, other binary files, other text files, otp, ovf, ovp, pdftex config, pk, subfont definition files, tex, texmfscripts, texpool, tfm, truetype fonts, typel fonts, type42 fonts, vf, web, web2c files

The default type is `tex`. Note: this is different from `kpsewhich`, which tries to deduce the file type itself from looking at the supplied extension.
mustexist

is similar to kpsewhich’s -must-exist, and the default is false. If you specify true (or a non-zero integer), then the kpse library will search the disk as well as the ls-R databases.

dpi

This is used for the size argument of the formats pk, gf, and bitmap font.

If --output-directory is specified and the value is a relative pathname, the file is searched first here and if it fails it will be searched in the standard tree.

10.7.4 lookup

A more powerful (but slower) generic method for finding files is also available. It returns a string for each found file.

<string> f, ... = kpse.lookup(<string> filename, <table> options)

The options match commandline arguments from kpsewhich:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>number</td>
<td>set debugging flags for this lookup</td>
</tr>
<tr>
<td>format</td>
<td>string</td>
<td>use specific file type (see list above)</td>
</tr>
<tr>
<td>dpi</td>
<td>number</td>
<td>use this resolution for this lookup; default 600</td>
</tr>
<tr>
<td>path</td>
<td>string</td>
<td>search in the given path</td>
</tr>
<tr>
<td>all</td>
<td>boolean</td>
<td>output all matches, not just the first</td>
</tr>
<tr>
<td>mustexist</td>
<td>boolean</td>
<td>search the disk as well as ls-R if necessary</td>
</tr>
<tr>
<td>mktexpk</td>
<td>boolean</td>
<td>disable/enable mktexpk generation for this lookup</td>
</tr>
<tr>
<td>mktetexm</td>
<td>boolean</td>
<td>disable/enable mktexfm generation for this lookup</td>
</tr>
<tr>
<td>mktexmf</td>
<td>boolean</td>
<td>disable/enable mktexmf generation for this lookup</td>
</tr>
<tr>
<td>subdir</td>
<td>string or table</td>
<td>only output matches whose directory part ends with the given string(s)</td>
</tr>
</tbody>
</table>

If --output-directory is specified and the value is a relative pathname, the file is searched first here and then in the standard tree.

10.7.5 init_prog

Extra initialization for programs that need to generate bitmap fonts.

kpse.init_prog(<string> prefix, <number> base_dpi, <string> mfmode)
kpse.init_prog(<string> prefix, <number> base_dpi, <string> mfmode, <string> fallback)

10.7.6 readable_file

Test if an (absolute) file name is a readable file.

<string> f = kpse.readable_file(<string> name)
The return value is the actual absolute filename you should use, because the disk name is not always the same as the requested name, due to aliases and system-specific handling under e.g. msdos. Returns nil if the file does not exist or is not readable.

10.7.7 expand_path
Like kpsewhich’s -expand-path:

<string> r = kpse.expand_path(<string> s)

10.7.8 expand_var
Like kpsewhich’s -expand-var:

<string> r = kpse.expand_var(<string> s)

10.7.9 expand_braces
Like kpsewhich’s -expand-braces:

<string> r = kpse.expand_braces(<string> s)

10.7.10 show_path
Like kpsewhich’s -show-path:

<string> r = kpse.show_path(<string> ftype)

10.7.11 var_value
Like kpsewhich’s -var-value:

<string> r = kpse.var_value(<string> s)

10.7.12 version
Returns the kpse which version string.

<string> r = kpse.version()

10.7.13 check_permission
Checks if the filename can be executed or not. Returns 1, filename or a safe cmd alternative if it can be executed, otherwise returns 0 and an error message.

<string> res, cmd = kpse.check_permission(filename)
11 The graphic libraries

11.1 The \texttt{img} library

The \texttt{img} library can be used as an alternative to \texttt{\pdfximage} and \texttt{\pdfrefximage}, and the associated ‘satellite’ commands like \texttt{\pdfximagebbox}. Image objects can also be used within virtual fonts via the \texttt{image} command listed in section 6.3.

11.1.1 \texttt{new}

\begin{verbatim}
<image> var = img.new()
<image> var = img.new(<table> image_spec)
\end{verbatim}

This function creates a userdata object of type ‘image’. The \texttt{image_spec} argument is optional. If it is given, it must be a table, and that table must contain a filename key. A number of other keys can also be useful, these are explained below.

You can either say

\begin{verbatim}
a = img.new()
\end{verbatim}

followed by

\begin{verbatim}
a.filename = "foo.png"
\end{verbatim}

or you can put the file name (and some or all of the other keys) into a table directly, like so:

\begin{verbatim}
a = img.new({filename='foo.pdf', page=1})
\end{verbatim}

The generated \texttt{<image>} userdata object allows access to a set of user-specified values as well as a set of values that are normally filled in and updated automatically by Lua\TeX{} itself. Some of those are derived from the actual image file, others are updated to reflect the pdf output status of the object.

There is one required user-specified field: the file name (filename). It can optionally be augmented by the requested image dimensions (width, depth, height), user-specified image attributes (attr), the requested pdf page identifier (page), the requested boundingbox (pagebox) for pdf inclusion, the requested color space object (colorspace).

The function \texttt{img.new} does not access the actual image file, it just creates the \texttt{<image>} userdata object and initializes some memory structures. The \texttt{<image>} object and its internal structures are automatically garbage collected.

Once the image is scanned, all the values in the \texttt{<image>} except width, height and depth, become frozen, and you cannot change them any more.

You can use \texttt{pdf.setignoreunknownimages(1)} (or at the \TeX{} end the \texttt{\pdfvariable ignoreunknownimages}) to get around a quit when no known image type is found (based on name or preamble). Beware: this will not catch invalid images and we cannot guarantee side effects.
A zero dimension image is still included when requested. No special flags are set. A proper workflow will not rely in such a catch but make sure that images are valid.

11.1.2 fields

<table> keys = img.fields()

This function returns a list of all the possible image_spec keys, both user-supplied and automatic ones.

<table>
<thead>
<tr>
<th>FIELD NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>string</td>
<td>the image attributes for LuaTeX</td>
</tr>
<tr>
<td>bbox</td>
<td>table</td>
<td>table with 4 boundingbox dimensions llx, lly, urx and ury overruling the pagebox entry</td>
</tr>
<tr>
<td>colordepth</td>
<td>number</td>
<td>the number of bits used by the color space</td>
</tr>
<tr>
<td>colorspace</td>
<td>number</td>
<td>the color space object number</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the image depth for LuaTeX</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>the image file name</td>
</tr>
<tr>
<td>filepath</td>
<td>string</td>
<td>the full (expanded) file name of the image</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the image height for LuaTeX</td>
</tr>
<tr>
<td>imagetype</td>
<td>string</td>
<td>one of pdf, png, jpg, JP2 or jbig2</td>
</tr>
<tr>
<td>index</td>
<td>number</td>
<td>the pdf image name suffix</td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the pdf image object number</td>
</tr>
<tr>
<td>page</td>
<td>number</td>
<td>the identifier for the requested image page</td>
</tr>
<tr>
<td>pagebox</td>
<td>string</td>
<td>the requested bounding box, one of none, media, crop, bleed, trim, art</td>
</tr>
<tr>
<td>pages</td>
<td>number</td>
<td>the total number of available pages</td>
</tr>
<tr>
<td>rotation</td>
<td>number</td>
<td>the image rotation from included pdf file, in multiples of 90 deg.</td>
</tr>
<tr>
<td>stream</td>
<td>string</td>
<td>the raw stream data for an /XObject /Form object</td>
</tr>
<tr>
<td>transform</td>
<td>number</td>
<td>the image transform, integer number 0..7</td>
</tr>
<tr>
<td>orientation</td>
<td>number</td>
<td>the (jpeg) image orientation, integer number 1..8 (0 for unset)</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the image width for LuaTeX</td>
</tr>
<tr>
<td>xres</td>
<td>number</td>
<td>the horizontal natural image resolution (in dpi)</td>
</tr>
<tr>
<td>xsize</td>
<td>number</td>
<td>the natural image width</td>
</tr>
<tr>
<td>yres</td>
<td>number</td>
<td>the vertical natural image resolution (in dpi)</td>
</tr>
<tr>
<td>ysize</td>
<td>number</td>
<td>the natural image height</td>
</tr>
<tr>
<td>visiblefilename</td>
<td>string</td>
<td>when set, this name will find its way in the pdf file as PTEX specification; when an empty string is assigned nothing is written to file; otherwise the natural filename is taken</td>
</tr>
<tr>
<td>userpassword</td>
<td>string</td>
<td>the userpassword needed for opening a pdf file</td>
</tr>
<tr>
<td>ownerpassword</td>
<td>string</td>
<td>the ownerpassword needed for opening a pdf file</td>
</tr>
<tr>
<td>keepopen</td>
<td>boolean</td>
<td>keep the pdf file open</td>
</tr>
<tr>
<td>nobbox</td>
<td>boolean</td>
<td>don’t add a boundingbox specification for streams</td>
</tr>
<tr>
<td>nolength</td>
<td>boolean</td>
<td>don’t add length key nor compress for streams</td>
</tr>
<tr>
<td>nosize</td>
<td>boolean</td>
<td>don’t add size fields for streams</td>
</tr>
</tbody>
</table>

226 The graphic libraries
A running (undefined) dimension in width, height, or depth is represented as nil in Lua, so if you want to load an image at its ‘natural’ size, you do not have to specify any of those three fields.

The stream parameter allows to fabricate an /XObject /Form object from a string giving the stream contents, e.g., for a filled rectangle:

```lua
a.stream = "0 0 20 10 re f"
```

When writing the image, an /XObject /Form object is created, like with embedded pdf file writing. The object is written out only once. The stream key requires that also the bbox table is given. The stream key conflicts with the filename key. The transform key works as usual also with stream.

The bbox key needs a table with four boundingbox values, e.g.:

```lua
a.bbox = { "30bp", 0, "225bp", "200bp" }
```

This replaces and overrules any given pagebox value; with given bbox the box dimensions coming with an embedded pdf file are ignored. The xsize and ysize dimensions are set accordingly, when the image is scaled. The bbox parameter is ignored for non-pdf images.

The transform allows to mirror and rotate the image in steps of 90 deg. The default value 0 gives an unmirrored, unrotated image. Values 1 – 3 give counterclockwise rotation by 90, 180, or 270 degrees, whereas with values 4 – 7 the image is first mirrored and then rotated counterclockwise by 90, 180, or 270 degrees. The transform operation gives the same visual result as if you would externally preprocess the image by a graphics tool and then use it by LuaTeX. If a pdf file to be embedded already contains a /Rotate specification, the rotation result is the combination of the /Rotate rotation followed by the transform operation.

### 11.1.3 scan

```lua
<image> var = img.scan(<image> var)
<image> var = img.scan(<table> image_spec)
```

When you say `img.scan(a)` for a new image, the file is scanned, and variables such as xsize, ysize, image type, number of pages, and the resolution are extracted. Each of the width, height, depth fields are set up according to the image dimensions, if they were not given an explicit value already. An image file will never be scanned more than once for a given image variable. With all subsequent `img.scan(a)` calls only the dimensions are again set up (if they have been changed by the user in the meantime).

For ease of use, you can do right-away a

```lua
<image> a = img.scan { filename = "foo.png" }
```

without a prior `img.new`.

Nothing is written yet at this point, so you can do `a=img.scan`, retrieve the available info like image width and height, and then throw away a again by saying `a=nil`. In that case no image object will be reserved in the PDF, and the used memory will be cleaned up automatically.
11.1.4 copy

\[
\text{var} = \text{img.copy(var)}
\]
\[
\text{var} = \text{img.copy(table image_spec)}
\]

If you say \( a = b \), then both variables point to the same \(<image>\) object. If you want to write out an image with different sizes, you can do \( b = \text{img.copy(a)} \).

Afterwards, \( a \) and \( b \) still reference the same actual image dictionary, but the dimensions for \( b \) can now be changed from their initial values that were just copies from \( a \).

11.1.5 write, immediatewrite, immediatewriteobject

\[
\text{var} = \text{img.write(var)}
\]
\[
\text{var} = \text{img.write(table image_spec)}
\]

By \( \text{img.write(a)} \) a pdf object number is allocated, and a rule node of subtype image is generated and put into the output list. By this the image \( a \) is placed into the page stream, and the image file is written out into an image stream object after the shipping of the current page is finished.

Again you can do a terse call like

\[
\text{img.write \{ filename = "foo.png" \}}
\]

The \(<image>\) variable is returned in case you want it for later processing. You can also write an object.

By \( \text{img.immediatewrite(a)} \) a pdf object number is allocated, and the image file for image \( a \) is written out immediately into the pdf file as an image stream object (like with \immediate\pdfx-image). The object number of the image stream dictionary is then available by the objnum key. No pdf_refximage whatsit node is generated. You will need an \( \text{img.write(a)} \) or \( \text{img.node(a)} \) call to let the image appear on the page, or reference it by another trick; else you will have a dangling image object in the pdf file.

\[
\text{var} = \text{img.immediatewrite(var)}
\]
\[
\text{var} = \text{img.immediatewrite(table image_spec)}
\]

Also here you can do a terse call like

\[
\text{a} = \text{img.immediatewrite \{ filename = "foo.png" \}}
\]

The \(<image>\) variable is returned and you will most likely need it.

The next function is kind of special as it copies an object from a (pdf) image file. This features is experimental and might disappear.

\[
\text{objnum} = \text{img.immediatewriteobject(var, objnum)}
\]
\[
\text{objnum} = \text{img.immediatewriteobject(table image_spec, objnum)}
\]
11.1.6 node

\<node\> n = img.node(\<image\> var)\n\<node\> n = img.node(\<table\> image_spec)

This function allocates a pdf object number and returns a whatsis node of subtype pdf_refimage, filled with the image parameters width, height, depth, and objnum. Also here you can do a terse call like:

n = img.node(\{ filename = "foo.png" \})

This example outputs an image:

node.write(img.node(\{ filename="foo.png"\}))

11.1.7 types

\<table\> types = img.types()

This function returns a list with the supported image file type names, currently these are pdf, png, jpg, jp2 (JPEG 2000), and jbig2.

11.1.8 boxes

\<table\> boxes = img.boxes()

This function returns a list with the supported pdf page box names, currently these are media, crop, bleed, trim, and art, all in lowercase.

The pdf file is kept open after its properties are determined. After inclusion, which happens when the page that references the image is flushed, the file is closed. This means that when you have thousands of images on one page, your operating system might decide to abort the run. When you include more than one page from a pdf file you can set the keepopen flag when you allocate an image object, or pass the keepopen directive when you refer to the image with \useimageresource. This only makes sense when you embed many pages. An \immediate applied to \saveimageresource will also force a close after inclusion.

\immediate\useimageresource{foo.pdf} % closed
\useimageresource{foo.pdf} % kept open
\saveimageresource{foo.pdf} % kept open
\saveimageresource keepopen\lastsavedimageresourceindex % kept open

\directlua{img.write(img.scan{ file = "foo.pdf" })} % closed
\directlua{img.write(img.scan{ file = "foo.pdf", keepopen = true })} % kept open
11.2 The mplib library

The MetaPost library interface registers itself in the table mplib. It is based on mplib version 2.02.

11.2.1 new

To create a new MetaPost instance, call

```latex
<mpinstance> mp = mplib.new({...})
```

This creates the `mp` instance object. The argument hash can have a number of different fields, as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>DEFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_line</td>
<td>number</td>
<td>error line width</td>
<td>79</td>
</tr>
<tr>
<td>print_line</td>
<td>number</td>
<td>line length in ps output</td>
<td>100</td>
</tr>
<tr>
<td>random_seed</td>
<td>number</td>
<td>the initial random seed</td>
<td>variable</td>
</tr>
<tr>
<td>math_mode</td>
<td>string</td>
<td>the number system to use:</td>
<td>scaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled, double or decimal</td>
<td></td>
</tr>
<tr>
<td>interaction</td>
<td>string</td>
<td>the interaction mode:</td>
<td>errorstop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>batch, nonstop, scroll or errorstop</td>
<td></td>
</tr>
<tr>
<td>job_name</td>
<td>string</td>
<td>--jobname</td>
<td>mpout</td>
</tr>
<tr>
<td>find_file</td>
<td>function</td>
<td>a function to find files</td>
<td>only local files</td>
</tr>
</tbody>
</table>

The binary mode is no longer available in the Lua\TeX{} 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 off performance wise.

The `find_file` function should be of this form:

```latex
<string> found = finder (<string> name, <string> mode, <string> type)
```

with:

<table>
<thead>
<tr>
<th>NAME</th>
<th>THE REQUESTED FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>the file mode: r or w</td>
</tr>
<tr>
<td>type</td>
<td>the kind of file, one of: mp, tfm, map, pfb, enc</td>
</tr>
</tbody>
</table>

Return either the full path name of the found file, or `nil` if the file cannot be found.

Note that the new version of mplib no longer uses binary mem files, so the way to preload a set of macros is simply to start off with an input command in the first execute call.

When you are processing a snippet of text starting with `btex` and ending with either `etex` or `verbatimtex`, the MetaPost `texscriptmode` parameter controls how spaces and newlines get honoured. The default value is 1. Possible values are:

<table>
<thead>
<tr>
<th>NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no newlines</td>
</tr>
</tbody>
</table>
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.

### 11.2.2 statistics

You can request statistics with:

```plaintext
<table>
  stats = mp:statistics()
</table>
```

This function returns the vital statistics for an mplib instance. There are four fields, giving the maximum number of used items in each of four allocated object classes:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>main_memory</td>
<td>number</td>
<td>memory size</td>
</tr>
<tr>
<td>hash_size</td>
<td>number</td>
<td>hash size</td>
</tr>
<tr>
<td>param_size</td>
<td>number</td>
<td>simultaneous macro parameters</td>
</tr>
<tr>
<td>max_in_open</td>
<td>number</td>
<td>input file nesting levels</td>
</tr>
</tbody>
</table>

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.

### 11.2.3 execute

You can ask the MetaPost interpreter to run a chunk of code by calling

```plaintext
<table>
  retable = execute(mp,"metapost code")
</table>
```

for various bits of MetaPost language input. Be sure to check the `retable.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.

### 11.2.4 finish

```plaintext
<table>
  retable = finish(mp)
</table>
```
If for some reason you want to stop using an mplib instance while processing is not yet actually done, you can call `finish`. 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.

### 11.2.5 Result table

The return value of `execute` and `finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>string</td>
<td>output to the ‘log’ stream</td>
</tr>
<tr>
<td>term</td>
<td>string</td>
<td>output to the ‘term’ stream</td>
</tr>
<tr>
<td>error</td>
<td>string</td>
<td>output to the ‘error’ stream (only used for ‘out of memory’)</td>
</tr>
<tr>
<td>status</td>
<td>number</td>
<td>the return value: 0 = good, 1 = warning, 2 = errors, 3 = fatal error</td>
</tr>
<tr>
<td>fig</td>
<td>table</td>
<td>an array of generated figures (if any)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundingbox</td>
<td>function</td>
<td>returns the bounding box, as an array of 4 values</td>
</tr>
<tr>
<td>postscript</td>
<td>function</td>
<td>returns a string that is the ps output of the fig. this function accepts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two optional integer arguments for specifying the values of prologues</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(first argument) and procset (second argument)</td>
</tr>
<tr>
<td>svg</td>
<td>function</td>
<td>returns a string that is the svg output of the fig. This function accepts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>an optional integer argument for specifying the value of prologues</td>
</tr>
<tr>
<td>objects</td>
<td>function</td>
<td>returns the actual array of graphic objects in this fig</td>
</tr>
<tr>
<td>copy_objects</td>
<td>function</td>
<td>returns a deep copy of the array of graphic objects in this fig</td>
</tr>
<tr>
<td>filename</td>
<td>function</td>
<td>the filename this fig’s PostScript output would have written to in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stand alone mode</td>
</tr>
<tr>
<td>width</td>
<td>function</td>
<td>the fontcharwd value</td>
</tr>
<tr>
<td>height</td>
<td>function</td>
<td>the fontcharht value</td>
</tr>
<tr>
<td>depth</td>
<td>function</td>
<td>the fontchardp value</td>
</tr>
<tr>
<td>italcorr</td>
<td>function</td>
<td>the fontcharit value</td>
</tr>
<tr>
<td>charcode</td>
<td>function</td>
<td>the (rounded) charcode value</td>
</tr>
</tbody>
</table>

Note: you can call `fig:objects()` only once for any one fig object!

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 that each has a different list of accessible values. The types are: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `special`. 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 represented as a floating point number, unless stated otherwise. Field values that are of type table are explained in the next section.

### 11.2.5.1 fill

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>table</td>
<td>the list of knots</td>
</tr>
<tr>
<td>htap</td>
<td>table</td>
<td>the list of knots for the reversed trajectory</td>
</tr>
<tr>
<td>pen</td>
<td>table</td>
<td>knots of the pen</td>
</tr>
<tr>
<td>color</td>
<td>table</td>
<td>the object’s color</td>
</tr>
<tr>
<td>linejoin</td>
<td>number</td>
<td>line join style (bare number)</td>
</tr>
<tr>
<td>miterlimit</td>
<td>number</td>
<td>miterlimit</td>
</tr>
<tr>
<td>prescript</td>
<td>string</td>
<td>the prescript text</td>
</tr>
<tr>
<td>postscript</td>
<td>string</td>
<td>the postscript text</td>
</tr>
</tbody>
</table>

The entries htap and pen are optional.

### 11.2.5.2 outline

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>table</td>
<td>the list of knots</td>
</tr>
<tr>
<td>pen</td>
<td>table</td>
<td>knots of the pen</td>
</tr>
<tr>
<td>color</td>
<td>table</td>
<td>the object’s color</td>
</tr>
<tr>
<td>linejoin</td>
<td>number</td>
<td>line join style (bare number)</td>
</tr>
<tr>
<td>miterlimit</td>
<td>number</td>
<td>miterlimit</td>
</tr>
<tr>
<td>linelcap</td>
<td>number</td>
<td>line cap style (bare number)</td>
</tr>
<tr>
<td>dash</td>
<td>table</td>
<td>representation of a dash list</td>
</tr>
<tr>
<td>prescript</td>
<td>string</td>
<td>the prescript text</td>
</tr>
<tr>
<td>postscript</td>
<td>string</td>
<td>the postscript text</td>
</tr>
</tbody>
</table>

The entry dash is optional.

### 11.2.5.3 text

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>string</td>
<td>the text</td>
</tr>
<tr>
<td>font</td>
<td>string</td>
<td>font tfm name</td>
</tr>
<tr>
<td>dsize</td>
<td>number</td>
<td>font size</td>
</tr>
<tr>
<td>color</td>
<td>table</td>
<td>the object’s color</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td>table</td>
<td>a text transformation</td>
</tr>
</tbody>
</table>
11.2.5.4 special

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>prescript</td>
<td>string</td>
<td>special text</td>
</tr>
</tbody>
</table>

11.2.5.5 start_bounds, start_clip

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>table</td>
<td>the list of knots</td>
</tr>
</tbody>
</table>

11.2.5.6 stop_bounds, stop_clip

Here are no fields available.

11.2.6 Subsidiary table formats

11.2.6.1 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.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>left_type</td>
<td>string</td>
<td>when present: endpoint, but usually absent</td>
</tr>
<tr>
<td>right_type</td>
<td>string</td>
<td>like left_type</td>
</tr>
<tr>
<td>x_coord</td>
<td>number</td>
<td>X coordinate of this knot</td>
</tr>
<tr>
<td>y_coord</td>
<td>number</td>
<td>Y coordinate of this knot</td>
</tr>
<tr>
<td>left_x</td>
<td>number</td>
<td>X coordinate of the precontrol point of this knot</td>
</tr>
<tr>
<td>left_y</td>
<td>number</td>
<td>Y coordinate of the precontrol point of this knot</td>
</tr>
<tr>
<td>right_x</td>
<td>number</td>
<td>X coordinate of the postcontrol point of this knot</td>
</tr>
<tr>
<td>right_y</td>
<td>number</td>
<td>Y coordinate of the postcontrol point of this knot</td>
</tr>
</tbody>
</table>

There is one special case: pens that are (possibly transformed) ellipses have an extra string B A valued key type with value elliptical besides the array part containing the knot list.

11.2.6.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>marking only</td>
<td>no values</td>
</tr>
<tr>
<td>1</td>
<td>greyscale</td>
<td>one value in the range (0, 1), ‘black’ is 0</td>
</tr>
</tbody>
</table>
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.

### 11.2.6.3 Transforms

Each transform is a six-item array.

<table>
<thead>
<tr>
<th>INDEX</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number</td>
<td>represents x</td>
</tr>
<tr>
<td>2</td>
<td>number</td>
<td>represents y</td>
</tr>
<tr>
<td>3</td>
<td>number</td>
<td>represents xx</td>
</tr>
<tr>
<td>4</td>
<td>number</td>
<td>represents yy</td>
</tr>
<tr>
<td>5</td>
<td>number</td>
<td>represents xy</td>
</tr>
<tr>
<td>6</td>
<td>number</td>
<td>represents yy</td>
</tr>
</tbody>
</table>

Note that the translation (index 1 and 2) comes first. This differs from the ordering in PostScript, where the translation comes last.

### 11.2.6.4 Dashes

Each dash is two-item hash, using 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.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashes</td>
<td>hash</td>
<td>an array of on-off numbers</td>
</tr>
<tr>
<td>offset</td>
<td>number</td>
<td>the starting offset value</td>
</tr>
</tbody>
</table>

### 11.2.7 Pens and pen_info

There is helper function (pen_info(obj)) that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

<table>
<thead>
<tr>
<th>FIELD</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>number</td>
<td>width of the pen</td>
</tr>
<tr>
<td>sx</td>
<td>number</td>
<td>x scale</td>
</tr>
<tr>
<td>rx</td>
<td>number</td>
<td>x y multiplier</td>
</tr>
<tr>
<td>ry</td>
<td>number</td>
<td>y x multiplier</td>
</tr>
<tr>
<td>sy</td>
<td>number</td>
<td>y scale</td>
</tr>
<tr>
<td>tx</td>
<td>number</td>
<td>x offset</td>
</tr>
<tr>
<td>ty</td>
<td>number</td>
<td>y offset</td>
</tr>
</tbody>
</table>
11.2.8 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.

11.2.8.1 char_width

<number> w = char_width(mp,<string> fontname, <number> char)

11.2.8.2 char_height

<number> w = char_height(mp,<string> fontname, <number> char)

11.2.8.3 char_depth

<number> w = char_depth(mp,<string> fontname, <number> char)

11.2.8.4 get_[boolean|numeric|string|path]

When a script call brings you from the MetaPost run (temporarily) back to Lua you can access variables, but only if they are known (so for instance anonymous capsules like loop variables are not accessible).

<boolean> w = get_boolean(mp,<string> name)
<number> n = get_numeric(mp,<string> name)
<string> s = get_string (mp,<string> name)
<table> p = get_path  (mp,<string> name)

The path is returned as a table with subtables that have six numbers: the coordinates of the point, pre- and postcontrol. A cycle fields indicates if a path is cyclic.
12 The fontloader

The fontloader library is sort of independent of the rest in the sense that it can load font into a Lua table that then can be converted into a table suitable for \LaTeX. The library is an adapted subset of FontForge and as such gives a similar view on a font (which has advantages when you want to debug). We will not discuss OpenType in detail here as the Microsoft website offers enough information about it. The tables returned by the loader are not that far from the standard. We have no plans to extend the loader (it may even become an external module at some time).

12.1 Getting quick information on a font

When you want to locate font by name you need some basic information that is hidden in the font files. For that reason we provide an efficient helper that gets the basic information without loading all of the font. Normally this helper is used to create a font (name) database.

```lua
<table> info =
    fontloader.info(<string> filename)
</table>
```

This function returns either nil, or a table, or an array of small tables (in the case of a TrueType collection). The returned table(s) will contain some fairly interesting information items from the font(s) defined by the file:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>fontname</td>
<td>string</td>
<td>the PostScript name of the font</td>
</tr>
<tr>
<td>fullname</td>
<td>string</td>
<td>the formal name of the font</td>
</tr>
<tr>
<td>familyname</td>
<td>string</td>
<td>the family name this font belongs to</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>a string indicating the color value of the font</td>
</tr>
<tr>
<td>version</td>
<td>string</td>
<td>the internal font version</td>
</tr>
<tr>
<td>italicangle</td>
<td>float</td>
<td>the slant angle</td>
</tr>
<tr>
<td>units_per_em</td>
<td>number</td>
<td>1000 for PostScript-based fonts, usually 2048 for TrueType</td>
</tr>
<tr>
<td>pfminfo</td>
<td>table</td>
<td>(see section 12.6.6)</td>
</tr>
</tbody>
</table>

Getting information through this function is (sometimes much) more efficient than loading the font properly, and is therefore handy when you want to create a dictionary of available fonts based on a directory contents.

12.2 Loading an OPENTYPE or TRUETYPE file

If you want to use an OpenType font, you have to get the metric information from somewhere. Using the fontloader library, the simplest way to get that information is thus:

```lua
function load_font (filename)
    local metrics = nil
    local font = fontloader.open(filename)
    if font then
```
metrics = fontloader.to_table(font)
fontloader.close(font)
end
return metrics
end

myfont = load_font('/opt/tex/texmf/fonts/data/arial.ttf')

The main function call is

<userdata> f, <table> w = fontloader.open(<string> filename)
<userdata> f, <table> w = fontloader.open(<string> filename, <string> fontname)

The first return value is a userdata representation of the font. The second return value is a table containing any warnings and errors reported by fontloader while opening the font. In normal typesetting, you would probably ignore the second argument, but it can be useful for debugging purposes.

For TrueType collections (when filename ends in 'ttc') and dfont collections, you have to use a second string argument to specify which font you want from the collection. Use the fontname strings that are returned by fontloader.info for that.

To turn the font into a table, fontloader.to_table is used on the font returned by fontloader.open.

<table> f = fontloader.to_table(<userdata> font)

This table cannot be used directly by LuaTEX and should be turned into another one as described in chapter 6. Do not forget to store the fontname value in the psname field of the metrics table to be returned to LuaTEX, otherwise the font inclusion backend will not be able to find the correct font in the collection.

See section 12.5 for details on the userdata object returned by fontloader.open and the layout of the metrics table returned by fontloader.to_table.

The font file is parsed and partially interpreted by the font loading routines from FontForge. The file format can be OpenType, TrueType, TrueType Collection, cff, or Type1.

There are a few advantages to this approach compared to reading the actual font file ourselves:

- The font is automatically re-encoded, so that the metrics table for TrueType and OpenType fonts is using Unicode for the character indices.
- Many features are pre-processed into a format that is easier to handle than just the bare tables would be.
- PostScript-based OpenType fonts do not store the character height and depth in the font file, so the character bounding box has to be calculated in some way.

A loaded font is discarded with:

fontloader.close(<userdata> font)
12.3 Applying a ‘feature file’

You can apply a ‘feature file’ to a loaded font:

```
<table>
errors = fontloader.apply_featurefile(<userdata> font, <string> filename)
```

A ‘feature file’ is a textual representation of the features in an OpenType font. See
and
http://fontforge.sourceforge.net/featurefile.html
for a more detailed description of feature files.
If the function fails, the return value is a table containing any errors reported by fontloader while applying the feature file. On success, nil is returned.

12.4 Applying an ‘AFM file’

You can apply an ‘afm file’ to a loaded font:

```
<table>
errors = fontloader.apply_afmfile(<userdata> font, <string> filename)
```

An afm file is a textual representation of (some of) the meta information in a Type1 font. See
for more information about afm files.
Note: If you fontloader.open a Type1 file named font.pfb, the library will automatically search for and apply font.afm if it exists in the same directory as the file font.pfb. In that case, there is no need for an explicit call to apply_afmfile().
If the function fails, the return value is a table containing any errors reported by fontloader while applying the AFM file. On success, nil is returned.

12.5 Fontloader font tables

As mentioned earlier, the return value of fontloader.open is a userdata object. One way to have access to the actual metrics is to call fontloader.to_table on this object, returning the table structure that is explained in the following sections. In the following sections we will not explain each field in detail. Most fields are self descriptive and for the more technical aspects you need to consult the relevant font references.

It turns out that the result from fontloader.to_table sometimes needs very large amounts of memory (depending on the font’s complexity and size) so it is possible to access the userdata object directly.
• All top-level keys that would be returned by to_table() can also be accessed directly.
• The top-level key `glyphs` returns a virtual array that allows indices from \( f.glyph\text{min} \) to \( f.glyph\text{max} \).
• The items in that virtual array (the actual glyphs) are themselves also userdata objects, and each has accessors for all of the keys explained in the section ‘Glyph items’ below.
• The top-level key `subfonts` returns an actual array of userdata objects, one for each of the subfonts (or nil, if there are no subfonts).

A short example may be helpful. This code generates a printout of all the glyph names in the font PunkNova.kern.otf:

```lua
local f = fontloader.open('PunkNova.kern.otf')
print (f.fontname)
local i = 0
if f.glyphcnt > 0 then
    for i=f.glyphmin,f.glyphmax do
        local g = f.glyphs[i]
        if g then
            print(g.name)
        end
        i = i + 1
    end
end
fontloader.close(f)
```

In this case, the LuaTeX memory requirement stays below 100MB on the test computer, while the internal structure generated by to_table() needs more than 2GB of memory (the font itself is 6.9MB in disk size).

Only the top-level font, the subfont table entries, and the glyphs are virtual objects, everything else still produces normal Lua values and tables.

If you want to know the valid fields in a font or glyph structure, call the fields function on an object of a particular type (either glyph or font):

```lua
<table> fields = fontloader.fields(<userdata> font)
<table> fields = fontloader.fields(<userdata> font_glyph)
```

For instance:

```lua
local fields = fontloader.fields(f)
local fields = fontloader.fields(f.glyphs[0])
```

### 12.6 Table types

#### 12.6.1 The main table

The top-level keys in the returned table are (the explanations in this part of the documentation are not yet finished):
<table>
<thead>
<tr>
<th><strong>KEY</strong></th>
<th><strong>TYPE</strong></th>
<th><strong>explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>table_version</td>
<td>string</td>
<td>indicates the metrics version (currently 0.3)</td>
</tr>
<tr>
<td>fontname</td>
<td>string</td>
<td>PostScript font name</td>
</tr>
<tr>
<td>fullname</td>
<td>string</td>
<td>official (human-oriented) font name</td>
</tr>
<tr>
<td>familyname</td>
<td>string</td>
<td>family name</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>weight indicator</td>
</tr>
<tr>
<td>copyright</td>
<td>string</td>
<td>copyright information</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>the file name</td>
</tr>
<tr>
<td>version</td>
<td>string</td>
<td>font version</td>
</tr>
<tr>
<td>italicangle</td>
<td>float</td>
<td>slant angle</td>
</tr>
<tr>
<td>units_per_em</td>
<td>number</td>
<td>1000 for PostScript-based fonts, usually 2048 for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TrueType</td>
</tr>
<tr>
<td>ascent</td>
<td>number</td>
<td>height of ascender in units_per_em</td>
</tr>
<tr>
<td>descent</td>
<td>number</td>
<td>depth of descender in units_per_em</td>
</tr>
<tr>
<td>upos</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>uwidth</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>uniqueid</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>glyphs</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>glyphcnt</td>
<td>number</td>
<td>number of included glyphs</td>
</tr>
<tr>
<td>glyphmax</td>
<td>number</td>
<td>maximum used index the glyphs array</td>
</tr>
<tr>
<td>glyphmin</td>
<td>number</td>
<td>minimum used index the glyphs array</td>
</tr>
<tr>
<td>notdef_loc</td>
<td>number</td>
<td>location of the .notdef glyph or -1 when not present</td>
</tr>
<tr>
<td>hasvmetrics</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>onlybitmaps</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>serifcheck</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>isserif</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>issans</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>encodingchanged</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>strokedfont</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>use_typo_metrics</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>weight_width_slope_only</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>head_optimized_for_cleartype</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>uni_interp</td>
<td>enum</td>
<td>unset, none, adobe, greek, japanese, trad_chinese, simp_chinese, korean, ams</td>
</tr>
<tr>
<td>origname</td>
<td>string</td>
<td>the file name, as supplied by the user</td>
</tr>
<tr>
<td>map</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>xuid</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>pfinfo</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>names</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>cidinfo</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>subfonts</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>comments</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>fontlog</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cvt_names</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>anchor_classes</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>
ttf_tables table
ttf_tab_saved table
kerns table
vkerns table
texdata table
lookups table
gpos table
gsub table
mm table
chosenname string
macstyle number
tfonname string
fontstyle_id number
fontstyle_name table
strokewidth float
mark_classes table
creationtime number
modificationtime number
os2_version number
math table
validation_state table
horiz_base table
vert_base table
extrema_bound number
true_type boolean signals a TrueType font

12.6.2 glyphs

The glyphs is an array containing the per-character information (quite a few of these are only present if non-zero).

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>the glyph name</td>
</tr>
<tr>
<td>unicode</td>
<td>number</td>
<td>unicode code point, or -1</td>
</tr>
<tr>
<td>boundingbox</td>
<td>array</td>
<td>array of four numbers, see note below</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>only for horizontal fonts</td>
</tr>
<tr>
<td>vwidth</td>
<td>number</td>
<td>only for vertical fonts</td>
</tr>
<tr>
<td>tsidebearing</td>
<td>number</td>
<td>only for vertical ttf/otf fonts, and only if non-zero</td>
</tr>
<tr>
<td>lsidebearing</td>
<td>number</td>
<td>only if non-zero and not equal to boundingbox[1]</td>
</tr>
<tr>
<td>class</td>
<td>string</td>
<td>one of &quot;none&quot;, &quot;base&quot;, &quot;ligature&quot;, &quot;mark&quot;, &quot;component&quot; (if not present, the glyph class is ‘automatic’)</td>
</tr>
<tr>
<td>kerns</td>
<td>array</td>
<td>only for horizontal fonts, if set</td>
</tr>
<tr>
<td>vkerns</td>
<td>array</td>
<td>only for vertical fonts, if set</td>
</tr>
<tr>
<td>dependents</td>
<td>array</td>
<td>linear array of glyph name strings, only if nonempty</td>
</tr>
<tr>
<td>lookups</td>
<td>table</td>
<td>only if nonempty</td>
</tr>
<tr>
<td>ligatures</td>
<td>table</td>
<td>only if nonempty</td>
</tr>
</tbody>
</table>
On boundingbox: The boundingbox information for TrueType fonts and TrueType-based otf fonts is read directly from the font file. PostScript-based fonts do not have this information, so the boundingbox of traditional PostScript fonts is generated by interpreting the actual bezier curves to find the exact boundingbox. This can be a slow process, so the boundingboxes of PostScript-based otf fonts (and raw cff fonts) are calculated using an approximation of the glyph shape based on the actual glyph points only, instead of taking the whole curve into account. This means that glyphs that have missing points at extrema will have a too-tight boundingbox, but the processing is so much faster that in our opinion the tradeoff is worth it.

The kerns and vkerns are linear arrays of small hashes:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>off</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lookup</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

The lookups is a hash, based on lookup subtable names, with the value of each key inside that a linear array of small hashes:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>enum</td>
<td>position, pair, substitution, alternate, multiple, ligature, lcaret, kerning, vkerning, anchors, contextpos, contextsub, chainpos, chain-sub, reversesub, max, kernback, vkernback</td>
</tr>
<tr>
<td>specification</td>
<td>table</td>
<td>extra data</td>
</tr>
</tbody>
</table>

For the first seven values of type, there can be additional sub-information, stored in the sub-table specification:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>table</td>
<td>a table of the offset_specs type</td>
</tr>
<tr>
<td>pair</td>
<td>table</td>
<td>one string: paired, and an array of one or two offset_specs tables: offsets</td>
</tr>
<tr>
<td>substitution</td>
<td>table</td>
<td>one string: variant</td>
</tr>
<tr>
<td>alternate</td>
<td>table</td>
<td>one string: components</td>
</tr>
<tr>
<td>multiple</td>
<td>table</td>
<td>one string: components</td>
</tr>
</tbody>
</table>
The ligature table two strings: components, char
caret array linear array of numbers

Tables for offset_specs contain up to four number-valued fields: x (a horizontal offset), y (a vertical offset), h (an advance width correction) and v (an advance height correction).

The ligatures is a linear array of small hashes:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>lig</td>
<td>table</td>
<td>uses the same substructure as a single item in the lookups table explained above</td>
</tr>
<tr>
<td>char</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>components</td>
<td>array</td>
<td>linear array of named components</td>
</tr>
<tr>
<td>ccnt</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

The anchor table is indexed by a string signifying the anchor type, which is one of:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>table</td>
<td>placement mark</td>
</tr>
<tr>
<td>basechar</td>
<td>table</td>
<td>mark for attaching combining items to a base char</td>
</tr>
<tr>
<td>baselig</td>
<td>table</td>
<td>mark for attaching combining items to a ligature</td>
</tr>
<tr>
<td>basemark</td>
<td>table</td>
<td>generic mark for attaching combining items to connect to</td>
</tr>
<tr>
<td>centry</td>
<td>table</td>
<td>cursive entry point</td>
</tr>
<tr>
<td>cexit</td>
<td>table</td>
<td>cursive exit point</td>
</tr>
</tbody>
</table>

The content of these is a short array of defined anchors, with the entry keys being the anchor names. For all except baselig, the value is a single table with this definition:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>number</td>
<td>x location</td>
</tr>
<tr>
<td>y</td>
<td>number</td>
<td>y location</td>
</tr>
<tr>
<td>ttf_pt_index</td>
<td>number</td>
<td>true type point index, only if given</td>
</tr>
</tbody>
</table>

For baselig, the value is a small array of such anchor sets sets, one for each constituent item of the ligature.

For clarification, an anchor table could for example look like this:

```python
[‘anchor’] = {
    [‘basemark’] = {
        [‘Anchor-7’] = { [‘x’]=170, [‘y’]=1080 }  
    },
    [‘mark’] ={
        [‘Anchor-1’] = { [‘x’]=160, [‘y’]=810 },
        [‘Anchor-4’] = { [‘x’]=160, [‘y’]=800 }  
    },
    [‘baselig’] = {
        [1] = { [‘Anchor-2’] = { [‘x’]=160, [‘y’]=650 } },
        [2] = { [‘Anchor-2’] = { [‘x’]=460, [‘y’]=640 } }
}"
```
Note: The baselig table can be sparse!

### 12.6.3 map

The top-level map is a list of encoding mappings. Each of those is a table itself.

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>enccount</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>encmax</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>backmax</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>remap</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>array</td>
<td>non-linear array of mappings</td>
</tr>
<tr>
<td>backmap</td>
<td>array</td>
<td>non-linear array of backward mappings</td>
</tr>
<tr>
<td>enc</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The remap table is very small:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstenc</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastenc</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>infont</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

The enc table is a bit more verbose:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>enc_name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>char_cnt</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>char_max</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>unicode</td>
<td>array</td>
<td>of Unicode position numbers</td>
</tr>
<tr>
<td>psnames</td>
<td>array</td>
<td>of PostScript glyph names</td>
</tr>
<tr>
<td>builtin</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hidden</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>only_1byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>has_1byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>has_2byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>is_unicodebmp</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_unicodefull</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_custom</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_original</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_compact</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_japanese</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_korean</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
<tr>
<td>is_tradchinese</td>
<td>number</td>
<td>only if non-zero [name?]</td>
</tr>
<tr>
<td>is_simplechinese</td>
<td>number</td>
<td>only if non-zero</td>
</tr>
</tbody>
</table>
12.6.4 private

This is the font’s private PostScript dictionary, if any. Keys and values are both strings.

12.6.5 cidinfo

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>registry</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>ordering</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>supplement</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

12.6.6 pfminfo

The pfminfo table contains most of the OS/2 information:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfmset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>winascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>windescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheadascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheaddescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>typoascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>typodescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>subsuper_set</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>panose_set</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheadset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>vheadset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pfmfamily</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>avgwidth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>firstchar</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastchar</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>fstype</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>linegap</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>vlinegap</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hhead_ascent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hhead_descent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_typoascent</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>
The panose subtable has exactly 10 string keys:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>familytype</td>
<td>string</td>
<td>Values as in the OpenType font specification: Any, No Fit, Text and Display, Script, Decorative, Pictorial</td>
</tr>
<tr>
<td>serifstyle</td>
<td>string</td>
<td>See the OpenType font specification for values</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>proportion</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>contrast</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>strokevariation</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>armstyle</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>letterform</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>midline</td>
<td>string</td>
<td>idem</td>
</tr>
<tr>
<td>xheight</td>
<td>string</td>
<td>idem</td>
</tr>
</tbody>
</table>

### 12.6.7 names

Each item has two top-level keys:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>lang</td>
<td>string</td>
<td>language for this entry</td>
</tr>
<tr>
<td>names</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>
The names keys are the actual TrueType name strings. The possible keys are: copyright, family, subfamily, uniqueid, fullname, version, postscriptname, trademark, manufacturer, designer, descriptor, venderurl, designerurl, license, licenseurl, idontknow, preffamily-name, preffamily-name, prefmodules, compatfull, sampletext, cidfindfontname, wwsfamily and wwssubfamily.

### 12.6.8 anchor_classes

The anchor_classes classes:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>a descriptive id of this anchor class</td>
</tr>
<tr>
<td>lookup</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>one of mark, mkmk, curs, mklg</td>
</tr>
</tbody>
</table>

### 12.6.9 gpos

The gpos table has one array entry for each lookup. (The gpos_ prefix is somewhat redundant.)

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>one of gpos_single, gpos_pair, gpos_cursive, gpos_mark2base,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gpos_mark2ligature, gpos_mark2mark, gpos_context,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gpos_contextchain</td>
</tr>
<tr>
<td>flags</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>features</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>subtables</td>
<td>array</td>
<td></td>
</tr>
</tbody>
</table>

The flags table has a true value for each of the lookup flags that is actually set:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2l</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignorebaseglyphs</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignoreligatures</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignorecombiningmarks</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>mark_class</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

The features subtable items of gpos have:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>scripts</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The scripts table within features has:
### The Fontloader

#### 12.6.10 gsub

This has identical layout to the gpos table, except for the type:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>one of gsub_single, gsub_multiple, gsub_alternate, gsub_ligature,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gsub_context, gsub_contextchain, gsub_reversecontextchain</td>
</tr>
</tbody>
</table>

#### 12.6.11 ttf_tables and ttf_tab_saved

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>len</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>maxlen</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

#### 12.6.12 mm

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>table</td>
<td>array of axis names</td>
</tr>
</tbody>
</table>
instance_count  number
positions    table   array of instance positions (#axes * instances)
defweights   table   array of default weights for instances
cdv          string
ndv          string
axismaps     table

The axismaps:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>blends</td>
<td>table</td>
<td>an array of blend points</td>
</tr>
<tr>
<td>designs</td>
<td>table</td>
<td>an array of design values</td>
</tr>
<tr>
<td>min</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

12.6.13 mark_classes

The keys in this table are mark class names, and the values are a space-separated string of glyph names in this class.

12.6.14 math

The math table has the variables that are also discussed in the chapter about math: Script-PercentScaleDown, ScriptScriptPercentScaleDown, DelimitedSubFormulaMinHeight, DisplayOperatorMinHeight, MathLeading, AxisHeight, AccentBaseHeight, FlattenedAccentBaseHeight, SubscriptShiftDown, SubscriptTopMax, SubscriptBaselineDropMin, SuperscriptShiftUp, SuperscriptShiftUpCramped, SuperscriptBottomMin, SuperscriptBaselineDropMax, SubSuperscriptGapMin, SuperscriptBottomMaxWithSubscript, SpaceAfterScript, UpperLimitGapMin, UpperLimitBaselineRiseMin, LowerLimitGapMin, LowerLimitBaselineDropMin, StackTopShiftUp, StackTopDisplayStyleShiftUp, StackBottomShiftDown, StackDisplayStyleGapMin, StretchStackTopShiftUp, StretchStackBottomShiftDown, StretchStackGapAboveMin, StretchStackGapBelowMin, FractionNumeratorDisplayStyleShiftUp, FractionDenominatorDisplayStyleShiftDown, FractionNumeratorDisplayStyleGapMin, FractionDenominatorDisplayStyleGapMax, FractionRuleThickness, FractionNumeratorGapMin, FractionDenominatorDisplayStyleGapMin, SkewedFractionHorizontalGap, SkewedFractionVerticalGap, OverbarVerticalGap, OverbarRuleThickness, OverbarExtraAscender, UnderbarVerticalGap, UnderbarRuleThickness, UnderbarExtraDescender, RadicalVerticalGap, RadicalDisplayStyleVerticalGap, RadicalRuleThickness, RadicalExtraAscender, RadicalKernBeforeDegree, RadicalKernAfterDegree, RadicalDegreeBottomRaisePercent, MinConnectorOverlap, FractionDelimiterSize and FractionDelimiterDisplayStyleSize.
12.6.15 validation_state

This is just a bonus table with keys: bad_ps_fontname, bad_glyph_table, bad_cff_table, bad_metrics_table, bad_cmap_table, bad_bitmaps_table, bad_gx_table, bad_ot_table, bad_os2_version and bad_sfnt_header.

12.6.16 horiz_base and vert_base

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>tags</td>
<td>table</td>
<td>an array of script list tags</td>
</tr>
<tr>
<td>scripts</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The scripts subtable:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>default_baseline</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lang</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The lang subtable:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>string</td>
<td>a script tag</td>
</tr>
<tr>
<td>ascent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>descent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>features</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The features points to an array of tables with the same layout except that in those nested tables, the tag represents a language.

12.6.17 altuni

An array of alternate Unicode values. Inside that array are hashes with:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicode</td>
<td>number</td>
<td>this glyph is also used for this unicode</td>
</tr>
<tr>
<td>variant</td>
<td>number</td>
<td>the alternative is driven by this unicode selector</td>
</tr>
</tbody>
</table>

12.6.18 vert_variants and horiz_variants

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>variants</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>italic_correction</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>parts</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The parts table is an array of smaller tables:
### KEY TYPE EXPLANATION

| component | string  |  
| extender   | number  |  
| start      | number  |  
| end        | number  |  
| advance    | number  |  

#### 12.6.19 mathkern

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_right</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>bottom_right</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>top_left</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>bottom_left</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

Each of the subtables is an array of small hashes with two keys:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>kern</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

#### 12.6.20 kerns

Substructure is identical to the per-glyph subtable.

#### 12.6.21 vkerns

Substructure is identical to the per-glyph subtable.

#### 12.6.22 texdata

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>unset, text, math, mathext</td>
</tr>
<tr>
<td>params</td>
<td>array</td>
<td>22 font numeric parameters</td>
</tr>
</tbody>
</table>

#### 12.6.23 lookups

Top-level lookups is quite different from the ones at character level. The keys in this hash are strings, the values the actual lookups, represented as dictionary tables.

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>enum</td>
<td>one of glyphs, class, coverage, reversecoverage</td>
</tr>
</tbody>
</table>
tag  string
current_class  array
before_class  array
after_class  array
rules  array  an array of rule items

Rule items have one common item and one specialized item:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookups</td>
<td>array</td>
<td>a linear array of lookup names</td>
</tr>
<tr>
<td>glyphs</td>
<td>array</td>
<td>only if the parent’s format is glyphs</td>
</tr>
<tr>
<td>class</td>
<td>array</td>
<td>only if the parent’s format is class</td>
</tr>
<tr>
<td>coverage</td>
<td>array</td>
<td>only if the parent’s format is coverage</td>
</tr>
<tr>
<td>reversecoverage</td>
<td>array</td>
<td>only if the parent’s format is reversecoverage</td>
</tr>
</tbody>
</table>

A glyph table is:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>fore</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

A class table is:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>array</td>
<td>of numbers</td>
</tr>
<tr>
<td>before</td>
<td>array</td>
<td>of numbers</td>
</tr>
<tr>
<td>after</td>
<td>array</td>
<td>of numbers</td>
</tr>
</tbody>
</table>

for coverage:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>before</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>after</td>
<td>array</td>
<td>of strings</td>
</tr>
</tbody>
</table>

and for reverse coverage:

<table>
<thead>
<tr>
<th>KEY</th>
<th>TYPE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>before</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>after</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>replacements</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
13 The HarfBuzz libraries

13.1 The luaharfbuzz library

At the moment the documentation of the library is a raw "html-to-pdf" rendering of the index.html file under luaharfbuzz/docs folder of the source code. The example is almost the verbatim copy of the file under luaharfbuzz/examples.
**Module harfbuzz**

Lua bindings to Harfbuzz.

**Info:**
- **Copyright:** 2016
- **License:** MIT
- **Author:** Deepak Jois <deepak.jois@gmail.com>

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version ()</td>
<td>Wraps hb_version</td>
</tr>
<tr>
<td>shape (font, buffer[, options])</td>
<td>Wraps hb_shape.</td>
</tr>
</tbody>
</table>

**Class Blob**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob.new (data)</td>
<td>Wraps hb_blob_create.</td>
</tr>
<tr>
<td>Blob.new_from_file (filename)</td>
<td>Wraps hb_blob_create_from_file.</td>
</tr>
<tr>
<td>Blob:get_length ()</td>
<td>Wraps hb_blob_get_length.</td>
</tr>
<tr>
<td>Blob:get_data ()</td>
<td>Wraps hb_blob_get_data.</td>
</tr>
</tbody>
</table>

**Class Face**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face.new_from_blob (blob[, font_index=0])</td>
<td>Wraps hb_face_create.</td>
</tr>
<tr>
<td>Face.new (file[, font_index=0])</td>
<td>Create a new Face from a file.</td>
</tr>
<tr>
<td>Face:collect_unicodes ()</td>
<td>Wraps hb_face_collect unicodes.</td>
</tr>
<tr>
<td>Face:get_glyph_count ()</td>
<td>Wraps hb_face_getGlyph_count.</td>
</tr>
<tr>
<td>Face:get_table (tag)</td>
<td>Wraps hb_face_reference_table.</td>
</tr>
<tr>
<td>Face:get_table_tags ()</td>
<td>Wraps hb_face_getTable_tags.</td>
</tr>
<tr>
<td>Face:get_upem ()</td>
<td>Wraps hb_face_getUpem.</td>
</tr>
<tr>
<td>Face:ot_color_has_palettes ()</td>
<td>Wraps hb_ot_color_hasPalettes.</td>
</tr>
<tr>
<td>Face:ot_color_palette_get_count ()</td>
<td>Wraps hb_ot_color_palette_get_count.</td>
</tr>
<tr>
<td>Face:ot_color_palette_get_colors ()</td>
<td>Wraps hb_ot_color_palette_get_colors.</td>
</tr>
<tr>
<td>Face:ot_color_has_layers ()</td>
<td>Wraps hb_ot_color_has_layers.</td>
</tr>
<tr>
<td>Face:ot_color_glyph_get_layers ()</td>
<td>Wraps hb_ot_color_glyph_get_layers.</td>
</tr>
<tr>
<td>Face:ot_color_has_png ()</td>
<td>Wraps hb_ot_color_has_png.</td>
</tr>
<tr>
<td>Face:ot_layout_get_script_tags ()</td>
<td>Wraps hb_ot_layout_table_get_script_tags.</td>
</tr>
<tr>
<td>Face:ot_layout_get_language_tags ()</td>
<td>Wraps hb_ot_layout_table_get_language_tags.</td>
</tr>
<tr>
<td>Face:ot_layout_get_feature_tags ()</td>
<td>Wraps hb_ot_layout_table_get_feature_tags.</td>
</tr>
<tr>
<td>Face:ot_layout_find_script ()</td>
<td>Wraps hb_ot_layout_table_find_script.</td>
</tr>
<tr>
<td>Face:ot_layout_find_language ()</td>
<td>Wraps hb_ot_layout_table_find_language.</td>
</tr>
<tr>
<td>Face:ot_layout_find_feature ()</td>
<td>Wraps hb_ot_layout_table_find_feature.</td>
</tr>
</tbody>
</table>

**Class Font**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font.new (face)</td>
<td>Wraps hb_font_create, and sets up some defaults for scale and shaping functions.</td>
</tr>
<tr>
<td>Font:get_scale ()</td>
<td>Wraps hb_font_get_scale.</td>
</tr>
<tr>
<td>Font:set_scale (x_scale, y_scale)</td>
<td>Wraps hb_font_set_scale.</td>
</tr>
<tr>
<td>Font:get_h_extents ()</td>
<td>Wraps hb_font_get_h_extents.</td>
</tr>
<tr>
<td>Font:get_v_extents ()</td>
<td>Wraps hb_font_get_v_extents.</td>
</tr>
<tr>
<td>Font:get_glyph_extents (glyph)</td>
<td>Wraps hb_font_get_glyph_extents.</td>
</tr>
</tbody>
</table>
### Font

- **Font:get_glyph_name (glyph)** Wraps `hb_font_get_glyph_name`
- **Font:get_glyph_from_name (name)** Wraps `hb_font_get_glyph_from_name`
- **Font:get_glyph_h_advance (glyph)** Wraps `hb_font_get_glyph_h_advance`
- **Font:get_glyph_v_advance (glyph)** Wraps `hb_font_get_glyph_v_advance`
- **Font:get_nominal_glyph (codepoint)** Wraps `hb_font_get_nominal_glyph`
- **Font:ot_color_glyph_get_png ()** Wraps `hb_ot_color_glyph_get_png`

### Class Buffer

- **Buffer.new ()** Wraps `hb_buffer_create`
- **Buffer:add_utf8 (text[, item_offset=0[, item_length=-1]])** Wraps `hb_buffer_add_utf8`
- **Buffer:add_codepoints (text[, item_offset=0[, item_length=-1]])** Wraps `hb_buffer_add_codepoints`
- **Buffer:set_direction (dir)** Wraps `hb_buffer_set_direction`
- **Buffer:get_direction ()** Wraps `hb_buffer_get_direction`
- **Buffer:set_script (script)** Wraps `hb_buffer_set_script`
- **Buffer:get_script ()** Wraps `hb_buffer_get_script`
- **Buffer:set_language (lang)** Wraps `hb_buffer_set_language`
- **Buffer:get_language ()** Wraps `hb_buffer_get_language`
- **Buffer:reverse ()** Wraps `hb_buffer_reverse`
- **Buffer:get_length ()** Wraps `hb_buffer_get_length`
- **Buffer:get_cluster_level ()** Wraps `hb_buffer_get_cluster_level`
- **Buffer:set_cluster_level (level)** Wraps `hb_buffer_set_cluster_level`
- **Buffer:guess_segment_properties ()** Wraps `hb_buffer_guess_segment_properties`
- **Buffer:get_glyphs ()** Helper method to get shaped glyph data.

### Cluster Levels

- **Buffer.CLUSTER_LEVEL_MONOTONE_GRAPHEMES** Wraps `HB_BUFFER_CLUSTER_LEVEL_MONOTONE_GRAPHEMES`
- **Buffer.CLUSTER_LEVEL_MONOTONE_CHARACTERS** Wraps `HB_BUFFER_CLUSTER_LEVEL_MONOTONE_CHARACTERS`
- **Buffer.CLUSTER_LEVEL_CHARACTERS** Wraps `HB_BUFFER_CLUSTER_LEVEL_CHARACTERS`
- **Buffer.CLUSTER_LEVEL_DEFAULT** Wraps `HB_BUFFER_CLUSTER_LEVEL_DEFAULT`
- **Buffer.GLYPH_FLAG_UNSAFE_TO_BREAK** Wraps `HB_GLYPH_FLAG_UNSAFE_TO_BREAK`
- **Buffer.GLYPH_FLAG_DEFINED** Wraps `HB_GLYPH_FLAG_DEFINED`

### Class Feature

- **Feature.new (feature_string)** Wraps `hb_feature_from_string`
- **Feature:__tostring ()** Wraps `hb_feature_to_string`

### Class Tag

- **Tag.new (string)** Wraps `hb_tag_from_string`
- **Tag:__to_string ()** Wraps `hb_tag_to_string`
- **Tag:__eq ()** Enables equality comparisons with `==` between two tags.

### Class Script

- **Script.new (script)** Wraps `hb_script_from_string`
- **Script.from_iso15924_tag (tag)** Wraps `hb_script_from_iso15924_tag`
### Predefined Script Codes

<table>
<thead>
<tr>
<th>Script</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script.COMMON</td>
<td>HB_SCRIPT_COMMON</td>
</tr>
<tr>
<td>Script.INHERITED</td>
<td>HB_SCRIPT_INHERITED</td>
</tr>
<tr>
<td>Script.UNKNOWN</td>
<td>HB_SCRIPT_UNKNOWN</td>
</tr>
<tr>
<td>Script.INVALID</td>
<td>HB_SCRIPT_INVALID</td>
</tr>
</tbody>
</table>

### Class Direction

<table>
<thead>
<tr>
<th>Method</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction.new (dir)</td>
<td>hb_direction_from_string</td>
</tr>
<tr>
<td>Direction.__to_string</td>
<td>hb_direction_to_string</td>
</tr>
<tr>
<td>Direction.__eq</td>
<td>Enables equality comparisons with == between two directions.</td>
</tr>
<tr>
<td>Direction:is_valid</td>
<td>HB_DIRECTION_IS_VALID</td>
</tr>
<tr>
<td>Direction:is_horizontal</td>
<td>HB_DIRECTION_IS_HORIZONTAL</td>
</tr>
<tr>
<td>Direction:is_vertical</td>
<td>HB_DIRECTION_IS_VERTICAL</td>
</tr>
<tr>
<td>Direction:is_forward</td>
<td>HB_DIRECTION_IS_FORWARD</td>
</tr>
<tr>
<td>Direction:is_backward</td>
<td>HB_DIRECTION_IS_BACKWARD</td>
</tr>
</tbody>
</table>

### Predefined directions

<table>
<thead>
<tr>
<th>Direction</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction.LTR</td>
<td>HB_DIRECTION_LTR</td>
</tr>
<tr>
<td>Direction.RTL</td>
<td>HB_DIRECTION_RTL</td>
</tr>
<tr>
<td>Direction.TTB</td>
<td>HB_DIRECTION_TTB</td>
</tr>
<tr>
<td>Direction.BTT</td>
<td>HB_DIRECTION_LTR</td>
</tr>
</tbody>
</table>

### Class Language

<table>
<thead>
<tr>
<th>Method</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language.new (lang)</td>
<td>hb_language_from_string</td>
</tr>
<tr>
<td>Language.__to_string</td>
<td>hb_language_to_string</td>
</tr>
<tr>
<td>Language.__eq</td>
<td>Enables equality comparisons with == between two languages.</td>
</tr>
</tbody>
</table>

### Predefined languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language.INVALID</td>
<td>HB_LANGUAGE_INVALID</td>
</tr>
</tbody>
</table>

### Unicode functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicode.script (char)</td>
<td>hb_unicode_script</td>
</tr>
</tbody>
</table>

### Predefined Name IDs

<table>
<thead>
<tr>
<th>Name ID</th>
<th>Wraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>ot.NAME_ID_COPYRIGHT</td>
<td>HB_OT_NAME_ID_COPYRIGHT</td>
</tr>
<tr>
<td>ot.NAME_ID_FONT_FAMILY</td>
<td>HB_OT_NAME_ID_FONT_FAMILY</td>
</tr>
<tr>
<td>ot.NAME_ID_FONT_SUBFAMILY</td>
<td>HB_OT_NAME_ID_FONT_SUBFAMILY</td>
</tr>
<tr>
<td>ot.NAME_ID_UNIQUE_ID</td>
<td>HB_OT_NAME_ID_UNIQUE_ID</td>
</tr>
<tr>
<td>ot.NAME_ID_FULL_NAME</td>
<td>HB_OT_NAME_ID_FULL_NAME</td>
</tr>
<tr>
<td>ot.NAME_ID_VERSION_STRING</td>
<td>HB_OT_NAME_ID_VERSION_STRING</td>
</tr>
<tr>
<td>ot.NAME_ID_POSTSCRIPT_NAME</td>
<td>HB_OT_NAME_ID_POSTSCRIPT_NAME</td>
</tr>
<tr>
<td>ot.NAME_ID_TRADEMARK</td>
<td>HB_OT_NAME_ID_TRADEMARK</td>
</tr>
<tr>
<td>ot.NAME_ID_MANUFACTURER</td>
<td>HB_OT_NAME_ID_MANUFACTURER</td>
</tr>
</tbody>
</table>
Functions

version ()

Wraps hb_version

shape (font, buffer[, options])

Wraps hb_shape.

Parameters:
- font: Font to use for shaping
- buffer: Buffer to shape
- options:
  table containing one or more supported options:
  - direction: A Direction object representing the object.
  - script: A Script object representing the script.
  - language: A Language object representing the language.
  - features: features to enable, specified as either of the following.
    - comma-separated list of features. See feature string syntax reference
    - table of Feature objects
      (optional)

Class Blob

Lua wrapper for hb_blob_t type

Blob.new (data)

Wraps hb_blob_create. Initializes a new hb_blob_t.
Parameters:
- data: lua string containing binary or character data.

Blob.new_from_file (filename)
Wraps hb_blob_create_from_file. Initializes a new hb_blob_t.

Parameters:
- filename: lua string.

Blob:get_length ()
Wraps hb_blob_get_length.

Blob:get_data ()
Wraps hb_blob_get_data.

Class Face
Lua wrapper for hb_face_t type

Face.new_from_blob (blob[, font_index=0])
Wraps hb_face_create. Initializes a new hb_face_t from a Blob object.

Parameters:
- blob: Blob to read the font from.
- font_index: index of font to read. (default 0)

Face.new (file[, font_index=0])
Create a new Face from a file. Makes a call to Face:new_from_blob after creating a Blob from the file contents.

Parameters:
- file: path to font file.
- font_index: index of font to read. (default 0)

Face:collectunicodes ()
Wraps hb_face_collectunicodes.

Returns:
table of codepoints supported by the face.

Face:getglyphcount ()
Wraps hb_face_get_glyph_count.

Face:get_table (tag)
Wraps hb_face_reference_table.

Parameters:
- tag: Tag object of the table.

Returns:
Blob object for the face table of tag.

260 The HarfBuzz libraries
Face:get_table_tags ()
Wraps hb_face_get_table_tags.
Returns:
table of Tags representing face table tags.

Face:get_upem ()
Wraps hb_face_get_upem.

Face:ot_color_has_palettes ()
Wraps hb_ot_color_has_pallettes.

Face:ot_color_palette_get_count ()
Wraps hb_ot_color_palette_get_count.

Face:ot_color_palette_get_colors ()
Wraps hb_ot_color_palette_get_colors.

Face:ot_color_has_layers ()
Wraps hb_ot_color_has_layers.

Face:ot_color_glyph_get_layers ()
Wraps hb_ot_color_glyph_get_layers.

Face:ot_color_has_png ()
Wraps hb_ot_color_has_png.

Face:ot_layout_get_script_tags ()
Wraps hb_ot_layout_table_get_script_tags.

Face:ot_layout_get_language_tags ()
Wraps hb_ot_layout_script_get_language_tags.

Face:ot_layout_get_feature_tags ()
Wraps hb_ot_layout_language_get_feature_tags.

Face:ot_layout_find_script ()
Wraps hb_ot_layout_table_find_script.

Face:ot_layout_find_language ()
Wraps hb_ot_layout_script_find_language.

Face:ot_layout_find_feature ()
Wraps hb_ot_layout_language_find_feature.
Class Font
Lua wrapper for \texttt{hb\_font\_t} type

\texttt{Font.new (face)}
Wraps \texttt{hb\_font\_create}, and sets up some defaults for scale and shaping functions. Initializes a new \texttt{hb\_font\_t} from a \texttt{Face} object. Sets the default scale to the face’s upem value, and sets the font shaping functions by calling \texttt{hb\_ot\_font\_set\_funcs} on it.

Parameters:
\begin{itemize}
  \item face: \texttt{Face} object.
\end{itemize}

\texttt{Font:get\_scale ()}
Wraps \texttt{hb\_font\_get\_scale}.

Returns:
two values for the x-scale and y-scale of the font.

\texttt{Font:set\_scale (x\_scale, y\_scale)}
Wraps \texttt{hb\_font\_set\_scale}.

Parameters:
\begin{itemize}
  \item x\_scale: desired x-scale of font.
  \item y\_scale: desired y-scale of font.
\end{itemize}

\texttt{Font:get\_h\_extents ()}
Wraps \texttt{hb\_font\_get\_h\_extents}.

Returns:
font extents table for horizontal direction, contains the following or \texttt{nil} if HarfBuzz fails to load font extents:
\begin{itemize}
  \item ascender: typographic ascender.
  \item descender: typographic descender.
  \item line\_gap: line spacing gap.
\end{itemize}

\texttt{Font:get\_v\_extents ()}
Wraps \texttt{hb\_font\_get\_v\_extents}.

Returns:
font extents table for vertical direction, similar to \texttt{Font:get\_h\_extents}, or \texttt{nil} if HarfBuzz fails to load font extents:

\texttt{Font:get\_glyph\_extents (glyph)}
Wraps \texttt{hb\_font\_get\_glyph\_extents}.

Parameters:
\begin{itemize}
  \item glyph: index inside the font.
\end{itemize}

Returns:
extents table contains the following or \texttt{nil} if HarfBuzz fails to load glyph extents:
\begin{itemize}
  \item x\_bearing: left side of glyph from origin.
  \item y\_bearing: top side of glyph from origin.
  \item width: distance from left to right side.
  \item height: distance from top to bottom side.
\end{itemize}
**Font:get_glyph_name (glyph)**

Wraps `hb_font_get_glyph_name`.

**Parameters:**
- `glyph`: index inside the font.

**Returns:**
name of the glyph or nil.

**Font:get_glyph_from_name (name)**

Wraps `hb_font_get_glyph_from_name`.

**Parameters:**
- `name`: of the glyph.

**Returns:**
glyph index inside the font or nil.

**Font:get_glyph_h_advance (glyph)**

Wraps `hb_font_get_glyph_h_advance`.

**Parameters:**
- `glyph`: index inside the font.

**Returns:**
advance glyph advance of the glyph in horizontal direction.

**Font:get_glyph_v_advance (glyph)**

Wraps `hb_font_get_glyph_v_advance`.

**Parameters:**
- `glyph`: index inside the font.

**Returns:**
advance glyph advance of the glyph in vertical direction.

**Font:get_nominal_glyph (codepoint.)**

Wraps `hb_font_get_nominal_glyph`.

**Parameters:**
- `codepoint`: 

**Returns:**
glyph index or nil if `codepoint` is not supported by the font.

**Font:ot_color_glyph_get_png ()**

Wraps `hb_ot_color_glyph_get_png`.

**Class Buffer**

Lua wrapper for `hb_buffer_t` type.

---

The HarfBuzz libraries 263
Buffer.new ()
    Wraps hb_buffer_create.

Buffer:add_utf8 (text[, item_offset=0[, item_length=-1]])
    Wraps hb_buffer_add_utf8.
    Parameters:
    • text: UTF8 encoded string.
    • item_offset: 0-indexed offset in text, from where to start adding. (default 0)
    • item_length: length to add from item_offset. -1 adds till end of text. (default -1)

Buffer:add_codepoints (text[, item_offset=0[, item_length=-1]])
    Wraps hb_buffer_add_codepoints.
    Parameters:
    • text: table with codepoints as lua numbers.
    • item_offset: 0-indexed offset in text, from where to start adding. (default 0)
    • item_length: length to add from item_offset. -1 adds till end of text. (default -1)

Buffer:set_direction (dir)
    Wraps hb_buffer_set_direction.
    Parameters:
    • dir: A Direction object.

Buffer:get_direction ()
    Wraps hb_buffer_get_direction.
    Returns:
    A Direction object.

Buffer:set_script (script)
    Wraps hb_buffer_set_script.
    Parameters:
    • script: A Script object.

Buffer:get_script ()
    Wraps hb_buffer_get_script.
    Returns:
    A Script object.

Buffer:set_language (lang)
    Wraps hb_buffer_set_language.
    Parameters:
    • lang: A Language object

Buffer:get_language ()
Wraps `hb_buffer_get_language`.

**Returns:**
A `Language` object

---

**Buffer:reverse ()**

Wraps `hb_buffer_reverse`.

---

**Buffer:get_length ()**

Wraps `hb_buffer_get_length`.

---

**Buffer:get_cluster_level ()**

Wraps `hb_buffer_get_cluster_level`.

**Returns:**
see Cluster Levels

---

**Buffer:set_cluster_level (level)**

Wraps `hb_buffer_set_cluster_level`.

**Parameters:**
- **level**: see Cluster Levels

---

**Buffer:guess_segment_properties ()**

Wraps `hb_buffer_guess_segment_properties`.

---

**Buffer:get_glyphs ()**

Helper method to get shaped glyph data. Calls `hb_buffer_get_glyph_infos`, `hb_buffer_get_glyph_positions` and `hb_glyph_info_get_glyph_flags`, and assembles the data into a Lua table.

**Returns:**
table containing data for each glyph, in a nested table. Each nested table contains the following:
- **x_advance**: horizontal advance.
- **y_advance**: vertical advance.
- **x_offset**: horizontal displacement.
- **y_offset**: vertical displacement.
- **cluster**: glyph cluster index within input.
- **codepoint**: glyph index inside the font (this field name is a bit misleading, but that's what Harfbuzz uses).
- **flags**: glyph flags

---

**Cluster Levels**

See Harfbuzz docs for more details about what each of these levels mean.

---

**Buffer.CLUSTER_LEVEL_MONOTONE_GRAPHEMES**

Wraps HB_BUFFER_CLUSTER_LEVEL_MONOTONE_GRAPHEMES.

---

**Buffer.CLUSTER_LEVEL_MONOTONE_CHARACTERS**

Wraps HB_BUFFER_CLUSTER_LEVEL_MONOTONE_CHARACTERS.

---

The HarfBuzz libraries 265
Buffer.CLUSTER_LEVEL_CHARACTERS
Wraps HB_BUFFER_CLUSTER_LEVEL_CHARACTERS.

Buffer.CLUSTER_LEVEL_DEFAULT
Wraps HB_BUFFER_CLUSTER_LEVEL_DEFAULT.

Buffer.GlyphFlagUnsafeToBreak
Wraps HB_GLYPH_FLAG_UNUSED_TO_BREAK.

Buffer.GlyphFlagDefined
Wraps HB_GLYPH_FLAG_DEFINED.

Class Feature
Lua wrapper for hb_feature_t type

Feature.new (feature_string)
Wraps hb_feature_from_string

Parameters:
• feature_string: See feature string syntax reference

Feature.__tostring ()
Wraps hb_feature_to_string. Enables nice output with tostring(...).

Class Tag
Lua wrapper for hb_tag_t type.

Tag.new (string)
Wraps hb_tag_from_string.

Parameters:
• string: to be converted to a Tag object.

Returns:
a Tag object.

Tag:__to_string ()
Wraps hb_tag_to_string. Enable nice output with tostring(...).

Returns:
Returns a string representation for the tag object.

Tag:__eq ()
Enables equality comparisons with == between two tags.

Returns:
true or false depending on whether the two tags are equal.
Class Script
Lua wrapper for `hb_script_t` type.

Script.new (script)
Wraps `hb_script_from_string`.
Parameters:
- `script`: 4-letter script code according to the ISO 15924 standard.
Returns:
a `Script` object.

Script.from_iso15924_tag (tag)
Wraps `hb_script_from_iso15924_tag`
Parameters:
- `tag`: a `Tag` object representing a ISO 15924 script.

Script:to_iso15924_tag ()
Wraps `hb_script_to_iso15924_tag`.
Returns:
a `Tag` object representing the script.

Script:__to_string ()
Enable nice output with `tostring(…)`
Returns:
Returns a 4-letter ISO 15924 script code for the script object.

Script:__eq ()
Enables equality comparisions with `==` between two scripts.
Returns:
true or false depending on whether the two scripts are equal.

Predefined Script Codes
Predefined directions that correspond to their original definitions in Harfbuzz.

Script.COMMON
Wraps `HB_SCRIPT_COMMON`.

Script.INHERITED
Wraps `HB_SCRIPT_INHERITED`.

Script.UNKNOWN
Wraps `HB_SCRIPT_UNKNOWN`.

Script.INVALID
Wraps `HB_SCRIPT_INVALID`.

## Class Direction

Lua wrapper for `hb_direction_t` type.

### Direction.new (dir)

Wraps `hb_direction_from_string`.

**Parameters:**
- `dir`: can be one of `ltr`, `rtl`, `ttb`, `btt` or `invalid`.

**Returns:**
- a `Direction` object.

### Direction:__to_string ()

Wraps `hb_direction_to_string`. Enable nice output with `tostring(...)`.

**Returns:**
- Returns a string representation for direction.

### Direction:__eq ()

Enables equality comparisons with `==` between two directions.

**Returns:**
- `true` or `false` depending on whether the two tags are equal.

### Direction:is_valid ()

Wraps `HB_DIRECTION_IS_VALID`.

**Returns:**
- a boolean value

### Direction:is_horizontal ()

Wraps `HB_DIRECTION_IS_HORIZONTAL`.

**Returns:**
- a boolean value

### Direction:is_vertical ()

Wraps `HB_DIRECTION_IS_VERTICAL`.

**Returns:**
- a boolean value

### Direction:is_forward ()

Wraps `HB_DIRECTION_IS_FORWARD`.

**Returns:**
- a boolean value

### Direction:is_backward ()

Wraps `HB_SCRIPT_INVALID`.

---

**268 The HarfBuzz libraries**
Wraps HB_DIRECTION_IS_BACKWARD.

**Predefined directions**

Predefined directions that correspond to their original definitions in Harfbuzz.

- **Direction.LTR**
  Wraps HB_DIRECTION_LTR.

- **Direction.RTL**
  Wraps HB_DIRECTION_RTL.

- **Direction.TTB**
  Wraps HB_DIRECTION_TTB.

- **Direction.BTT**
  Wraps HB_DIRECTION_LTR.

**Class Language**

Lua wrapper for `hb_language_t` type.

- **Language.new (lang)**
  Wraps `hb_language_from_string`.
  
  **Parameters:**
  - `lang`: three-letter language tag to be converted to a `Language` object.
  
  **Returns:**
  a `Language` object.

- **Language:__to_string ()**
  Wraps `hb_language_to_string`. Enable nice output with `tostring(...)`.
  
  **Returns:**
  Returns a string representation for the language object.

- **Language:__eq ()**
  Enables equality comparisons with `==` between two languages.
  
  **Returns:**
  `true` or `false` depending on whether the two languages are equal.

**Predefined languages**

Predefined languages that correspond to their original definitions in Harfbuzz.

- **Language.INVALID**
  Wraps HB_LANGUAGE_INVALID.
Unicode functions

unicode.script (char)
Wraps hb_unicode_script
Parameters:
• char: Unicode codepoint
Returns:
a Script object.

Predefined Name IDs
Predefined OpenType 'name' table name identifier.

ot.NAME_ID_COPYRIGHT
Wraps HB_OT_NAME_ID_COPYRIGHT

ot.NAME_ID_FONT_FAMILY
Wraps HB_OT_NAME_ID_FONT_FAMILY

ot.NAME_ID_FONT_SUBFAMILY
Wraps HB_OT_NAME_ID_FONT_SUBFAMILY

ot.NAME_ID_UNIQUE_ID
Wraps HB_OT_NAME_ID_UNIQUE_ID

ot.NAME_ID_FULL_NAME
Wraps HB_OT_NAME_ID_FULL_NAME

ot.NAME_ID_VERSION_STRING
Wraps HB_OT_NAME_ID_VERSION_STRING

ot.NAME_ID_POSTSCRIPT_NAME
Wraps HB_OT_NAME_ID_POSTSCRIPT_NAME

ot.NAME_ID_TRADEMARK
Wraps HB_OT_NAME_ID_TRADEMARK

ot.NAME_ID_MANUFACTURER
Wraps HB_OT_NAME_ID_MANUFACTURER

ot.NAME_ID.Designer
Wraps HB_OT_NAME_IDDESIGNER

ot.NAME_ID_DESCRIPTION
Wraps HB_OT_NAME_ID_DESCRIPTION
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME_ID_VENDOR_URL</td>
<td>Wraps HB_OT_NAME_ID_VENDOR_URL</td>
</tr>
<tr>
<td>NAME_ID_DESIGNER_URL</td>
<td>Wraps HB_OT_NAME_ID_DESIGNER_URL</td>
</tr>
<tr>
<td>NAME_ID_LICENSE</td>
<td>Wraps HB_OT_NAME_ID_LICENSE</td>
</tr>
<tr>
<td>NAME_ID_LICENSE_URL</td>
<td>Wraps HB_OT_NAME_ID_LICENSE_URL</td>
</tr>
<tr>
<td>NAME_ID_TYPOGRAPHIC_FAMILY</td>
<td>Wraps HB_OT_NAME_ID_TYPOGRAPHIC_FAMILY</td>
</tr>
<tr>
<td>NAME_ID_TYPOGRAPHIC_SUBFAMILY</td>
<td>Wraps HB_OT_NAME_ID_TYPOGRAPHIC_SUBFAMILY</td>
</tr>
<tr>
<td>NAME_ID_MAC_FULL_NAME</td>
<td>Wraps HB_OT_NAME_ID_MAC_FULL_NAME</td>
</tr>
<tr>
<td>NAME_ID_SAMPLE_TEXT</td>
<td>Wraps HB_OT_NAME_ID_SAMPLETextUtils</td>
</tr>
<tr>
<td>NAME_ID_CID_FINDFONT_NAME</td>
<td>Wraps HB_OT_NAME_ID_CID_FINDFONT_NAME</td>
</tr>
<tr>
<td>NAME_ID_WWS_FAMILY</td>
<td>Wraps HB_OT_NAME_ID_WWS_FAMILY</td>
</tr>
<tr>
<td>NAME_ID_WWS_SUBFAMILY</td>
<td>Wraps HB_OT_NAME_ID_WWS_SUBFAMILY</td>
</tr>
<tr>
<td>NAME_ID_LIGHT_BACKGROUND</td>
<td>Wraps HB_OT_NAME_ID_LIGHT_BACKGROUND</td>
</tr>
<tr>
<td>NAME_ID_DARK_BACKGROUND</td>
<td>Wraps HB_OT_NAME_ID_DARK_BACKGROUND</td>
</tr>
<tr>
<td>NAME_ID_VARIATIONS_PS_PREFIX</td>
<td>Wraps HB_OT_NAME_ID_VARIATIONS_PS_PREFIX</td>
</tr>
<tr>
<td>NAME_ID_INVALID</td>
<td>Wraps HB_OT_NAME_ID_INVALID</td>
</tr>
</tbody>
</table>

The HarfBuzz libraries 271
<table>
<thead>
<tr>
<th>ot.LAYOUT_NO_SCRIPT_INDEX</th>
<th>Wraps HB_OT_LAYOUT_NO_SCRIPT_INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ot.LAYOUT_NO_FEATURE_INDEX</td>
<td>Wraps HB_OT_LAYOUT_NO_FEATURE_INDEX</td>
</tr>
<tr>
<td>ot.LAYOUT_DEFAULT_LANGUAGE_INDEX</td>
<td>Wraps HB_OT_LAYOUT_DEFAULT_LANGUAGE_INDEX</td>
</tr>
<tr>
<td>ot.LAYOUT_NO_VARIATIONS_INDEX</td>
<td>Wraps HB_OT_LAYOUT_NO_VARIATIONS_INDEX</td>
</tr>
</tbody>
</table>

*Last updated 2019-07-31 18:39:39*
13.1.1 Example

The example is (a small modification of) the file core_types.lua and it requires the file harfbuzz.lua; both are under luaharfbuzz folder of the source code. The fonts notonastaliq.ttf and amiri-regular.ttf are under luaharfbuzz/fonts.

The file harfbuzz.lua:

```lua
local hb = require("luaharfbuzz")

-- special tags
hb.Tag.NONE = hb.Tag.new()

-- special script codes (ISO 15924)
hb.Script.COMMON = hb.Script.new("Zyyy")
hb.Script.INHERITED = hb.Script.new("Zinh")
hb.Script.UNKNOWN = hb.Script.new("Zzzz")
hb.Script.INVALID = hb.Script.from_iso15924_tag(hb.Tag.NONE)

-- directions
hb.Direction.INVALID = hb.Direction.new("invalid")
hb.Direction.LTR = hb.Direction.new("ltr")
hb.Direction.RTL = hb.Direction.new("rtl")
hb.Direction.TTB = hb.Direction.new("ttb")
hb.Direction.BTT = hb.Direction.new("btt")

-- special languages
hb.Language.INVALID = hb.Language.new()

hb.shape = function(font, buf, options)
  options = options or {

    -- Apply options to buffer if they are set.
    if options.language then buf:set_language(options.language) end
    if options.script then buf:set_script(options.script) end
    if options.direction then buf:set_direction(options.direction) end

    -- Guess segment properties, in case all steps above have failed
    -- to set the right properties.
    buf:guess_segment_properties()

    local features = {}

    -- Parse features
    if type(options.features) == "string" then
      for fs in string.gmatch(options.features, '([^,]+)') do
        --
      end
    end

  }
```
local feature = hb.Feature.new(fs)
if feature then
    table.insert(features, hb.Feature.new(fs))
else
    error(string.format("Invalid feature string: '%s'", fs))
end
end
elseif type(options.features) == "table" then
    features = options.features
elseif options.features then -- non-nil but not a string or table
    error("Invalid features option")
end

return hb.shape_full(font,buf,features,options.shapers or {})
end

-- For backward compatibility
hb.Buffer.HB_BUFFER_CLUSTER_LEVEL_MONOTONE_GRAPHEMES = hb.Buffer.CLUSTER_LEVEL_MONOTONE_GRAPHEMES
hb.Buffer.HB_BUFFER_CLUSTER_LEVEL_MONOTONE_CHARACTERS = hb.Buffer.CLUSTER_LEVEL_MONOTONE_CHARACTERS
hb.Buffer.HB_BUFFER_CLUSTER_LEVEL_CHARACTERS = hb.Buffer.CLUSTER_LEVEL_CHARACTERS
hb.Buffer.HB_BUFFER_CLUSTER_LEVEL_DEFAULT = hb.Buffer.CLUSTER_LEVEL_DEFAULT

hb.Tag.HB_TAG_NONE = hb.Tag.NONE

hb.Script.HB_SCRIPT_COMMON = hb.Script.COMMON
hb.Script.HB_SCRIPT_INHERITED = hb.Script.INHERITED
hb.Script.HB_SCRIPT_UNKNOWN = hb.Script.UNKNOWN
hb.Script.HB_SCRIPT_INVALID = hb.Script.INVALID

hb.Language.HB_LANGUAGE_INVALID = hb.Language.INVALID

hb.Direction.HB_DIRECTION_INVALID = hb.Direction.INVALID
hb.Direction.HB_DIRECTION_LTR = hb.Direction.LTR
hb.Direction.HB_DIRECTION_RTL = hb.Direction.RTL
hb.Direction.HB_DIRECTION_TTB = hb.Direction.TTB
hb.Direction.HB_DIRECTION_BTT = hb.Direction.BTT

hb.Direction.HB_DIRECTION_IS_VALID = hb.Direction.is_valid
hb.Direction.HB_DIRECTION_IS_HORIZONTAL = hb.Direction.is_horizontal
hb.Direction.HB_DIRECTION_IS_VERTICAL = hb.Direction.is_vertical
hb.Direction.HB_DIRECTION_IS_FORWARD = hb.Direction.is_forward
hb.Direction.HB_DIRECTION_IS_BACKWARD = hb.Direction.is_backward
The example:

```lua
local harfbuzz = require('harfbuzz')

-- Harfbuzz API Version
print("Harfbuzz API version", harfbuzz.version())

-- Shapers available
print("Shapers:", harfbuzz.shapers())

-- harfbuzz.Face
local face = harfbuzz.Face.new('notonastaliq.ttf')
print('\nFace upem = '..face:get_upem())

-- harfbuzz.Font
local font = harfbuzz.Font.new(face)
local xs, xy = font:get_scale()
print('\nDefault font scale = X: "{xs}", Y: "{xy}"')

-- harfbuzz.Buffer
local text = "ی" -- U+06CC U+06C1
local buf = harfbuzz.Buffer.new()
buf:add_utf8(text)

-- harfbuzz.shape (Shapes text)
print("Shaping '{text}' set with Noto Nastaliq Urdu")
harfbuzz.shape(font, buf, { language = harfbuzz.Language.new("urd"),
script = harfbuzz.Script.new("Arab"), direction = harfbuzz.Direction.RTL})

local glyphs = buf:get_glyphs()
print("No. of glyphs", #glyphs)
for k,v in pairs(glyphs) do
    print(k)
    for k1,v1 in pairs(v) do
        print("",k1,v1)
    end
end

local opts = { language = harfbuzz.Language.new("eng"),
script = harfbuzz.Script.new("Latn"), direction = harfbuzz.Direction.LTR }

local amiri_face = harfbuzz.Face.new('amiri-regular.ttf')
local amiri_font = harfbuzz.Font.new(amiri_face)
```
-- shaping '123' w/o features
print("\nShaping '123' set with Amiri Regular and no features")
buf= harfbuzz.Buffer.new()
buf:add_utf8("123")
harfbuzz.shape(amiri_font, buf, opts)
glyphs = buf:get_glyphs()
for k,v in pairs(glyphs) do
    print(k)
    for k1,v1 in pairs(v) do
        print("",k1,v1)
    end
end

-- shaping '123' with '+numr' (numerators)
print("\nShaping '123' set with Amiri Regular with 'numr' feature turned on")
buf= harfbuzz.Buffer.new()
buf:add_utf8("123")
opts.features = "+numr"
harfbuzz.shape(amiri_font, buf, opts)
glyphs = buf:get_glyphs()
for k,v in pairs(glyphs) do
    print(k)
    for k1,v1 in pairs(v) do
        print("",k1,v1)
    end
end

The result:

Harfbuzz API version 2.6.4
Shapers: graphite2 ot fallback

Face upem = 2048

Default font scale = X: 2048, Y: 2048

Shaping '۰' set with Noto Nastaliq Urdu
No. of glyphs 4
1
    flags 1.0
    cluster 2
    codepoint 277
    x_advance 472.0
    y_advance 0.0
    x_offset 0.0
    y_offset 0.0
Shaping '123' set with Amiri Regular and no features

Shaping '123' set with Amiri Regular with 'numr' feature turned on
cluster 0
codepoint 6673
x_advance 600.0
y_advance 0.0
x_offset 0.0
y_offset 0.0

2
cluster 1
codepoint 6674
x_advance 600.0
y_advance 0.0
x_offset 0.0
y_offset 0.0

3
cluster 2
codepoint 6675
x_advance 600.0
y_advance 0.0
x_offset 0.0
y_offset 0.0
14 The backend libraries

14.1 The pdf library

This library contains variables and functions that are related to the pdf backend. You can find more details about the expected values to setters in section 3.2.

14.1.1 mapfile, mapline

pdf.mapfile(<string> map file)
pdf.mapline(<string> map line)

These two functions can be used to replace primitives \pdfmapfile and \pdfmapline inherited from pdf\TeX. They expect a string as only parameter and have no return value. The first character in a map line can be -, + or = which means as much as remove, add or replace this line. They are not state setters but act immediately.

14.1.2 [set|get][catalog|info|names|trailer]

These functions complement the corresponding pdf backend token lists dealing with metadata. The value types are strings and they are written to the pdf file directly after the token registers set at the \TeX end are written.

14.1.3 [set|get][pageattributes|pageresources|pagesattributes]

These functions complement the corresponding pdf backend token lists dealing with page resources. The variables have no interaction with the corresponding pdf backend token register. They are written to the pdf file directly after the token registers set at the \TeX end are written.

14.1.4 [set|get][xformattributes|xformresources]

These functions complement the corresponding pdf backend token lists dealing with reuseable boxes and images. The variables have no interaction with the corresponding pdf backend token register. They are written to the pdf file directly after the token registers set at the \TeX end are written.

14.1.5 [set|get][major|minor]version

You can set both the major and minor version of the output. The major version is normally 1 but when set to 2 some data will not be written to the file in order to comply with the standard. What minor version you set depends on what pdf features you use. This is out of control of Lua\TeX.

One can set the major version number to 2 but we cannot guarantee that the engine adapts itself correctly, because there is no public and free specification that we know of. Also, user
constructed annotations are not checked and just passed to the file. On the other hand, the pdf that the engine generated is rather simple and not that version depending.

14.1.6 **getcreationdate**

This function returns a string with the date in the format that ends up in the pdf file, in this case it’s: .

14.1.7 **[set|get]inclusionerrorlevel and [set|get]ignoreunknownimages**

These variable control how error in included image are treated. They are modeled after the pdfTeX equivalents.


The optional info bitset (a number) determines what kind of info gets flushed. By default we flush all. See section 3.2.2 for more details.

You can set your own trailer id. This has to be string containing valid pdf array content with checksums.

The cidset, charset and info flags (numbers) disables inclusion of a so called CIDSet and CharSet entries, which can be handy when aiming at some of the many pdf substandards.

When it is omitted, one should provide the MediaBox via the page attribute options, because it is a mandate field. No checking is done.

14.1.9 **[set|get][obj||]compresslevel and [set|get]recompress**

These functions set the level stream compression. When object compression is enabled multiple objects will be packed in a compressed stream which saves space. The minimum values are 0, the maxima are 9.

When recompression is to 1 compressed objects will be decompressed and when compresslevel is larger than zero they will then be recompressed. This is mostly a debugging feature and should not be relied upon.

14.1.10 **[set|get]gentounicode**

This flag enables tounicode generation (like in pdfTeX). Normally the values are provided by the font loader.

14.1.11 **[set|get]decimaldigits**

These two functions set the accuracy of floats written to the pdf file. You can set any value but the backend will not go below 3 and above 6.
14.1.12 [set|get]pkresolution
These setter takes two arguments: the resolution and an optional zero or one that indicates if this is a fixed one. The getter returns these two values.

14.1.13 getlast[obj|link|annot] and getretval
These status variables are similar to the ones traditionally used in the backend interface at the \TeX end.

14.1.14 getmaxobjnum and getobjtype, getfontname, getfontobjnum, getfontsize, getxformname
These introspective helpers are mostly used when you construct pdf objects yourself and need for instance information about a (to be) embedded font.

14.1.15 [set|get]origin
This one is used to set the horizontal and/or vertical offset, a traditional backend property.

```latex
pdf.setorigin() -- sets both to 0pt
pdf.setorigin(tex.sp("lin")) -- sets both to lin
pdf.setorigin(tex.sp("lin"),tex.sp("lin"))
```

The counterpart of this function returns two values.

14.1.16 [set|get]imageresolution
These two functions relate to the imageresolution that is used when the image itself doesn’t provide a non-zero x or y resolution.

14.1.17 [set|get][link|dest|thread|xform]margin
These functions can be used to set and retrieve the margins that are added to the natural bounding boxes of the respective objects.

14.1.18 get[pos|hpos|vpos]
These functions get current location on the output page, measured from its lower left corner. The values return scaled points as units.

```latex
local h, v = pdf.getpos()
```

14.1.19 [has|get]matrix
The current matrix transformation is available via the getmatrix command, which returns 6 values: sx, rx, ry, sy, tx, and ty. The hasmatrix function returns true when a matrix is applied.
if pdf.hasmatrix() then
    local sx, rx, ry, sy, tx, ty = pdf.getmatrix()
    -- do something useful or not
end

14.1.20 print

You can print a string to the pdf document from within a `\tex` call. This function is not to be used inside `\directlua` unless you know exactly what you are doing.

```lua
pdf.print(<string> s)
pdf.print(<string> type, <string> s)
```

The optional parameter can be used to mimic the behavior of pdf literals: the type is `direct` or `page`.

14.1.21 immediateobj

This function creates a pdf object and immediately writes it to the pdf file. It is modelled after pdfTeX's `\immediate \pdfobj` primitives. All function variants return the object number of the newly generated object.

```lua
<number> n =
    pdf.immediateobj(<string> objtext)
<number> n =
    pdf.immediateobj("file", <string> filename)
<number> n =
    pdf.immediateobj("stream", <string> streamtext, <string> attrtext)
<number> n =
    pdf.immediateobj("streamfile", <string> filename, <string> attrtext)
```

The first version puts the `objtext` raw into an object. Only the object wrapper is automatically generated, but any internal structure (like `<< >>` dictionary markers) needs to be provided by the user. The second version with keyword `file` as first argument puts the contents of the file with name `filename` raw into the object. The third version with keyword `stream` creates a stream object and puts the `streamtext` raw into the stream. The stream length is automatically calculated. The optional `attrtext` goes into the dictionary of that object. The fourth version with keyword `streamfile` does the same as the third one, it just reads the stream data raw from a file.

An optional first argument can be given to make the function use a previously reserved pdf object.

```lua
<number> n =
    pdf.immediateobj(<integer> n, <string> objtext)
<number> n =
    pdf.immediateobj(<integer> n, "file", <string> filename)
<number> n =
```
pdf.immediateobj(<integer> n, "stream", <string> streamtext, <string> attrtext)
<number> n =
    pdf.immediateobj(<integer> n, "streamfile", <string> filename, <string> attrtext)

14.1.22 obj

This function creates a pdf object, which is written to the pdf file only when referenced, e.g., by refobj().

All function variants return the object number of the newly generated object, and there are two separate calling modes. The first mode is modelled after pdf\TeX's \pdfobj primitive.

<number> n =
    pdf.obj(<string> objtext)
<number> n =
    pdf.obj("file", <string> filename)
<number> n =
    pdf.obj("stream", <string> streamtext, <string> attrtext)
<number> n =
    pdf.obj("streamfile", <string> filename, <string> attrtext)

An optional first argument can be given to make the function use a previously reserved pdf object.

<number> n =
    pdf.obj(<integer> n, <string> objtext)
<number> n =
    pdf.obj(<integer> n, "file", <string> filename)
<number> n =
    pdf.obj(<integer> n, "stream", <string> streamtext, <string> attrtext)
<number> n =
    pdf.obj(<integer> n, "streamfile", <string> filename, <string> attrtext)

The second mode accepts a single argument table with key-value pairs.

<number> n = pdf.obj {
    type = <string>,
immediate = <boolean>,
objnum = <number>,
attr = <string>,
compresslevel = <number>,
objcompression = <boolean>,
file = <string>,
string = <string>,
nolength = <boolean>,
}
The type field can have the values `raw` and `stream`, this field is required, the others are optional (within constraints). When `nolength` is set, there will be no `/Length` entry added to the dictionary.

Note: this mode makes `obj` look more flexible than it actually is: the constraints from the separate parameter version still apply, so for example you can’t have both string and file at the same time.

### 14.1.23 refobj

This function, the Lua version of the `\pdfrefobj` primitive, references an object by its object number, so that the object will be written to the pdf file.

```lua
pdf.refobj(<integer> n)
```

This function works in both the `\directlua` and `\latelua` environment. Inside `\directlua` a new whatsis node 'pdf_refobj' is created, which will be marked for flushing during page output and the object is then written directly after the page, when also the resources objects are written to the pdf file. Inside `\latelua` the object will be marked for flushing.

This function has no return values.

### 14.1.24 reserveobj

This function creates an empty pdf object and returns its number.

```lua
<number> n = pdf.reserveobj()
<number> n = pdf.reserveobj("annot")
```

### 14.1.25 getpageref

The object number of a page can be fetched with this function. This can be a forward reference so when you ask for a future page, you do get a number back.

```lua
<number> n = pdf.getpageref(123)
```

### 14.1.26 registerannot

This function adds an object number to the `/Annots` array for the current page without doing anything else. This function can only be used from within `\latelua`.

```lua
pdf.registerannot (<number> objnum)
```

### 14.1.27 newcolorstack

This function allocates a new color stack and returns it’s id. The arguments are the same as for the similar backend extension primitive.
14.1.28 setfontattributes

This function will force some additional code into the font resource. It can for instance be used to add a custom ToUnicode vector to a bitmap file.

pdf.setfontattributes(<number> font id, <string> pdf code)

14.2 The pdfe library

14.2.1 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.

<table>
<thead>
<tr>
<th>pdf</th>
<th>Lua</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>nil</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>float</td>
<td>number</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>array</td>
<td>array userdata</td>
</tr>
<tr>
<td>dictionary</td>
<td>dictionary userdata</td>
</tr>
<tr>
<td>stream</td>
<td>stream userdata (with related dictionary)</td>
</tr>
<tr>
<td>reference</td>
<td>reference userdata</td>
</tr>
</tbody>
</table>

The regular getters return these Lua data types but one can also get more detailed information.

14.2.2 open, new, getstatus, close, unencrypt

A document is loaded from a file or string

<pdfe document> = pdfe.open(filename)
<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:

```python
defe.getstatus(<pdfe document>)
```

The returned codes are:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>the document is (still) protected</td>
</tr>
<tr>
<td>-1</td>
<td>the document failed to open</td>
</tr>
<tr>
<td>0</td>
<td>the document is not encrypted</td>
</tr>
<tr>
<td>1</td>
<td>the document has been unencrypted</td>
</tr>
</tbody>
</table>

An encrypted document can be unencrypted by the next command where instead of either password you can give `nil`:

```python
defe.unencrypt(<pdfe document>,userpassword,ownerpassword)
```

### 14.2.3 `getsize`, `getversion`, `getnofobjects`, `getnofpages`, `getmemoryusage`<br>

A successfully opened document can provide some information:

```python
bytes = getsize(<pdfe document>)
major, minor = getversion(<pdfe document>)
n = getnofobjects(<pdfe document>)
n = getnofpages(<pdfe document>)
bytes, waste = getmemoryusage(<pdfe document>)
```

### 14.2.4 `get[catalog|trailer|info]`

For accessing the document structure you start with the so called catalog, a dictionary:

```python
<pdfe dictionary> = pdfe.getcatalog(<pdfe document>)
```

The other two root dictionaries are accessed with:

```python
<pdfe dictionary> = pdfe.gettrailer(<pdfe document>)
<pdfe dictionary> = pdfe.getinfo(<pdfe document>)
```

### 14.2.5 `getpage`, `getbox`

A specific page can conveniently be reached with the next command, which returns a dictionary. The first argument is to be a page dictionary.

```python
<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`.

---

286 The backend libraries
pages = pdfe.getbox(<pdfe dictionary>,boxname)

14.2.6 get[string|integer|number|boolean|name], type

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)
\text{n} = getnumber (<pdfe array|dictionary>,index|key)
b = getboolean(<pdfe array|dictionary>,index|key)
\text{n} = getname  (<pdfe array|dictionary>,index|key)
s = type  (<pdfe array|dictionary|document|reference|stream)

The type returns a string describing the type of the object, i.e. "pdfe.array", "pdfe.dictionary", "pdfe", "pdfe.reference", "pdfe.stream".
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.

14.2.7 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])

14.2.8 [open|close|readfrom|readfromwhole]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.

### 14.2.9 `getfrom[dictionary|array]`

In addition to the interface described before, there is also a bit lower level interface available.

```lua
key, type, value, detail = getfromdictionary(<pdfe dictionary>, index)
type, value, detail = getfromarray(<pdfe array>, index)
```

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MEANING</th>
<th>VALUE</th>
<th>DETAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>null</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>boolean</td>
<td>1 or 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>integer</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>number</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>string</td>
<td>string</td>
<td>hex</td>
</tr>
<tr>
<td>7</td>
<td>array</td>
<td>arrayobject</td>
<td>size</td>
</tr>
<tr>
<td>8</td>
<td>dictionary</td>
<td>dictionaryobject</td>
<td>size</td>
</tr>
<tr>
<td>9</td>
<td>stream</td>
<td>streamobject</td>
<td>dictionary size</td>
</tr>
<tr>
<td>10</td>
<td>reference</td>
<td>integer</td>
<td></td>
</tr>
</tbody>
</table>

A hex string is (in the pdf file) surrounded by <> while plain strings are bounded by <>.

### 14.2.10 `[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.

```lua
hash = dictionarytotable(<pdfe dictionary>)
list = arraytotable(<pdfe array>)
```

You can get a list of pages with:

```lua
{ { <pdfe dictionary>, size, objnum }, ... } = pagestotable(<pdfe document>)
```

### 14.2.11 `getfromreference`

Because you can have unresolved references, a reference object can be resolved with:
So, as second value you get back a new pdfe userdata object that you can query.

### 14.3 Memory streams

The `pdfe.new` that takes three arguments:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>this is a (in low level Lua speak) light userdata object, i.e. a pointer to a sequence of bytes</td>
</tr>
<tr>
<td>length</td>
<td>this is the length of the stream in bytes (the stream can have embedded zeros)</td>
</tr>
<tr>
<td>name</td>
<td>optional, this is a unique identifier that is used for hashing the stream, so that multiple doesn’t use more memory</td>
</tr>
</tbody>
</table>

The third argument is optional. When it is not given the function will return an `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 an `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 its 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 `graphicsmagick`, 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.

### 14.4 The `pdfscanner` library

The `pdfscanner` library allows interpretation of pdf content streams and `/ToUnicode` (cmap) streams. You can get those streams from the `pdfe` library, as explained in an earlier section. There is only a single top-level function in this library:

```lua
pdfscanner.scan (<pdfe stream>, <table> operatortable, <table> info)
pdfscanner.scan (<pdfe array>, <table> operatortable, <table> info)
pdfscanner.scan (<string>, <table> operatortable, <table> info)
```

The first argument should be a Lua string or a stream or array object coming from the `pdfe` library. The second argument, `operatortable`, should be a Lua table where the keys are pdf operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these pdf operators.
in the content stream(s). The functions are called with two arguments: the scanner object itself, and the info table that was passed are the third argument to pdfscanner.scan.

Internally, pdfscanner.scan loops over the pdf operators in the stream(s), collecting operands on an internal stack until it finds a pdf operator. If that pdf operator’s name exists in operator table, then the associated function is executed. After the function has run (or when there is no function to execute) the internal operand stack is cleared in preparation for the next operator, and processing continues.

The scanner argument to the processing functions is needed because it offers various methods to get the actual operands from the internal operand stack.

A simple example of processing a pdf’s document stream could look like this:

```lua
local operatortable = { }

operatortable.Do = function(scanner,info)
    local resources = info.resources
    if resources then
        local val = scanner:pop()
        local name = val[2]
        local xobject = resources.XObject
        print(info.space .. "Uses XObject " .. name)
        local resources = xobject.Resources
        if resources then
            local newinfo = {
                space = info.space .. " ",
                resources = resources,
            }
            pdfscanner.scan(entry, operatortable, newinfo)
        end
    end
end

local function Analyze(filename)
    local doc = pdfe.open(filename)
    if doc then
        local pages = doc.Pages
        for i=1,#pages do
            local page = pages[i]
            local info = {
                space = " ",
                resources = page.Resources,
            }
            print("Page " .. i)
            pdfscanner.scan(page.Contents, operatortable, info)
            pdfscanner.scan(page.Contents(), operatortable, info)
        end
    end
end
```

290 The backend libraries
Analyze("foo.pdf")

This example iterates over all the actual content in the pdf, and prints out the found XObject names. While the code demonstrates quite some of the pdfe functions, let's focus on the type pdfs scanner specific code instead.

From the bottom up, the following line runs the scanner with the pdf page's top-level content given in the first argument.

The third argument, info, contains two entries: space is used to indent the printed output, and resources is needed so that embedded XForms can find their own content.

The second argument, operator table defines a processing function for a single pdf operator, Do.

The function Do prints the name of the current XObject, and then starts a new scanner for that object's content stream, under the condition that the XObject is in fact a /Form. That nested scanner is called with new info argument with an updated space value so that the indentation of the output nicely nests, and with a new resources field to help the next iteration down to properly process any other, embeddedXObject s.

Of course, this is not a very useful example in practice, but for the purpose of demonstrating pdfs scanner, it is just long enough. It makes use of only one scanner method: scanner: pop(). That function pops the top operand of the internal stack, and returns a Lua table where the object at index one is a string representing the type of the operand, and object two is its value.

The list of possible operand types and associated Lua value types is:

<table>
<thead>
<tr>
<th>TYPES</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>&lt;number&gt;</td>
</tr>
<tr>
<td>real</td>
<td>&lt;number&gt;</td>
</tr>
<tr>
<td>boolean</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>name</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>operator</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>string</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>array</td>
<td>&lt;table&gt;</td>
</tr>
<tr>
<td>dict</td>
<td>&lt;table&gt;</td>
</tr>
</tbody>
</table>

In case of integer or real, the value is always a Lua (floating point) number. In case of name, the leading slash is always stripped.

In case of string, please bear in mind that pdf actually supports different types of strings (with different encodings) in different parts of the pdf document, so you may need to reencode some of the results; pdfs scanner always outputs the byte stream without reencoding anything. pdfs scanner does not differentiate between literal strings and hexadecimal strings (the hexadecimal values are decoded), and it treats the stream data for inline images as a string that is the single operand for EI.

In case of array, the table content is a list of pop return values and in case of dict, the table keys are pdf name strings and the values are pop return values.
There are a few more methods defined that you can ask scanner:

<table>
<thead>
<tr>
<th>METHOD</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>see above</td>
</tr>
<tr>
<td>popnumber</td>
<td>return only the value of a real or integer</td>
</tr>
<tr>
<td>popname</td>
<td>return only the value of a name</td>
</tr>
<tr>
<td>popstring</td>
<td>return only the value of a string</td>
</tr>
<tr>
<td>poparray</td>
<td>return only the value of an array</td>
</tr>
<tr>
<td>popdictionary</td>
<td>return only the value of a dict</td>
</tr>
<tr>
<td>popboolean</td>
<td>return only the value of a boolean</td>
</tr>
<tr>
<td>done</td>
<td>abort further processing of this scan() call</td>
</tr>
</tbody>
</table>

The pop* are convenience functions, and come in handy when you know the type of the operands beforehand (which you usually do, in pdf). For example, the Do function could have used local name = scanner:popname() instead, because the single operand to the Do operator is always a pdf name object.

The done function allows you to abort processing of a stream once you have learned everything you want to learn. This comes in handy while parsing /ToUnicode, because there usually is trailing garbage that you are not interested in. Without done, processing only ends at the end of the stream, possibly wasting cpu cycles.

*We keep the older names popNumber, popName, popString, popArray, popDict and popBool around.*
Topics

a
Aleph 43, 51
adjust 130
attributes 22, 156, 194

b
backend 36, 44, 279
banner 19
boundary 134
boxes 17, 22, 197
reuse 198
split 198
bytecodes 187

c
callbacks 169
building pages 175
closing files 173
contributions 174, 177
data files 171
dump 180
editing 182
errors 181, 182
files 182
font files 170, 171
fonts 184, 185
format file 170
hyphenation 179
image content 184
image files 172
input buffer 174
inserts 175
job run 181
jobname 174
kerning 180
ligature building 179
linebreaks 176, 177
math 180
opening files 172
output 179
output buffer 174
output file 170
pdf file 183

packing 177, 178
pages 181
reader 172
readers 173
rules 179
synctex 183
wrapping up 183
catcodes 27
characters 69
codes 196
command line 59
conditions 33
configuration 212
convert commands 193
csnames 56
d
direct nodes 159
directions 51, 134
discretionaries 79, 83, 130
e
$\varepsilon$-\TeX 40
engines 39
errors 28, 29, 204
escaping 25
exceptions 77
expansion 32
f
files
binary 56
finding 222
map 279
names 37
recording 222
writing 37
fontloader
tables 239
fonts 29, 87
current 100
define 99
defining 205
extend 99
id 99, 100
information 237
iterate 100
library 97
loading 237
real 92
tables 87
tfm 97
used 317
vf 98
virtual 92, 94, 96, 98
format 20, 56

g
glue 131
glyphs 69, 132
graphics 225

h
hash 205
helpers 203
history 39
hyperlinks 57
hyphenation 35, 69, 75, 77
discretionaries 79
exceptions 77
how it works 79
patterns 77

i
io 214
images 225
  immediate 228
  injection 228
  library 225
  MetaPost 230
  mplib 230
  object 228
  types 229
initialization 59, 206
  bitmaps 223
insertions 129

k
kerning 81

kerns 132
suppress 29

l
Lua 17
  byte code 59
  extensions 62
  interpreter 59
  libraries 62, 67
  modules 67
languages 35, 69
  library 83
last items 194
leaders 35
libraries
  kpse 221
  lua 187
  status 188
  tex 190
  texconfig 212
  texio 214
  token 215
ligatures 81
  suppress 29
linebreaks 83, 210
lists 128, 200

m
MetaPost 230
  mplib 230
macros 219
main loop 75
map files 279
marks 31, 129
math 28, 37, 103
  accents 119
  codes 123
  cramped 106
delimiters 120, 122
  extensibles 120
fences 118
flattening 125
fractions 122
italics 116
kerning 116
last line 123
s
shipout 211
space 57
spaces
  suppress 30
splitting 34
synctex 212
t
TeX 39
TrueType 237
Type1 239
testing 68
text
  math 125
tokens 215
  scanning 30
  tracing 36
u
Unicode 20,21
  math 103
v
version 19,187
w
web2c 44
Primitives

This register contains the primitives that are mentioned in the manual. There are of course many more primitives. The Lua\TeX\ primitives are typeset in bold. The primitives from pdf\TeX\ are not supported that way but mentioned anyway.

\abovedisplayskip 115
\belowdisplayskip 115
\abovewithdelims 122
\accent 33, 75, 76
\addafterocplist 43
\addbeforeocplist 43
\adjustspacing 41, 91
\alignmark 31
\aligntab 31
\atop 106, 108
\atopwithdelims 106
\attribute 194
\attributedef 194
\automaticdiscretionary 75
\automaticphenmode 73
\automaticphenpenalty 77
\batchmode 214
\begincsname 31
\begingroup 106
\bodydir 43
\bodydirection 54
\box 21
\boxdir 43
\breakafterdirmode 53
\catcode 19, 20, 56, 196
\catcodetable 27, 201
\char 18, 20, 76, 77, 132
\chardef 20, 77, 218, 219
\clearmarks 31
\clearocplists 43
\clubpenalties 210
\count 21
\countdef 21, 194
\crampedscriptstyle 107
\csname 28, 31
\csstring 31
\edef 26, 32, 47, 216
\edef 20, 41, 91
\endcsname 28
\endgroup 106
\endinput 215
\endlinechar 30, 39, 201, 202, 203
\errhelp 204
\errmessage 204
\edef 60
\etoksapp 30
\etokspre 30
\everyeof 30
\everyjob 60
\exceptionpenalty 78
\expandafter 32
\expanded 32, 41
\DefaultInputMode 43
\DefaultInputTranslation 43
\DefaultOutputMode 43
\DefaultOutputTranslation 43
\def 47
\delcode 56, 103, 196, 197
\delimiter 103
\detokenize 216
\dimen 21, 62, 194
\dimendef 21, 194
\directlua 17
\directlua 19, 23, 24, 25, 187, 201, 206, 282, 284
\discretionary 18, 77, 78, 80, 130
\displaystyle 113
\displaywidowpenalties 211
\dp 21
\draftmode 36, 42
\edef 26, 32, 47, 216
\efcode 20, 41, 91
\endcsname 28
\endgroup 106
\endinput 215
\endlinechar 30, 39, 201, 202, 203
\errhelp 204
\errmessage 204
\edef 60
\etoksapp 30
\etokspre 30
\everyeof 30
\everyjob 60
\exceptionpenalty 78
\expandafter 32
\expanded 32, 41
\DefaultInputMode 43
\DefaultInputTranslation 43
\DefaultOutputMode 43
\DefaultOutputTranslation 43
\def 47
\delcode 56, 103, 196, 197
\delimiter 103
\detokenize 216
\dimen 21, 62, 194
\dimendef 21, 194
\directlua 17
\directlua 19, 23, 24, 25, 187, 201, 206, 282, 284
\discretionary 18, 77, 78, 80, 130
\displaystyle 113
\displaywidowpenalties 211
\dp 21
\draftmode 36, 42

Primitives 297
\texttt{\textbackslash luatexbanner 19}
\texttt{\textbackslash luatexrevision 19, 20}
\texttt{\textbackslash luatexversion 19, 20}

\texttt{\textbackslash mag 39}
\texttt{\textbackslash mark 129}
\texttt{\textbackslash marks 21, 148}
\texttt{\textbackslash mathaccent 103}
\texttt{\textbackslash mathchar 103, 125}
\texttt{\textbackslash mathchardef 103, 125}
\texttt{\textbackslash mathchoice 105}
\texttt{\textbackslash mathcode 56, 103, 196}
\texttt{\textbackslash mathdefaultsmode 126}
\texttt{\textbackslash mathdelimitersmode 118}
\texttt{\textbackslash mathdir 43}
\texttt{\textbackslash mathdir 54}
\texttt{\textbackslash mathdir 201}
\texttt{\textbackslash mathdirection 54}
\texttt{\textbackslash mathdisplayskipmode 115}
\texttt{\textbackslash matheqdirmode 115}
\texttt{\textbackslash matheqnogapstep 118}
\texttt{\textbackslash mathflattennmode 125}
\texttt{\textbackslash mathitalicsmode 116, 118}
\texttt{\textbackslash mathnolimitsmode 115, 116}
\texttt{\textbackslash mathoption 126}
\texttt{\textbackslash mathpenaltiesmode 117}
\texttt{\textbackslash mathscriptboxmode 116}
\texttt{\textbackslash mathscriptcharmode 116}
\texttt{\textbackslash mathscriptsmode 117}
\texttt{\textbackslash mathstyle 105, 106, 201}
\texttt{\textbackslash mathsurround 113, 131}
\texttt{\textbackslash mathsurroundmode 113}
\texttt{\textbackslash mathsurroundskip 113}
\texttt{\textbackslash maxdepth 178}
\texttt{\textbackslash medmuskip 115}
\texttt{\textbackslash middle 201}
\texttt{\textbackslash muskip 21, 114, 115, 194}
\texttt{\textbackslash muskipdef 21}

\texttt{\textbackslash newlinechar 39}
\texttt{\textbackslash noboundary 35, 76, 81, 134}
\texttt{\textbackslash noDefaultInputMode 43}
\texttt{\textbackslash noDefaultInputTranslation 43}
\texttt{\textbackslash noDefaultOutputMode 43}
\texttt{\textbackslash noDefaultOutputTranslation 43}
\texttt{\textbackslash noexpand 32}
\texttt{\textbackslash nohrule 35}
\texttt{\textbackslash noInputMode 43}
\texttt{\textbackslash noInputTranslation 43}
\texttt{\textbackslash nokerns 29}
\texttt{\textbackslash noligs 29}
\texttt{\textbackslash noOutputMode 43}
\texttt{\textbackslash noOutputTranslation 43}
\texttt{\textbackslash nospaces 30}
\texttt{\textbackslash novrule 35}
\texttt{\textbackslash nullfont 28}
\texttt{\textbackslash number 29, 204}

\texttt{\textbackslash OutputMode 43}
\texttt{\textbackslash OutputTranslation 43}
\texttt{\textbackslash ocp 43}
\texttt{\textbackslash ocplist 43}
\texttt{\textbackslash ocptracelevel 43}
\texttt{\textbackslash omathcode 43}
\texttt{\textbackslash openin 170}
\texttt{\textbackslash openout 37, 44, 170}
\texttt{\textbackslash outer 28}
\texttt{\textbackslash output 179, 189}
\texttt{\textbackslash outputbox 34}
\texttt{\textbackslash outputmode 36, 42}
\texttt{\textbackslash over 106, 108, 201}
\texttt{\textbackslash overline 107}
\texttt{\textbackslash overwithdelims 106}

\texttt{\textbackslash pagebottomoffset 43}
\texttt{\textbackslash pagedir 43}
\texttt{\textbackslash pagedir 54}
\texttt{\textbackslash pagedir 54}
\texttt{\textbackslash pagerightoffset 43}
\texttt{\textbackslash pagewidth 41, 43}
\texttt{\textbackslash par 22, 28, 175}
\texttt{\textbackslash pardir 43}
\texttt{\textbackslash pardir 54}
\texttt{\textbackslash pardir 54}
\texttt{\textbackslash pardir 54}
\texttt{\textbackslash parfillskip 176, 211}
\texttt{\textbackslash parindent 190}
\texttt{\textbackslash parshape 54}
\texttt{\textbackslash patterns 77, 79, 80}
\texttt{\textbackslash pdfadjjustinterwordglue 40}
\texttt{\textbackslash pdfappendkern 40}
\texttt{\textbackslash pdfcopyfont 41}
\pdfdraftmode 42
\pdfeachlinedepth 41
\pdfeachlineheight 41
\pdfelapsedtime 40
\pdfescapehex 40
\pdfescapename 40
\pdfescapestring 40
\pdfextension 40, 45
\pdffeedback 40, 42, 45
\pdffilemoddate 40
\pdffilesize 40
\pdffirstlineheight 41
\pdffontattr 88
\pdffontexpand 41
\pdfforcepagebox 40
\pdfignoreddimen 41
\pdfimageaddfilename 42
\pdfinsertht 42
\pdflastlinedepth 41
\pdflastmatch 40
\pdflastxform 42
\pdflastximage 42
\pdflastximagepages 42
\pdfliteral 25
\pdfmapfile 279
\pdfmapline 279
\pdffilesize 40
\pdfmdfivesum 41
\pdfmovechars 41
\pdfnoligatures 41
\pdfnormaldeviate 41
\pdfobj 282, 283
\pdfoptionalwaysusepdfpagebox 41
\pdfoptionretainpdfinclusionerrorlevel 41
\pdfobject 284
\pdfoutput 42
\pdfpageheight 41
\pdfpagewidth 41
\pdfprependkern 40
\pxdimen 42
\pdfrandomseed 41
\pdfrefobj 284
\pdfresetafterocplist 43
\pdfresetbeforeocplist 43
\pdfresetomega 41
\pdfresettimer 41
\pdfRRcode 20, 41, 90
\pdfsetrandomseed 41
\pdfshellescape 41
\pdfsnaprefpoint 40
\pdfsnappy 40
\pdfsnappycomp 40
\pdfstrcmp 41
\pdftexbanner 41
\pdftexversion 41
\pdfunichar 41
\pdfuniformdeviate 41
\pdfvariable 40, 45, 225
\pdfxform 41, 42
\pdfxformattr 41
\pdfxformresources 41
\pdfximage 42, 225, 228
\penalty 132
\pushocpplist 43
\postexhyphenchar 75, 80
\posthyphenchar 80
\preexhyphenchar 75, 80
\prehyphenchar 80
\primitive 41
\protrudechars 41, 91
\protrusionboundary 35, 134
\pushocpplist 43
\pxdimen 42
\quitvmode 41
\radical 103
\read 170
\relax 77, 202, 206, 218
\removeafterocplist 43
\removebeforeocplist 43
\right 118
\rightghost 70, 76
\righthyphenmin 35, 70
\rightmargin 41
\romannumeral 105, 204
\rule 128
\saveboxresource 35, 42
\scalebox 27, 28
\saveimageresource 35, 42, 229
\Unosubscript 124
\Unosuperscript 124
\Uoverdelimiter 104, 120, 121
\Uradical 104, 120
\Uright 122
\Uroot 104, 120, 138
\Ustack 106
\Ustartdisplaymath 124
\Ustartmath 124
\Ustopdisplaymath 124
\Ustopmath 124
\Usubscript 124
\Usuperscript 124
\Uunderdelimiter 104, 120, 121
\uccode 20, 56, 196
\uchyph 70, 133
\unexpanded 216
\unhbox 21
\unhcopy 21
\unvbox 21
\unvcopy 21
\uppercase 31, 78
\useboxresource 35, 42, 198
\useimageresource 35, 42, 229
\vadjust 130, 175, 200
\valign 176
\variablefam 104
\vbox 18, 21, 34, 176, 197, 211
\vcenter 176
\voffset 43
\vpack 34
\vrule 18
\vskip 18, 131
\vsplit 21, 34, 176, 198
\vtop 18, 34, 176, 197
\wd 21
\widowpenalties 211
\wordboundary 35, 71, 134
\write 25, 60, 170, 174
\xtoksapp 30
\xtokspre 30
\- 75, 77, 130
Callbacks

b
buildpage_filter 175
build_page_insert 175

c
call_edit 182
contribute_filter 174

d
define_font 87, 94, 184

e
find_data_file 171
find_enc_file 171
find_font_file 170, 171
find_format_file 170
find_image_file 172
find_map_file 171
find_opentype_file 171
find_output_file 170
find_pk_file 171
find_read_file 170, 172
find_truetype_file 171
find_type1_file 171, 172
find_vf_file 171
find_write_file 170
finish_pdffile 183
finish_pdfpage 183
finish_synctex 183

f
glyph_info 185
glyph_not_found 185

h
hpack_filter 176, 177, 178
hyphenate 179

k
kerning 180, 243

l
ligaturing 179, 180
linebreak_filter 177, 211

m
mlist_to_hlist 118, 150, 180

o
open_read_file 172

p
page_order_index 183
post_linebreak_filter 177
pre_dump 180
pre_linebreak_filter 176, 211
process_input_buffer 174
process_jobname 174
process_output_buffer 174
process_pdf_image_content 184
process_rule 179

s
show_error_hook 181
show_error_message 182
show_lua_error_hook 182
start_file 182
start_page_number 181
start_run 181
stop_file 182
stop_page_number 181
stop_run 181

v
vpack_filter 176, 178

w
wrapup_run 183
Nodes

This register contains the nodes that are known to LuaTeX. The primary nodes are in bold, whatsit nodes that are determined by their subtype are normal. The names prefixed by pdf_ are backend specific.

a
accent 137
adjust 72, 130
attr 157
attribute_list 156, 157

b
boundary 35, 72, 134

c
choice 137
close 139
color_stack 127

d
delim 136
delta 203
dir 18, 72, 127, 134
disc 18, 21, 130

f
fence 138
fraction 119, 138

g
glue 18, 21, 72, 127, 131
glue_spec 131, 190, 193, 194, 195
glyph 18, 21, 69, 70, 75, 132, 152

h
hlist 18, 21, 22, 23, 72, 128, 153

i
ins 72, 129

k
kern 18, 21, 72, 132

l
late_lua 140
local_par 134, 211

m
margin kern 135
mark 129, 248
math 130, 252
math_char 135
math_text_char 135

n
noad 137

o
open 139

p
pdf_action 127, 142
pdf_annot 141
pdf_colorstack 143
pdf_dest 142
pdf_end_link 142
pdf_end_thread 143
pdf_late_literal 141
pdf_literal 127, 141
pdf_refobj 141
pdf_restore 144
pdf_save 144
pdf_setmatrix 144
pdf_start_link 141
pdf_start_thread 143
pdf_thread 143
pdf_window 127
penalty 72, 132

r
radical 138
rule 18, 72, 97, 128

s
save_pos 140
special 140
style 137
sub_box 135, 136
sub_mlist 135, 136

t
temp 128

u
unset 144, 242, 252
user_defined 139

v
vlist 18, 21, 72, 128, 153

w
whatsit 72, 146
write 139
Libraries

This register contains the functions available in libraries. Not all functions are documented, for instance because they can be experimental or obsolete.

**callback**
- find 169
- list 169
- register 169

**fio**
- getposition 66
- readbytes 66
- readbytetable 66
- readcardinaltable 66
- readcardinal1 66
- readcardinal2 66
- readcardinal3 66
- readcardinal4 66
- readfixed2 66
- readfixed4 66
- readinteger1 66
- readinteger2 66
- readinteger3 66
- readinteger4 66
- read2dot14 66
- setposition 66
- skipposition 66

**fontloader**
- apply_afmfile 239
- apply_featurefile 239
- close 237
- fields 239
- info 237
- open 237
- to_table 237

**img**
- boxes 229
- copy 228
- fields 226
- immediatewrite 228
- immediatewriteobject 228
- new 225
- node 229
- scan 227

**kpse**
- check_permission 224
- default_texmfconf 221
- expand_braces 224
- expand_path 224
- expand_var 224
- find_file 222
- init_prog 223
- lookup 223
- new 221
- readable_file 223
- record_input_file 222
- record_output_file 222
- set_program_name 221
- show_path 224
- var_value 224
- version 224

**lang**
- clean 84
- clear_hyphenation 84
- clear_patterns 84
- gethjcode 85
- hyphenate 85
- hyphenation 84
- hyphenationmin 84
- id 83
- new 83
- patterns 84
- postexhyphenchar 84
- posthyphenchar 84
- preexhyphenchar 84
- prehyphenchar 84
- sethjcode 85

**lua**
- bytecode 187
- getbytecode 187
- getcalllevel 188
- getluaname 188
getstacktop 188
name 187
setbytecode 187
setluaname 187
version 187

mplib
char_depth 236
char_height 236
char_width 236
execute 231
fields 232
finish 231
get_boolean 236
get_numeric 236
get_path 236
get_string 236
new 230
pen_info 235
statistics 231
version 230

node
check_discretionaries 158, 161
check_discretionary 158, 161
copy 147, 161
copy_list 147, 161
count 151, 161
current_attr 147, 161
dimensions 149, 161
effective_glue 161
end_of_math 153, 161
family_font 158, 161
fields 127, 146, 161
find_attribute 157, 161
first_glyph 154, 161
fix_node_lists 161
flatten_discretionaries 158, 161
flush_list 146, 161
flush_node 146, 161
flush_properties_table 161, 164
free 146, 161
getboth 161
getchar 161
getdisc 161
getfield 161
getfont 161
getglue 156, 161
getid 161
getleader 162
getlist 162
getnext 162
getprev 162
getproperty 162
getsubtype 162
getwhd 162
get_attribute 157, 161
get_properties_table 161, 164
has_attribute 157, 162
has_field 146, 162
has_glyph 153, 162
hpack 148, 162
id 145, 162
insert_after 154, 162
insert_before 153, 162
is_char 151, 162
is_glyph 151, 162
is_node 145, 162
is_zero_glue 156, 162
kerning 154, 162
last_node 155, 162
length 151, 162
ligaturing 154, 162
make_extensible 162
mlist_to_hlist 150, 162
new 146, 162
next 147, 162
prepend_prevdepth 149, 162
prev 147, 162
protect_glyph 155, 162
protect_glyphs 155, 162
protrusion_skippable 155, 162
rangedimensions 149, 162
remove 153, 162
setfield 163
setglue 155, 163
setproperty 163
set_attribute 157, 162
set_properties_mode 162, 164
slide 158, 163
subtype 145, 163
subtypes 127, 163
tail 150, 163
todirect 159
tonode 159
tostring 159, 163
traverse 151, 164
traverse_char 152, 163
traverse_glyph 152, 163
traverse_id 152, 163
traverse_list 153
type 145, 164
types 145, 164
unprotect_glyph 155, 164
unprotect_glyphs 155, 164
unset_attribute 158, 164
usedlist 164
uses_font 164
values 127, 164
vpack 148, 164
whatsits 145, 164
write 155, 164

node.direct
check_discretionaries 161
check_discretionary 161
copy 161
copy_list 161
count 161
current_attr 161
dimensions 161
effective_glue 161
end_of_math 161
find_attribute 161
first_glyph 161
flatten_discretionaries 161
flush_list 161
flush_node 161
flush_properties_table 161
free 161
getattributelist 161
getboth 161
getcbox 161
getchar 161
getcomponents 161
getdata 161
getdepth 161
getdir 161
getdirection 161
getdisc 161
getexpansion 161

getfam 161
getfield 161
getfont 161
getglue 161
getheight 161
getid 161
getkern 162
getlang 162
getleader 162
getlist 162
getnext 162
getnucleus 162
getoffsets 162
getpenalty 162
getprev 162
getproperty 162
getshift 162
getsub 162
getsubtype 162
getsup 162
getwidth 162
get_attribute 161
get_properties_table 161
get_synctex_fields 161
has_attribute 162
has_field 162
has_glyph 162
hpack 162
hyphenating 162
insert_after 162
insert_before 162
is_char 162
is_direct 162
is_glyph 162
is_node 162
is_zero_glue 162
kerning 162
last_node 162
length 162
ligaturing 162
new 162
prepend_prevdepth 162
protect_glyph 162
protect_glyphs 162
protrusion_skippable 162
rangedimensions 162
remove 162
setattributelist 163
setboth 163
setbox 163
setchar 163
setcomponents 163
setdata 163
setdepth 163
setdir 163
setdirection 163
setdisc 163
setexpansion 163
setfam 163
setfield 163
setfont 163
setglue 163
setheight 163
setkern 163
setlang 163
setleader 163
setlink 163
setlist 163
setnext 163
setnucleus 163
setoffsets 163
setpenalty 163
setprev 163
setproperty 163
setshift 163
setsplit 163
setsub 163
setsubtype 163
setsup 163
setwdh 163
setwidth 163
set_attribute 162
set_properties_mode 162
set_synctex_fields 163
slide 163
tail 163
todirect 163
tonode 163
tostring 163
traverse 164
traverse_char 163
traverse_glyph 163
traverse_id 163
traverse_list 163
unprotect_glyph 164
unprotect_glyphs 164
unset_attribute 164
usedlist 164
uses_font 164
vpack 164
write 164
os
env 64
exec 64
gmtimeofday 64
ksepopen 64
name 64
selfdir 64
setenv 64
sleep 64
socketgettime 64
socketsleep 64
spawn 64
times 64
tmpdir 64
type 64
uname 64
pdf
gccatalog 279
gcompresslevel 280
gcreationaldate 280
gdecimaldigits 280
gdestmargin 281
gfontname 281
gfontobjnum 281
gfontsize 281
gentounicode 280
gethpos 281
gignoreunknownimages 280
gimageresolution 281
ginclusionerrorlevel 280
ginfo 279
glastannot 281
glastlink 281
glastobj 281
glinkmargin 281
gmajorversion 279
getmarginmargin 281
getmatrix 281
getmaxobjnum 281
getminversion 279
getnames 279
getobjccompresslevel 280
getobjtype 281
getomitcharset 280
getomitcidset 280
getomitinfo 280
getomitmediabox 280
getorigin 281
getpageattributes 279
getpageref 284
getpageresources 279
getpagesattributes 279
getpkresolution 281
getpos 281
getrecompress 280
getretval 281
getsuppressoptionalinfo 280
getthreadmargin 281
gettrailer 279
gettrailerid 280
getvpos 281
getxformattributes 279
getxformmargin 281
getxformname 281
getxformresources 279
hasmatrix 281
immediateobj 282
newcolorstack 284
obj 283
print 282
reobj 284
reserveannot 284
setcatalog 279
setcompresslevel 280
setdecimaldigits 280
setdestmargin 281
setfontattributes 285
setgentounicode 280
setignoreunknownimages 280
setimageresolution 281
setinclusionerrorlevel 280
setinfo 279
setlastannot 281
setlastlink 281
setlastobj 281
setlinkmargin 281
setmajorversion 279
setmarginmargin 281
setminorversion 279
setnames 279
setobjccompresslevel 280
setomitcharset 280
setomitcidset 280
setomitinfo 280
setomitmediabox 280
setorigin 281
setpageattributes 279
setpageref 284
setpageresources 279
setpagesattributes 279
setpkresolution 281
setrecompress 280
setsuppressoptionalinfo 280
setthreadmargin 281
settrailer 279
settrailerid 280
setxformattributes 279
setxformmargin 281
setxformresources 279

pdfe
arraytotable 288
close 285
closestream 287
dictionarytotable 288
getarray 287
getboolean 287
getbox 286
getcatalog 286
getdictionary 287
getfromarray 287, 288
getfromdictionary 287, 288
getfromreference 288
getfromstream 287
getinfo 286
getinteger 287
getmemoryusage 286
glename 287
getnofobjects 286
getnofpages 286
getnumber 287
getpage 286
getsize 286
getstatus 285
getstream 287
getstring 287
gettrailer 286
getversion 286
new 285, 289
open 285
openstream 287
readfromstream 287
readfromwholestream 287
type 287
unencrypt 285
pdfscanner
done 292
pop 292
poparray 292
popboolean 292
popdictionary 292
popname 292
popnumber 292
popstring 292
scan 289
sha2
digest256 67
digest384 67
digest512 67
sio
getposition 67
readbytes 67
readbytetable 67
readcardinaltable 67
readcardinal1 67
readcardinal2 67
readcardinal3 67
readcardinal4 67
readfixed2 67
readfixed4 67
readintegeretable 67
readinteger1 67
readinteger2 67
readinteger3 67
readinteger4 67
read2dot14 67
setposition 67
skipposition 67
status
list 188
resetmessages 188
setexitcode 188
string
bytepairs 64
bytes 64
characterpairs 64
characters 64
explode 63
utfcharacter 64
utfcharacters 64
utflength 64
utfvalue 64
utfvalues 64
tex
attribute 194
badness 210
box 194, 197
catcode 196
count 194
cprint 203
definefont 205
delcode 196
dimen 194
enableprimitives 206
error 204
extraprimatives 206
finish 204
fontidentifier 204
fontname 204
forcehmode 205
force_synctex_line 212
force_synctex_tag 212
get 190
getattribute 194
getbox 194, 197
getboxresourcedimensions 198
getchcatcode 196
getcount 194
getdelcode 196
getdelcodes 196
getdimen 194
getglue 194
getlccode 196
getlist 200
getlocallevel 212
getmark 194
getmath 198
getmathcode 196
getmathcodes 196
getmuglue 194
getmuskip 194
getnest 200
getpagestate 211
getsfcode 196
getskip 194
gettoks 194
getuuccode 196
get_synctex_line 212
get_synctex_mode 212
get_synctex_tag 212

gleu 194
hashtokens 205
init_rand 212
isattribute 194
isbox 194
iscount 194
isdimen 194
isglue 194
ismuglue 194
ismuskip 194
isskip 194
istoks 194
lccode 196
linebreak 210
lists 200
lua_math_random 212
lua_math_randomseed 212
mathcode 196
muglue 194
muskip 194
nest 200
normal_rand 212
number 204
primitives 210
print 201
ptr 200
resetparagraph 210
romanumeral 204
round 203
run 204
saveboxresource 198
scale 203
scantoks 194
set 190
setattribute 194
setbox 194, 197
setcatcode 196
setcount 194
setdelcode 196
setdelcodes 196
setdimen 194
setglue 194
setlccode 196
setlist 200
setmath 198
setmathcode 196
setmathcodes 196
setmuglue 194
setmuskip 194
setsfcode 196
setskip 194
settoks 194
setattribute 194
setbox 194, 197
setcatcode 196
setcount 194
setdelcode 196
setdelcodes 196
setdimen 194
setglue 194
setlccode 196
setlist 200
setmath 198
setmathcode 196
setmathcodes 196
setmuglue 194
setmuskip 194
setsfcode 196
setskip 194
settoks 194
setuuccode 196
set_synctex_line 212
set_synctex_mode 212
set_synctex_no_files 212
set_synctex_tag 212
sfcode 196
shipout 211
show_context 204
skip 194
sp 204
splitbox 198
sprint 201
toks 194
tprint 202
triggerbuildpage 198
uuccode 196
uniformdeviate 212
uniform_rand 212
useboxresource 198
write 203
texio
  closeinput 215
  setescape 215
  write 214
  write_nl 214

token
  biggest_char 218
  commands 218
  command_id 218
  create 218
  expand 217
  get_active 218
  get_cmdname 218
  get_command 218
  get_csname 218
  get_expandable 218
  get_functions_table 219
  get_id 218
  get_index 218
  get_macro 219
  get_meaning 219
  get_mode 218
  get_next 217, 218, 219
  get_protected 218
  get_tok 218
  is_defined 218
  is_token 218
  new 218
  put_next 219
  scan_argument 215
  scan_code 215
  scan_csname 215
  scan_dimen 215
  scan_float 215
  scan_glue 215
  scan_int 215
  scan_keyword 215
  scan_keywordcs 215
  scan_list 215
  scan_real 215
  scan_string 215
  scan_token 217
  scan_toks 215
  scan_word 215
  set_char 219
  set_lua 219
  set_macro 219
Statistics

The following fonts are used in this document:

<table>
<thead>
<tr>
<th>used</th>
<th>filesize</th>
<th>version</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>733.736</td>
<td>1.959</td>
<td>latinmodern-math.otf</td>
</tr>
<tr>
<td>1</td>
<td>64.684</td>
<td>2.004</td>
<td>lmmono10-regular.otf</td>
</tr>
<tr>
<td>1</td>
<td>64.160</td>
<td>2.004</td>
<td>lmmonol10-regular.otf</td>
</tr>
<tr>
<td>4</td>
<td>111.536</td>
<td>2.004</td>
<td>lmroman10-regular.otf</td>
</tr>
<tr>
<td>30</td>
<td>525.008</td>
<td>1.106</td>
<td>texgyredejavu-math.otf</td>
</tr>
<tr>
<td>24</td>
<td>601.220</td>
<td>1.632</td>
<td>texgyrepagella-math.otf</td>
</tr>
<tr>
<td>4</td>
<td>218.100</td>
<td>2.501</td>
<td>texgyrepagella-regular.otf</td>
</tr>
<tr>
<td>1</td>
<td>693.876</td>
<td>2.340</td>
<td>DejaVuSans-Bold.ttf</td>
</tr>
<tr>
<td>1</td>
<td>741.536</td>
<td>2.340</td>
<td>DejaVuSans.ttf</td>
</tr>
<tr>
<td>4</td>
<td>318.392</td>
<td>2.340</td>
<td>DejaVuSansMono-Bold.ttf</td>
</tr>
<tr>
<td>1</td>
<td>245.948</td>
<td>2.340</td>
<td>DejaVuSansMono-Oblique.ttf</td>
</tr>
<tr>
<td>3</td>
<td>335.068</td>
<td>2.340</td>
<td>DejaVuSansMono.ttf</td>
</tr>
<tr>
<td>9</td>
<td>345.364</td>
<td>2.340</td>
<td>DejaVuSerif-Bold.ttf</td>
</tr>
<tr>
<td>1</td>
<td>336.884</td>
<td>2.340</td>
<td>DejaVuSerif-BoldItalic.ttf</td>
</tr>
<tr>
<td>1</td>
<td>343.388</td>
<td>2.340</td>
<td>DejaVuSerif-Italic.ttf</td>
</tr>
<tr>
<td>5</td>
<td>367.260</td>
<td>2.340</td>
<td>DejaVuSerif.ttf</td>
</tr>
<tr>
<td>28</td>
<td>1.622.732</td>
<td>5.960</td>
<td>cambria.ttc</td>
</tr>
<tr>
<td>24</td>
<td>348.296</td>
<td>1.802</td>
<td>LucidaBrightMathOT.otf</td>
</tr>
<tr>
<td>4</td>
<td>73.284</td>
<td>1.801</td>
<td>LucidaBrightOT.otf</td>
</tr>
</tbody>
</table>

**170  8.090.472  19 files loaded**