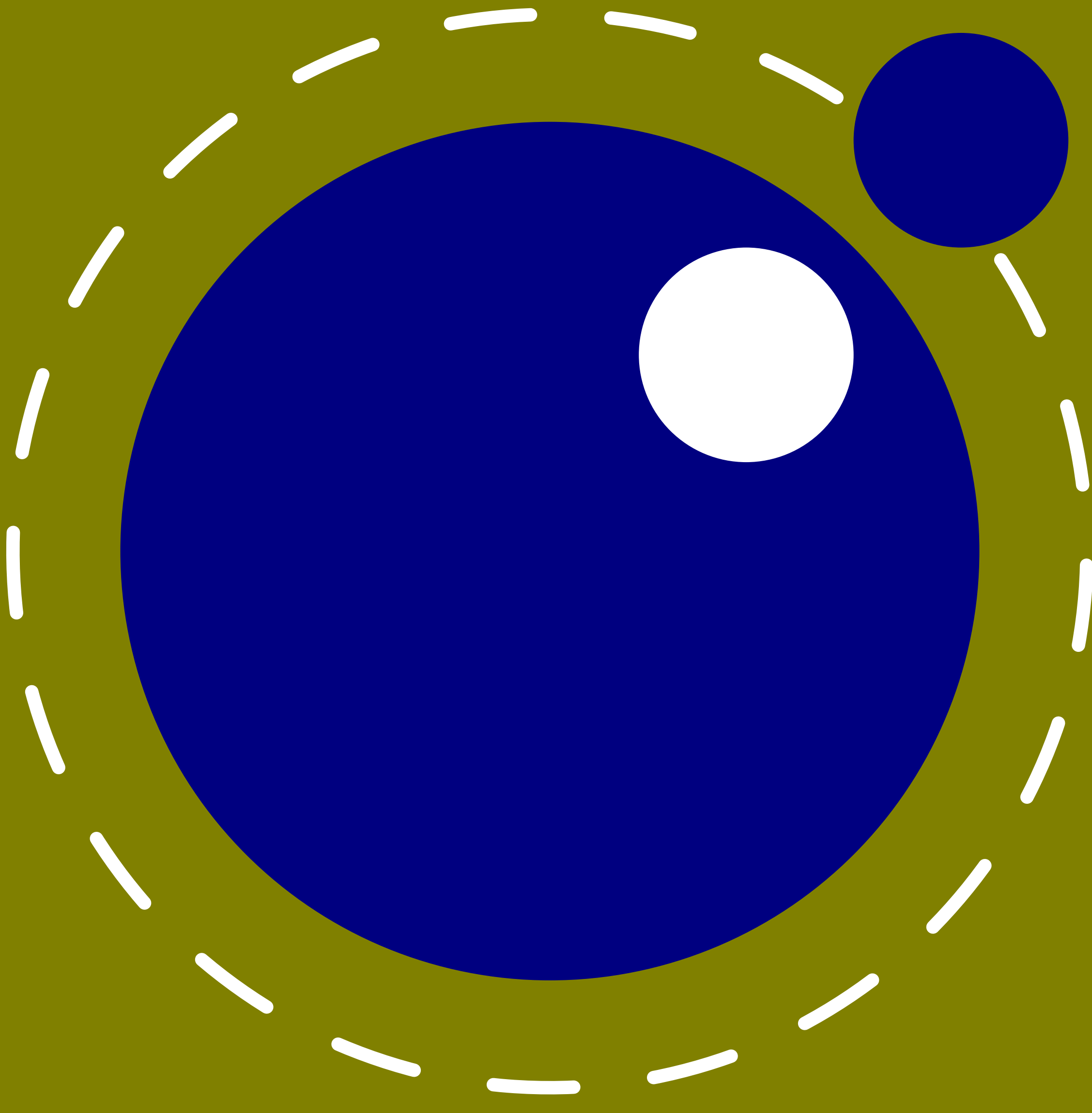
LuaMetaTEX Reference Manual



July 2020

Version 2.06.17

LuaMetaTeX Reference Manual

copyright : LuaTeX development team
: ConTeXt development team
more info : www.luatex.org
: contextgarden.net
version : July 19, 2020

Contents

Introduction

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

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

The LUAMETATEX engine is, as said, a lightweight version of LUATEX, that for now targets CONTEXT. We will use it for possibly drastic experiments but without affecting LUATEX. As we can easily adapt CONTEXT to support both, no other macro package will be harmed when (for instance) interfaces change as part of an experiment. Of course, when we consider something to be useful, it can be ported back to LUATEX, but only when there are good reasons for doing so and when no compatibility issues are involved. When considering this follow up one consideration was that a lean and mean version with an extension mechanism is a bit closer to original TEX. Of course, because we also have new primitives, this is not entirely true. The move to LUA already meant that some aspects, especially system dependent ones, no longer made sense and therefore had consequences for the interface at the system level.

This manual currently has quite a bit of overlap with the LUATEX manual but some chapters are removed, others added and the rest has been (and will be further) adapted. It also discusses the (main) differences. Some of the new primitives or functions that show up in LUAMETATEX might show up in LUATEX at some point, others might not, so don't take this manual as reference for LUATEX! For now it is an experimental engine in which we can change things at will but with CONTEXT in tandem so that this macro package will keep working.

For CONTEXT users the LUAMETATEX engine will become the default. The CONTEXT variant for this engine is tagged LMTX. The pair can be used in production, just as with LUATEX and MKIV. In fact, most users will probably not really notice the difference. In some cases there will be a drop in performance, due to more work being delegated to LUA, but on the average performance will be better, also due to some changes below the hood of the engine.

As this follow up is closely related to CONTEXT development, and because we expect stock LUATEX to be used outside the CONTEXT proper, there will be no special mailing list nor coverage (or pollution) on the LUATEX related mailing lists. We have the CONTEXT mailing lists for that. In due time the source code will be part of the regular CONTEXT distribution.

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

Hans Hagen

Version	:	July 19, 2020
LuaMetaTeX	:	luametatex 2.0617 / 20200717
ConTeXt	:	MkIV 2020.07.14 20:20
LuaTeX Team	:	Hans Hagen, Hartmut Henkel, Taco Hoekwater, Luigi Scarso

remark: LuaMetaTeX development is mostly done by Hans Hagen and Alan Braslau, who love playing with the three languages involved. And as usual Mojca Miklavec make sure all compiles well on the buildbot infrastructure. Testing is done by ConTeXt developers and users. Many thanks for their patience!

remark: When there are non-intrusive features that also make sense in LuaTeX, these will be applied in the experimental branch first, so that there is no interference with the stable release.

remark: Most ConTeXt users seem always willing to keep up with the latest versions which means that LMTX is tested well. We can therefore safely claim that end of 2019 the code has become quite stable. There are no complaints about performance (on my laptop this manual compiles at 22.5 pps with LMTX versus 20.7 pps for the LuaTeX manual with MkIV). Probably no one notices it, but memory consumption stepwise got reduced too. And ... the binary is still below 3 MegaBytes on all platforms.

1 The internals

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. Because the LUATEX and LUAMETATEX engines open up TEX there's suddenly quite some more to explain, especially about the way a (to be) typeset stream moves through the machinery. However, discussing all that in detail makes not much sense, because deep knowledge is only relevant for those who write code not possible with regular TEX and who are already familiar with these internals (or willing to spend time on figuring it out).

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

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

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

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

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

```
Hi There
```

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

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

When we have a paragraph, we actually get something:

```
[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. Sometimes multiple characters are represented by one glyphs, so one can also get:

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

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

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

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

The LUATEX engine provides hooks for LUA code at nearly every reasonable point in the process: collecting content, hyphenating, applying font features, breaking into lines, etc. This means that you can overload TEX's natural behaviour, which still is the benchmark. When we refer to 'callbacks' we means these hooks. The TEX engine itself is pretty well optimized but when you kick in much LUA code, you will notices that performance drops. Don't blame and bother the authors with performance issues. In CONTEXT over 50% of the time can be spent in LUA, but so far we didn't get many complaints about efficiency.

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. Each successive step expects a certain result and if you mess around to much, the engine eventually might bark and quit. It can even crash, because testing everywhere for what users can do wrong is no real option.

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

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

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

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

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

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

16 The internals

2 Differences with LUATEX

As LUAMETATEX is a leaner and meaner LUATEX, this chapter will discuss what is gone. We start with the primitives that were dropped.

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

The resources and positioning primitives are actually useful but can be defined as macros that (via LUA) inject nodes in the input that suit the macro package and backend. The three–letter direction primitives are gone and the numeric variants are now leading. There is no need for page and body related directions and they don't work well in LUATEX anyway. We only have two directions left.

The primitive related extensions were not that useful and reliable so they have been removed. There are some new variants that will be discussed later. The `\outer` and `\long` prefixes are gone as they don't make much sense nowadays and them becoming dummies opened the way to something new, again to be discussed elsewhere. I don't think that (CONTEXT) users will notice it. The `\suppress..` features are now default.

The `\shipout` primitive does no ship out but just erases the content of the box, if that hasn't happened already in another way.

The extension primitives relate to the backend (when not immediate) and can be implemented as part of a backend design using generic whatsits. There is only one type of whatsit now. In fact we're now closer to original TEX with respect to the extensions.

The `img` library has been removed as it's rather bound to the backend. The `slunicode` library is also gone. There are some helpers in the string library that can be used instead and one can write additional LUA code if needed. There is no longer a `pdf` backend library.

In the node, tex and status library we no longer have helpers and variables that relate to the backend. The LuaMetaTeX engine is in principle DVI and PDF unaware. There are only generic whatsit nodes that can be used for some management related tasks. For instance you can use them to implement user nodes.

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

The KPSE library is no longer built-in. Because there is no backend, quite some file related callbacks could go away. The following file related callbacks remained (till now):

```
find_write_file find_data_file find_format_file
open_data_file read_data_file
```

Also callbacks related to errors stay:

```
show_error_hook show_lua_error_hook,
show_error_message show_warning_message
```

The (job) management hooks are kept:

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

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

```
show_whatsit
```

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

```
find_log_file, find_data_file, find_format_file, open_data_file, read_data_file,
process_jobname, start_run, stop_run, define_font, pre_output_filter,
buildpage_filter, hpack_filter, vpack_filter, hyphenate, ligaturing, kerning,
pre_linebreak_filter, linebreak_filter, post_linebreak_filter,
append_to_vlist_filter, mlist_to_hlist, pre_dump, start_file, stop_file,
handle_error_hook, show_error_hook, show_lua_error_hook, show_error_message,
show_warning_message, hpack_quality, vpack_quality, insert_local_par,
contribute_filter, build_page_insert, wrapup_run, new_graf, make_extensible,
show_whatsit, terminal_input,
```

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

```
define_font
```

There are all kinds of subtle differences in the implementation, for instance we no longer intercept * and & as these were already replaced long ago in TeX engines by command line options. Talking of options, only a few are left.

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

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

The error reporting subsystem has been redone a little but is still fundamentally the same. We don't really assume interactive usage but if someone uses it, it might be noticed that it is not possible to backtrack or inject something. Of course it is no big deal to implement all that in LUA if needed. It removes a system dependency and makes for a bit cleaner code.

There are new primitives too as well as some extensions to existing primitive functionality. These are described in following chapters but there might be hidden treasures in the binary. If you locate them, don't automatically assume them to stay, some might be part of experiments!

The following primitives are available in LuaTeX but not in LuaMetaTeX.Some of these are emulated in ConTeXt.

bodydir
bodydirection
boxdir
closeout
copyfont
draftmode
dviextension
dvifeedback
dvivariable
eTeXVersion
eTeXglueshrinkorder
eTeXgluestretchorder
eTeXminorversion
eTeXrevision
eTeXversion
expandglyphsinfont
ifprimitive
ignoreligaturesinfont
immediate
lastsavedboxresourceindex
lastsavedimageresourceindex
lastsavedimageresourcepages
lastxpos
lastypos
latelua
lateluafunction
leftghost
letterspacefont
linedir
mathdir
mathoption
nolocaldirs
nolocalwhatsits
normaldeviate
openout
outputmode
pagebottomoffset
pagedir
pagedirection
pageheight
pageleftoffset
pagerightoffset
pagetopoffset
pagewidth
pardir
pdfextension
pdffeedback
pdfvariable
primitive
randomseed
rightghost
saveboxresource
saveimageresource
savepos
setrandomseed
special
suppressfontnotfounderror
suppressifcsnameerror
suppresslongerror
suppressmathparerror
suppressoutererror
suppressprimitiveerror
synctex
tagcode
textdir
uniformdeviate
useboxresource
useimageresource
write

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

UUskewed
UUskewedwithdelims
Uabove
Uabovewithdelims
Uatop
Uatopwithdelims
Umathclass
Umathspacebeforescript
Umathspacingmode
Unosubprescript
Unosuperprescript
Uover
Uoverwithdelims
Ustyle
Usubprescript
Usuperprescript
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
afterassigned
aftergrouped
atendofgroup
atendofgrouped
beginlocalcontrol
boxattr
boxorientation
boxtotal
boxxmove
boxxoffset
boxymove
boxyoffset
everytab
expand
expandafterpars
expandafterspaces
expandcstoken
expandtoken
frozen
futuredef
futureexpand
futureexpandis
futureexpandisap
glyphdatafield
glyphscriptfield
glyphstatefield
ifarguments
ifboolean
ifchkdim
ifchknum
ifcmpdim
ifcmpnum
ifcstok
ifdimval
ifempty
iffrozen
ifhastok
ifhastoks
ifhasxtoks
ifmathparameter
ifmathstyle
ifnumval
ifprotected
iftok
ifusercmd
ignorearguments
ignorepars
internalcodesmode
lastarguments
lastnodesubtype
letdatacode
letfrozen
letprotected
luavaluefunction
matholdmode
ordlimits
orelse
shownodedetails
supmarkmode
thewithoutunit
tokenized
unletfrozen
unletprotected

22 Differences with LuaTeX

3 The original engines

3.1 The merged engines

3.1.1 The rationale

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

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

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

3.1.2 Changes from TEX 3.1415926

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

- The current code base is written in C, not PASCAL. The original CWEB documentation is kept when possible and not wrapped in tagged comments. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like `tex`, `luaf`, `languages`, `fonts`, `libraries`, etc. There are some artifacts of the conversion to C, but these got (and get) removed stepwise. The documentation, which actually comes from the mix of engines (via so called change files), is kept as much as possible. 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. However, because we use C, some documentation is a bit off. Also, most global variables are now collected in structures, but the original names were kept. There are lots of so called macros too.
- See chapter 7 for many small changes related to paragraph building, language handling and hyphenation. The most important change is that adding a brace group in the middle of a word (like in `of{}fice`) does not prevent ligature creation. Also, the hyphenation, ligature

building and kerning has been split so that we can hook in alternative or extra code wherever we like. There are various options to control discretionary injection and related penalties are now integrated in these nodes. Language information is now bound to glyphs. The number of languages in LuaMetaTeX is smaller than in LuaTeX.

- There is no pool file, all strings are embedded during compilation. This also removed some memory constraints. We kept token and node memory management because it is convenient and efficient but parts were reimplemented in order to remove some constraints. Token memory management is largely the same.
- The specifier `plus 1 fillll` does not generate an error. The extra 'l' is simply typeset.
- The upper limit to `\endlinechar` and `\newlinechar` is 127.
- Because the backend is not built-in, the magnification (`\mag`) primitive is not doing nothing. A `shipout` just discards the content of the given box. The write related primitives have to be implemented in the used macro package using Lua. None of the pdfTeX derived primitives is present.
- There is more control over some (formerly hard-coded) math properties. In fact, there is a whole extra bit of math related code because we need to deal with OpenType fonts.
- The `\outer` and `\long` prefixed are silently ignored. It is permitted to use `\par` in math.
- Because there is no font loader, a Lua variant is free to either support or not the Omega `ofm` file format. As there are hardly any such fonts it probably makes no sense.
- The lack of a backend means that some primitives related to it are not implemented. This is no big deal because it is possible to use the scanner library to implement them as needed, which depends on the macro package and backend.
- When detailed logging is enabled more detail is output with respect to what nodes are involved. This is a side effect of the core nodes having more detailed subtype information. The benefit of more detail wins from any wish to be byte compatible in the logging. One can always write additional logging in Lua.

3.1.3 Changes from ε-TeX 2.2

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

- The ε-TeX functionality is always present and enabled so the prepended asterisk or `-etex` switch for iniTeX is not needed.
- The TeXXeT extension is not present, so the primitives `\TeXXeTstate`, `\beginR`, `\beginL`, `\endR` and `\endL` are missing. Instead we used the Omega/Aleph approach to directionality as starting point, albeit it has been changed quite a bit, so that we're probably not that far from TeXXeT.
- Some of the tracing information that is output by ε-TeX's `\tracingassigns` and `\tracingrestores` is not there. Also keep in mind that tracing doesn't involve what Lua does.
- Register management in LuaMetaTeX uses the Omega/Aleph model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat & sparse model from ε-TeX.
- Because we don't use change files on top of original TeX, the integration of ε-TeX functionality is bit more natural, code wise.

3.1.4 Changes from PDFTEX 1.40

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

- The experimental primitives `\ifabsnum` and `\ifabsdim` have been promoted to core primitives.
- The primitives `\ifincsname`, `\expanded` and `\quitvmode` have become core primitives.
- As the hz (expansion) and protrusion mechanism are part of the core the related primitives `\lpcode`, `\rpcode`, `\efcode`, `\leftmarginkern`, `\rightmarginkern` are promoted to core primitives. The two commands `\protrudechars` and `\adjustspacing` control these processes.
- In LUAMETATEX three extra primitives can be used to overload the font specific settings: `\adjustspacingstep` (max: 100), `\adjustspacingstretch` (max: 1000) and `\adjustspacingshrink` (max: 500).
- The hz optimization code has been partially redone so that we no longer need to create extra font instances. The front- and backend have been decoupled and the glyph and kern nodes carry the used values. In LUATEX that made a more efficient generation of PDF code possible. It also resulted in much cleaner code. The backend code is gone, but of course the information is still carried around.
- When `\adjustspacing` has value 2, hz optimization will be applied to glyphs and kerns. When the value is 3, only glyphs will be treated. A value smaller than 2 disables this feature. 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 PDFTEX). But ... this is a left-over from the early days of PDFTEX when this feature was part of a research topic. At some point level 1 might be dropped from LUAMETATEX.
- When `\protrudechars` has a value larger than zero characters at the edge of a line can be made to hang out. A value of 2 will take the protrusion into account when breaking a paragraph into lines. A value of 3 will try to deal with right-to-left rendering; this is a still experimental feature.
- The pixel multiplier dimension `\pxdimen` has be inherited as core primitive.
- The primitive `\tracingfonts` is now a core primitive but doesn't relate to the backend.

3.1.5 Changes from ALEPH RC4

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

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

Here is a summary of inherited functionality:

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

3.1.6 Changes from standard WEB2C

The LUAMETATEX codebase is not dependent on the WEB2C framework. The interaction with the file system and TDS is up to LUA. There still might be traces but eventually the code base should be lean and mean. The METAPOST library is coded in CWEB and in order to be independent from related tools, conversion to C is done with a LUA script ran by, surprise, LUAMETATEX.

3.2 Implementation notes

3.2.1 Memory allocation

The single internal memory heap that traditional TEX used for tokens and nodes is split into two separate arrays. Each of these will grow dynamically when needed. Internally a token or node is an index into these arrays. This permits for an efficient implementation and is also responsible for the performance of the core. The original documentation in TEX The Program mostly applies!

3.2.2 Sparse arrays

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

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

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.

3.2.3 Simple single-character csnames

Single-character commands are no longer treated specially in the internals, they are stored in the hash just like the multiletter 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.

3.2.4 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). So far we never had issues with this approach.

3.2.5 Tabs and spaces

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

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

3.2.6 Logging

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

In a similar fashion, there is a bit different behaviour when TEX expects input, which in turn is a side effect of removing the interception of * and & which made for cleaner code (quite a bit had accumulated as side effect of continuous adaptations in the TEX ecosystems). There was already code that was never executed, simply as side effect of the way LUATEX initializes itself (one needs to enable classes of primitives for instance).

4 Using LuaMetaTeX

4.1 Initialization

4.1.1 LuaMetaTeX as a Lua interpreter

Although LuaMetaTeX is primarily meant as a TeX engine, it can also serve as a stand alone Lua interpreter. There are two ways to make LuaMetaTeX behave like a standalone Lua interpreter:

- if a `--luaonly` option is given on the commandline, or
- if the only non-option argument (file) on the commandline has the extension `lua` or `luc`.

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

LuaMetaTeX will exit immediately after executing the specified Lua script and is, in effect, a somewhat bulky stand alone Lua interpreter with a bunch of extra preloaded libraries.

When no argument is given, LuaMetaTeX will look for a Lua file with the same name as the binary and run that one when present. This makes it possible to use the engine as a stub. For instance, in ConTeXt a symlink from `mtxrun` to type luametatex will run the `mtxrun.lua` script when present in the same path as the binary itself

4.1.2 Other commandline processing

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

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

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

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.

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

Command line processing happens very early on. So early, in fact, that none of TEX's initializations have taken place yet. The LUA libraries that don't deal with TEX are initialized early.

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

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

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

These names are initialized according to `--luaonly` or the first filename seen in the list of options. Special treatment of & and * as well as interactive startup is gone.

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

When running on MS WINDOWS the command line, filenames, environment variable access etc. internally uses the current code page but to the user is exposed as UTF8. Normally users won't notice this.

4.2 LUA behaviour

4.2.1 The LUA version

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

LUAs `tostring` function (and `string.format`) may return values in scientific notation, thereby confusing the TEX end of things when it is used as the right-hand side of an assignment to a

`\dimen` or `\count`. The output of these serializers also depend on the LUA version, so in LUA 5.3 you can get different output than from 5.2. It is best not to depend the automatic cast from string to number and vise versa as this can change in future versions.

4.2.2 Locales

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

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

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

4.3 LUA modules

Of course the regular LUA modules are present. In addition we provide the `lpeg` library by Roberto Ierusalimschy, This library is not UNICODE-aware, but interprets strings on a byte-per-byte basis. This mainly means that `lpeg.S` cannot be used with UTF8 characters that need more than one byte, and thus `lpeg.S` will look for one of those two bytes when matching, not the combination of the two. The same is true for `lpeg.R`, although the latter will display an error message if used with multibyte characters. Therefore `lpeg.R('aä')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, `ä` is two 'characters' (bytes), so `aä` totals three. In practice this is no real issue and with some care you can deal with UNICODE just fine.

There are some more libraries present. These are discussed on a later chapter. For instance we embed `luasocket` but contrary to LUATEX don't embed the related LUA code. An adapted version of `luafilesystem` is also included. There is a more extensive math library and there are libraries that deal with encryption and compression.

4.4 Testing

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

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

5 Basic TEX enhancements

5.1 Introduction

5.1.1 Primitive behaviour

From day one, LUATEX has offered extra features compared to the superset of PDFTEX, which includes ε-TEX, and ALEPH. This has not been limited to the possibility to execute LUA code via `\directlua`, but LUATEX also adds functionality via new TEX-side primitives or extensions to existing ones. The same is true for LUAMETATEX. Some primitives have `luatex` in their name and there will be no `luametatex` variants. This is because we consider LUAMETATEX to be LUATEX2^{+}.

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

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

The `extraprimitives` function returns the whole list or a subset, specified by one or more keywords `core`, `tex`, `etex` or `luatex`.[1].

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 11.3.15. There are only three kinds of primitives: `tex`, `etex` and `luatex` but a future version might drop this and no longer make that distinction as it no longer serves a purpose.

5.1.2 Experiments

There are a few extensions to the engine regarding the macro machinery. Some are already well tested but others are (still) experimental. Although they are likely to stay, their exact behaviour might evolve. Because LUAMETATEX is also used for experiments, this is not a problem. We can always decide to also add some of what is discussed here to LUATEX, but it will happen with a delay.

There are all kinds of small improvements that might find their way into stock LUATEX: a few more helpers, some cleanup of code, etc. We'll see. In any case, if you play with these before they are declared stable, unexpected side effects are what you have to accept.

[1] At some point this function might be changed to return the whole list always

5.1.3 Version information

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

There are three primitives to test the version of LUATEX (and LUAMETATEX):

PRIMITIVE	VALUE	EXPLANATION
`\luatexbanner`	This is LuaMetaTeX, Version 2.06.17	the banner reported on the command line
`\luatexversion`	206	a combination of major and minor number
`\luatexrevision`	17	the revision number

A version is defined as follows:

- The major version is the integer result of `\luatexversion` divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with `\the` or `\number` depending on the context.
- The minor version is a number running from 0 upto 99.
- The revision is reported by `\luatexrevision`. Contrary to other engines in LUAMETATEX is also a number so one needs to prefix it with `\the` or `\number`.[2]
- The full version number consists of the major version (`X`), minor version (`YY`) and revision (`ZZ`), separated by dots, so `X.YY.ZZ`.

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

5.1.3.2 `\formatname`

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

5.2 UNICODE text support

5.2.1 Extended ranges

Text input and output is now considered to be UNICODE text, so input characters can use the full range of UNICODE ($2^{20} + 2^{16} - 1 = $ 0x10FFFF). Later chapters will talk of characters and glyphs. Although these are not interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific

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

font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside the engine there is no clear separation between the two concepts. Because the subtype of a glyph node can be changed in LUA it is up to the user. Subtypes larger than 255 indicate that font processing has happened.

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

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

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

Contrary to other TEX engines, the output to the terminal is as-is so there is no escaping with `^^`. We operate in a UTF universe.

5.2.2 `\Uchar`

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

5.2.3 Extended tables

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

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

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

5.3 Attributes

5.3.1 Nodes

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

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

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

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

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

5.3.2 Attribute registers

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

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

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

Attributes can be used as extra counter values, but their usefulness comes mostly from the fact that the numbers and values of all 'set' attributes are attached to all nodes created in their scope. These can then be queried from any LUA code that deals with node processing. Further information about how to use attributes for node list processing from LUA is given in chapter 9.

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

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

5.3.3 Box attributes

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

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

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

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

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

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

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

```
for b=0,2,2 do
  for a=997, 999 do
```

```
    tex.sprint("box ", b, " : attr ",a," : ",tostring(tex.box[b]     [a]))
    tex.sprint("\\quad\\quad")
    tex.sprint("list ",b, " : attr ",a," : ",tostring(tex.box[b].list[a]))
    tex.sprint("\\par")
  end
end
```

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

```
box 0 : attr 997 : 123   list 0 : attr 997 : 123
box 0 : attr 998 : 456   list 0 : attr 998 : 456
box 0 : attr 999 : nil   list 0 : attr 999 : nil
box 2 : attr 997 : 123   list 2 : attr 997 : 123
box 2 : attr 998 : nil   list 2 : attr 998 : 456
box 2 : attr 999 : 789   list 2 : attr 999 : nil
```

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

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

5.4 LUA related primitives

5.4.1 \directlua

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

`\directlua` ⟨general text⟩

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

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

```
\begingroup
\endlinechar=10
```

```
\directlua ...
\endgroup
```

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

The `\directlua` command is expandable. Since it passes LUA code to the LUA interpreter its expansion from the TEX viewpoint is usually empty. However, there are some LUA functions that produce material to be read by TEX, the so called print functions. The most simple use of these is `tex.print(<string> s)`. The characters of the string `s` will be placed on the TEX input buffer, that is, 'before TEX's eyes' to be read by TEX immediately. For example:

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

expands to

a25b

Here is another example:

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

will result in

$\pi = 3.14159265358 98$

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 ε-TEX's `\scantokens`. For a description of print functions look at section 11.3.13.

Because the ⟨general text⟩ is a chunk, the normal LUA error handling is triggered if there is a problem in the included code. The LUA error messages should be clear enough, but the contextual information is 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 LUAMETATEX 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.

5.4.2 `\luaescapestring`

This primitive converts a TEX token sequence so that it can be safely used as the contents of a LUA string: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `n` and `r` respectively. The token sequence is fully expanded.

`\luaescapestring` ⟨general text⟩

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

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

5.4.3 \luafunction, \luafunctioncall and \luadef

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

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

\luafunction1
\luafunction2
```

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

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

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

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

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

There are another three (still experimental) primitives that behave like `\luafunction` but they expect the function to return an integer, dimension (also an integer) or a gluespec node. The return values gets injected into the input.

```
\luacountfunction 997 123
\luadimenfunction 998 123pt
\luaskipfunction  999 123pt plus 10pt minus 20pt
```

Examples of function 997 in the above lines are:

```
function() return token.scan_int() end
function() return 1234 end
```

This itself is not spectacular so there is more. These functions can be called in two modes: either TEX is expecting a value, or it is not and just expanding the call.

```
local n = 0
function(slot,scanning)
    if scanning then
        return n
    else
        n = token.scan_int()
    end
end
```

So, assuming that the function is in slot 997, you can do this:

```
\luacountfunction 997 123
\count100=\luacountfunction 997
```

After which `\count 100` has the value `123`.

5.4.4 `\luabytecode` and `\luabytecodecall`

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

```
\directlua {
    lua.bytecode[9998] = function(s)
        tex.sprint(s*token.scan_int())
    end
    lua.bytecode[5555] = function(s)
        tex.sprint(s*token.scan_dimen())
    end
}
```

This works with:

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

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

5.5 Catcode tables

5.5.1 Catcodes

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

5.5.2 `\catcodetable`

`\catcodetable` ⟨15-bit number⟩

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

5.5.3 `\initcatcodetable`

`\initcatcodetable` ⟨15-bit number⟩

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

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

5.5.4 `\savecatcodetable`

`\savecatcodetable` ⟨15-bit number⟩

`\savecatcodetable` copies the current set of catcodes to a new table with the requested number. The definitions in this new table are all treated as if they were made in the outermost level.

The new table is allocated globally: it will not go away after the current group has ended. If the supplied number is the currently active table, an error is raised.

5.6 Tokens, commands and strings

5.6.1 `\scantextokens`

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

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

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

Instead of:

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

you can use:

```
\etoksapp0{foo}
```

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

5.6.3 `\csstring`, `\begincsname` and `\lastnamedcs`

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

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

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

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

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

This is slightly more efficient than constructing the string twice (deep down in LuaTeX this also involves some UTF8 juggling), but probably more relevant is that it saves a few tokens and can make code a bit more readable.

5.6.4 \clearmarks

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

`\clearmarks` ⟨16-bit number⟩

5.6.5 \alignmark and \aligntab

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

5.6.6 \letcharcode

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

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

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

5.6.7 \glet

This primitive is similar to:

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

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

5.6.8 \expanded, \immediateassignment and \immediateassigned

The `\expanded` primitive takes a token list and expands its content which can come in handy: it avoids a tricky mix of `\expandafter` and `\noexpand`. You can compare it with what happens inside the body of an `\edef`. But this kind of expansion 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{abc}{def}{yes}{nop}/%
    \expandeddoifelse{abc}{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).

5.6.9 `\ignorepars`

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

5.6.10 `\futureexpand`, `\futureexpandis`, `\futureexpandisap`

These commands are used as:

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

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

5.6.11 `\aftergrouped`

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

```
{
    \aftergroup A \aftergroup B \aftergroup C
test 1 : }

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

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

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

```
test 4 : }
```

This gives:

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

5.7 Conditions

5.7.1 \ifabsnum and \ifabsdim

There are two tests that we took from PDFTEX:

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

This gives

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

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

New are the ones that compare two numbers or dimensions:

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

```
less equal more
```

and

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

```
less equal more
```

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

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

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

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

```
< = > !
```

```
< = > !
```

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

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

```
okay okay okay bad
```

```
okay okay okay bad
```

5.7.3 \ifmathstyle and \ifmathparameter

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

5.7.4 \ifempty

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

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

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

5.7.5 \ifboolean

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

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

5.7.6 \iftok and \ifcstok

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

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

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

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

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

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

5.7.7 \ifcondition

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

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

\ifcondition\something{a}{b}%
    \ifcondition\something{a}{a}%
        true 1
    \else
        false 1
    \fi
\else
    \ifcondition\something{a}{a}%
        true 2
```

```
    \else
        false 2
    \fi
\fi
```

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

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

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

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

```
non-zero zero
```

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

5.7.8 \orelse

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

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

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

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

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

you can do this:

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

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

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

5.7.9 `\ifprotected`, `\frozen`, `\iffrozen` and `\ifusercmd`

These checkers deal with control sequences. You can check if a command is a protected one, that is, defined with the `\protected` prefix. A command is frozen when it has been defined with the `\frozen` prefix. Beware: only macros can be frozen. A user command is a command that is not part of the predefined set of commands. This is an experimental command.

5.8 Boxes, rules and leaders

5.8.1 `\outputbox`

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

```
\outputbox = 12345
```

5.8.2 `\vpack`, `\hpack` and `\tpack`

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

5.8.3 `\vsplit`

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

5.8.4 Images and reused box objects

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

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

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

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

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

COMMAND	EXPLANATION
`\saveboxresource`	save the box as an object to be included later
`\saveimageresource`	save the image as an object to be included later

`\useboxresource`	include the saved box object here (by index)
`\useimageresource`	include the saved image object here (by index)
`\lastsavedboxresourceindex`	the index of the last saved box object
`\lastsavedimageresourceindex`	the index of the last saved image object
`\lastsavedimageresourcepages`	the number of pages in the last saved image object

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

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

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

5.8.5 `\hpack`, `\vpack` and `\tpack`

These three primitives are the equivalents of `\hbox`, `\vbox` and `\vtop` but they don't trigger the packaging related callbacks. Of course one never know if content needs a treatment so using them should be done with care.

5.8.6 `\nohrule` and `\novrule`

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

5.8.7 `\gleaders`

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

5.9 Languages

5.9.1 `\hyphenationmin`

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

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

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

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

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

5.10 Control and debugging

5.10.1 Tracing

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

5.10.2 \lastnodetype, \lastnodesubtype, \currentiftype and \internalcodesmode.

The ε-TEX command \lastnodetype is limited to some nodes. When the parameter \internalcodesmode is set to a non-zero value the normal (internally used) numbers are reported. The same is true for \currentiftype, as we have more conditionals and also use a different order. The \lastnodesubtype is a bonus.

5.11 Files

5.11.1 File syntax

LUAMETATEX will accept a braced argument as a file name:

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

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

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

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

5.11.2 Writing to file

You can now open upto 127 files with `\openout`. When no file is open writes will go to the console and log. The `write` related primitives have to be implemented as part of a backend! As a consequence a system command is no longer possible but one can use `os.execute` to do the same.

5.12 Math

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

5.13 Fonts

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

5.14 Directions

5.14.1 Two directions

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

5.14.2 How it works

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

This is what happens:

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

This becomes stepwise:

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

And this:

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

becomes:

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

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

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

This becomes:

run RUN run run

Compare this to:

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

Which renders as:

run RUNrun run

So how do we deal with the next?

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

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

It gets typeset as:

run run RUNrun RUNrun run
run run runRUN runRUN run

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

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

and this time:

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

comes out as a properly spaced:

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

Anything more complex that this, like combination of skips and penalties, or kerns, should be handled in the input or macro package because there is no way we can predict the expected behaviour. In fact, the `\linedir` is just a convenience extra which could also have been implemented using node list parsing.

5.14.3 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.

	0	1

`pre {\textdirection 0 xxx} post`	`pre`
	`xxx post`
		`pre`
		`xxx`
		`post`

`pre {\textdirection 0 xxx }post`	`pre`
	`xxx`
	`post`
		`pre`
		`xxx`
		`post`

`pre{ \textdirection 0 xxx} post`	`pre`
	`xxx post`
		`pre`
		`xxx`
		`post`

`pre{ \textdirection 0 xxx }post`	`pre`
	`xxx`
	`post`
		`pre`
		`xxx`
		`post`

`pre { \textdirection 0 xxx } post`	`pre`
	`xxx`
	` post`
		`pre`
		`xxx`
		
		`post`

`pre {\textdirection 0\relax \space xxx} post`	`pre`
	` xxx post`
		`pre`
		
		`xxx`
		`post`

6 Fonts

6.1 Introduction

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

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

6.2 Defining fonts

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

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

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

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

The names and their internal remapping are:

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

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

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

KEY	TYPE	DESCRIPTION
width	number	width in sp (default 0)
height	number	height in sp (default 0)
depth	number	depth in sp (default 0)
italic	number	italic correction in sp (default 0)
top_accent	number	top accent alignment place in sp (default zero)
bot_accent	number	bottom accent alignment place, in sp (default zero)
left_protruding	number	left protruding factor (\lpcode)
right_protruding	number	right protruding factor (\rpcode)
expansion_factor	number	expansion factor (\efcode)
next	number	'next larger' character index
extensible	table	constituent parts of an extensible recipe
vert_variants	table	constituent parts of a vertical variant set
horiz_variants	table	constituent parts of a horizontal variant set
kerns	table	kerning information
ligatures	table	ligaturing information
mathkern	table	math cut-in specifications

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

```
[102] = {
    ["width"]  = 200250,
    ["height"] = 455111,
```

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

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

The values of `top_accent`, `bot_accent` and `mathkern` are used only for math accent and superscript placement, see page 89 in this manual for details. The values of `left_protruding` and `right_protruding` are used only when `\protrudechars` is non-zero. Whether or not `expansion_factor` is used depends on the font's global expansion settings, as well as on the value of `\adjustspacing`.

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

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

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

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

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

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

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

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

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

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

6.3 Virtual fonts

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

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

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

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

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

The `commands` array is a hash where each item is another small array, with the first entry representing a command and the extra items being the parameters to that command. The allowed commands and their arguments are:

COMMAND	ARGUMENTS	TYPE	DESCRIPTION
`font`	1	number	select a new font from the local `fonts` table
`char`	1	number	typeset this character number from the current font, and move right by the character's width
`node`	1	node	output this node (list), and move right by the width of this list
`slot`	2	2 numbers	a shortcut for the combination of a font and char command
`push`	0		save current position
`nop`	0		do nothing
`pop`	0		pop position
`rule`	2	2 numbers	output a rule $ht * wd$, and move right.
`down`	1	number	move down on the page
`right`	1	number	move right on the page
`special`	1	string	output a `\special` command
`pdf`	2	2 strings	output a PDF literal, the first string is one of `origin`, `page`, `text`, `font`, `direct` or `raw`; if you have one string only `origin` is assumed
`lua`	1	string, function	execute a Lua script when the glyph is embedded; in case of a function it gets the font id and character code passed
`image`	1	image	output an image (the argument can be either an <image> variable or an `image_spec` table)
`comment`	any	any	the arguments of this command are ignored

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

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

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

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

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

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

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

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

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

The fields mentioned above can be found in external fonts. It is good to keep in mind that we can extend this model, given that the backend knows what to do with it.

6.4 Additional TeX commands

6.4.1 Font syntax

LuaTeX will accept a braced argument as a font name:

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

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

6.4.2 `\fontid` and `\setfontid`

```
\fontid\font
```

This primitive expands into a number. The currently used font id is 29. Here are some more:[3]

STYLE	COMMAND	FONT ID
normal	`\tf`	29
bold	`\bf`	**38**
italic	`\it`	*59*
bold italic	`\bi`	***77***

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

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

6.4.3 `\noligs` and `\nokerns`

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by LuaTeX'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` ⟨integer⟩
`\nokerns` ⟨integer⟩

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.

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