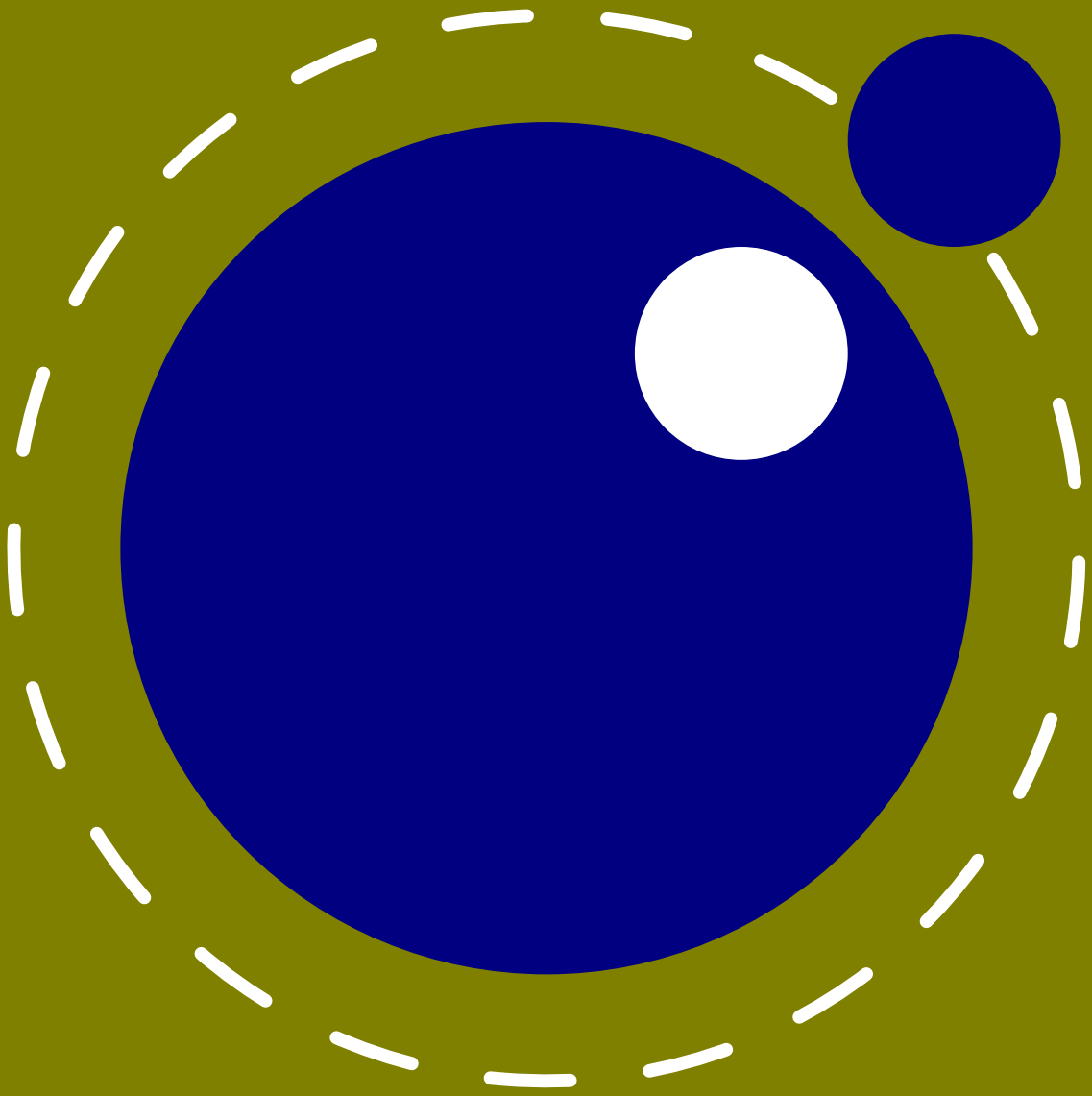
LuaMetaTEX Reference Manual



November 2020
Version 2.08.01

LuaMetaTEX Reference Manual

more info: www.luatex.org
: contextgarden.net
version : November 3, 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 follow up on LuaTeX. Just as we have ConTeXt MkII for pdfTeX and XeTeX, we have MkIV for LuaTeX. For LuaMetaTeX we have yet another version of ConTeXt: LMTX. By freezing MkII, and at soem point freezing MkIV, we can move on as we like, but we try to remain downward compatible where possible, something that the user interface makes possible. Although LuaMetaTeX can be used for production we can also use it for possibly drastic experiments but without affecting LuaTeX. Because we can easily adapt ConTeXt to support both, no other macro package will be harmed when (for instance) the interface that the engine provides change as part of an experiment or cleanup of code. Of course, when we consider something to be useful, it can be ported back to LuaTeX, but only when there are good reasons for doing so and when no compatibility issues are involved.

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

This manual started as an adaptation of the LuaTeX manual and therefore looks similar. Some chapters are removed, others were added and the rest has been (and will be further) adapted. It also discusses the (main) differences. Some of the new primitives or functions that show up in LuaMetaTeX might show up in LuaTeX at some point, but most will be exclusive to LuaMetaTeX,

so don't take this manual as reference for LuaTeX! As long as we're experimenting we can change things at will but as we keep ConTeXt LMTX synchronized users normally won't notice this. Often you can find examples of usage in ConTeXt related documents and the source code so that serves a reference too.

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

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

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

Hans Hagen

Version	:	November 3, 2020
LuaMetaTeX	:	luametatex 2.0801 / 20201103
ConTeXt	:	MkIV 2020.11.03 17:00
LuaTeX Team	:	Hans Hagen, Hartmut Henkel, Taco Hoekwater, Luigi Scarso

remark: LuaMetaTeX development is mostly done by Hans Hagen and in adapting the macros to the new features Wolfgang Schuster, who knows the code inside–out is a instrumental. In the initial phase Alan Braslau, who love playing with the three languages did extensive testing and compiled for several platforms. Later Mojca Miklavec make sure all compiles well on the buildbot infrastructure. After the first release more users got involved in testing. Many thanks for their patience! The development also triggered upgrading of the wiki support infrastructure where Taco Hoekwater and Paul Mazaitis have teamed up. So, progress all around.

remark: When there are non-intrusive features that also make sense in LuaTeX, these will be applied in the experimental branch first, so that there is no interference with the stable release. However, given that in the meantime the code bases differs a lot, it is unlikely that much will trickle back. This is no real problem as there's not much demand for that anyway.

remark: Most ConTeXt users seem always willing to keep up with the latest versions which means that LMTX is tested well. We can therefore safely claim that end of 2019 the code has become quite stable. There are no complaints about performance (on my 2013 laptop this manual

compiles at 24.5 pps with LMTX versus 20.7 pps for the LuaTeX manual with MkIV). 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 and not a tutorial. This means that we discuss changes relative to traditional TEX and also present new (or extended) functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later. Because the LUATEX and LUAMETATEX engines open up TEX there's suddenly quite some more to explain, especially about the way a (to be) typeset stream moves through the machinery. However, discussing all that in detail makes not much sense, because deep knowledge is only relevant for those who write code not possible with regular TEX and who are already familiar with these internals (or willing to spend time on figuring it out).

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

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

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

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

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

```
Hi There ...
```

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

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

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

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

Each character becomes a so called glyph node, a record with properties like the current font, the character code and the current language. Spaces become glue nodes. There are many node types 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:

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


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

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

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. Adding more callbacks makes no sense, also because at some point the performance hit gets too large. There are plenty ways to achieve one goals.

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 what commands relate to them. Here are a few:

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

Whatever we feed into TEX at some point becomes a token which is either interpreted directly or stored in a linked list. A token is just a number that encodes a specific command (operator) and

some value (operand) that further specifies what that command is supposed to do. In addition to an interface to nodes, there is an interface to tokens, as later chapters will demonstrate.

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

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

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

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

2 Differences with LuaTeX

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

fonts	`\letterspacefont` `\copyfont` `\expandglyphsinfont` `\ignoreligaturesinfont` `\tagcode` `\leftghost` `\rightghost`
backend	`\dviextension` `\dvivariable` `\dvifeedback` `\pdfextension` `\pdfvariable` `\pdffeedback` `\dviextension` `\draftmode` `\outputmode`
dimensions	`\pageleftoffset` `\pagerightoffset` `\pagetopoffset` `\pagebottomoffset` `\pageheight` `\pagewidth`
resources	`\saveboxresource` `\useboxresource` `\lastsavedboxresourceindex` `\saveimageresource` `\useimageresource` `\lastsavedimageresourceindex` `\lastsavedimageresourcepages`
positioning	`\savepos` `\lastxpos` `\lastypos`
directions	`\textdir` `\linedir` `\mathdir` `\pardir` `\pagedir` `\bodydir` `\pagedirection` `\bodydirection`
randomizer	`\randomseed` `\setrandomseed` `\normaldeviate` `\uniformdeviate`
utilities	`\synctex`
extensions	`\latelua` `\lateluafunction` `\openout` `\write` `\closeout` `\openin` `\read` `\readline` `\closein` `\ifeof`
control	`\suppressfontnotfounderror` `\suppresslongerror` `\suppressprimitiveerror` `\suppressmathparerror` `\suppressifcsnameerror` `\suppressoutererror` `\mathoption`
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. More extensive status information is provided in the overhauled status library.

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_format_file open_data_file
```

The callbacks related to errors are changed:

```
intercept_tex_error intercept_lua_error
show_error_message show_warning_message
```

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

```
trace_memory
```

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:

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

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

```
define_font
```

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

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.

```
automatichyphenmode
bodydir
bodydirection
boxdir
breakafterdirmode
closein
closeout
compoundhyphenmode
copyfont
draftmode
dviextension
dvifeedback
dvivariable
eTeXVersion
eTeXglueshrinkorder
eTeXgluestretchorder
eTeXminorversion
eTeXrevision
eTeXversion
expandglyphsinfont
fixupboxesmode
hoffset
hyphenationbounds
hyphenpenaltymode
ifeof
ifprimitive
ignoreligaturesinfont
immediateassigned
immediateassignment
lastsavedboxresourceindex
lastsavedimageresourceindex
lastsavedimageresourcepages
lastxpos
lastypos
latelua
lateluafunction
leftghost
letterspacefont
```


linedir
mag
mathdir
mathoption
nokerns
noligs
nolocaldirs
nolocalwhatsits
normaldeviate
openin
openout
outputmode
pagebottomoffset
pagedir
pagedirection
pageheight
pageleftoffset
pagerightoffset
pagetopoffset
pagewidth
pardir
pdfextension
pdffeedback
pdfvariable
primitive
randomseed
read
readline
rightghost
saveboxresource
saveimageresource
savepos
setrandomseed
shapemode
special
suppressfontnotfounderror
suppressifcsnameerror
suppresslongerror
suppressmathparerror
suppressoutererror
suppressprimitiveerror
synctex
tagcode
textdir
tracingscantokens
uniformdeviate
useboxresource
useimageresource
voffset
write

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

UUskewed
UUskewedwithdelims
Uabove
Uabovewithdelims
Uatop
Uatopwithdelims
Umathclass
Umathspacebeforescript
Umathspacingmode
Unosubprescript
Unosuperprescript
Uover
Uoverwithdelims
Ustyle
Usubprescript
Usuperprescript
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
afterassigned
aftergrouped
aliased
atendofgroup
atendofgrouped
automigrationmode
beginlocalcontrol
boxattribute
boxorientation
boxtotal
boxxmove
boxxoffset
boxymove
boxyoffset
defcsname

edefcsname
enforced
everytab
expand
expandafterpars
expandafterspaces
expandcstoken
expandtoken
fontspecifiedname
fontspecifiedsize
frozen
futuredef
futureexpand
futureexpandis
futureexpandisap
glyphdatafield
glyphoptions
glyphscriptfield
glyphstatefield
hyphenationmode
ifarguments
ifboolean
ifchkdim
ifchknum
ifcmpdim
ifcmpnum
ifcstok
ifdimval
ifempty
ifflags
ifhastok
ifhastoks
ifhasxtoks
ifmathparameter
ifmathstyle
ifnumval
ifparameter
iftok
ignorearguments
ignorepars
immutable
instance
integerdef
lastarguments
lastnodesubtype
letcsname
letfrozen
letprotected
linepar
localcontrol
localcontrolled
matholdmode
meaningfull
meaningless
mutable
noaligned
normalizelinemode
ordlimits
orelse
orunless
overloaded
overloadmode
overshoot
parattr
parfillleftskip
permanent
shownodedetails
snapshotpar
supmarkmode
swapcsvalues
thewithoutunit
todimension
tointeger
tokenized
tolerant
toscaled
tracingalignments
tracingmath
unhpack
unletfrozen
unletprotected
unvpack
wrapuppar

3 The original engines

3.1 The merged engines

3.1.1 The rationale

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

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

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

3.1.2 Changes from TeX 3.1415926

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

> The current code base is written in C, not Pascal. The original web documentation is kept when possible and not wrapped in tagged comments. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like `tex`, `lua`, `languages`, `fonts`, `libraries`, etc. There are some artifacts of the conversion to C, but these got (and get) removed stepwise. The documentation, which actually comes from the mix of engines (via so called change files), is 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 and level of abstraction were mostly kept. On the other hand, opening up had its impact on the code. See chapter 7 for many small changes related to paragraph building, language handling and hyphenation. The most important change is that adding a brace group in the middle of a

word (like in `of{}fice`) does not prevent ligature creation. Also, the hyphenation, ligature building and kerning has been split so that we can hook in alternative or extra code wherever we like. There are various options to control discretionary injection and related penalties are now integrated in these nodes. Language information is now bound to glyphs. The number of languages in LuaMetaTeX is smaller than in LuaTeX. Control over discretionaries is more granular and now managed by less variables.

There is no pool file, all strings are embedded during compilation. This also removed some memory constraints. We kept token and node memory management because it is convenient and efficient but parts were reimplemented in order to remove some constraints. Token memory management is largely the same. All the other large memory structures, like those related to nesting, the save stack, input levels, the hash table and table of equivalents, etc. now all start out small and are enlarged when needed, where maxima are controlled in the usual way. In principle the initial memory footprint is smaller while at the same time we can go real large.

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.

The math style related primitives can use numbers as well as symbolic names. There is some more (control over) math anyway, which is a side effect of supporting OpenType math.

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 have more nodes, conditionals, etc. the ε-TEX status related variables are adapted to LUAMETATEX: we use different 'constants', but that should be no problem because any sane macro package uses abstraction.
The `\scantokens` primitive is now using the same mechanism as LUA print-to-TEX uses, which simplifies the code. There is a little performance hit but it will not be noticed in CONTEXT, because we never use this primitive.
Because we don't use change files on top of original TEX, the integration of ε-TEX functionality is bit more natural, code wise.

3.1.4 Changes from PDFTEX 1.40

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

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

3.1.5 Changes from ALEPH RC4

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

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

Here is a summary of inherited functionality:

The ^^ notation has been extended: after ^^^^ four hexadecimal characters are expected and after ^^^^^^ six hexadecimal characters have to be given. The original TEX interpretation is still valid for the ^^ case but the four and six variants do no backtracking, i.e. when they are not followed by the right number of hexadecimal digits they issue an error message. Because ^^^ is a normal TEX case, we don't support the odd number of ^^^^^ either.

Glues *immediately after* direction change commands are not legal breakpoints. There is a bit more sanity testing for the direction state. This can be configured.

The placement of math formula numbers is direction aware and adapts accordingly. Boxes carry directional information but rules don't.

There are no direction related primitives for page and body directions. The paragraph, text and math directions are specified using primitives that take a number. The three letter codes are dropped.

3.1.6 Changes from standard WEB2C

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

3.2 Implementation notes

3.2.1 Memory allocation

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

3.2.2 Sparse arrays

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

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

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

3.2.3 Simple single-character csnames

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

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

3.2.4 Binary file reading

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

Setting a callback is expected now. Although reading input natively using `getc` calls is more efficient, we now fetch lines from LUA, put them in a buffer and then pick successive bytes (keep in mind that we read UTF) from that. The performance is quite ok, also because LUA is fast, todays operating systems cache, and storage media have become very fast. Also, TEX is spending more time messing around with what it has input than actually reading input.

3.2.5 Tabs and spaces

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

The previous behaviour was a known side effect and (as that kind of input normally comes from generated sources) it was normally dealt with by adding a comment token to the line in case the

spaces and/or tabs were intentional and to be kept. We are aware of the fact that this contradicts some of our other choices but consistency with other engines. We still stick to our view that at the log level we can (and might be) more incompatible. We already expose some more details anyway.

3.2.6 Logging

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

In a similar fashion, there is a bit different behaviour when TeX expects input, which in turn is a side effect of removing the interception of * and & which made for cleaner code (quite a bit had accumulated as side effect of continuous adaptations in the TeX ecosystems). There was already code that was never executed, simply as side effect of the way LuaTeX initializes itself (one needs to enable classes of primitives for instance). Keep in mind that over time system dependencies have been handles with TeX change files, the WEB2C infrastructure, KPSE features, compilation variables and flags, etc. In LuaMetaTeX we try to minimize all that.

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. But we really want to keep the binary small, if possible below the 3MB which is okay for a script engine.

When no argument is given, LuaMetaTeX will look for a Lua file with the same name as the binary and run that one when present. This makes it possible to use the engine as a stub. For instance, in ConTeXt a symlink from `mtxrun` to type luametatex will run the `mtxrun.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 what is called jobname is a bit complex. There can be explicit names set on the command line but when not set they can be taken from the `texconfig` table.

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

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

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

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

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

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

4.2 LUA behaviour

4.2.1 The LUA version

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

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

4.2.2 Locales

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

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

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

4.3 LUA modules

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

There are some more libraries present. These are discussed on a later chapter. For instance we embed `luasocket` but contrary to LUATEX don't embed the related LUA code. An adapted version of `luafilesystem` is also included. There are more extensive math libraries and there are libraries that deal with encryption and compression. At some point some of these might become so called optional libraries (read: the handful that we provide interfaces for but that get loaded on demand).

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`.[2].

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.

[2] 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 2.08.01	the banner reported on the console
\luatexversion	208	major and minor number combined
\luatexrevision	1	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.[3]
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 ($2^{20} + 2^{16} - 1 = 0$x10FFFF). Later chapters will talk of characters and glyphs. Although these are not interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside the engine there is no clear separation between the two concepts. Because

[3] 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.

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. Because we operate in a C universum, zero characters are special but because we also live in a UNICODE galaxy that is no real problem.

5.2.2 `\Uchar`

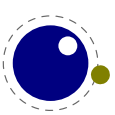
The expandable command `\Uchar` reads a number between 0 and 1,114,111 and expands to the associated UNICODE character.

5.2.3 Extended tables

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

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

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



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.

It is possible to change or add to the attributes assigned to a box:

```
\boxattr 0 123 456
```

5.4 LUA related primitives

5.4.1 \directlua

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

`\directlua` ⟨general text⟩

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

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

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

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

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

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

expands to

a25b

Here is another example:

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

will result in

$\pi = 3.1415926535898$

Note that the expansion of `\directlua` is a sequence of characters, not of tokens, contrary to all TEX commands. So formally speaking its expansion is null, but it collects material in a new level on the input stack to be immediately read by TEX after the LUA call as finished. It is a bit like ε-TEX's `\scantokens`, which now uses the same mechanism. For a description of print functions look at section 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 often suboptimal because it can come from deep down, and TEX has no knowledge about what you do in LUA. Often, you will only see the line number of the right brace at the end of the code.

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

5.4.2 `\luaescapestring`

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

\luaescapestring ⟨general text⟩

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

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

5.4.3 \luafunction, \luafunctioncall and \luadef

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

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

\luafunction1
\luafunction2
```

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

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

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

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

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

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`

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

`\catcodetable` ⟨15-bit number⟩

5.5.3 `\initcatcodetable`

`\initcatcodetable` ⟨15-bit number⟩

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

CATCODE	CHARACTER	EQUIVALENT	CATEGORY
0	\		escape

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

5.5.4 \savecatcodetable

\savecatcodetable ⟨15-bit number⟩

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

5.6 Tokens, commands and strings

5.6.1 \scantextokens and \tokenized

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

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

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

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

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

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

Instead of:

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

you can use:

```
\etoksapp0{foo}
```

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

5.6.3 \csstring, \begincsname and \lastnamedcs

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

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

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

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

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

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

5.6.4 \clearmarks

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 (no synchronization node is used).

`\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

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

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

5.6.9 \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` and `\orunless`

Sometimes you have successive tests that, when laid out in the source lead to deep trees. The `\ifcase` test is an exception. Experiments with `\ifcasex` worked out fine but eventually were

rejected because we have many tests so it would add a lot. As LuaMetaTeX permitted more experiments, eventually an alternative was cooked up, one that has some restrictions but is relative lightweight. It goes like this:

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

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

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

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

you can do this:

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

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

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


```
\fi
```

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

5.7.9 \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 \hrule and \vrule

Both rule drawing commands take an optional xoffset and yoffset parameter. The displacement is virtual and not taken into account when the dimensions are calculated.

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.

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

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

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

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

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

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

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

historic reasons, the only more or less standardized image inclusion interface one can expect to work in all macro packages.

5.8.5 \hpack, \vpack and \tpack

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

5.8.6 \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`

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

5.11 Files

5.11.1 File syntax

LUAMETATEX will accept a braced argument as a file name:

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

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

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 Normalizing lines

Experimental!

5.14.4 Orientations

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

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

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

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

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

5.15 Keywords

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

```
\hbox attr 123 456 attr 123 456 spread 10cm { }
```

```
\hrule width 10cm depth 3mm
\hskip 3pt plus 2pt minus 1pt
```

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

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

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

Here the order is not prescribed and, as demonstrated with the box specifier, for instance dimensions (specified by `to` or `spread` can be overloaded by later settings. In case you wonder if that breaks compatibility: in some way it does but bad or sloppy keyword usage breaks a run anyway. For instance `minuscule` results in `minus` with no dimension being seen. So, in the end the user should not noticed it and when a user does, the macro package already had an issue that had to be fixed.

5.16 Expressions

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

5.17 Nodes

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

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

6 Fonts

6.1 Introduction

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

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

6.2 Defining fonts

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

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

KEY	VALUE TYPE	DESCRIPTION
`name`	string	metric (file) name
`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 87 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:[4]

STYLE	COMMAND	FONT ID
normal	`\tf`	29
bold	`\bf`	**38**
italic	`\it`	*59*
bold italic	`\bi`	***71***

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.

[4] 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.

6.4.4 \nospaces

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

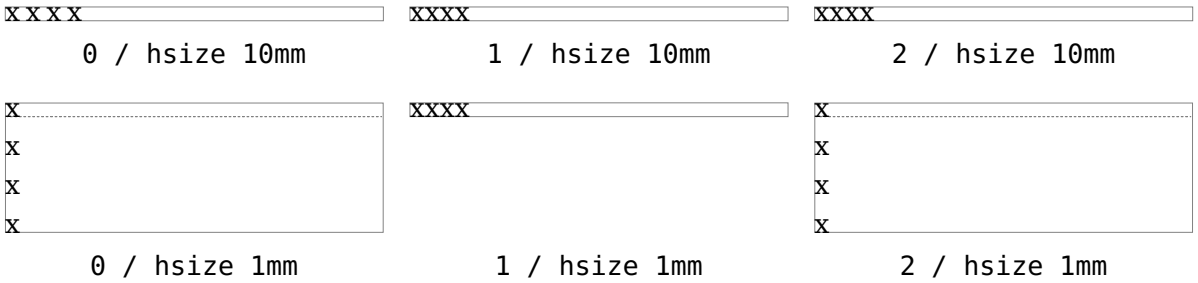


Figure 6.1 The \nospaces options.

6.4.5 \protrusionboundary

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

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

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

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

6.4.6 \glyphdimensionsmode

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

VALUE	EFFECT
0	the old behaviour: add the offset to the height and only subtract the offset only from the depth when it is positive

1	add the offset to the height and subtract it from the depth
2	add the offset to the height and subtract it from the depth but keep the maxima of the current and previous results
3	use the height and depth of the glyph, so no offset is applied

6.5 The LUA font library

6.5.1 Introduction

The LUA font library is reduced to a few commands. Contrary to LUATEX there is no loading of TFM or VF files. The explanation of the following commands is in the LUATEX manual.

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

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

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

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

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

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

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

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

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

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


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

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

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

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

6.5.4 Glyph data: `\glyphdata`, `\glyphscript`, `\glyphstate`

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

7 Languages, characters, fonts and glyphs

7.1 Introduction

LuaTeX's internal handling of the characters and glyphs that eventually become typeset is quite different from the way TeX82 handles those same objects. The easiest way to explain the difference is to focus on unrestricted horizontal mode (i.e. paragraphs) and hyphenation first. Later on, it will be easy to deal with the differences that occur in horizontal and math modes.

In TeX82, the characters you type are converted into `char` node records when they are encountered by the main control loop. TeX attaches and processes the font information while creating those records, so that the resulting 'horizontal list' contains the final forms of ligatures and implicit kerning. This packaging is needed because we may want to get the effective width of for instance a horizontal box.

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

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

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

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

7.2 Characters, glyphs and discretionaries

TeX82 (including pdfTeX) differentiates between `char` nodes and `lig` nodes. The former are simple items that contained nothing but a 'character' and a 'font' field, and they lived in the same memory as tokens did. The latter also contained a list of components, and a subtype indicating whether this ligature was the result of a word boundary, and it was stored in the same place as other nodes like boxes and kerns and glues. In LuaMetaTeX we no longer keep the list of components with the glyph node.

In LuaTeX, these two types are merged into one, somewhat larger structure called a `glyph` node. Besides having the old character, font, and component fields there are a few more, like 'attr' that we will see in section 9.2.12, these nodes also contain a subtype, that codes four main types and two additional ghost types. For ligatures, multiple bits can be set at the same time (in case of a single-glyph word).

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

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

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

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

Because the `\uchyph` value is saved in the actual nodes, its handling is subtly different from TeX82: changes to `\uchyph` become effective immediately, not at the end of the current partial paragraph.

Typeset boxes now always have their language information embedded in the nodes themselves, so there is no longer a possible dependency on the surrounding language settings. In TeX82, a mid-paragraph statement like `\unhbox0` would process the box using the current paragraph language unless there was a `\setlanguage` issued inside the box. In LuaTeX, all language variables are already frozen.

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

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

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

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

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

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

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

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

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

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

```
discrete---discrete
```

discrete—discrete

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

discrete
discrete

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

dis-
crete
discrete

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

dis-
crete
dis-
crete

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

dis-
crete—
dis-
crete

We only accept an explicit hyphen when there is a preceding glyph and we skip a sequence of explicit hyphens since that normally indicates a `--` or `---` ligature in which case we can in a worse case usage get bad node lists later on due to messed up ligature building as these dashes

are ligatures in base fonts. This is a side effect of separating the hyphenation, ligaturing and kerning steps.

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

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

The word start is determined as follows:

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

The word end is determined as follows:

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

Figures 7.1 upto 7.5 show some examples. In all cases we set the min values to 1 and make sure that the words hyphenate at each character.

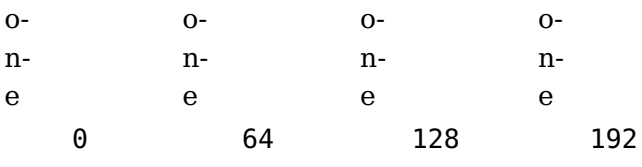


Figure 7.1 one

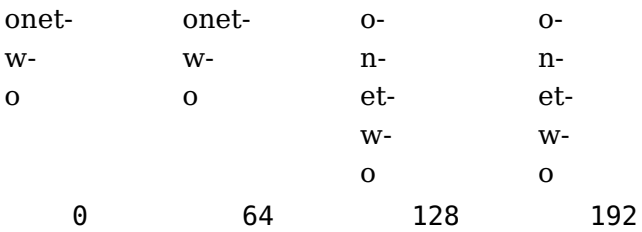


Figure 7.2 one\null two

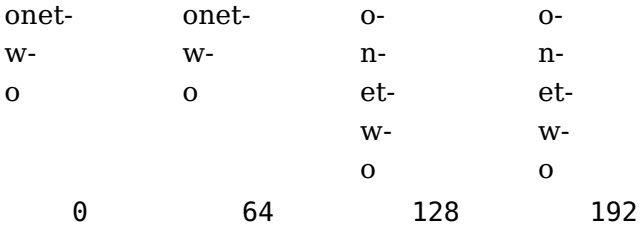


Figure 7.3 \null one\null two

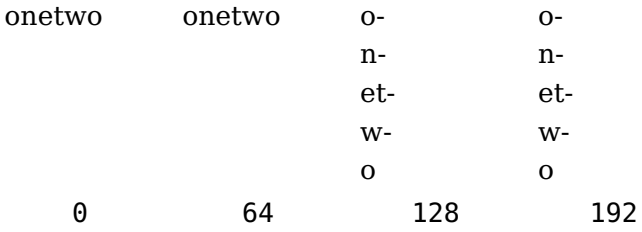


Figure 7.4 one\null two\null

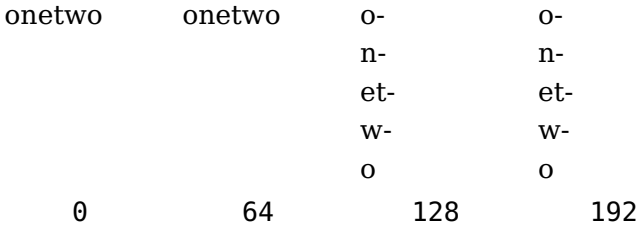


Figure 7.5 \null one\null two\null

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

Hyphenation and discretionary injection is driven by a mode parameter which is a bitset made from the following values, some of which we saw in the previous examples.

1	honour (normal) `\discretionary`'s
2	turn `-` into (automatic) discretionaries
4	turn `\-` into (explicit) discretionaries
8	hyphenate (syllable) according to language
16	hyphenate uppercase characters too (replaces `\uchyph`
32	permit break at an explicit hyphen (border cases)
64	traditional TEX compatibility wrt the start of a word
128	traditional TEX compatibility wrt the end of a word
256	use `\automatichyphenpenalty`
512	use `\explicithyphenpenalty`
1024	turn glue in discretionaries into kerns

7.3 The main control loop

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

1. The `\accent` primitive creates nodes with subtype 'glyph' instead of 'character': one for the actual accent and one for the accentee. The primary reason for this is that `\accent` in TEX82 is explicitly dependent on the current font encoding, so it would not make much sense to attach a new meaning to the primitive's name, as that would invalidate many old documents and macro packages. A secondary reason is that in TEX82, `\accent` prohibits hyphenation of the current word. Since in LUATEX hyphenation only takes place on 'character' nodes, it is possible to achieve the same effect. Of course, modern UNICODE aware macro packages will not use the `\accent` primitive at all but try to map directly on composed characters.
   This change of meaning did happen with `\char`, that now generates 'glyph' nodes with a character subtype. In traditional TEX there was a strong relationship between the 8-bit input encoding, hyphenation and glyphs taken from a font. In LUATEX we have UTF input, and in most cases this maps directly to a character in a font, apart from glyph replacement in the font engine. If you want to access arbitrary glyphs in a font directly you can always use LUA to do so, because fonts are available as LUA table.
2. All the results of processing in math mode eventually become nodes with 'glyph' subtypes. In fact, the result of processing math is just a regular list of glyphs, kerns, glue, penalties, boxes etc.
3. Automatic discretionaries are handled differently. TEX82 inserts an empty discretionary after sensing an input character that matches the `\hyphenchar` in the current font. This test is wrong in our opinion: whether or not hyphenation takes place should not depend on the current font, it is a language property.[5]
   In LUATEX, it works like this: if LUATEX senses a string of input characters that matches the value of the new integer parameter `\exhyphenchar`, it will insert an explicit discretionary after that series of nodes. Initially TEX sets the `\exhyphenchar=`\-`. Incidentally, this is a global parameter instead of a language-specific one because it may be useful to change the value depending on the document structure instead of the text language.

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

The insertion of discretionaries after a sequence of explicit hyphens happens at the same time as the other hyphenation processing, *not* inside the main control loop.
The only use LuaTeX has for `\hyphenchar` is at the check whether a word should be considered for hyphenation at all. If the `\hyphenchar` of the font attached to the first character node in a word is negative, then hyphenation of that word is abandoned immediately. This behaviour is added for backward compatibility only, and the use of `\hyphenchar=-1` as a means of preventing hyphenation should not be used in new LuaTeX documents.

4. The `\setlanguage` command no longer creates whatsits. The meaning of `\setlanguage` is changed so that it is now an integer parameter like all others. That integer parameter is used in `\glyph_node` creation to add language information to the glyph nodes. In conjunction, the `\language` primitive is extended so that it always also updates the value of `\setlanguage`.
5. The `\noboundary` command (that prohibits word boundary processing where that would normally take place) now does create nodes. These nodes are needed because the exact place of the `\noboundary` command in the input stream has to be retained until after the ligature and font processing stages.
6. There is no longer a `main_loop` label in the code. Remember that TeX82 did quite a lot of processing while adding `char_nodes` to the horizontal list? For speed reasons, it handled that processing code outside of the 'main control' loop, and only the first character of any 'word' was handled by that 'main control' loop. In LuaTeX, there is no longer a need for that (all hard work is done later), and the (now very small) bits of character-handling code have been moved back inline. When `\tracingcommands` is on, this is visible because the full word is reported, instead of just the initial character.

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

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

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

7.4 Loading patterns and exceptions

Although we keep the traditional approach towards hyphenation (which is still superior) the implementation of the hyphenation algorithm in LuaTeX is quite different from the one in TeX82.

After expansion, the argument for `\patterns` has to be proper UTF8 with individual patterns separated by spaces, no `\char` or `\chardefd` commands are allowed. The current implementation is quite strict and will reject all non-Unicode characters. Likewise, the expanded argument for `\hyphenation` also has to be proper UTF8, but here a bit of extra syntax is provided:

1. Three sets of arguments in curly braces (`{}{}{}`) indicate a desired complex discretionary, with arguments as in `\discretionary`'s command in normal document input.
2. A `-` indicates a desired simple discretionary, cf. `\-` and `\discretionary{-}{}{}` in normal document input.

3. Internal command names are ignored. This rule is provided especially for `\discretionary`, but it also helps to deal with `\relax` commands that may sneak in.
4. An = indicates a (non-discretionary) hyphen in the document input.

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

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

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

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

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

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

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

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

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

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

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

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

7.5 Applying hyphenation

The internal structures LuaTeX uses for the insertion of discretionaries in words is very different from the ones in TeX82, and that means there are some noticeable differences in handling as well.

First and foremost, there is no 'compressed trie' involved in hyphenation. The algorithm still reads pattern files generated by PATGEN, but LuaTeX uses a finite state hash to match the pat-

terns against the word to be hyphenated. This algorithm is based on the 'libhnj' library used by OPENOFFICE, which in turn is inspired by TEX.

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

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

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

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

This results in:

1:foobar

2:foobar

3:foo
bar

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

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

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

LuaTeX also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in TeX82). Longer words are ignored right now, but eventually either the limitation will be removed or perhaps it will become possible to silently ignore the excess characters (this is what happens in TeX82, but there the behaviour cannot be controlled).

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

7.6 Applying ligatures and kerning

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

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

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

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

initial	`{o}{f}{f}{i}{c}{e}`
after hyphenation	`{o}{f}{{-},{},{}}{f}{i}{c}{e}`
first ligature stage	`{o}{{f-},{f},{<ff>}}{i}{c}{e}`
final result	`{o}{{f-},{<fi>},{<ffi>}}{c}{e}`

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

```
{o}{{f-},
```


```
{{f-},
 {i},
 {<fi>}},
{{<ff>-},
 {i},
 {<ffi>}}}{c}{e}
```

with discretionaries in the post-break text as well as in the replacement text of the top-level discretionary that resulted from the first hyphenation point.

Here is that nested solution again, in a different representation:

	PRE		POST		REPLACE	
topdisc	f-	(1)	sub 1		sub 2	
sub 1	f-	(2)	i	(3)	<fi>	(4)
sub 2	<ff>-	(5)	i	(6)	<ffi>	(7)

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

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

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

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

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

FIELD	DESCRIPTION	
disc1.pre	f-	(1)
disc1.post	<fi>	(4)
disc1.replace	<ffi>	(7)
disc2.pre	f-	(2)
disc2.post	i	(3,6)
disc2.replace	<ff>-	(5)

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

```
{o}{{f-},
    {<fi>},
    {<ffi>}}
   {{f-},
    {i},
    {<ff>-}}{c}{e}
```

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

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

7.7 Breaking paragraphs into lines

This code is almost unchanged, but because of the above-mentioned changes with respect to discretionaries and ligatures, line breaking will potentially be different from traditional TeX. The actual line breaking code is still based on the TeX82 algorithms, and there can be no discretionaries inside of discretionaries. But, as patterns evolve and font handling can influence discretionaries, you need to be aware of the fact that long term consistency is not an engine matter only.

But that situation is now fairly common in LuaTeX, due to the changes to the ligaturing mechanism. And also, the LuaTeX discretionary nodes are implemented slightly different from the TeX82 nodes: the `no_break` text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list. In traditional TeX the discretionary node contains a counter indicating how many nodes to skip, but in LuaTeX we store the pre, post and replace text in the discretionary node.

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

7.8 The `language` library

7.8.1 `new` and `id`

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

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


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

This function creates a new userdata object. An object of type `<language>` is the first argument to most of the other functions in the `language` library. These functions can also be used as if they were object methods, using the colon syntax. Without an argument, the next available internal id number will be assigned to this object. With argument, an object will be created that links to the internal language with that id number.

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

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

7.8.2 hyphenation

You can load exceptions with:

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

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

7.8.3 clear_hyphenation and clean

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

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

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

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

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

7.8.4 patterns and clear_patterns

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

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

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

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

7.8.5 hyphenationmin

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

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

7.8.6 [pre|post][ex|]hyphenchar

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

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

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

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

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

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

7.8.7 hyphenate

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

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

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

7.8.8 [set|get]hjcode

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

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

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

8 Math

8.1 Traditional alongside OpenType

At this point there is no difference between LuaMetaTeX and LuaTeX with respect to math. The handling of mathematics in LuaTeX differs quite a bit from how TeX82 (and therefore pdfTeX) handles math. First, LuaTeX adds primitives and extends some others so that Unicode input can be used easily. Second, all of TeX82's internal special values (for example for operator spacing) have been made accessible and changeable via control sequences. Third, there are extensions that make it easier to use OpenType math fonts. And finally, there are some extensions that have been proposed or considered in the past that are now added to the engine.

8.2 Unicode math characters

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

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

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

The input for such primitives would look like this:

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

The altered TeX82 primitives are:

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

The unaltered ones are:

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

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

The following new primitives are compatible with XeTeX:

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

Specifications typically look like:

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

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

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

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

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

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

8.3 Math styles

8.3.1 \mathstyle

It is possible to discover the math style that will be used for a formula in an expandable fashion (while the math list is still being read). To make this possible, LuaTeX adds the new primitive:

\mathstyle. This is a 'convert command' like e.g. \romannumeral: its value can only be read, not set. Beware that contrary to LuaTeX this is now a proper number so you need to use \number o r\the in order to serialize it.

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

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

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

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

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


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

This gives:

$[a:0]$ $(\mathbf{x}:\mathbf{d}:\mathbf{4})$ $[b:0]$ $(\mathbf{y}:\mathbf{d}:\mathbf{4})$ $[c:0]$ $(\mathbf{z}:\mathbf{s}:\mathbf{6})$ $[d:0]$

$[a:2]$ $(\mathbf{x}:\mathbf{t}:\mathbf{6})$ $[b:2]$ $(\mathbf{y}:\mathbf{t}:\mathbf{6})$ $[c:2]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:2]$

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

$[a:0]$ $(\mathbf{x}:\mathbf{d}:\mathbf{4})$ $[b:0]$ $(\mathbf{y}:\mathbf{s}:\mathbf{6})$ $[c:0]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:0]$

$[a:2]$ $(\mathbf{x}:\mathbf{t}:\mathbf{6})$ $[b:2]$ $(\mathbf{y}:\mathbf{ss}:\mathbf{6})$ $[c:2]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:2]$

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

8.3.2 \Ustack

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

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

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

8.3.3 The new \cramped ...style commands

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

```
\crampeddisplaystyle
\crampedtextstyle
\crampedscriptstyle
\crampedscriptscriptstyle
```

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

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

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

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

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

Now we set the following parameters

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

This gives a different result:

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

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

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

Now we get:

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

8.4 Math parameter settings

8.4.1 Many new \Umath* primitives

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

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

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

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

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

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

8.4.2 Font-based math parameters

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

In the table below, the one-letter style abbreviations and symbolic tfm font dimension names match those used in the TeXbook. Assignments to `\textfont` set the values for the cramped and uncramped display and text styles, `\scriptfont` sets the script styles, and `\scriptscriptfont` sets the scriptscript styles, so we have eight parameters for three font sizes. In the TFM case, assignments only happen in family 2 and family 3 (and of course only for the parameters for which there are font dimensions).

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

VARIABLE / STYLE	TFM / OPENTYPE
`\Umathaxis`	`axis_height`

		AxisHeight
[6]	**\Umathoperatorsize**	–
	D, D’	DisplayOperatorMinHeight
[9]	**\Umathfractiondelsize**	delim1
	D, D’	FractionDelimiterDisplayStyleSize
[9]	**\Umathfractiondelsize**	delim2
	T, T’, S, S’, SS, SS’	FractionDelimiterSize
	\Umathfractiondenomdown	denom1
	D, D’	FractionDenominatorDisplayStyleShiftDown
	\Umathfractiondenomdown	denom2
	T, T’, S, S’, SS, SS’	FractionDenominatorShiftDown
	\Umathfractiondenomvgap	3*default_rule_thickness
	D, D’	FractionDenominatorDisplayStyleGapMin
	\Umathfractiondenomvgap	default_rule_thickness
	T, T’, S, S’, SS, SS’	FractionDenominatorGapMin
	\Umathfractionnumup	num1
	D, D’	FractionNumeratorDisplayStyleShiftUp
	\Umathfractionnumup	num2
	T, T’, S, S’, SS, SS’	FractionNumeratorShiftUp
	\Umathfractionnumvgap	3*default_rule_thickness
	D, D’	FractionNumeratorDisplayStyleGapMin
	\Umathfractionnumvgap	default_rule_thickness
	T, T’, S, S’, SS, SS’	FractionNumeratorGapMin
	\Umathfractionrule	default_rule_thickness
		FractionRuleThickness
	\Umathskewedfractionhgap	math_quad/2
		SkewedFractionHorizontalGap
	\Umathskewedfractionvgap	math_x_height
		SkewedFractionVerticalGap
	\Umathlimitabovebgap	big_op_spacing3
		UpperLimitBaselineRiseMin
[1]	**\Umathlimitabovekern**	big_op_spacing5
		0
	\Umathlimitabovevgap	big_op_spacing1
		UpperLimitGapMin
	\Umathlimitbelowbgap	big_op_spacing4
		LowerLimitBaselineDropMin
[1]	**\Umathlimitbelowkern**	big_op_spacing5
		0
	\Umathlimitbelowvgap	big_op_spacing2
		LowerLimitGapMin
	\Umathoverdelimitervgap	big_op_spacing1

	StretchStackGapBelowMin
\Umathoverdelimiterbgap	big_op_spacing3 StretchStackTopShiftUp
\Umathunderdelimitervgap	big_op_spacing2 StretchStackGapAboveMin
\Umathunderdelimiterbgap	big_op_spacing4 StretchStackBottomShiftDown
\Umathoverbarkern	default_rule_thickness OverbarExtraAscender
\Umathoverbarrule	default_rule_thickness OverbarRuleThickness
\Umathoverbarvgap	3*default_rule_thickness OverbarVerticalGap
[1] **\Umathquad**	math_quad <font_size(f)>
\Umathradicalkern	default_rule_thickness RadicalExtraAscender
[2] **\Umathradicalrule**	<not set> RadicalRuleThickness
[3] **\Umathradicalvgap** D, D'	default_rule_thickness+abs(math_x_height)/4 RadicalDisplayStyleVerticalGap
[3] **\Umathradicalvgap** T, T', S, S', SS, SS'	default_rule_thickness+abs(default_rule_thickness)/4 RadicalVerticalGap
[2] **\Umathradicaldegreebefore**	<not set> RadicalKernBeforeDegree
[2] **\Umathradicaldegreeafter**	<not set> RadicalKernAfterDegree
[2,7] **\Umathradicaldegreeraise**	<not set> RadicalDegreeBottomRaisePercent
[4] **\Umathspaceafterscript**	script_space SpaceAfterScript
\Umathstackdenomdown D, D'	denom1 StackBottomDisplayStyleShiftDown
\Umathstackdenomdown T, T', S, S', SS, SS'	denom2 StackBottomShiftDown
\Umathstacknumup D, D'	num1 StackTopDisplayStyleShiftUp
\Umathstacknumup T, T', S, S', SS, SS'	num3 StackTopShiftUp
\Umathstackvgap D, D'	7*default_rule_thickness StackDisplayStyleGapMin
\Umathstackvgap	3*default_rule_thickness

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

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

Note 2: Traditional TFM fonts do not set `\Umathradicalrule` because TEX82 uses the height of the radical instead. When this parameter is indeed not set when LUATEX has to typeset a radical, a backward compatibility mode will kick in that assumes that an oldstyle TEX font is used. Also, they do not set `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. These are then automatically initialized to 5/18quad, −10/18quad, and 60.

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

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

Note 5: Traditional TFM fonts set `\Umathconnectoroverlapmin` to zero because TeX82 always stacks extensibles without any overlap.

Note 6: The `\Umathoperatorsize` is only used in `\displaystyle`, and is only set in OpenType fonts. In TFM font mode, it is artificially set to one scaled point more than the initial attempt's size, so that always the 'first next' will be tried, just like in TeX82.

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

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

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

8.5 Math spacing

8.5.1 Setting inline surrounding space with `\mathsurround[skip]\mathsurround[skip]`

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

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

MODE	XXX	X X X	EFFECT
0	x x x	x x x	obey `\mathsurround` when `\mathsurroundskip` is 0pt
1	x xx	x x x	only add skip to the left
2	xx x	x x x	only add skip to the right
3	x x x	x x x	add skip to the left and right
4	x x x	x x x	ignore the skip setting, obey `\mathsurround`
5	xxx	x x x	disable all spacing around math
6	x x x	x x x	only apply `\mathsurroundskip` when also spacing
7	x x x	x x x	only apply `\mathsurroundskip` when no spacing

Anything more fancy, like checking the beginning or end of a paragraph (or edges of a box) would not be robust anyway. If you want that you can write a callback that runs over a list and analyzes a paragraph. Actually, in that case you could also inject glue (or set the properties of

a math node) explicitly. So, these modes are in practice mostly useful for special purposes and experiments (they originate in a tracker item). Keep in mind that this glue is part of the math node and not always treated as normal glue: it travels with the begin and end math nodes. Also, method 6 and 7 will zero the skip related fields in a node when applicable in the first occasion that checks them (linebreaking or packaging).

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

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

```
\Umathordordspacing
\Umathordopspacing
\Umathordbinspacing
\Umathordrelspacing
\Umathordopenspacing
\Umathordclosespacing
\Umathordpunctspacing
\Umathordinnerspacing
\Umathopordspacing
\Umathopopspacing
\Umathopbinspacing
\Umathoprelspacing
\Umathopopenspacing
\Umathopclosespacing
\Umathoppunctspacing
\Umathopinnerspacing
\Umathbinordspacing
\Umathbinopspacing
\Umathbinbinspacing
\Umathbinrelspacing
\Umathbinopenspacing
\Umathbinclosespacing
\Umathbinpunctspacing
\Umathbininnerspacing
\Umathrelordspacing
\Umathrelopspacing
\Umathrelbinspacing
\Umathrelrelspacing
\Umathrelopenspacing
\Umathrelclosespacing
\Umathrelpunctspacing
\Umathrelinnerspacing
\Umathopenordspacing
\Umathopenopspacing
\Umathopenbinspacing
\Umathopenrelspacing
\Umathopenopenspacing
\Umathopenclosespacing
\Umathopenpunctspacing
\Umathopeninnerspacing
\Umathcloseordspacing
\Umathcloseopspacing
\Umathclosebinspacing
\Umathcloserelspacing
\Umathcloseopenspacing
\Umathcloseclosespacing
\Umathclosepunctspacing
\Umathcloseinnerspacing
\Umathpunctordspacing
\Umathpunctopspacing
\Umathpunctbinspacing
\Umathpunctrelspacing
\Umathpunctopenspacing
\Umathpunctclosespacing
\Umathpunctpunctspacing
\Umathpunctinnerspacing
\Umathinnerordspacing
\Umathinneropspacing
\Umathinnerbinspacing
\Umathinnerrelspacing
\Umathinneropenspacing
\Umathinnerclosespacing
\Umathinnerpunctspacing
\Umathinnerinnerspacing
```

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

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

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

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

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

8.5.3 Local \frozen settings with

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

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

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

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

$$x_{x_{x_{x}}}\quad x_{x}\quad x_{x}\quad x$$

8.5.4 Checking a state with \ifmathparameter

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

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

8.5.5 Skips around display math and \mathdisplayskipmode

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

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

8.5.6 Nolimit correction with \mathnolimitsmode

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

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

When the mode is set to one, the math parameters are used. This way a macro package writer can decide what looks best. Given the current state of fonts in ConTeXt we currently use mode 1 with factor 0 for the superscript and 750 for the subscripts. Positive values are used for both

parameters but the subscript shifts to the left. A `\mathnolimitsmode` larger that 15 is considered to be a factor for the subscript correction. This feature can be handy when experimenting.

8.5.7 Controlling math italic mess with `\mathitalicsmode`

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

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

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

8.5.8 Influencing script kerning with `\mathscriptboxmode`

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

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

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

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

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

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

normal	modern	T_f	γ_e	γ_{ee}	T_{fluff}
	pagella	T_f	γ_e	γ_{ee}	T_{fluff}
	cambria	T_f	γ_e	γ_{ee}	T_{fluff}
	lucidaot	T_f	γ_e	γ_{ee}	T_{fluff}
bold	modern	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	pagella	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	cambria	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	lucidaot	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$

8.5.9 Forcing fixed scripts with \mathscriptsmode

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

PARAMETER	REGISTER
d	\Umathsubshiftdown
u	\Umathsupshiftup
s	\Umathsubsupshiftdown

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

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

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

8.5.10 Penalties: \mathpenaltiesmode

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

```
\mathpenaltiesmode = 1
```

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

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

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

8.5.11 Equation spacing: \matheqnogapstep

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

```
\matheqnogapstep = 1000
```

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

8.6 Math constructs

8.6.1 Unscaled fences and \mathdelimitersmode

The \mathdelimitersmode primitive is experimental and deals with the following (potential) problems. Three bits can be set. The first bit prevents an unwanted shift when the fence symbol is not scaled (a cambria side effect). The second bit forces italic correction between a preceding character ordinal and the fenced subformula, while the third bit turns that subformula into an ordinary so that the same spacing applies as with unfenced variants. Here we show Cambria (with \mathitalicsmode enabled).

```
\mathdelimitersmode = 0
\mathdelimitersmode = 1
\mathdelimitersmode = 2
\mathdelimitersmode = 3
\mathdelimitersmode = 4
\mathdelimitersmode = 5
\mathdelimitersmode = 6
\mathdelimitersmode = 7
```

So, when set to 7 fenced subformulas with unscaled delimiters come out the same as unfenced ones. This can be handy for cases where one is forced to use \left and \right always because of unpredictable content. As said, it's an experimental feature (which somehow fits in the exceptional way fences are dealt with in the engine). The full list of flags is given in the next table:

VALUE	MEANING
"01	don't apply the usual shift
"02	apply italic correction when possible
"04	force an ordinary subformula

"08	no shift when a base character
"10	only shift when an extensible

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

Sometimes you might want to act upon the size of a delimiter, something that is not really possible because of the fact that they are calculated *after* most has been typeset already. In the following example the all-zero specification is the trigger to make a fake box with the last delimiter dimensions and shift. It's an ugly hack but its relative simple and not intrusive implementation has no side effects. Any other heuristic solution would not satisfy possible demands anyway. Here is a rather low level example:

```
\startformula
\Uleft  \Udelimiter 5 0 "222B
\frac{\frac{a}{b}}{\frac{c}{d}}
\Uright \Udelimiter 5 0 "222B
\kern-2\fontcharwd\textfont0 "222B
\mathlimop{\Uvextensible \Udelimiter 0 0 0}_1^2 x
\stopformula
```

The last line, by passing zero values, results in a fake operator that has the dimensions of the previous delimiter. We can then backtrack over the (presumed) width and the two numbers become limit operators. As said, it's not pretty but it works.

$$\int_{\frac{c}{d}1}^{\frac{a}{b}2} x$$

8.6.2 Accent handling with \Umathaccent

LuaTeX supports both top accents and bottom accents in math mode, and math accents stretch automatically (if this is supported by the font the accent comes from, of course). Bottom and combined accents as well as fixed-width math accents are controlled by optional keywords following \Umathaccent.

The keyword bottom after \Umathaccent signals that a bottom accent is needed, and the keyword both signals that both a top and a bottom accent are needed (in this case two accents need to be specified, of course).

Then the set of three integers defining the accent is read. This set of integers can be prefixed by the fixed keyword to indicate that a non-stretching variant is requested (in case of both accents, this step is repeated).

A simple example:

```
\Umathaccent both fixed 0 0 "20D7 fixed 0 0 "20D7 {example}
```

If a math top accent has to be placed and the accentee is a character and has a non-zero top_accent value, then this value will be used to place the accent instead of the \skewchar kern used by TeX82.

The top_accent value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own top_accent line coincides with the one from the accentee. If the top_accent value of the accent is zero, then half the width of the accent followed by its italic correction is used instead.

The vertical placement of a top accent depends on the x_height of the font of the accentee (as explained in the TEXbook), but if a value turns out to be zero and the font had a MathConstants table, then AccentBaseHeight is used instead.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

Possible locations are top, bottom, both and center. When no location is given top is assumed. An additional parameter fraction can be specified followed by a number; a value of for instance 1200 means that the criterium is 1.2 times the width of the nucleus. The fraction only applies to the stepwise selected shapes and is mostly meant for the overlay location. It also works for the other locations but then it concerns the width.

8.6.3 Building radicals with \Uradical and \Uroot

The new primitive \Uroot allows the construction of a radical noad including a degree field. Its syntax is an extension of \Uradical:

```
\Uradical <fam integer> <char integer> <radicand>
\Uroot    <fam integer> <char integer> <degree> <radicand>
```

The placement of the degree is controlled by the math parameters \Umathradicaldegreebefore, \Umathradicaldegreeafter, and \Umathradicaldegreeraise. The degree will be typeset in \scriptscriptstyle.

8.6.4 Super- and subscripts

The character fields in a LUA-loaded OPENTYPE math font can have a 'mathkern' table. The format of this table is the same as the 'mathkern' table that is returned by the fontloader library, except that all height and kern values have to be specified in actual scaled points.

When a super- or subscript has to be placed next to a math item, LUATEX checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OPENTYPE fonts (as opposed to legacy TEX fonts), then LUATEX will use the OPENTYPE math algorithm for deciding on the horizontal placement of the super- or subscript.

This works as follows:

The vertical position of the script is calculated.
The default horizontal position is flat next to the base character.
For superscripts, the italic correction of the base character is added.
For a superscript, two vertical values are calculated: the bottom of the script (after shifting up), and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.

For each of these two locations:

find the math kern value at this height for the base (for a subscript placement, this is the bottom_right corner, for a superscript placement the top_right corner)

find the math kern value at this height for the script (for a subscript placement, this is the top_left corner, for a superscript placement the bottom_left corner)

add the found values together to get a preliminary result.

The horizontal kern to be applied is the smallest of the two results from previous step.

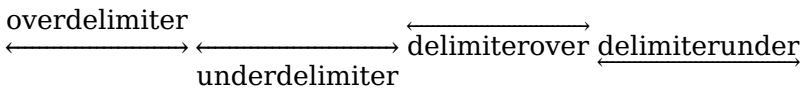
The math kern value at a specific height is the kern value that is specified by the next higher height and kern pair, or the highest one in the character (if there is no value high enough in the character), or simply zero (if the character has no math kern pairs at all).

8.6.5 Scripts on extensibles: \Uunderdelimiter, \Uoverdelimiter, \Udelimiterover, \Udelimiterunder and \Uhextensible

The primitives `\Uunderdelimiter` and `\Uoverdelimiter` allow the placement of a subscript or superscript on an automatically extensible item and `\Udelimiterunder` and `\Udelimiterover` allow the placement of an automatically extensible item as a subscript or superscript on a nucleus. The input:

```
$\Uoverdelimiter  0 "2194 {\hbox{\strut  overdelimiter}}$
$\Uunderdelimiter 0 "2194 {\hbox{\strut underdelimiter}}$
$\Udelimiterover  0 "2194 {\hbox{\strut  delimiterover}}$
$\Udelimiterunder 0 "2194 {\hbox{\strut delimiterunder}}$
```

will render this:



The vertical placements are controlled by `\Umathunderdelimiterbgap`, `\Umathunderdelimitervgap`, `\Umathoverdelimiterbgap`, and `\Umathoverdelimitervgap` in a similar way as limit placements on large operators. The superscript in `\Uoverdelimiter` is typeset in a suitable scripted style, the subscript in `\Uunderdelimiter` is cramped as well.

These primitives accepts an optional `width` specification. When used the also optional keywords `left`, `middle` and `right` will determine what happens when a requested size can't be met (which can happen when we step to successive larger variants).

An extra primitive `\Uhextensible` is available that can be used like this:

```
$\Uhextensible width 10cm 0 "2194$
```

This will render this:

Here you can also pass options, like:

```
$\Uhextensible width 1pt middle 0 "2194$
```

This gives:

↔

LuaTeX internally uses a structure that supports OpenType 'MathVariants' as well as TFM 'extensible recipes'. In most cases where font metrics are involved we have a different code path for traditional fonts end OpenType fonts.

8.6.6 Fractions and the new \Uskewed and \Uskewedwithdelims

The `\abovewithdelims` command accepts a keyword `exact`. When issued the extra space relative to the rule thickness is not added. One can of course use the `\Umathfraction..gap` commands to influence the spacing. Also the rule is still positioned around the math axis.

```
$$ { {a} \abovewithdelims() exact 4pt {b} }$$
```

The math parameter table contains some parameters that specify a horizontal and vertical gap for skewed fractions. Of course some guessing is needed in order to implement something that uses them. And so we now provide a primitive similar to the other fraction related ones but with a few options so that one can influence the rendering. Of course a user can also mess around a bit with the parameters `\Umathskewedfractionhgap` and `\Umathskewedfractionvgap`.

The syntax used here is:

```
{ {1} \Uskewed / <options> {2} }
{ {1} \Uskewedwithdelims / () <options> {2} }
```

where the options can be `noaxis` and `exact`. By default we add half the axis to the shifts and by default we zero the width of the middle character. For Latin Modern the result looks as follows:

	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact`	$x + {}^{a}\!/_{exactb} + x$	$x + {}^{1}\!/_{exact2} + x$	$x + ({}^{a}\!/_{exactb}) + x$	$x + ({}^{1}\!/_{exact2}) + x$
`noaxis`	$x + {}^{a}\!/_{noaxisb} + x$	$x + {}^{1}\!/_{noaxis2} + x$	$x + ({}^{a}\!/_{noaxisb}) + x$	$x + ({}^{1}\!/_{noaxis2}) + x$
`exact noaxis`	$x + {}^{a}\!/_{exactnoaxisb} + x$	$x + {}^{1}\!/_{exactnoaxis2} + x$	$x + ({}^{a}\!/_{exactnoaxisb}) + x$	$x + ({}^{1}\!/_{exactnoaxis2}) + x$

The `\over` and related primitives have the form:

```
{{top}\over{bottom}}
```

For convenience, which also avoids some of the trickery that makes this 'looking back' possible, the LuaMetaTeX also provides this variant:

```
\Uover{top}{bottom}
```

The optional arguments are also supported but we have one extra option: `style`. The style is applied to the numerator and denominator.

```
\Uover style \scriptstyle {top} {bottom}
```

The complete list of these commands is: `\Uabove`, `\Uatop`, `\Uover`, `\Uabovewithdelims`, `\Uatopwithdelims`, `\Uoverwithdelims`, `\UUskewed`, `\UUskewedwithdelims`. As with other extensions we use a leading `U` and because we already had extra skew related primitives we end up with a

UU there. This obscurity is not that big an issue because normally such primitives are wrapped in a macro. Here are a few examples:

```
$\Uover {                    1234} {                    5678} $\quad
$\Uover {\textstyle          1234} {\textstyle          5678} $\quad
$\Uover {\scriptstyle        1234} {\scriptstyle        5678} $\quad
$\Uover {\scriptscriptstyle 1234} {\scriptscriptstyle 5678} $\blank

$\Uover                           {1234} {5678} $\quad
$\Uover style \textstyle          {1234} {5678} $\quad
$\Uover style \scriptstyle        {1234} {5678} $\quad
$\Uover style \scriptscriptstyle {1234} {5678} $\blank
```

These render as: $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$

$\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$ $\frac{1234}{5678}$

8.6.7 Math styles: \Ustyle

This primitive accepts a style identifier:

```
\Ustyle \displaystyle
```

This in itself is not spectacular because it is equivalent to

```
\displaystyle
```

Both commands inject a style node and change the current style. However, as in other places where LuaMetaTeX expects a style you can also pass a number in the range zero upto seven (like the ones reported by the primitive `\mathstyle`). So, the next few lines give identical results:

Like: 0_7 0_7 0_7. Values outside the valid range are ignored.

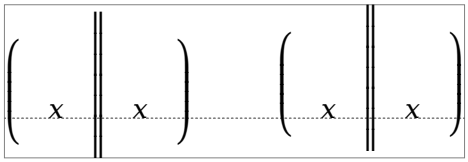
There is an extra option `norule` that can be used to suppress the rule while keeping the spacing compatible.

8.6.8 Delimiters: \Uleft, \Umiddle and \Uright

Normally you will force delimiters to certain sizes by putting an empty box or rule next to it. The resulting delimiter will either be a character from the stepwise size range or an extensible. The latter can be quite differently positioned than the characters as it depends on the fit as well as the fact whether the used characters in the font have depth or height. Commands like (plain TeXs) `\big` need to use this feature. In LuaTeX we provide a bit more control by three variants that support optional parameters `height`, `depth` and `axis`. The following example uses this:

```
\Uleft   height 30pt depth 10pt      \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt      \Udelimiter "0 "0 "002016
\quad x\quad
```

```
\Uright  height 30pt depth 10pt      \Udelimiter "0 "0 "000029
\quad \quad \quad
\Uleft   height 30pt depth 10pt axis \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt axis \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt axis \Udelimiter "0 "0 "000029
```



The keyword `exact` can be used as directive that the real dimensions should be applied when the criteria can't be met which can happen when we're still stepping through the successively larger variants. When no dimensions are given the `noaxis` command can be used to prevent shifting over the axis.

You can influence the final class with the keyword `class` which will influence the spacing. The numbers are the same as for character classes.

8.6.9 Accents: \mathlimitsmode

When you use `\limits` or `\nolimits` without scripts spacing might get messed up. This can be prevented by setting `\mathlimitsmode` to a non-zero value.

8.7 Extracting values

8.7.1 Codes and using \Umathcode, \Umathcharclass, \Umathcharfam and \Umathcharslot

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

```
\Umathcode 1 2 3 4
```

then

```
[\Umathcharclass1] [\Umathcharfam1] [\Umathcharslot1]
```

will return:

```
[2] [3] [4]
```

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

```
\def\Umathcharclass{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[1])}
```


```
\relax}
\def\Umathcharfam{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[2])}
\relax}
\def\Umathcharslot{\numexpr
    \directlua{tex.print(tex.getmathcode(token.scan_int())[3])}
\relax}
```

8.7.2 Last lines and \predisplaygapfactor

There is a new primitive to control the overshoot in the calculation of the previous line in mid-paragraph display math. The default value is 2 times the em width of the current font:

```
\predisplaygapfactor=2000
```

If you want to have the length of the last line independent of math i.e. you don't want to revert to a hack where you insert a fake display math formula in order to get the length of the last line, the following will often work too:

```
\def\lastlinelength{\dimexpr
    \directlua {tex.sprint (
        (nodes.dimensions(node.tail(tex.lists.page_head).list))
    )}sp
\relax}
```

8.8 Math mode

8.8.1 Verbose versions of single-character math commands like \Usuperscript and \Usubscript

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

PRIMITIVE	EXPLANATION
\Usuperscript	duplicates the functionality of ^
\Usubscript	duplicates the functionality of _
\Ustartmath	duplicates the functionality of $, when used in non-math mode.
\Ustopmath	duplicates the functionality of $, when used in inline math mode.
\Ustartdisplaymath	duplicates the functionality of $$, when used in non-math mode.
\Ustopdisplaymath	duplicates the functionality of $$, when used in display math mode.

The \Ustopmath and \Ustopdisplaymath primitives check if the current math mode is the correct one (inline vs. displayed), but you can freely intermix the four mathon/mathoff commands with explicit dollar sign(s).

8.8.2 Script commands \Unosuperscript and \Unosubscript

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

```
$
    x\Usuperscript  {1}\Usubscript  {2} =
    x\Unosuperscript{1}\Unosubscript{2} =
    x\Usuperscript  {1}\Unosubscript{2} =
    x\Unosuperscript{1}\Usubscript  {2}
$
```

results in $x^1_2 = x^1_2 = x^1_2 = x^1_2$.

8.8.3 Allowed math commands in non-math modes

The commands `\mathchar`, and `\Umathchar` and control sequences that are the result of `\mathchardef` or `\Umathchardef` are also acceptable in the horizontal and vertical modes. In those cases, the `\textfont` from the requested math family is used.

8.9 Goodies

8.9.1 Flattening: `\mathflattenmode`

The TEX math engine collapses `ord` noads without sub- and superscripts and a character as nucleus, which has the side effect that in OPENTYPE mode italic corrections are applied (given that they are enabled).

```
\switchtobodyfont[modern]
$V \mathbin{\mathbin{v}} V$\par
$V \mathord{\mathord{v}} V$\par
```

This renders as:

$V v\, V$
VvV

When we set `\mathflattenmode` to 31 we get:

$V\, v\, V$
VvV

When you see no difference, then the font probably has the proper character dimensions and no italic correction is needed. For Latin Modern (at least till 2018) there was a visual difference. In that respect this parameter is not always needed unless of course you want efficient math lists anyway.

You can influence flattening by adding the appropriate number to the value of the mode parameter. The default value is 1.

MODE	CLASS
1	ord

2	bin
4	rel
8	punct
16	inner

8.9.2 Less Tracing

Because there are quite some math related parameters and values, it is possible to limit tracing. Only when `tracingassigns` and/or `tracingrestores` are set to 2 or more they will be traced.

8.10 Experiments

There are a couple of experimental features. They will stay but details might change, for instance more control over spacing. We just show some examples and let your imagination work it out. First we have prescripts:

8.10.1 Prescripts with \Usuperprescript and Usubprescript

```
\hbox{$
    {\tf X}^1_2^^3__4 \quad
    {\tf X}^1  ^^3    \quad
    {\tf X}  _1   __4 \quad
    {\tf X}    ^^3    \quad
    {\tf X}       __4 \quad
    {\tf X}^^3    __4
$}
```

The question is: are these double super and subscript triggers the way to go? Anyway, you need to have them either being active (which in ConTeXt then boils down to them being other characters), or say `\supmarkmode = 1` to disable the normal multiple ^ treatment (a value larger than 1 will also disable that in text mode).

$^{3}_{4}X^{1}_{2}$ $^{3}X^{1}$ $_{4}X_{1}$ ^{3}X $_{4}X$ $^{3}_{4}X$

The more explicit commands are:

```
\hbox{$
{\tf X}\Usuperscript{1}                                          \quad
{\tf X}                \Usubscript{2}                            \quad
{\tf X}\Usuperscript{1}\Usubscript{2}                            \quad
{\tf X}\Usuperscript{1}             \Usuperprescript{3}          \quad
{\tf X}                \Usubscript{2}                 \Usubprescript{4}\quad
{\tf X}\Usuperscript{1}\Usubscript{2}\Usuperprescript{3}\Usubprescript{4}\quad
{\tf X}                             \Usuperprescript{3}          \quad
{\tf X}                                                \Usubprescript{4}\quad
{\tf X}                             \Usuperprescript{3}\Usubprescript{4}
```

```
$}
```

These more verbose triggers can be used to build interfaces:

$X^1 \quad X_2 \quad X_2^1 \quad {}^3X^1 \quad {}_4X_2 \quad {}_4^3X_2^1 \quad {}^3X \quad {}_4X \quad {}_4^3X$

8.10.2 Prescripts with \Usuperprescript and Usubprescript

You can change the class of a math character on the fly:

```
$x\mathopen   {!}+123+\mathclose  {!}x$
$x\Umathclass4 ! +123+\Umathclass5 ! x$
$x             ! +123+             ! x$
$x\mathclose  {!}+123+\mathopen   {!}x$
$x\Umathclass5 ! +123+\Umathclass4 ! x$
```

Watch how the spacing changes:

$x!+123+!x$
$x!+123+!x$
$x! + 123+!x$
$x! + 123 + !x$
$x! + 123 + !x$

9 Nodes

9.1 LUA node representation

TEX's nodes are represented in LUA as userdata objects with a variable set of fields or by a numeric identifier when requested. When you print a node userdata object you will see these numbers. In the following syntax tables the type of such a userdata object is represented as ⟨node⟩.

The return values of `node.types` are: `hlist` (0), `vlist` (1), `rule` (2), `insert` (3), `mark` (4), `adjust` (5), `boundary` (6), `disc` (7), `whatsit` (8), `par` (9), `dir` (10), `math` (11), `glue` (12), `kern` (13), `penalty` (14), `style` (15), `choice` (16), `parameter` (17), `noad` (18), `radical` (19), `fraction` (20), `accent` (21), `fence` (22), `math_char` (23), `math_text_char` (24), `sub_box` (25), `sub_mlist` (26), `delimiter` (27), `glyph` (28), `unset` (29), `attribute_list` (32), `attribute` (33), `glue_spec` (34), `temp` (35) and `split` (36)

In ε-TEX the `\lastnodetype` primitive has been introduced. With this primitive the valid range of numbers is still $[-1, 15]$ and glyph nodes (formerly known as char nodes) have number 0. That way macro packages can use the same symbolic names as in traditional ε-TEX. But you need to keep in mind that these ε-TEX node numbers are different from the real internal ones. When you set `\internalcodesmode` to a non-zero value, the internal codes will be used in the ε-TEX introspection commands `\lastnodetype` and `\currentiftype`.

You can ask for a list of fields with `node.fields` and for valid subtypes with `node.subtypes`. The `node.values` function reports some used values. Valid arguments are `glue`, `style` and `math`. Keep in mind that the setters normally expect a number, but this helper gives you a list of what numbers matter. For practical reason the `pagestate` values are also reported with this helper, but they are backend specific.

The return values of `node.values("glue")` are: `normal` (0), `fi` (1), `fil` (2), `fill` (3) and `filll` (4)

The return values of `node.values("style")` are: `display` (0), `crampeddisplay` (1), `text` (2), `crampedtext` (3), `script` (4), `crampedscript` (5), `scriptscript` (6) and `crampedscriptscript` (7)

The return values of `node.values("math")` are: `quad` (0), `axis` (1), `spacingmode` (2), `operatorsize` (3), `overbarkern` (4), `overbarrule` (5), `overbarvgap` (6), `underbarkern` (7), `underbarrule` (8), `underbarvgap` (9), `radicalkern` (10), `radicalrule` (11), `radicalvgap` (12), `radicaldegreebefore` (13), `radicaldegreeafter` (14), `radicaldegreeraise` (15), `stackvgap` (16), `stacknumup` (17), `stackdenomdown` (18), `fractionrule` (19), `fractionnumvgap` (20), `fractionnumup` (21), `fractiondenomvgap` (22), `fractiondenomdown` (23), `fractiondelsize` (24), `skewedfractionhgap` (25), `skewedfractionvgap` (26), `limitabovevgap` (27), `limitabovebgap` (28), `limitabovekern` (29), `limitbelowvgap` (30), `limitbelowbgap` (31), `limitbelowkern` (32), `nolimitsubfactor` (33), `nolimitsupfactor` (34), `underdelimitervgap` (35), `underdelimiterbgap` (36), `overdelimitervgap` (37), `overdelimiterbgap` (38), `subshiftdrop` (39), `supshiftdrop` (40), `subshiftdown` (41), `subsupshiftdown` (42), `subtopmax` (43), `supshiftup`

(44), supbottommin (45), supsubbottommax (46), subsupvgap (47), spacebeforescript (48), spaceafterscript (49), connectoroverlapmin (50), ordordspacing (51), ordopspacing (52), ordbinspacing (53), ordrelspacing (54), ordopenspacing (55), ordclosespacing (56), ordpunctspacing (57), ordinnerspacing (58), opordspacing (59), opopspacing (60), opbinspacing (61), oprelspacing (62), opopenspacing (63), opclosespacing (64), oppunctspacing (65), opinnerspacing (66), binordspacing (67), binopspacing (68), binbinspacing (69), binrelspacing (70), binopenspacing (71), binclosespacing (72), binpunctspacing (73), bininnerspacing (74), relordspacing (75), relopspacing (76), relbinspacing (77), relrelspacing (78), relopenspacing (79), relclosespacing (80), relpunctspacing (81), relinnerspacing (82), openordspacing (83), openopspacing (84), openbinspacing (85), openrelspacing (86), openopenspacing (87), openclosespacing (88), openpunctspacing (89), openinnerspacing (90), closeordspacing (91), closeopspacing (92), closebinspacing (93), closerelspacing (94), closeopenspacing (95), closeclosespacing (96), closepunctspacing (97), closeinnerspacing (98), punctordspacing (99), punctopspacing (100), punctbinspacing (101), punctrelspacing (102), punctopenspacing (103), punctclosespacing (104), punctpunctspacing (105), punctinnerspacing (106), innerordspacing (107), inneropspacing (108), innerbinspacing (109), innerrelspacing (110), inneropenspacing (111), innerclosespacing (112), innerpunctspacing (113) and innerinnerspacing (114)

The return values of `node.values("pagestate")` are:

9.2 Main text nodes

These are the nodes that comprise actual typesetting commands. A few fields are present in all nodes regardless of their type, these are:

FIELD	TYPE	EXPLANATION
next	node	the next node in a list, or nil
id	number	the node's type (id) number
subtype	number	the node subtype identifier

The subtype is sometimes just a dummy entry because not all nodes actually use the subtype, but this way you can be sure that all nodes accept it as a valid field name, and that is often handy in node list traversal. In the following tables next and id are not explicitly mentioned.

Besides these three fields, almost all nodes also have an attr field, and there is a also a field called prev. That last field is always present, but only initialized on explicit request: when the function node.slide is called, it will set up the prev fields to be a backwards pointer in the argument node list. By now most of TEX's node processing makes sure that the prev nodes are valid but there can be exceptions, especially when the internal magic uses a leading temp nodes to temporarily store a state.

The LUAMETATEX engine provides a lot of freedom and it is up to the user to make sure that the node lists remain sane. There are some safeguards but there can be cases where the engine just quits out of frustration. And, of course you can make the engine crash.

9.2.1 hlist and vlist nodes

These lists share fields and subtypes although some subtypes can only occur in horizontal lists while others are unique for vertical lists. The possible fields are attr, depth, direction, doffset, glue_order, glue_set, glue_sign, height, hoffset, list, orientation, shift, state, width, woffset, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	accent, alignment, box, cell, degree, denominator, equation, equationnumber, fraction, hdelimiter, hextensible, indent, limits, line, math, mathchar, nucleus, numerator, over, overdelimiter, radical, scripts, sub, sup, under, underdelimiter, unknown, vdelimiter and vextensible
attr	node	list of attributes
width	number	the width of the box
height	number	the height of the box
depth	number	the depth of the box
direction	number	the direction of this box, see 9.2.15
shift	number	a displacement perpendicular to the character (hlist) or line (vlist) progression direction
glue_order	number	a number in the range [0, 4], indicating the glue order
glue_set	number	the calculated glue ratio
glue_sign	number	0 = normal, 1 = stretching, 2 = shrinking
list	node	the first node of the body of this list

The orientation, woffset, hoffset, doffset, xoffset and yoffset fields are special. They can be used to make the backend rotate and shift boxes which can be handy in for instance vertical typesetting. Because they relate to (and depend on the) the backend they are not discussed here (yet).

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

Note: the field name head and list are both valid. Sometimes it makes more sense to refer to a list by head, sometimes list makes more sense.

9.2.2 rule nodes

Contrary to traditional TEX, LUATEX has more \rule subtypes because we also use rules to store reuseable objects and images. User nodes are invisible and can be intercepted by a callback. The supported fields are attr, data, depth, height, left, right, width, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	box, empty, fraction, image, normal, outline, over, radical, under and user
attr	node	list of attributes
width	number	the width of the rule where the special value −1073741824 is used for 'running' glue dimensions

height	number	the height of the rule (can be negative)
depth	number	the depth of the rule (can be negative)
left	number	shift at the left end (also subtracted from width)
right	number	(subtracted from width)
dir	string	the direction of this rule, see 9.2.15
index	number	an optional index that can be referred to
transform	number	an private variable (also used to specify outline width)

The left and type right keys are somewhat special (and experimental). When rules are auto adapting to the surrounding box width you can enforce a shift to the right by setting left. The value is also subtracted from the width which can be a value set by the engine itself and is not entirely under user control. The right is also subtracted from the width. It all happens in the backend so these are not affecting the calculations in the frontend (actually the auto settings also happen in the backend). For a vertical rule left affects the height and right affects the depth. There is no matching interface at the TEX end (although we can have more keywords for rules it would complicate matters and introduce a speed penalty.) However, you can just construct a rule node with LUA and write it to the TEX input. The outline subtype is just a convenient variant and the transform field specifies the width of the outline.

The xoffset and yoffset fields are special. They can be used to shift rules. Because they relate to (and depend on the) the backend they are not discussed here (yet).

9.2.3 insert nodes

This node relates to the \insert primitive and support the fields: attr, cost, depth, height, list and spec.

FIELD	TYPE	EXPLANATION
subtype	number	the insertion class
attr	node	list of attributes
cost	number	the penalty associated with this insert
height	number	height of the insert
depth	number	depth of the insert
list	node	the first node of the body of this insert

There is a set of extra fields that concern the associated glue: width, stretch, stretch_order, shrink and shrink_order. These are all numbers.

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may result. You can use list instead (often in functions you want to use local variable with similar names and both names are equally sensible).

9.2.4 mark nodes

This one relates to the \mark primitive and only has a few fields: attr, class and mark.

FIELD	TYPE	EXPLANATION
subtype	number	unused

attr	node	list of attributes
class	number	the mark class
mark	table	a table representing a token list

9.2.5 adjust nodes

This node comes from \vadjust primitive and has fields: attr and list.

FIELD	TYPE	EXPLANATION
subtype	number	normal and pre
attr	node	list of attributes
list	node	adjusted material

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

9.2.6 disc nodes

The \discretionary and \-, the - character but also the hyphenation mechanism produces these nodes. The available fields are: attr, options, penalty, post, pre and replace.

FIELD	TYPE	EXPLANATION
subtype	number	automatic, discretionary, explicit, first, regular and second
attr	node	list of attributes
pre	node	pointer to the pre-break text
post	node	pointer to the post-break text
replace	node	pointer to the no-break text
penalty	number	the penalty associated with the break, normally \hyphenpenalty or \exhyphenpenalty

The subtype numbers 4 and 5 belong to the 'of-f-ice' explanation given elsewhere. These disc nodes are kind of special as at some point they also keep information about breakpoints and nested ligatures.

The pre, post and replace fields at the LUA end are in fact indirectly accessed and have a prev pointer that is not nil. This means that when you mess around with the head of these (three) lists, you also need to reassign them because that will restore the proper prev pointer, so:

```
pre = d.pre
-- change the list starting with pre
d.pre = pre
```

Otherwise you can end up with an invalid internal perception of reality and LUAMETATEX might even decide to crash on you. It also means that running forward over for instance pre is ok but backward you need to stop at pre. And you definitely must not mess with the node that prev points to, if only because it is not really a node but part of the disc data structure (so freeing it again might crash LUAMETATEX).

9.2.7 math nodes

Math nodes represent the boundaries of a math formula, normally wrapped into $ signs. The following fields are available: attr, shrink, shrink_order, stretch, stretch_order, surround and width.

FIELD	TYPE	EXPLANATION
subtype	number	beginmath and endmath
attr	node	list of attributes
surround	number	width of the \mathsurround kern
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

The glue fields only kick in when the surround fields is zero.

9.2.8 glue nodes

Skips are about the only type of data objects in traditional TEX that are not a simple value. They are inserted when TEX sees a space in the text flow but also by \hskip and \vskip. The structure that represents the glue components of a skip internally is called a glue_spec. In LUAMETATEX we don't use the spec itself but just its values. A glue node has the fields: attr, font, leader, shrink, shrink_order, stretch, stretch_order and width.

FIELD	TYPE	EXPLANATION
subtype	number	abovedisplayshortskip, abovedisplayskip, baselineskip, belowdisplayshortskip, belowdisplayskip, cleaders, conditionalmathskip, correctionskip, gleaders, indentskip, intermathskip, leaders, lefthangskip, leftskip, lineskip, mathskip, medmuskip, muglue, parfillleftskip, parfillskip, parskip, righthangskip, rightskip, spaceskip, splittopskip, tabskip, thickmuskip, thinmuskip, topskip, userskip, xleaders, xspaceskip and zerospaceskip
attr	node	list of attributes
leader	node	pointer to a box or rule for leaders
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

Note that we use the key width in both horizontal and vertical glue. This suits the TEX internals well so we decided to stick to that naming.

The effective width of some glue subtypes depends on the stretch or shrink needed to make the encapsulating box fit its dimensions. For instance, in a paragraph lines normally have glue

representing spaces and these stretch or shrink to make the content fit in the available space. The `effective_glue` function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item. When you pass `true` as third argument the value will be rounded.

9.2.9 glue_spec nodes

Internally LUAMETATEX (like its ancestors) also uses nodes to store data that is not seen in node lists. For instance the state of expression scanning (`\dimexpr` etc.) and conditionals (`\ifcase` etc.) is also kept in lists of nodes. A glue, which has five components, is stored in a node as well, so, where most registers store just a number, a skip register (of internal quantity) uses a pointer to a glue spec node. It has similar fields as glue nodes: `shrink`, `shrink_order`, `stretch`, `stretch_order` and `width`, which is not surprising because in the past (and other engines than LUATEX) a glue node also has its values stored in a glue spec. This has some advantages because often the values are the same, so for instance spacing related skips were not resolved immediately but pointed to the current value of a space related internal register (like `\spaceskip`). But, in LUATEX we do resolve these quantities immediately and we put the current values in the glue nodes.

FIELD	TYPE	EXPLANATION
`width`	number	the horizontal or vertical displacement
`stretch`	number	extra (positive) displacement or stretch amount
`stretch_order`	number	factor applied to stretch amount
`shrink`	number	extra (negative) displacement or shrink amount
`shrink_order`	number	factor applied to shrink amount

You will only find these nodes in a few places, for instance when you query an internal quantity. In principle we could do without them as we have interfaces that use the five numbers instead. For compatibility reasons we keep glue spec nodes exposed but this might change in the future.

9.2.10 kern nodes

The `\kern` command creates such nodes but for instance the font and math machinery can also add them. There are not that many fields: `attr`, `expansion_factor` and `kern`.

FIELD	TYPE	EXPLANATION
`subtype`	number	`accentkern`, `fontkern`, `italiccorrection`, `leftmarginkern`, `mathlistkern`, `rightmarginkern` and `userkern`
`attr`	node	list of attributes
`kern`	number	fixed horizontal or vertical advance
`expansion_factor`	number	multiplier related to hz for font kerns

9.2.11 penalty nodes

The `\penalty` command is one that generates these nodes. It is one of the type of nodes often found in vertical lists. It has the fields: `attr` and `penalty`.

FIELD	TYPE	EXPLANATION
subtype	number	afterdisplaypenalty, beforedisplaypenalty, equationnumberpenalty, finalpenalty, linebreakpenalty, linepenalty, noadpenalty, userpenalty and wordpenalty
attr	node	list of attributes
penalty	number	the penalty value

The subtypes are just informative and TEX itself doesn't use them. When you run into an linebreakpenalty you need to keep in mind that it's a accumulation of club, widow and other relevant penalties.

9.2.12 glyph nodes

These are probably the mostly used nodes and although you can push them in the current list with for instance \char TEX will normally do it for you when it considers some input to be text. Glyph nodes are relatively large and have many fields: attr, char, data, depth, expansion_factor, font, height, language, left, right, state, uchyph, width, xoffset and yoffset.

FIELD	TYPE	EXPLANATION
subtype	number	bit field
attr	node	list of attributes
char	number	the character index in the font
font	number	the font identifier
language	number	the language identifier
left	number	the frozen \lefthyphenmnin value
right	number	the frozen \righthyphenmnin value
uchyph	boolean	the frozen \uchyph value
state	number	a user field (replaces the component list)
xoffset	number	a virtual displacement in horizontal direction
yoffset	number	a virtual displacement in vertical direction
width	number	the (original) width of the character
height	number	the (original) height of the character
depth	number	the (original) depth of the character
expansion_factor	number	the to be applied expansion_factor
data	number	a general purpose field for users (we had room for it)

The width, height and depth values are read-only. The expansion_factor is assigned in the par builder and used in the backend. Valid bits for the subtype field are:

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

The expansion_factor has been introduced as part of the separation between front- and backend. It is the result of extensive experiments with a more efficient implementation of expansion. Early versions of LuaTeX already replaced multiple instances of fonts in the backend by scaling but contrary to pdfTeX in LuaTeX we now also got rid of font copies in the frontend and replaced them by expansion factors that travel with glyph nodes. Apart from a cleaner approach this is also a step towards a better separation between front- and backend.

The is_char function checks if a node is a glyph node with a subtype still less than 256. This function can be used to determine if applying font logic to a glyph node makes sense. The value nil gets returned when the node is not a glyph, a character number is returned if the node is still tagged as character and false gets returned otherwise. When nil is returned, the id is also returned. The is_glyph variant doesn't check for a subtype being less than 256, so it returns either the character value or nil plus the id. These helpers are not always faster than separate calls but they sometimes permit making more readable tests. The uses_font helpers takes a node and font id and returns true when a glyph or disc node references that font.

9.2.13 boundary nodes

This node relates to the \noboundary, \boundary, \protrusionboundary and \wordboundary primitives. These are small nodes: attr and data are the only fields.

FIELD	TYPE	EXPLANATION
subtype	number	cancel, protrusion, user and word
attr	node	list of attributes
data	number	values 0–255 are reserved

9.2.14 par nodes

This node is inserted at the start of a paragraph. You should not mess too much with this one. Valid fields are: attr, box_left, box_left_width, box_right, box_right_width, brokenpenalty, dir and interlinepenalty.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
pen_inter	number	local interline penalty (from \localinterlinepenalty)
pen_broken	number	local broken penalty (from \localbrokenpenalty)
dir	string	the direction of this par. see 9.2.15
box_left	node	the \localleftbox
box_left_width	number	width of the \localleftbox
box_right	node	the \localrightbox
box_right_width	number	width of the \localrightbox

A warning: never assign a node list to the box_left or box_right field unless you are sure its internal link structure is correct, otherwise an error may result.

9.2.15 dir nodes

Direction nodes mark parts of the running text that need a change of direction and the `\textdir` command generates them. Again this is a small node, we just have `attr`, `dir` and `level`.

FIELD	TYPE	EXPLANATION
`subtype`	number	`cancel` and `normal`
`attr`	node	list of attributes
`dir`	string	the direction (`0` = l2r, `1` = r2l)
`level`	number	nesting level of this direction

There are only two directions: left-to-right (`0`) and right-to-left (`1`). This is different from LuaTeX that has four directions.

9.2.16 Whatsits

A whatsit node is a real simple one and it only has a subtype. It is even less than a user node (which it actually could be) and uses hardly any memory. What you do with it it entirely up to you: it's is real minimalistic. You can assign a subtype and it has attributes. It is all up to the user how they are handled.

9.2.17 Math noads

9.2.17.1 The concept

These are the so–called 'noad's and the nodes that are specifically associated with math processing. When you enter a formula, TeX creates a node list with regular nodes and noads. Then it hands over the list the math processing engine. The result of that is a nodelist without noads. Most of the noads contain subnodes so that the list of possible fields is actually quite small. Math formulas are both a linked list and a tree. For instance in $e = mc^2$ there is a linked list `e = m c` but the `c` has a superscript branch that itself can be a list with branches.

First, there are the objects (the TeXbook calls them 'atoms') that are associated with the simple math objects: ord, op, bin, rel, open, close, punct, inner, over, under, vcenter. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation: `attr`, `nucleus`, `options`, `sub` and `sup`.

Many object fields in math mode are either simple characters in a specific family or math lists or node lists: `math_char`, `math_text_char`, sub_box and `sub_mlist` and `delimiter`. These are endpoints and therefore the `next` and `prev` fields of these these subnodes are unused.

Some of the more elaborate noads have an option field. The values in this bitset are common:

MEANING	BITS
set	`0x08`
internal	`0x00` + `0x08`
internal	`0x01` + `0x08`
axis	`0x02` + `0x08`

no axis	0x04 + 0x08
exact	0x10 + 0x08
left	0x11 + 0x08
middle	0x12 + 0x08
right	0x14 + 0x08
no subscript	0x21 + 0x08
no superscript	0x22 + 0x08
no script	0x23 + 0x08

9.2.17.2 math_char and math_text_char subnodes

These are the most common ones, as they represent characters, and they both have the same fields: attr, char and fam.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
char	number	the character index
fam	number	the family number

The math_char is the simplest subnode field, it contains the character and family for a single glyph object. The family eventually resolves on a reference to a font. The math_text_char is a special case that you will not normally encounter, it arises temporarily during math list conversion (its sole function is to suppress a following italic correction).

9.2.17.3 sub_box and sub_mlist subnodes

These two subnode types are used for subsidiary list items. For sub_box, the list points to a 'normal' vbox or hbox. For sub_mlist, the list points to a math list that is yet to be converted. Their fields are: attr and list.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
list	node	list of nodes

A warning: never assign a node list to the list field unless you are sure its internal link structure is correct, otherwise an error is triggered.

9.2.17.4 delimiter subnodes

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the next and prev fields are unused, but we do have: attr, large_char, large_fam, small_char and small_fam.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
small_char	number	character index of base character

small_fam	number	family number of base character
large_char	number	character index of next larger character
large_fam	number	family number of next larger character

The fields large_char and large_fam can be zero, in that case the font that is set for the small_fam is expected to provide the large version as an extension to the small_char.

9.2.17.5 simple noad nodes

In these noads, the nucleus, sub and sup fields can branch of. Its fields are: attr, nucleus, options, sub and sup.

FIELD	TYPE	EXPLANATION
subtype	number	bin, close, inner, opdisplaylimits, open, oplimits, opnolimits, ord, ordlimits, over, punct, rel, under and vcenter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
options	number	bitset of rendering options

9.2.17.6 accent nodes

Accent nodes deal with stuff on top or below a math constructs. They support: accent, attr, bot_accent, fraction, nucleus, overlay_accent, sub, sup and top_accent.

FIELD	TYPE	EXPLANATION
subtype	number	bothflexible, fixedboth, fixedbottom and fixedtop
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
accent	kernel node	top accent
bot_accent	kernel node	bottom accent
fraction	number	larger step criterium (divided by 1000)

9.2.17.7 style nodes

These nodes are signals to switch to another math style. They are quite simple: attr and style. Currently the subtype is actually used to store the style but don't rely on that for the future. Fields are: attr and style.

FIELD	TYPE	EXPLANATION
style	string	contains the style

Valid styles are: display (0), crampeddisplay (1), text (2), crampedtext (3), script (4), crampedscript (5), scriptscript (6) and crampedscriptscript (7).

9.2.17.8 parameter nodes

These nodes are used to (locally) set math parameters: list, name, style and value. Fields are: list, name, style and value.

FIELD	TYPE	EXPLANATION
style	string	contains the style
name	string	defines the parameter
value	number	holds the value, in case of a muglue multiple

9.2.17.9 choice nodes

Of its fields attr, display, script, scriptscript and text most are lists. Warning: never assign a node list unless you are sure its internal link structure is correct, otherwise an error can occur.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
display	node	list of display size alternatives
text	node	list of text size alternatives
script	node	list of scriptsize alternatives
scriptscript	node	list of scriptscriptsize alternatives

9.2.17.10 radical nodes

Radical nodes are the most complex as they deal with scripts as well as constructed large symbols. Many fields: attr, degree, left, nucleus, options, sub, sup and width. Warning: never assign a node list to the nucleus, sub, sup, left, or degree field unless you are sure its internal link structure is correct, otherwise an error can be triggered.

FIELD	TYPE	EXPLANATION
subtype	number	radical, udelimiterover, udelimiterunder, uhextensible, uoverdelimiter, uradical, uroot and uunderdelimiter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
left	delimiter node	
degree	kernel node	only set by \Uroot
width	number	required width
options	number	bitset of rendering options

9.2.17.11 fraction nodes

Fraction nodes are also used for delimited cases, hence the left and right fields among: attr, denom, fam, left, middle, num, options, right and width.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	(optional) width of the fraction
num	kernel node	numerator
denom	kernel node	denominator
left	delimiter node	left side symbol
right	delimiter node	right side symbol
middle	delimiter node	middle symbol
options	number	bitset of rendering options

Warning: never assign a node list to the num, or denom field unless you are sure its internal link structure is correct, otherwise an error can result.

9.2.17.12 fence nodes

Fence nodes come in pairs but either one can be a dummy (this period driven empty fence). Fields are: attr, class, delimiter, depth, height, italic and options. Some of these fields are used by the renderer and might get adapted in the process.

FIELD	TYPE	EXPLANATION
subtype	number	left, middle, no, right and unset
attr	node	list of attributes
delimiter	delimiter node	delimiter specification
italic	number	italic correction
height	number	required height
depth	number	required depth
options	number	bitset of rendering options
class	number	spacing related class

9.3 The node library

9.3.1 Introduction

The node library provides methods that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, copy, delete, and insert node, the core objects within the typesetter. Nodes are represented in LUA as userdata. The various parts within a node can be accessed using named fields.

Each node has at least the three fields next, id, and subtype. The other available fields depend on the id.

The next field returns the userdata object for the next node in a linked list of nodes, or nil, if there is no next node.
The id indicates TEX's 'node type'. The field id has a numeric value for efficiency reasons, but some of the library functions also accept a string value instead of id.
The subtype is another number. It often gives further information about a node of a particular id.

Nodes can be compared to each other, but: you are actually comparing indices into the node memory. This means that equality tests can only be trusted under very limited conditions. It will not work correctly in any situation where one of the two nodes has been freed and/or reallocated: in that case, there will be false positives. The general approach to a node related callback is as follows:

Assume that the node list that you get is okay and properly double linked. If for some reason the links are not right, you can apply `node.slide` to the list.
When you insert a node, make sure you use a previously removed one, a new one or a copy. Don't simply inject the same node twice.
When you remove a node, make sure that when this is permanent, you also free the node or list.
Although you can fool the system, normally you will trigger an error when you try to copy a nonexisting node, or free an already freed node. There is some overhead involved in this checking but the current compromise is acceptable.
When you're done, pass back (if needed) the result. It's your responsibility to make sure that the list is properly linked (you can play safe and again apply `node.slide`. In principle you can put nodes in a list that are not acceptable in the following up actions. Some nodes get ignored, others will trigger an error, and sometimes the engine will just crash.

So, from the above it will be clear then memory management of nodes has to be done explicitly by the user. Nodes are not 'seen' by the LUA garbage collector, so you have to call the node freeing functions yourself when you are no longer in need of a node (list). Nodes form linked lists without reference counting, so you have to be careful that when control returns back to LUATEX itself, you have not deleted nodes that are still referenced from a `next` pointer elsewhere, and that you did not create nodes that are referenced more than once. Normally the setters and getters handle this for you.

A good example are discretionary nodes that themselves have three sublists. Internally they use special pointers, but the user never sees them because when you query them or set fields, this property is hidden and taken care of. You just see a list. But, when you mess with these sub lists it is your responsibility that it only contains nodes that are permitted in a discretionary.

There are statistics available with regards to the allocated node memory, which can be handy for tracing. Normally the amount of used nodes is not that large. Typesetting a page can involve thousands of them but most are freed when the page has been shipped out. Compared to other programs, node memory usage is not that excessive. So, if for some reason your application leaks nodes, if at the end of your run you lost as few hundred it's not a real problem. In fact, if you created boxes and made copies but not flushed them for good reason, your run will for sure end with used nodes and the statistics will mention that. The same is true for attributes and skips (glue spec nodes): keeping the current state involves using nodes.

9.3.2 Housekeeping

9.3.2.1 types

This function returns an array that maps node id numbers to node type strings, providing an overview of the possible top-level `id` types.

```
<table> t = node.types()
```

When we issue this command, we get a table. The currently visible types are `{ [0] = "hlist", "vlist", "rule", "insert", "mark", "adjust", "boundary", "disc", "whatsit", "par", "dir", "math", "glue", "kern", "penalty", "style", "choice", "parameter", "noad", "radical", "fraction", "accent", "fence", "math_char", "math_text_char", "sub_box", "sub_mlist", "delimiter", "glyph", "unset", [32] = "attribute_list", [33] = "attribute", [34] = "glue_spec", [35] = "temp", [36] = "split", }` where the numbers are the internal identifiers. Only those nodes are reported that make sense to users so there can be gaps in the range of numbers.

9.3.2.2 id and type

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

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

The `node.id("glyph")` command returns the number `28` and `node.id("hlist")` returns `0` where the numbers don't relate to importance or some ordering; they just appear in the order that is handy for the engine. Commands like this are rather optimized so performance should be ok but you can of course always store the id in a LUA number.

The reverse operation is: `node.type` If the argument is a number, then the next function converts an internal numeric representation to an external string representation. Otherwise, it will return the string `node` if the object represents a node, and `nil` otherwise.

```
<string> type = node.type(<any> n)
```

The `node.type(4)` command returns the string `mark` and `node.id(99)` returns `NIL` because there is no node with that id.

9.3.2.3 fields and has_field

This function returns an indexed table with valid field names for a particular type of node.

```
<table> t = node.fields(<number|string> id)
```

The function accepts a string or number, so `node.fields ("glyph")` returns `{ [-1] = "prev", [0] = "next", "id", "subtype", "attr", "char", "font", "language", "left", "right", "uchyph", "state", "xoffset", "yoffset", "width", "height", "depth", "expansion_factor", "data", }` and `node.fields (12)` gives `{ [-1] = "prev", [0] = "next", "id", "subtype", "attr", "leader", "width", "stretch", "shrink", "stretch_order", "shrink_order", "font", }`.

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

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

This function probably is not that useful but some nodes don't have a `subtype`, `attr` or `prev` field and this is a way to test for that.

9.3.2.4 is_node

```
<boolean|integer> t = node.is_node(<any> item)
```

This function returns a number (the internal index of the node) if the argument is a userdata object of type <node> and false when no node is passed.

9.3.2.5 new

The new function creates a new node. All its fields are initialized to either zero or nil except for id and subtype. Instead of numbers you can also use strings (names). If you pass a second argument the subtype will be set too.

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

As already has been mentioned, you are responsible for making sure that nodes created this way are used only once, and are freed when you don't pass them back somehow.

9.3.2.6 free, flush_node and flush_list

The next one frees node n from TEX's memory. Be careful: no checks are done on whether this node is still pointed to from a register or some next field: it is up to you to make sure that the internal data structures remain correct. Fields that point to nodes or lists are flushed too. So, when you used their content for something else you need to set them to nil first.

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

The free function returns the next field of the freed node, while the flush_node alternative returns nothing.

A list starting with node n can be flushed from TEX's memory too. Be careful: no checks are done on whether any of these nodes is still pointed to from a register or some next field: it is up to you to make sure that the internal data structures remain correct.

```
node.flush_list(<node> n)
```

When you free for instance a discretionary node, flush_list is applied to the pre, post, replace so you don't need to do that yourself. Assigning them nil won't free those lists!

9.3.2.7 copy and copy_list

This creates a deep copy of node n, including all nested lists as in the case of a hlist or vlist node. Only the next field is not copied.

```
<node> m = node.copy(<node> n)
```

A deep copy of the node list that starts at n can be created too. If m is also given, the copy stops just before node m.

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

Note that you cannot copy attribute lists this way. However, there is normally no need to copy attribute lists as when you do assignments to the `attr` field or make changes to specific attributes, the needed copying and freeing takes place automatically. When you change a value of an attribute *in* a list, it will affect all the nodes that share that list.

9.3.2.8 write

```
node.write(<node> n)
```

This function will append a node list to TEX's 'current list'. The node list is not deep-copied! There is no error checking either! You might need to enforce horizontal mode in order for this to work as expected.

9.3.3 Manipulating lists

9.3.3.1 slide

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

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

In most cases TEX itself only uses `next` pointers but your other callbacks might expect proper `prev` pointers too. So, when you run into issues or are in doubt, apply the slide function before you return the list.

9.3.3.2 tail

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

Returns the last node of the node list that starts at `n`.

9.3.3.3 length and count

```
<number> i = node.length(<node> n)
<number> i = node.length(<node> n, <node> m)
```

Returns the number of nodes contained in the node list that starts at `n`. If `m` is also supplied it stops at `m` instead of at the end of the list. The node `m` is not counted.

```
<number> i = node.count(<number> id, <node> n)
<number> i = node.count(<number> id, <node> n, <node> m)
```

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

9.3.3.4 remove

```
<node> head, current, removed =
    node.remove(<node> head, <node> current)
<node> head, current =
    node.remove(<node> head, <node> current, <boolean> true)
```

This function removes the node `current` from the list following `head`. It is your responsibility to make sure it is really part of that list. The return values are the new `head` and `current` nodes. The returned `current` is the node following the `current` in the calling argument, and is only passed back as a convenience (or `nil`, if there is no such node). The returned `head` is more important, because if the function is called with `current` equal to `head`, it will be changed. When the third argument is passed, the node is freed.

9.3.3.5 insert_before

```
<node> head, new = node.insert_before(<node> head, <node> current, <node> new)
```

This function inserts the node `new` before `current` into the list following `head`. It is your responsibility to make sure that `current` is really part of that list. The return values are the (potentially mutated) `head` and the node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

9.3.3.6 insert_after

```
<node> head, new = node.insert_after(<node> head, <node> current, <node> new)
```

This function inserts the node `new` after `current` into the list following `head`. It is your responsibility to make sure that `current` is really part of that list. The return values are the `head` and the node `new`, set up to be part of the list (with correct `next` field). If `head` is initially `nil`, it will become `new`.

9.3.3.7 last_node

```
<node> n = node.last_node()
```

This function pops the last node from TEX's 'current list'. It returns that node, or `nil` if the current list is empty.

9.3.3.8 traverse

```
<node> t, id, subtype = node.traverse(<node> n)
```

This is a LUA iterator that loops over the node list that starts at `n`. Typically code looks like this:

```
for n in node.traverse(head) do
   ...
end
```


is functionally equivalent to:

```
do
  local n
  local function f (head,var)
    local t
    if var == nil then
       t = head
    else
       t = var.next
    end
    return t
  end
  while true do
    n = f (head, n)
    if n == nil then break end
    ...
  end
end
```

It should be clear from the definition of the function `f` that even though it is possible to add or remove nodes from the node list while traversing, you have to take great care to make sure all the `next` (and `prev`) pointers remain valid.

If the above is unclear to you, see the section 'For Statement' in the LUA Reference Manual.

9.3.3.9 traverse_id

```
<node> t, subtype = node.traverse_id(<number> id, <node> n)
```

This is an iterator that loops over all the nodes in the list that starts at `n` that have a matching `id` field.

See the previous section for details. The change is in the local function `f`, which now does an extra while loop checking against the upvalue `id`:

```
 local function f(head,var)
   local t
   if var == nil then
      t = head
   else
      t = var.next
   end
   while not t.id == id do
      t = t.next
   end
   return t
 end
```

9.3.3.10 traverse_char and traverse_glyph

The traverse_char iterator loops over the glyph nodes in a list. Only nodes with a subtype less than 256 are seen.

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

The traverse_glyph iterator loops over a list and returns the list and filters all glyphs:

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

9.3.3.11 traverse_list

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

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

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

9.3.3.12 find_node

This helper returns the location of the first match at or after node n:

```
<node> n = node.find_node(<node> n, <integer> subtype)
<node> n, subtype = node.find_node(<node> n)
```

9.3.4 Glue handling

9.3.4.1 setglue

You can set the five properties of a glue in one go. If a non-numeric value is passed the property becomes zero.

```
node.setglue(<node> n)
node.setglue(<node> n,width,stretch,shrink,stretch_order,shrink_order)
```

When you pass values, only arguments that are numbers are assigned so

```
node.setglue(n,655360,false,65536)
```

will only adapt the width and shrink.

When a list node is passed, you set the glue, order and sign instead.

9.3.4.2 getglue

The next call will return 5 values or nothing when no glue is passed.

```
<integer> width, <integer> stretch, <integer> shrink, <integer> stretch_order,
    <integer> shrink_order = node.getglue(<node> n)
```

When the second argument is false, only the width is returned (this is consistent with `tex.get`).

When a list node is passed, you get back the glue that is set, the order of that glue and the sign.

9.3.4.3 `is_zero_glue`

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

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

9.3.5 Attribute handling

9.3.5.1 Attributes

Assignments to attributes registers result in assigning lists with set attributes to nodes and the implementation is non-trivial because the value that is attached to a node is essentially a (sorted) sparse array of key-value pairs. It is generally easiest to deal with attribute lists and attributes by using the dedicated functions in the `node` library.

9.3.5.2 `attribute_list` nodes

An `attribute_list` item is used as a head pointer for a list of attribute items. It has only one user-visible field:

FIELD	TYPE	EXPLANATION
`next`	node	pointer to the first attribute

9.3.5.3 `attr` nodes

A normal node's attribute field will point to an item of type `attribute_list`, and the `next` field in that item will point to the first defined 'attribute' item, whose `next` will point to the second 'attribute' item, etc.

FIELD	TYPE	EXPLANATION
`next`	node	pointer to the next attribute
`number`	number	the attribute type id
`value`	number	the attribute value

As mentioned it's better to use the official helpers rather than edit these fields directly. For instance the `prev` field is used for other purposes and there is no double linked list.

9.3.5.4 `current_attr`

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

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

The intended usage of `current_attr` is as follows:

```
local x1 = node.new("glyph")
x1.attr = node.current_attr()
local x2 = node.new("glyph")
x2.attr = node.current_attr()
```

or:

```
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.current_attr()
x1.attr = ca
x2.attr = ca
```

The attribute lists are ref counted and the assignment takes care of incrementing the refcount. You cannot expect the value `ca` to be valid any more when you assign attributes (using `tex.setattribute`) or when control has been passed back to TEX.

9.3.5.5 has_attribute

```
<number> v = node.has_attribute(<node> n, <number> id)
<number> v = node.has_attribute(<node> n, <number> id, <number> val)
```

Tests if a node has the attribute with number `id` set. If `val` is also supplied, also tests if the value matches `val`. It returns the value, or, if no match is found, `nil`.

9.3.5.6 get_attribute

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

Tests if a node has an attribute with number `id` set. It returns the value, or, if no match is found, `nil`. If no `id` is given then the zero attributes is assumed.

9.3.5.7 find_attribute

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

Finds the first node that has attribute with number `id` set. It returns the value and the node if there is a match and otherwise nothing.

9.3.5.8 set_attribute

```
node.set_attribute(<node> n, <number> id, <number> val)
```

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

9.3.5.9 unset_attribute

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

Unsets the attribute with number `id`. If `val` is also supplied, it will only perform this operation if the value matches `val`. Missing attributes or attribute-value pairs are ignored.

If the attribute was actually deleted, returns its old value. Otherwise, returns `nil`.

9.3.6 Glyph handling

9.3.6.1 first_glyph

```
<node> n = node.first_glyph(<node> n)
<node> n = node.first_glyph(<node> n, <node> m)
```

Returns the first node in the list starting at `n` that is a glyph node with a subtype indicating it is a glyph, or `nil`. If `m` is given, processing stops at (but including) that node, otherwise processing stops at the end of the list.

9.3.6.2 is_char and is_glyph

The subtype of a glyph node signals if the glyph is already turned into a character reference or not.

```
<boolean> b = node.is_char(<node> n)
<boolean> b = node.is_glyph(<node> n)
```

9.3.6.3 has_glyph

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

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

9.3.6.4 ligaturing

```
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n)
<node> h, <node> t, <boolean> success = node.ligaturing(<node> n, <node> m)
```

Apply TEX-style ligaturing to the specified nodelist. The tail node `m` is optional. The two returned nodes `h` and `t` are the new head and tail (both `n` and `m` can change into a new ligature).

9.3.6.5 kerning

```
<node> h, <node> t, <boolean> success = node.kerning(<node> n)
```

```
<node> h, <node> t, <boolean> success = node.kerning(<node> n, <node> m)
```

Apply TEX-style kerning to the specified node list. The tail node `m` is optional. The two returned nodes `h` and `t` are the head and tail (either one of these can be an inserted kern node, because special kernings with word boundaries are possible).

9.3.6.6 unprotect_glyph[s]

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

Subtracts 256 from all glyph node subtypes. This and the next function are helpers to convert from `characters` to `glyphs` during node processing. The second argument is optional and indicates the end of a range.

9.3.6.7 protect_glyph[s]

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

Adds 256 to all glyph node subtypes in the node list starting at `n`, except that if the value is 1, it adds only 255. The special handling of 1 means that `characters` will become `glyphs` after subtraction of 256. A single character can be marked by the singular call. The second argument is optional and indicates the end of a range.

9.3.6.8 protrusion_skippable

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

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

9.3.6.9 check_discretionary, check_discretionaries

When you fool around with disc nodes you need to be aware of the fact that they have a special internal data structure. As long as you reassign the fields when you have extended the lists it's ok because then the tail pointers get updated, but when you add to list without reassigning you might end up in trouble when the linebreak routine kicks in. You can call this function to check the list for issues with disc nodes.

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

The plural variant runs over all disc nodes in a list, the singular variant checks one node only (it also checks if the node is a disc node).

9.3.6.10 flatten_discretionaries

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

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

9.3.7 Packaging

9.3.7.1 hpack

This function creates a new hlist by packaging the list that begins at node `n` into a horizontal box. With only a single argument, this box is created using the natural width of its components. In the three argument form, `info` must be either `additional` or `exactly`, and `w` is the additional (`\hbox spread`) or exact (`\hbox to`) width to be used. The second return value is the badness of the generated box.

```
<node> h, <number> b =
    node.hpack(<node> n)
<node> h, <number> b =
    node.hpack(<node> n, <number> w, <string> info)
<node> h, <number> b =
    node.hpack(<node> n, <number> w, <string> info, <string> dir)
```

Caveat: there can be unexpected side-effects to this function, like updating some of the `\marks` and `\inserts`. Also note that the content of `h` is the original node list `n`: if you call `node.free(h)` you will also free the node list itself, unless you explicitly set the `list` field to `nil` beforehand. And in a similar way, calling `node.free(n)` will invalidate `h` as well!

9.3.7.2 vpack

This function creates a new vlist by packaging the list that begins at node `n` into a vertical box. With only a single argument, this box is created using the natural height of its components. In the three argument form, `info` must be either `additional` or `exactly`, and `w` is the additional (`\vbox spread`) or exact (`\vbox to`) height to be used.

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

The second return value is the badness of the generated box. See the description of `hpack` for a few memory allocation caveats.

9.3.7.3 prepend_prevdepth

This function is somewhat special in the sense that it is an experimental helper that adds the interlinespace to a line keeping the baselineskip and lineskip into account.

```
<node> n, <number> delta =
```

```
    node.prepend_prevdepth(<node> n,<number> prevdepth)
```

9.3.7.4 dimensions, rangedimensions, naturalwidth

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

This function calculates the natural in-line dimensions of the node list starting at node `n` and terminating just before node `t` (or the end of the list, if there is no second argument). The return values are scaled points. An alternative format that starts with glue parameters as the first three arguments is also possible:

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

This calling method takes glue settings into account and is especially useful for finding the actual width of a sublist of nodes that are already boxed, for example in code like this, which prints the width of the space in between the `a` and `b` as it would be if `\box0` was used as-is:

```
\setbox0 = \hbox to 20pt {a b}

\directlua{print (node.dimensions(
    tex.box[0].glue_set,
    tex.box[0].glue_sign,
    tex.box[0].glue_order,
    tex.box[0].head.next,
    node.tail(tex.box[0].head)
)) }
```

You need to keep in mind that this is one of the few places in TEX where floats are used, which means that you can get small differences in rounding when you compare the width reported by `hpack` with `dimensions`.

The second alternative saves a few lookups and can be more convenient in some cases:

```
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first)
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first, <node> last)
```

A simple and somewhat more efficient variant is this:

```
<number> w =
```


```
node.naturalwidth(<node> start, <node> stop)
```

9.3.8 Math

9.3.8.1 mlist_to_hlist

```
<node> h =
    node.mlist_to_hlist(<node> n, <string> display_type, <boolean> penalties)
```

This runs the internal mlist to hlist conversion, converting the math list in `n` into the horizontal list `h`. The interface is exactly the same as for the callback `mlist_to_hlist`.

9.3.8.2 end_of_math

```
<node> t = node.end_of_math(<node> start)
```

Looks for and returns the next `math_node` following the `start`. If the given node is a math end node this helper returns that node, else it follows the list and returns the next math endnote. If no such node is found nil is returned.

9.4 Two access models

Deep down in TEX a node has a number which is a numeric entry in a memory table. In fact, this model, where TEX manages memory is real fast and one of the reasons why plugging in callbacks that operate on nodes is quite fast too. Each node gets a number that is in fact an index in the memory table and that number often is reported when you print node related information. You go from userdata nodes and there numeric references and back with:

```
<integer> d = node.todirect(<node> n))
<node> n = node.tonode(<integer> d))
```

The userdata model is rather robust as it is a virtual interface with some additional checking while the more direct access which uses the node numbers directly. However, even with userdata you can get into troubles when you free nodes that are no longer allocated or mess up lists. if you apply `tostring` to a node you see its internal (direct) number and id.

The first model provides key based access while the second always accesses fields via functions:

```
nodeobject.char
getfield(nodenumber,"char")
```

If you use the direct model, even if you know that you deal with numbers, you should not depend on that property but treat it as an abstraction just like traditional nodes. In fact, the fact that we use a simple basic datatype has the penalty that less checking can be done, but less checking is also the reason why it's somewhat faster. An important aspect is that one cannot mix both methods, but you can cast both models. So, multiplying a node number makes no sense.

So our advice is: use the indexed (table) approach when possible and investigate the direct one when speed might be a real issue. For that reason LuaTeX also provide the `get*` and `set*` functions in the top level node namespace. There is a limited set of getters. When implementing this direct approach the regular index by key variant was also optimized, so direct access only makes sense when nodes are accessed millions of times (which happens in some font processing for instance).

We're talking mostly of getters because setters are less important. Documents have not that many content related nodes and setting many thousands of properties is hardly a burden contrary to millions of consultations.

Normally you will access nodes like this:

```
local next = current.next
if next then
    -- do something
end
```

Here `next` is not a real field, but a virtual one. Accessing it results in a metatable method being called. In practice it boils down to looking up the node type and based on the node type checking for the field name. In a worst case you have a node type that sits at the end of the lookup list and a field that is last in the lookup chain. However, in successive versions of LuaTeX these lookups have been optimized and the most frequently accessed nodes and fields have a higher priority.

Because in practice the `next` accessor results in a function call, there is some overhead involved. The next code does the same and performs a tiny bit faster (but not that much because it is still a function call but one that knows what to look up).

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

In the direct namespace there are more helpers and most of them are accompanied by setters. The getters and setters are clever enough to see what node is meant. We don't deal with whatsit nodes: their fields are always accessed by name. It doesn't make sense to add getters for all fields, we just identifier the most likely candidates. In complex documents, many node and fields types never get seen, or seen only a few times, but for instance glyphs are candidates for such optimization. The `node.direct` interface has some more helpers.[6]

The `setdisc` helper takes three (optional) arguments plus an optional fourth indicating the subtype. Its `getdisc` takes an optional boolean; when its value is `true` the tail nodes will also be returned. The `setfont` helper takes an optional second argument, it being the character. The directmode setter `setlink` takes a list of nodes and will link them, thereby ignoring `nil` entries. The first valid node is returned (beware: for good reason it assumes single nodes). For rarely used fields no helpers are provided and there are a few that probably are used seldom too but

[6] We can define the helpers in the node namespace with `getfield` which is about as efficient, so at some point we might provide that as module.

were added for consistency. You can of course always define additional accessors using `getfield` and `setfield` with little overhead. When the second argument of `setattributelist` is `true` the current attribute list is assumed.

The `reverse` function reverses a given list. The `exchange` function swaps two nodes; it takes upto three arguments: a head node, and one or two to be swapped nodes. When there is no third argument, it will assume that the node following node is to be used. So we have:

```
head = node.direct.reverse(head)
head = node.direct.exchange(head,first,[second])
```

In ConTeXt some of the not performance-critical userdata variants are emulated in Lua and not in the engine, so we retain downward compatibility.

FUNCTION	NODE	DIRECT	emulated
check_discretionaries	–	+	+
check_discretionary	–	+	+
copy	+	+	
copy_list	+	+	
count	–	+	+
current_attr	+	+	
dimensions	–	+	+
effective_glue	–	+	+
end_of_math	–	+	+
find_attribute	–	+	+
first_glyph	–	+	+
flatten_discretionaries	–	+	+
flush_list	+	+	
flush_node	+	+	
free	+	+	
get_attribute	+	+	
get_properties_table	+	+	
get_synctex_fields	–	+	
getattributelist	–	+	
getboth	–	+	
getbox	–	+	
getchar	–	+	
getstate	–	+	
getdata	–	+	
getdepth	–	+	
getdirection	–	+	
getdisc	–	+	
getexpansion	–	+	
getfam	–	+	
getfield	+	+	
getfont	–	+	
getglue	–	+	+
getglyphdata	–	+	

getglyphstate	−	+	
getglyphscript	−	+	
getheight	−	+	
getid	−	+	
getkern	−	+	
getlang	−	+	
getleader	−	+	
getlist	−	+	
getnext	−	+	
getnormalizedline	−	+	
getnucleus	−	+	
getoffsets	−	+	
getorientation	−	+	
getpenalty	−	+	
getpost	−	+	
getpre	−	+	
getprev	−	+	
getproperty	+	+	
getreplace	−	+	
getshift	−	+	
getsub	−	+	
getsubtype	−	+	
getsup	−	+	
getwhd	−	+	
getwidth	−	+	
has_attribute	+	+	
has_dimensions	−	+	
has_field	+	+	
has_glyph	−	+	+
hpack	−	+	+
insert_after	+	+	
insert_before	+	+	
is_char	−	+	
is_direct	−	+	
is_glyph	−	+	
is_node	+	+	
is_valid	−	+	
is_zero_glue	−	+	+
kerning	−	+	+
last_node	−	+	+
length	−	+	+
ligaturing	−	+	+
make_extensible	−	+	+
mlist_to_hlist	−	+	+
naturalwidth	−	+	+
new	+	+	

prepend_prevdepth	−	+	+
protect_glyphs	−	+	+
protect_glyph	−	+	+
protrusion_skippable	−	+	+
rangedimensions	−	+	+
remove	+	+	
set_attribute	+	+	
set_synctex_fields	−	+	
setattributelist	−	+	
setboth	−	+	
setbox	−	+	
setchar	−	+	
setstate	−	+	
setdata	−	+	
setdepth	−	+	
setdirection	−	+	
setdisc	−	+	
setexpansion	−	+	
setfam	−	+	
setfield	+	+	
setfont	−	+	
setglue	+	+	
setglyphdata	−	+	
setglyphstate	−	+	
setglyphscript	−	+	
setheight	−	+	
setkern	−	+	
setlang	−	+	
setleader	−	+	
setlink	−	+	
setlist	−	+	
setnext	−	+	
setnucleus	−	+	
setoffsets	−	+	
setorientation	−	+	
setpenalty	−	+	
setprev	−	+	
setproperty	+	+	
setshift	−	+	
setsplit	−	+	
setsub	−	+	
setsubtype	−	+	
setsup	−	+	
setwhd	−	+	
setwidth	−	+	
slide	−	+	+

reverse	−	+	
exchange	−	+	
start_of_par	−	+	
subtype	−	−	
tail	+	+	
todirect	−	+	
tonode	−	+	
tostring	+	−	
traverse	+	+	
traverse_char	+	+	
traverse_glyph	+	+	
traverse_id	+	+	
traverse_list	+	+	
type	+	−	
unprotect_glyphs	−	+	+
unprotect_glyph	−	+	+
unset_attribute	+	+	
usedlist	−	+	+
uses_font	−	+	+
vpack	−	+	+
write	+	+	

The `node.next` and `node.prev` functions will stay but for consistency there are variants called `getnext` and `getprev`. We had to use `get` because `node.id` and `node.subtype` are already taken for providing meta information about nodes. Note: The getters do only basic checking for valid keys. You should just stick to the keys mentioned in the sections that describe node properties.

Some of the getters and setters handle multiple node types, given that the field is relevant. In that case, some field names are considered similar (like `kern` and `width`, or `data` and `value`). In retrospect we could have normalized field names better but we decided to stick to the original (internal) names as much as possible. After all, at the LUA end one can easily create synonyms.

Some nodes have indirect references. For instance a math character refers to a family instead of a font. In that case we provide a virtual font field as accessor. So, `getfont` and `.font` can be used on them. The same is true for the `width`, `height` and `depth` of glue nodes. These actually access the spec node properties, and here we can set as well as get the values.

In some places LUATEX can do a bit of extra checking for valid node lists and you can enable that with:

```
node.fix_node_lists(<boolean> b)
```

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

```
node.set_synctex_fields(<integer> f, <integer> l)
<integer> f, <integer> l =
    node.get_synctex_fields(<node> n)
```


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

9.5 Normalization

As an experiment the lines resulting from paragraph construction can be normalized. There are several modes, that can be set and queried with:

```
node.direct.setnormalize(<integer> n)
<integer> n = node.direct.getnormalize()
```

The state of a line (a hlist) can be queried with:

```
<integer> leftskip, <integer> rightskip,
    <integer> lefthangskip, <integer> righthangskip,
    <node> head, <node> tail,
    <integer> parindent, <integer> parfillskip = node.direct.getnormalized()
```

The modes accumulate, so mode 4 includes 1 upto 3:

VALUE	EXPLANATION
1	left and right skips and directions
2	indentation and parfill skip
3	hanging indentation and par shapes
4	idem but before left and right skips
5	inject compensation for overflow

This is experimental code and might take a while to become frozen.

9.6 Properties

Attributes are a convenient way to relate extra information to a node. You can assign them at the TEX end as well as at the LUA end and consult them at the LUA end. One big advantage is that they obey grouping. They are linked lists and normally checking for them is pretty efficient, even if you use a lot of them. A macro package has to provide some way to manage these attributes at the TEX end because otherwise clashes in their usage can occur.

Each node also can have a properties table and you can assign values to this table using the `setproperty` function and get properties using the `getproperty` function. Managing properties is way more demanding than managing attributes.

Take the following example:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,"foo")
    print(node.getproperty(n))
```

```
    node.setproperty(n,"bar")
    print(node.getproperty(n))

    node.free(n)
}
```

This will print `foo` and `bar` which in itself is not that useful when multiple mechanisms want to use this feature. A variant is:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,{ one = "foo", two = "bar" })
    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

This time we store two properties with the node. It really makes sense to have a table as property because that way we can store more. But in order for that to work well you need to do it this way:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)

    if not t then
        t = { }
        node.setproperty(n,t)
    end
    t.one = "foo"
    t.two = "bar"

    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

Here our own properties will not overwrite other users properties unless of course they use the same keys. So, eventually you will end up with something:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)
```


```
    if not t then
        t = { }
        node.setproperty(n,t)
    end
    t.myself = { one = "foo", two = "bar" }

    print(node.getproperty(n).myself.one)
    print(node.getproperty(n).myself.two)

    node.free(n)
}
```

This assumes that only you use `myself` as subtable. The possibilities are endless but care is needed. For instance, the generic font handler that ships with CONTEXT uses the `injections` subtable and you should not mess with that one!

There are a few helper functions that you normally should not touch as user: `flush_properties_table` will wipe the table (normally a bad idea), `get_properties_table` and will give the table that stores properties (using direct entries) and you can best not mess too much with that one either because LUATEX itself will make sure that entries related to nodes will get wiped when nodes get freed, so that the LUA garbage collector can do its job. In fact, the main reason why we have this mechanism is that it saves the user (or macro package) some work. One can easily write a property mechanism in LUA where after a shipout properties gets cleaned up but it's not entirely trivial to make sure that with each freed node also its properties get freed, due to the fact that there can be nodes left over for a next page. And having a callback bound to the node deallocator would add way to much overhead.

When we copy a node list that has a table as property, there are several possibilities: we do the same as a new node, we copy the entry to the table in properties (a reference), we do a deep copy of a table in the properties, we create a new table and give it the original one as a metatable. After some experiments (that also included timing) with these scenarios we decided that a deep copy made no sense, nor did nilling. In the end both the shallow copy and the metatable variant were both ok, although the second one is slower. The most important aspect to keep in mind is that references to other nodes in properties no longer can be valid for that copy. We could use two tables (one unique and one shared) or metatables but that only complicates matters.

When defining a new node, we could already allocate a table but it is rather easy to do that at the lua end e.g. using a metatable `__index` method. That way it is under macro package control. When deleting a node, we could keep the slot (e.g. setting it to false) but it could make memory consumption raise unneeded when we have temporary large node lists and after that only small lists. Both are not done.

So in the end this is what happens now: when a node is copied, and it has a table as property, the new node will share that table. If the second argument of `set_properties_mode` is `true` then a metatable approach is chosen: the copy gets its own table with the original table as metatable. If you use the generic font loader the mode is enabled that way.

A few more experiments were done. For instance: copy attributes to the properties so that we have fast access at the LUA end. In the end the overhead is not compensated by speed and convenience, in fact, attributes are not that slow when it comes to accessing them. So this was rejected.

Another experiment concerned a bitset in the node but again the gain compared to attributes was neglectable and given the small amount of available bits it also demands a pretty strong agreement over what bit represents what, and this is unlikely to succeed in the TEX community. It doesn't pay off.

Just in case one wonders why properties make sense: it is not so much speed that we gain, but more convenience: storing all kinds of (temporary) data in attributes is no fun and this mechanism makes sure that properties are cleaned up when a node is freed. Also, the advantage of a more or less global properties table is that we stay at the LUA end. An alternative is to store a reference in the node itself but that is complicated by the fact that the register has some limitations (no numeric keys) and we also don't want to mess with it too much.

10 LUA callbacks

10.1 Registering callbacks

The callbacks are a moving target. Don't bother me with questions about them.

This library has functions that register, find and list callbacks. Callbacks are LUA functions that are called in well defined places. There are two kinds of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In mosty cases the second category is expected to behave similar to the built in functionality because in a next step specific data is expected. For instance, you can replace the hyphenation routine. The function gets a list that can be hyphenated (or not). The final list should be valid and is (normally) used for constructing a paragraph. Another function can replace the ligature builder and/or kerner. Doing something else is possible but in the end might not give the user the expected outcome.

The first thing you need to do is registering a callback:

```
id = callback.register(<string> callback_name, <function> func)
id = callback.register(<string> callback_name, nil)
id = callback.register(<string> callback_name, false)
```

Here the `callback_name` is a predefined callback name, see below. The function returns the internal `id` of the callback or `nil`, if the callback could not be registered.

LUATEX internalizes the callback function in such a way that it does not matter if you redefine a function accidentally.

Callback assignments are always global. You can use the special value `nil` instead of a function for clearing the callback.

For some minor speed gain, you can assign the boolean `false` to the non-file related callbacks, doing so will prevent LUATEX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know *exactly* what you are doing!

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

The keys in the table are the known callback names, the value is a boolean where `true` means that the callback is currently set (active).

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

If the callback is not set, `find` returns `nil`. The `known` function can be used to check if a callback is supported.

```
if callback.known("foo") then ... end
```


10.2 File related callbacks

10.2.1 find_format_file and find_log_file

These callbacks are called as:

```
<string> actualname =
    function (<string> askedname)
```

The `askedname` is a format file for reading (the format file for writing is always opened in the current directory) or a log file for writing.

10.2.2 open_data_file

This callback function gets a filename passed:

```
<table> env = function (<string> filename)
```

The return value is either the boolean value false or a table with two functions. A mandate `reader` function fill be called once for each new line to be read, the optional `close` function will be called once LuaTeX is done with the file.

LuaTeX never looks at the rest of the table, so you can use it to store your private per-file data. Both the callback functions will receive the table as their only argument.

10.3 Data processing callbacks

10.3.1 process_jobname

This callback allows you to change the jobname given by `\jobname` in TeX and `tex.jobname` in Lua. It does not affect the internal job name or the name of the output or log files.

```
function(<string> jobname)
    return <string> adjusted_jobname
end
```

The only argument is the actual job name; you should not use `tex.jobname` inside this function or infinite recursion may occur. If you return `nil`, LuaTeX will pretend your callback never happened. This callback does not replace any internal code.

10.4 Node list processing callbacks

The description of nodes and node lists is in chapter 9.

10.4.1 contribute_filter

This callback is called when LuaTeX adds contents to list:

```
function(<string> extrainfo)
end
```

The string reports the group code. From this you can deduce from what list you can give a treat.

VALUE	EXPLANATION
pre_box	interline material is being added
pre_adjust	\vadjust material is being added
box	a typeset box is being added (always called)
adjust	\vadjust material is being added

10.4.2 buildpage_filter

This callback is called whenever LuaTeX is ready to move stuff to the main vertical list. You can use this callback to do specialized manipulation of the page building stage like imposition or column balancing.

```
function(<string> extrainfo)
end
```

The string extrainfo gives some additional information about what TeX's state is with respect to the 'current page'. The possible values for the buildpage_filter callback are:

VALUE	EXPLANATION
alignment	a (partial) alignment is being added
after_output	an output routine has just finished
new_graf	the beginning of a new paragraph
vmode_par	\par was found in vertical mode
hmode_par	\par was found in horizontal mode
insert	an insert is added
penalty	a penalty (in vertical mode)
before_display	immediately before a display starts
after_display	a display is finished
end	LuaTeX is terminating (it's all over)

10.4.3 build_page_insert

This callback is called when the pagebuilder adds an insert. There is not much control over this mechanism but this callback permits some last minute manipulations of the spacing before an insert, something that might be handy when for instance multiple inserts (types) are appended in a row.

```
function(<number> n, <number> i)
    return <number> register
end
```

with

VALUE	EXPLANATION
n	the insert class
i	the order of the insert

The return value is a number indicating the skip register to use for the prepended spacing. This permits for instance a different top space (when i equals one) and intermediate space (when i is larger than one). Of course you can mess with the insert box but you need to make sure that LuaTeX is happy afterwards.

10.4.4 pre_linebreak_filter

This callback is called just before LuaTeX starts converting a list of nodes into a stack of \hboxes, after the addition of \parfillskip.

```
function(<node> head, <string> groupcode)
    return true | false | <node> newhead
end
```

The string called groupcode identifies the nodelist's context within TeX's processing. The range of possibilities is given in the table below, but not all of those can actually appear in pre_linebreak_filter, some are for the hpack_filter and vpack_filter callbacks that will be explained in the next two paragraphs.

VALUE	EXPLANATION
<empty>	main vertical list
hbox	\hbox in horizontal mode
adjusted_hbox	\hbox in vertical mode
vbox	\vbox
vtop	\vtop
align	\halign or \valign
disc	discretionaries
insert	packaging an insert
vcenter	\vcenter
local_box	\localleftbox or \localrightbox
split_off	top of a \vsplit
split_keep	remainder of a \vsplit
align_set	alignment cell
fin_row	alignment row

As for all the callbacks that deal with nodes, the return value can be one of three things:

boolean true signals successful processing
<node> signals that the 'head' node should be replaced by the returned node
boolean false signals that the 'head' node list should be ignored and flushed from memory

This callback does not replace any internal code.

10.4.5 linebreak_filter

This callback replaces LuaTeX's line breaking algorithm.

```
function(<node> head, <boolean> is_display)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the main vertical list, the boolean argument is true if this paragraph is interrupted by a following math display.

If you return something that is not a <node>, LuaTeX will apply the internal linebreak algorithm on the list that starts at <head>. Otherwise, the <node> you return is supposed to be the head of a list of nodes that are all allowed in vertical mode, and at least one of those has to represent an \hbox. Failure to do so will result in a fatal error.

Setting this callback to false is possible, but dangerous, because it is possible you will end up in an unfixable 'deadcycles loop'.

10.4.6 append_to_vlist_filter

This callback is called whenever LuaTeX adds a box to a vertical list (the mirrored argument is obsolete):

```
function(<node> box, <string> locationcode, <number> prevdepth)
    return list [, prevdepth [, checkdepth ] ]
end
```

It is ok to return nothing or nil in which case you also need to flush the box or deal with it yourself. The prevdepth is also optional. Locations are box, alignment, equation, equation_number and post_linebreak. When the third argument returned is true the normal prevdepth correction will be applied, based on the first node.

10.4.7 post_linebreak_filter

This callback is called just after LuaTeX has converted a list of nodes into a stack of \hboxes.

```
function(<node> head, <string> groupcode)
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

10.4.8 hpack_filter

This callback is called when TeX is ready to start boxing some horizontal mode material. Math items and line boxes are ignored at the moment.

```
function(<node> head, <string> groupcode, <number> size,
```


```
        <string> packtype [, <string> direction] [, <node> attributelist])
    return true | false | <node> newhead
end
```

The `packtype` is either `additional` or `exactly`. If `additional`, then the `size` is a `\hbox spread ...` argument. If `exactly`, then the `size` is a `\hbox to ...`. In both cases, the number is in scaled points.

The `direction` is either one of the three-letter direction specifier strings, or `nil`.

This callback does not replace any internal code.

10.4.9 vpack_filter

This callback is called when TEX is ready to start boxing some vertical mode material. Math displays are ignored at the moment.

This function is very similar to the `hpack_filter`. Besides the fact that it is called at different moments, there is an extra variable that matches TEX's `\maxdepth` setting.

```
function(<node> head, <string> groupcode, <number> size, <string> packtype,
        <number> maxdepth [, <string> direction] [, <node> attributelist]))
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

10.4.10 hpack_quality

This callback can be used to intercept the overfull messages that can result from packing a horizontal list (as happens in the par builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
        <number> last)
    return <node> whatever
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed (when protrusion or expansion is enabled, this is an intermediate list). Optionally you can return a node, for instance an overfull rule indicator. That node will be appended to the list (just like TEX's own rule would).

10.4.11 vpack_quality

This callback can be used to intercept the overfull messages that can result from packing a vertical list (as happens in the page builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
```

```
        <number> last)
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed.

10.4.12 process_rule

This is an experimental callback. It can be used with rules of subtype 4 (user). The callback gets three arguments: the node, the width and the height. The callback can use `pdf.print` to write code to the PDF file but beware of not messing up the final result. No checking is done.

10.4.13 pre_output_filter

This callback is called when TEX is ready to start boxing the box 255 for `\output`.

```
function(<node> head, <string> groupcode, <number> size, <string> packtype,
        <number> maxdepth [, <string> direction])
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

10.4.14 hyphenate

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to insert discretionary nodes in the node list it receives.

Setting this callback to `false` will prevent the internal discretionary insertion pass.

10.4.15 ligaturing

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply ligaturing to the node list it receives.

You don't have to worry about return values because the `head` node that is passed on to the callback is guaranteed not to be a glyph_node (if need be, a temporary node will be prepended), and therefore it cannot be affected by the mutations that take place. After the callback, the internal value of the 'tail of the list' will be recalculated.

The `next` of `head` is guaranteed to be non-nil.

The `next` of `tail` is guaranteed to be nil, and therefore the second callback argument can often be ignored. It is provided for orthogonality, and because it can sometimes be handy when special processing has to take place.

Setting this callback to `false` will prevent the internal ligature creation pass.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTeX into panic mode.

10.4.16 kerning

```
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply kerning between the nodes in the node list it receives. See `ligaturing` for calling conventions.

Setting this callback to `false` will prevent the internal kern insertion pass.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTeX into panic mode.

10.4.17 insert_par

Each paragraph starts with a local par node that keeps track of for instance the direction. You can hook a callback into the creator:

```
function(<node> par, <string> location)
end
```

There is no return value and you should make sure that the node stays valid as otherwise TeX can get confused.

10.4.18 mlist_to_hlist

This callback replaces LuaTeX's math list to node list conversion algorithm.

```
function(<node> head, <string> display_type, <boolean> need_penalties)
    return <node> newhead
end
```

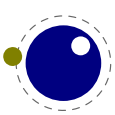
The returned node is the head of the list that will be added to the vertical or horizontal list, the string argument is either 'text' or 'display' depending on the current math mode, the boolean argument is `true` if penalties have to be inserted in this list, `false` otherwise.

Setting this callback to `false` is bad, it will almost certainly result in an endless loop.

10.5 Information reporting callbacks

10.5.1 pre_dump

```
function()
```



```
end
```

This function is called just before dumping to a format file starts. It does not replace any code and there are neither arguments nor return values.

10.5.2 start_run

```
function()
end
```

This callback replaces the code that prints LuaTeX's banner. Note that for successful use, this callback has to be set in the Lua initialization script, otherwise it will be seen only after the run has already started.

10.5.3 stop_run

```
function()
end
```

This callback replaces the code that prints LuaTeX's statistics and 'output written to' messages. The engine can still do housekeeping and therefore you should not rely on this hook for postprocessing the PDF or log file.

10.5.4 intercept_tex_error, intercept_lua_error

```
function()
end
```

This callback is run from inside the TeX error function, and the idea is to allow you to do some extra reporting on top of what TeX already does (none of the normal actions are removed). You may find some of the values in the `status` table useful. The TeX related callback gets two arguments: the current processing mode and a boolean indicating if there was a runaway.

10.5.5 show_error_message and show_warning_message

```
function()
end
```

These callback replaces the code that prints the error message. The usual interaction after the message is not affected.

10.5.6 start_file

```
function(category,filename)
end
```


This callback replaces the code that LuaTeX prints when a file is opened like `(filename` for regular files. The category is a number:

VALUE	MEANING
1	a normal data file, like a source
2	a font map coupling font names to resources
3	an image file (`png`, `pdf`, etc)
4	an embedded font subset
5	a fully embedded font

10.5.7 stop_file

```
function(category)
end
```

This callback replaces the code that LuaTeX prints when a file is closed like the `)` for regular files.

10.5.8 wrapup_run

This callback is called after the PDF and log files are closed. Use it at your own risk.

10.6 Font-related callbacks

10.6.1 define_font

```
function(<string> name, <number> size)
    return <number> id
end
```

The string `name` is the filename part of the font specification, as given by the user.

The number `size` is a bit special:

> If it is positive, it specifies an 'at size' in scaled points.
> If it is negative, its absolute value represents a 'scaled' setting relative to the design size of the font.

The font can be defined with `font.define` which returns a font identifier that can be returned in the callback. So, contrary to LuaTeX, in LuaMetaTeX we only accept a number.

The internal structure of the `font` table that is passed to `font.define` is explained in chapter 6. That table is saved internally, so you can put extra fields in the table for your later Lua code to use. In alternative, `retval` can be a previously defined fontid. This is useful if a previous definition can be reused instead of creating a whole new font structure.

Setting this callback to `false` is pointless as it will prevent font loading completely but will nevertheless generate errors.

11 The TEX related libraries

11.1 The lua library

11.1.1 Version information

This library contains two read-only items:

```
<string> v = lua.version
<string> s = lua.startupfile
```

This returns the LUA version identifier string. The value currently is Lua 5.4.

11.1.2 Table allocators

Sometimes performance (and memory usage) can benefit a little from it preallocating a table with `newtable`:

```
<table> t = lua.newtable(100,5000)
```

This preallocates 100 hash entries and 5000 index entries. The `newindex` function create an indexed table with preset values:

```
<table> t = lua.newindex(2500,true)
```

11.1.3 Bytecode registers

LUA registers can be used to store LUA code chunks. The accepted values for assignments are functions and `nil`. Likewise, the retrieved value is either a function or `nil`.

```
lua.bytecode[<number> n] = <function> f
lua.bytecode[<number> n]()
```

The contents of the `lua.bytecode` array is stored inside the format file as actual LUA bytecode, so it can also be used to preload LUA code. The function must not contain any upvalues. The associated function calls are:

```
<function> f = lua.getbytecode(<number> n)
lua.setbytecode(<number> n, <function> f)
```

Note: Since a LUA file loaded using `loadfile(filename)` is essentially an anonymous function, a complete file can be stored in a bytecode register like this:

```
lua.bytecode[n] = loadfile(filename)
```

Now all definitions (functions, variables) contained in the file can be created by executing this bytecode register:

```
lua.bytecode[n]()
```

Note that the path of the file is stored in the LUA bytecode to be used in stack backtraces and therefore dumped into the format file if the above code is used in INITEX. If it contains private information, i.e. the user name, this information is then contained in the format file as well. This should be kept in mind when preloading files into a bytecode register in INITEX.

11.1.4 Chunk name registers

There is an array of 65536 (0–65535) potential chunk names for use with the `\directlua` and `\latelua` primitives.

```
lua.name[<number> n] = <string> s
<string> s = lua.name[<number> n]
```

If you want to unset a LUA name, you can assign `nil` to it. The function accessors are:

```
lua.setluaname(<string> s,<number> n)
<string> s = lua.getluaname(<number> n)
```

11.1.5 Introspection

The `getstacktop` function return a number indicating how full the LUA stack is. This function only makes sense as breakpoint when checking some mechanism going haywire.

There are four time related helpers. The `getruntime` function returns the time passed since startup. The `getcurrenttime` does what its name says. Just play with them to see how it pays off. The `getpreciseticks` returns a number that can be used later, after a similar call, to get a difference. The `getpreciseseconds` function gets such a tick (delta) as argument and returns the number of seconds. Ticks can differ per operating system, but one always creates a reference first and then deltas to this reference.

11.2 The `status` library

This contains a number of run-time configuration items that you may find useful in message reporting, as well as an iterator function that gets all of the names and values as a table.

```
<table> info = status.list()
```

The keys in the table are the known items, the value is the current value. Almost all of the values in `status` are fetched through a metatable at run-time whenever they are accessed, so you cannot use `pairs` on `status`, but you *can* use `pairs` on `info`, of course. If you do not need the full list, you can also ask for a single item by using its name as an index into `status`. The current list is:

This table is under reconstrction!

KEY	EXPLANATION
`banner`	terminal display banner

`buf_size`	current allocated size of the line buffer
`callbacks`	total number of executed callbacks so far
`cs_count`	number of control sequences
`dest_names_size`	PDF destination table size
`dvi_gone`	written DVI bytes
`dvi_ptr`	not yet written DVI bytes
`dyn_used`	token (multi-word) memory in use
`filename`	name of the current input file
`fix_mem_end`	maximum number of used tokens
`fix_mem_min`	minimum number of allocated words for tokens
`fix_mem_max`	maximum number of allocated words for tokens
`font_ptr`	number of active fonts
`hash_extra`	extra allowed hash
`hash_size`	size of hash
`indirect_callbacks`	number of those that were themselves a result of other callbacks (e.g. file readers)
`ini_version`	`true` if this is an INITEX run
`init_pool_ptr`	INITEX string pool index
`init_str_ptr`	number of INITEX strings
`input_ptr`	the level of input we're at
`inputid`	numeric id of the current input
`largest_used_mark`	max referenced marks class
`lasterrorcontext`	last error context string (with newlines)
`lasterrorstring`	last TEX error string
`lastluaerrorstring`	last LUA error string
`lastwarningstring`	last warning tag, normally an indication of in what part
`lastwarningtag`	last warning string
`linenumber`	location in the current input file
`log_name`	name of the log file
`luabytecode_bytes`	number of bytes in LUA bytecode registers
`luabytecodes`	number of active LUA bytecode registers
`luastate_bytes`	number of bytes in use by LUA interpreters
`luatex_engine`	the LUATEX engine identifier
`luatex_hashchars`	length to which LUA hashes strings (2^n)
`luatex_hashtype`	the hash method used (in LUAJITTEX)
`luatex_version`	the LUATEX version number
`luatex_revision`	the LUATEX revision string
`max_buf_stack`	max used buffer position
`max_in_stack`	max used input stack entries
`max_nest_stack`	max used nesting stack entries
`max_param_stack`	max used parameter stack entries
`max_save_stack`	max used save stack entries
`max_strings`	maximum allowed strings
`nest_size`	nesting stack size
`node_mem_usage`	a string giving insight into currently used nodes
`obj_ptr`	max PDF object pointer

obj_tab_size	PDF object table size
output_active	true if the \output routine is active
output_file_name	name of the PDF or DVI file
param_size	parameter stack size
pdf_dest_names_ptr	max PDF destination pointer
pdf_gone	written PDF bytes
pdf_mem_ptr	max PDF memory used
pdf_mem_size	PDF memory size
pdf_os_cntr	max PDF object stream pointer
pdf_os_objidx	PDF object stream index
pdf_ptr	not yet written PDF bytes
pool_ptr	string pool index
pool_size	current size allocated for string characters
save_size	save stack size
shell_escape	0 means disabled, 1 means anything is permitted, and 2 is restricted
safer_option	1 means safer is enforced
kpse_used	1 means that kpse is used
stack_size	input stack size
str_ptr	number of strings
total_pages	number of written pages
var_mem_max	number of allocated words for nodes
var_used	variable (one-word) memory in use
lc_collate	the value of LC_COLLATE at startup time (becomes C at startup)
lc_ctype	the value of LC_CTYPE at startup time (becomes C at startup)
lc_numeric	the value of LC_NUMERIC at startup time

The error and warning messages can be wiped with the resetmessages function. A return value can be set with setexitcode.

11.3 The tex library

11.3.1 Introduction

The tex table contains a large list of virtual internal TEX parameters that are partially writable.

The designation 'virtual' means that these items are not properly defined in LUA, but are only frontends that are handled by a metatable that operates on the actual TEX values. As a result, most of the LUA table operators (like pairs and #) do not work on such items.

At the moment, it is possible to access almost every parameter that you can use after \the, is a single token or is sort of special in TEX. This excludes parameters that need extra arguments, like \the\scriptfont. The subset comprising simple integer and dimension registers are writable as well as readable (like \tracingcommands and \parindent).

11.3.2 Internal parameter values, set and get

For all the parameters in this section, it is possible to access them directly using their names as index in the tex table, or by using one of the functions tex.get and tex.set.

The exact parameters and return values differ depending on the actual parameter, and so does whether `tex.set` has any effect. For the parameters that *can* be set, it is possible to use `global` as the first argument to `tex.set`; this makes the assignment global instead of local.

```
tex.set (["global",] <string> n, ...)
... = tex.get (<string> n)
```

Glue is kind of special because there are five values involved. The return value is a `glue_spec` node but when you pass `false` as last argument to `tex.get` you get the width of the glue and when you pass `true` you get all five values. Otherwise you get a node which is a copy of the internal value so you are responsible for its freeing at the LUA end. When you set a glue quantity you can either pass a `glue_spec` or upto five numbers.

11.3.2.1 Integer parameters

The integer parameters accept and return LUA numbers. These are read-write:

```
tex.adjdemerits
tex.binoppenalty
tex.brokenpenalty
tex.catcodetable
tex.clubpenalty
tex.day
tex.defaulthyphenchar
tex.defaultskewchar
tex.delimiterfactor
tex.displaywidowpenalty
tex.doublehyphendemerits
tex.endlinechar
tex.errorcontextlines
tex.escapechar
tex.exhyphenpenalty
tex.fam
tex.finalhyphendemerits
tex.floatingpenalty
tex.globaldefs
tex.hangafter
tex.hbadness
tex.holdinginserts
tex.hyphenpenalty
tex.interlinepenalty
tex.language
tex.lastlinefit
tex.lefthyphenmin
tex.linepenalty
tex.localbrokenpenalty
tex.localinterlinepenalty
tex.looseness
tex.mag
tex.maxdeadcycles
tex.month
tex.newlinechar
tex.outputpenalty
tex.pausing
tex.postdisplaypenalty
tex.predisplaydirection
tex.predisplaypenalty
tex.pretolerance
tex.relpenalty
tex.righthyphenmin
tex.savinghyphcodes
tex.savingvdiscards
tex.showboxbreadth
tex.showboxdepth
tex.time
tex.tolerance
tex.tracingassigns
tex.tracingcommands
tex.tracinggroups
tex.tracingifs
tex.tracinglostchars
tex.tracingmacros
tex.tracingnesting
tex.tracingonline
tex.tracingoutput
tex.tracingpages
tex.tracingparagraphs
```


```
tex.tracingrestores                    tex.vbadness
tex.tracingscantokens                  tex.widowpenalty
tex.tracingstats                       tex.year
tex.uchyph
```

These are read-only:

```
tex.deadcycles              tex.interlinepenalties      tex.displaywidowpenalties
tex.insertpenalties         tex.clubpenalties           tex.prevgraf
tex.parshape                tex.widowpenalties          tex.spacefactor
```

11.3.2.2 Dimension parameters

The dimension parameters accept LUA numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points. These are read-write:

```
tex.boxmaxdepth             tex.mathsurround            tex.parindent
tex.delimitershortfall      tex.maxdepth                tex.predisplaysize
tex.displayindent           tex.nulldelimiterspace      tex.scriptspace
tex.displaywidth            tex.overfullrule            tex.splitmaxdepth
tex.emergencystretch        tex.pagebottomoffset        tex.vfuzz
tex.hangindent              tex.pageheight              tex.voffset
tex.hfuzz                   tex.pageleftoffset          tex.vsize
tex.hoffset                 tex.pagerightoffset         tex.prevdepth
tex.hsize                   tex.pagetopoffset           tex.prevgraf
tex.lineskiplimit           tex.pagewidth               tex.spacefactor
```

These are read-only:

```
tex.pagedepth               tex.pagefilstretch          tex.pagestretch
tex.pagefilllstretch        tex.pagegoal                tex.pagetotal
tex.pagefillstretch         tex.pageshrink
```

Beware: as with all LUA tables you can add values to them. So, the following is valid:

```
tex.foo = 123
```

When you access a TEX parameter a look up takes place. For read–only variables that means that you will get something back, but when you set them you create a new entry in the table thereby making the original invisible.

There are a few special cases that we make an exception for: `prevdepth`, `prevgraf` and `spacefactor`. These normally are accessed via the `tex.nest` table:

```
tex.nest[tex.nest.ptr].prevdepth   = p
tex.nest[tex.nest.ptr].spacefactor = s
```

However, the following also works:

```
tex.prevdepth   = p
tex.spacefactor = s
```

Keep in mind that when you mess with node lists directly at the LUA end you might need to update the top of the nesting stack's `prevdepth` explicitly as there is no way LUATEX can guess your intentions. By using the accessor in the `tex` tables, you get and set the values at the top of the nesting stack.

11.3.2.3 Direction parameters

The direction states can be queried and set with:

```
tex.gettextdir()          tex.getpardir()           tex.setmathdir(<number>)
tex.getlinedir()          tex.settextdir(<number>)  tex.setpardir(<number>)
tex.getmathdir()          tex.setlinedir(<number>)
```

and also with:

```
tex.textdirection         tex.mathdirection
tex.linedirection         tex.pardirection
```

11.3.2.4 Glue parameters

The glue parameters accept and return a userdata object that represents a `glue_spec` node.

```
tex.abovedisplayshortskip   tex.leftskip              tex.spaceskip
tex.abovedisplayskip        tex.lineskip              tex.splittopskip
tex.baselineskip            tex.parfillskip           tex.tabskip
tex.belowdisplayshortskip   tex.parskip               tex.topskip
tex.belowdisplayskip        tex.rightskip             tex.xspaceskip
```

11.3.2.5 Muglue parameters

All muglue parameters are to be used read-only and return a LUA string.

```
tex.medmuskip             tex.thickmuskip           tex.thinmuskip
```

11.3.2.6 Tokenlist parameters

The tokenlist parameters accept and return LUA strings. LUA strings are converted to and from token lists using `\the \toks` style expansion: all category codes are either space (10) or other (12). It follows that assigning to some of these, like 'tex.output', is actually useless, but it feels bad to make exceptions in view of a coming extension that will accept full-blown token strings.

```
tex.errhelp               tex.everyeof              tex.everymath
tex.everycr               tex.everyhbox             tex.everypar
tex.everydisplay          tex.everyjob              tex.everyvbox
```


```
tex.output
```

11.3.3 Convert commands

All 'convert' commands are read-only and return a LUA string. The supported commands at this moment are:

```
tex.formatname                      tex.uniformdeviate(number)
tex.jobname                         tex.number(number)
tex.luatexbanner                    tex.romannumeral(number)
tex.luatexrevision                  tex.fontidentifier(number)
tex.fontname(number)
```

If you are wondering why this list looks haphazard; these are all the cases of the 'convert' internal command that do not require an argument, as well as the ones that require only a simple numeric value. The special (LUA-only) case of `tex.fontidentifier` returns the `csname` string that matches a font id number (if there is one).

11.3.4 Last item commands

All 'last item' commands are read-only and return a number. The supported commands at this moment are:

```
tex.lastpenalty          tex.lastxpos             tex.currentgrouptype
tex.lastkern             tex.lastypos             tex.currentiflevel
tex.lastskip             tex.randomseed           tex.currentiftype
tex.lastnodetype         tex.luatexversion        tex.currentifbranch
tex.inputlineno          tex.currentgrouplevel
```

11.3.5 Accessing registers: `set*`, `get*` and `is*`

TEX's attributes (`\attribute`), counters (`\count`), dimensions (`\dimen`), skips (`\skip`, `\muskip`) and token (`\toks`) registers can be accessed and written to using two times five virtual sub-tables of the `tex` table:

```
tex.attribute            tex.skip                 tex.muglue
tex.count                tex.glue                 tex.toks
tex.dimen                tex.muskip
```

It is possible to use the names of relevant `\attributedef`, `\countdef`, `\dimendef`, `\skipdef`, or `\toksdef` control sequences as indices to these tables:

```
tex.count.scratchcounter = 0
enormous = tex.dimen['maxdimen']
```

In this case, LUATEX looks up the value for you on the fly. You have to use a valid `\countdef` (or

\attributedef, or \dimendef, or \skipdef, or \toksdef), anything else will generate an error (the intent is to eventually also allow <chardef tokens> and even macros that expand into a number).

The count registers accept and return LUA numbers.
The dimension registers accept LUA numbers (in scaled points) or strings (with an included absolute dimension; em and ex and px are forbidden). The result is always a number in scaled points.
The token registers accept and return LUA strings. LUA strings are converted to and from token lists using \the \toks style expansion: all category codes are either space (10) or other (12).
The skip registers accept and return glue_spec userdata node objects (see the description of the node interface elsewhere in this manual).
The glue registers are just skip registers but instead of userdata are verbose.
Like the counts, the attribute registers accept and return LUA numbers.

As an alternative to array addressing, there are also accessor functions defined for all cases, for example, here is the set of possibilities for \skip registers:

```
tex.setskip (["global",] <number> n, <node> s)
tex.setskip (["global",] <string> s, <node> s)
<node> s = tex.getskip (<number> n)
<node> s = tex.getskip (<string> s)
```

We have similar setters for count, dimen, muskip, and toks. Counters and dimen are represented by numbers, skips and muskips by nodes, and toks by strings.

Again the glue variants are not using the glue-spec userdata nodes. The setglue function accepts upto five arguments: width, stretch, shrink, stretch order and shrink order. Non-numeric values set the property to zero. The getglue function reports all five properties, unless the second argument is false in which case only the width is returned.

Here is an example using a threesome:

```
local d = tex.getdimen("foo")
if tex.isdimen("bar") then
    tex.setdimen("bar",d)
end
```

There are six extra skip (glue) related helpers:

```
tex.setglue (["global"], <number> n,
    width, stretch, shrink, stretch_order, shrink_order)
tex.setglue (["global"], <string> s,
    width, stretch, shrink, stretch_order, shrink_order)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<number> n)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<string> s)
```


The other two are `tex.setmuglue` and `tex.getmuglue`.

There are such helpers for `dimen`, `count`, `skip`, `muskip`, `box` and `attribute` registers but the glue ones are special because they have to deal with more properties.

As with the general `get` and `set` function discussed before, for the skip registers `getskip` returns a node and `getglue` returns numbers, while `setskip` accepts a node and `setglue` expects upto 5 numbers. Again, when you pass `false` as second argument to `getglue` you only get the width returned. The same is true for the `mu` variants `getmuskip`, `setmuskip`, `getmuskip` and`setmuskip`.

For tokens registers we have an alternative where a catcode table is specified:

```
tex.scantoks(0,3,"$e=mc^2$")
tex.scantoks("global",0,3,"$\int\limits^1_2$")
```

In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

There is a dedicated getter for marks: `getmark` that takes two arguments. The first argument is one of `top`, `bottom`, `first`, `splitbottom` or `splitfirst`, and the second argument is a marks class number. When no arguments are given the current maximum number of classes is returned.

When `tex.gettoks` gets an extra argument `true` it will return a table with userdata tokens.

11.3.6 Character code registers: `[get|set]*code[s]`

TEX's character code tables (`\lccode`, `\uccode`, `\sfcode`, `\catcode`, `\mathcode`, `\delcode`) can be accessed and written to using six virtual subtables of the `tex` table

```
tex.lccode                  tex.sfcode                  tex.mathcode
tex.uccode                  tex.catcode                 tex.delcode
```

The function call interfaces are roughly as above, but there are a few twists. `sfcodes` are the simple ones:

```
tex.setsfcode (["global",] <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
```

The function call interface for `lccode` and `uccode` additionally allows you to set the associated sibling at the same time:

```
tex.setlccode (["global"], <number> n, <number> lc)
tex.setlccode (["global"], <number> n, <number> lc, <number> uc)
<number> lc = tex.getlccode (<number> n)
tex.setuccode (["global"], <number> n, <number> uc)
tex.setuccode (["global"], <number> n, <number> uc, <number> lc)
<number> uc = tex.getuccode (<number> n)
```

The function call interface for `catcode` also allows you to specify a category table to use on assignment or on query (default in both cases is the current one):

```
tex.setcatcode (["global"], <number> n, <number> c)
tex.setcatcode (["global"], <number> cattable, <number> n, <number> c)
<number> lc = tex.getcatcode (<number> n)
<number> lc = tex.getcatcode (<number> cattable, <number> n)
```

The interfaces for `delcode` and `mathcode` use small array tables to set and retrieve values:

```
tex.setmathcode (["global"], <number> n, <table> mval )
<table> mval = tex.getmathcode (<number> n)
tex.setdelcode (["global"], <number> n, <table> dval )
<table> dval = tex.getdelcode (<number> n)
```

Where the table for `mathcode` is an array of 3 numbers, like this:

```
{
    <number> class,
    <number> family,
    <number> character
}
```

And the table for `delcode` is an array with 4 numbers, like this:

```
{
    <number> small_fam,
    <number> small_char,
    <number> large_fam,
    <number> large_char
}
```

You can also avoid the table:

```
tex.setmathcode (["global"], <number> n, <number> class,
    <number> family, <number> character)
class, family, char =
    tex.getmathcodes (<number> n)
tex.setdelcode (["global"], <number> n, <number> smallfam,
    <number> smallchar, <number> largefam, <number> largechar)
smallfam, smallchar, largefam, largechar =
    tex.getdelcodes (<number> n)
```

Normally, the third and fourth values in a delimiter code assignment will be zero according to `\Udelcode` usage, but the returned table can have values there (if the delimiter code was set using `\delcode`, for example). Unset `delcode`'s can be recognized because `dval[1]` is −1.

11.3.7 Box registers: `[get|set]box`

It is possible to set and query actual boxes, coming for instance from `\hbox`, `\vbox` or `\vtop`, using the node interface as defined in the `node` library:

```
tex.box
```

for array access, or

```
tex.setbox(["global",] <number> n, <node> s)
tex.setbox(["global",] <string> cs, <node> s)
<node> n = tex.getbox(<number> n)
<node> n = tex.getbox(<string> cs)
```

for function-based access. In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

Be warned that an assignment like

```
tex.box[0] = tex.box[2]
```

does not copy the node list, it just duplicates a node pointer. If `\box2` will be cleared by TEX commands later on, the contents of `\box0` becomes invalid as well. To prevent this from happening, always use `node.copy_list` unless you are assigning to a temporary variable:

```
tex.box[0] = node.copy_list(tex.box[2])
```

11.3.8 triggerbuildpage

You should not expect to much from the `triggerbuildpage` helpers because often TEX doesn't do much if it thinks nothing has to be done, but it might be useful for some applications. It just does as it says it calls the internal function that build a page, given that there is something to build.

11.3.9 splitbox

You can split a box:

```
local vlist = tex.splitbox(n,height,mode)
```

The remainder is kept in the original box and a packaged vlist is returned. This operation is comparable to the `\vsplit` operation. The mode can be `additional` or `exactly` and concerns the split off box.

11.3.10 Accessing math parameters: [get|set]math

It is possible to set and query the internal math parameters using:

```
tex.setmath(["global",] <string> n, <string> t, <number> n)
<number> n = tex.getmath(<string> n, <string> t)
```

As before an optional first parameter `global` indicates a global assignment.

The first string is the parameter name minus the leading 'Umath', and the second string is the style name minus the trailing 'style'. Just to be complete, the values for the math parameter name are:

```
quad                 axis                 operatorsize
overbarkern          overbarrule          overbarvgap
underbarkern         underbarrule         underbarvgap
radicalkern          radicalrule          radicalvgap
radicaldegreebefore  radicaldegreeafter   radicaldegreeraise
stackvgap            stacknumup           stackdenomdown
fractionrule         fractionnumvgap      fractionnumup
fractiondenomvgap    fractiondenomdown    fractiondelsize
limitabovevgap       limitabovebgap       limitabovekern
limitbelowvgap       limitbelowbgap       limitbelowkern
underdelimitervgap   underdelimiterbgap
overdelimitervgap    overdelimiterbgap
subshiftdrop         supshiftdrop         subshiftdown
subsupshiftdown      subtopmax            supshiftup
supbottommin         supsubbottommax      subsupvgap
spaceafterscript     connectoroverlapmin
ordordspacing        ordopspacing         ordbinspacing     ordrelspacing
ordopenspacing       ordclosespacing      ordpunctspacing   ordinnerspacing
opordspacing         opopspacing          opbinspacing      oprelspacing
opopenspacing        opclosespacing       oppunctspacing    opinnerspacing
binordspacing        binopspacing         binbinspacing     binrelspacing
binopenspacing       binclosespacing      binpunctspacing   bininnerspacing
relordspacing        relopspacing         relbinspacing     relrelspacing
relopenspacing       relclosespacing      relpunctspacing   relinnerspacing
openordspacing       openopspacing        openbinspacing    openrelspacing
openopenspacing      openclosespacing     openpunctspacing  openinnerspacing
closeordspacing      closeopspacing       closebinspacing   closerelspacing
closeopenspacing     closeclosespacing    closepunctspacing closeinnerspacing
punctordspacing      punctopspacing       punctbinspacing   punctrelspacing
punctopenspacing     punctclosespacing    punctpunctspacing punctinnerspacing
innerordspacing      inneropspacing       innerbinspacing   innerrelspacing
inneropenspacing     innerclosespacing    innerpunctspacing innerinnerspacing
```

The values for the style parameter are:

```
display       crampeddisplay
text          crampedtext
script        crampedscript
scriptscript  crampedscriptscript
```

The value is either a number (representing a dimension or number) or a glue spec node representing a muskip for `ordordspacing` and similar spacing parameters.

11.3.11 Special list heads: [get|set]list

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.

FIELD	EXPLANATION
page_ins_head	circular list of pending insertions
contribute_head	the recent contributions
page_head	the current page content
hold_head	used for held-over items for next page
adjust_head	head of the current \vadjust list
pre_adjust_head	head of the current \vadjust pre list
page_discards_head	head of the discarded items of a page break
split_discards_head	head of the discarded items in a vsplit

The getter and setter functions are getlist and setlist. You have to be careful with what you set as TEX can have expectations with regards to how a list is constructed or in what state it is.

11.3.12 Semantic nest levels: getnest and ptr

The virtual table nest contains the currently active semantic nesting state. It has two main parts: a zero-based array of userdata for the semantic nest itself, and the numerical value ptr, which gives the highest available index. Neither the array items in nest[] nor ptr can be assigned to (as this would confuse the typesetting engine beyond repair), but you can assign to the individual values inside the array items, e.g. tex.nest[tex.nest.ptr].prevdepth.

tex.nest[tex.nest.ptr] is the current nest state, nest[0] the outermost (main vertical list) level. The getter function is getnest. You can pass a number (which gives you a list), nothing or top, which returns the topmost list, or the string ptr which gives you the index of the topmost list.

The known fields are:

KEY	TYPE	MODES	EXPLANATION
mode	number	all	the meaning of these numbers depends on the engine and sometimes even the version; you can use tex.getmodevalues() to get the mapping: positive values signal vertical, horizontal and math mode, while negative values indicate inner and inline variants
modeline	number	all	source input line where this mode was entered in, negative inside the output routine
head	node	all	the head of the current list
tail	node	all	the tail of the current list
prevgraf	number	vmode	number of lines in the previous paragraph
prevdepth	number	vmode	depth of the previous paragraph
spacefactor	number	hmode	the current space factor
direction	node	hmode	stack used for temporary storage by the line break algorithm
noad	node	mmode	used for temporary storage of a pending fraction numerator, for \over etc.
delimptr	node	mmode	used for temporary storage of the previous math delimiter, for \middle
mathdir	boolean	mmode	true when during math processing the \mathdir is not the same as the surrounding \textdir

`mathstyle`	number	mmode	the current `\mathstyle`

When a second string argument is given to the `getnest`, the value with that name is returned. Of course the level must be valid. When `setnest` gets a third argument that value is assigned to the field given as second argument.

11.3.13 Print functions

The `tex` table also contains the three print functions that are the major interface from LUA scripting to TEX. The arguments to these three functions are all stored in an in-memory virtual file that is fed to the TEX scanner as the result of the expansion of `\directlua`.

The total amount of returnable text from a `\directlua` command is only limited by available system RAM. However, each separate printed string has to fit completely in TEX's input buffer. The result of using these functions from inside callbacks is undefined at the moment.

11.3.13.1 `print`

```
tex.print(<string> s, ...)
tex.print(<number> n, <string> s, ...)
tex.print(<table> t)
tex.print(<number> n, <table> t)
```

Each string argument is treated by TEX as a separate input line. If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional parameter can be used to print the strings using the catcode regime defined by `\catcodetable` `n`. If `n` is −1, the currently active catcode regime is used. If `n` is −2, the resulting catcodes are the result of `\the \toks`: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, if `n` is not a valid catcode table, then it is ignored, and the currently active catcode regime is used instead.

The very last string of the very last `tex.print` command in a `\directlua` will not have the `\endlinechar` appended, all others do.

11.3.13.2 `sprint`

```
tex.sprint(<string> s, ...)
tex.sprint(<number> n, <string> s, ...)
tex.sprint(<table> t)
tex.sprint(<number> n, <table> t)
```

Each string argument is treated by TEX as a special kind of input line that makes it suitable for use as a partial line input mechanism:

TEX does not switch to the 'new line' state, so that leading spaces are not ignored.
No `\endlinechar` is inserted.
Trailing spaces are not removed. Note that this does not prevent TEX itself from eating spaces as result of interpreting the line. For example, in

```
before\directlua{tex.sprint("\\relax")tex.sprint(" in between")}after
```

the space before `in between` will be gobbled as a result of the 'normal' scanning of `\relax`.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

The optional argument sets the catcode regime, as with `tex.print`. This influences the string arguments (or numbers turned into strings).

Although this needs to be used with care, you can also pass token or node userdata objects. These get injected into the stream. Tokens had best be valid tokens, while nodes need to be around when they get injected. Therefore it is important to realize the following:

When you inject a token, you need to pass a valid token userdata object. This object will be collected by LUA when it no longer is referenced. When it gets printed to TEX the token itself gets copied so there is no interference with the LUA garbage collection. You manage the object yourself. Because tokens are actually just numbers, there is no real extra overhead at the TEX end.
When you inject a node, you need to pass a valid node userdata object. The node related to the object will not be collected by LUA when it no longer is referenced. It lives on at the TEX end in its own memory space. When it gets printed to TEX the node reference is used assuming that node stays around. There is no LUA garbage collection involved. Again, you manage the object yourself. The node itself is freed when TEX is done with it.

If you consider the last remark you might realize that we have a problem when a printed mix of strings, tokens and nodes is reused. Inside TEX the sequence becomes a linked list of input buffers. So, `"123"` or `"\foo{123}"` gets read and parsed on the fly, while `<token userdata>` already is tokenized and effectively is a token list now. A `<node userdata>` is also tokenized into a token list but it has a reference to a real node. Normally this goes fine. But now assume that you store the whole lot in a macro: in that case the tokenized node can be flushed many times. But, after the first such flush the node is used and its memory freed. You can prevent this by using copies which is controlled by setting `\luacopyinputnodes` to a non-zero value. This is one of these fuzzy areas you have to live with if you really mess with these low level issues.

11.3.13.3 tprint

```
tex.tprint({<number> n, <string> s, ...}, {...})
```

This function is basically a shortcut for repeated calls to `tex.sprint(<number> n, <string> s, ...)`, once for each of the supplied argument tables.

11.3.13.4 cprint

This function takes a number indicating the to be used catcode, plus either a table of strings or an argument list of strings that will be pushed into the input stream.

```
tex.cprint( 1," 1: $&{\\foo}") tex.print("\\par") -- a lot of \bgroup s
tex.cprint( 2," 2: $&{\\foo}") tex.print("\\par") -- matching \egroup s
```

```
tex.cprint( 9," 9: $&{\\foo}") tex.print("\\par") -- all get ignored
tex.cprint(10,"10: $&{\\foo}") tex.print("\\par") -- all become spaces
tex.cprint(11,"11: $&{\\foo}") tex.print("\\par") -- letters
tex.cprint(12,"12: $&{\\foo}") tex.print("\\par") -- other characters
tex.cprint(14,"12: $&{\\foo}") tex.print("\\par") -- comment triggers
```

11.3.13.5 write

```
tex.write(<string> s, ...)
tex.write(<table> t)
```

Each string argument is treated by TEX as a special kind of input line that makes it suitable for use as a quick way to dump information:

All catcodes on that line are either 'space' (for ' ') or 'character' (for all others).
There is no `\endlinechar` appended.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process).

11.3.14 Helper functions

11.3.14.1 round

```
<number> n = tex.round(<number> o)
```

Rounds LUA number `o`, and returns a number that is in the range of a valid TEX register value. If the number starts out of range, it generates a 'number too big' error as well.

11.3.14.2 scale

```
<number> n = tex.scale(<number> o, <number> delta)
<table> n = tex.scale(table o, <number> delta)
```

Multiplies the LUA numbers `o` and `delta`, and returns a rounded number that is in the range of a valid TEX register value. In the table version, it creates a copy of the table with all numeric top–level values scaled in that manner. If the multiplied number(s) are of range, it generates 'number too big' error(s) as well.

Note: the precision of the output of this function will depend on your computer's architecture and operating system, so use with care! An interface to LUATEX's internal, 100% portable scale function will be added at a later date.

11.3.14.3 number and romannumeral

These are the companions to the primitives `\number` and `\romannumeral`. They can be used like:

```
tex.print(tex.romannumeral(123))
```


11.3.14.4 fontidentifier and fontname

The first one returns the name only, the second one reports the size too.

```
tex.print(tex.fontname(tex.fontname))
tex.print(tex.fontname(tex.fontidentidier))
```

11.3.14.5 sp

```
<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)
```

Converts the number `o` or a string `s` that represents an explicit dimension into an integer number of scaled points.

For parsing the string, the same scanning and conversion rules are used that LuaTeX would use if it was scanning a dimension specifier in its TeX-like input language (this includes generating errors for bad values), expect for the following:

1. only explicit values are allowed, control sequences are not handled
2. infinite dimension units (`fil...`) are forbidden
3. `mu` units do not generate an error (but may not be useful either)

11.3.14.6 tex.getlinenumber and tex.setlinenumber

You can mess with the current line number:

```
local n = tex.getlinenumber()
tex.setlinenumber(n+10)
```

which can be shortcut to:

```
tex.setlinenumber(10,true)
```

This might be handy when you have a callback that reads numbers from a file and combines them in one line (in which case an error message probably has to refer to the original line). Interference with TeX's internal handling of numbers is of course possible.

11.3.14.7 error, show_context and gethelptext

```
tex.error(<string> s)
tex.error(<string> s, <table> help)
<string> s = tex.gethelptext()
```

This creates an error somewhat like the combination of `\errhelp` and `\errmessage` would. During this error, deletions are disabled.

The array part of the `help` table has to contain strings, one for each line of error help.

In case of an error the `show_context` function will show the current context where we're at (in the expansion).

11.3.14.8 getfamilyoffont

When you pass a proper family identifier the next helper will return the font currently associated with it.

```
<integer> id = font.getfamilyoffont(<integer> fam)
```

11.3.14.9 [set|get]interaction

The engine can be in one of four modes:

VALUE	mode	MEANING
0	batch	omits all stops and omits terminal output
1	nonstop	omits all stops
2	scroll	omits error stops
3	errorstop	stops at every opportunity to interact

The mode can be queried and set with:

```
<integer> i = tex.getinteraction()
tex.setinteraction(<integer> i)
```

11.3.14.10 runtoks and quittoks

Because of the fact that TEX is in a complex dance of expanding, dealing with fonts, typesetting paragraphs, messing around with boxes, building pages, and so on, you cannot easily run a nested TEX run (read nested main loop). However, there is an option to force a local run with `runtoks`. The content of the given token list register gets expanded locally after which we return to where we triggered this expansion, at the LUA end. Instead a function can get passed that does some work. You have to make sure that at the end TEX is in a sane state and this is not always trivial. A more complex mechanism would complicate TEX itself (and probably also harm performance) so this simple local expansion loop has to do.

```
tex.runtoks(<token register>)
tex.runtoks(<lua function>)
tex.runtoks(<macro name>)
tex.runtoks(<register name>)
```

When the `\tracingnesting` parameter is set to a value larger than 2 some information is reported about the state of the local loop. The return value indicates an error:

VALUE	meaning
0	no error
1	bad register number
2	unknown macro or register name
3	macro is unsuitable for runtoks (has arguments)

This function has two optional arguments in case a token register is passed:

```
tex.runtoks(<token register>,force,grouped,obeymode)
```

Inside for instance an `\edef` the `runtoks` function behaves (at least tries to) like it were an `\the`. This prevents unwanted side effects: normally in such an definition tokens remain tokens and (for instance) characters don't become nodes. With the second argument you can force the local main loop, no matter what. The third argument adds a level of grouping. The last argument tells the scanner to stay in the current mode.

You can quit the local loop with `\endlocalcontrol` or from the LUA end with `tex.quittoks`. In that case you end one level up! Of course in the end that can mean that you arrive at the main level in which case an extra end will trigger a redundancy warning (not an abort!).

11.3.14.11 forcehmode

An example of a (possible error triggering) complication is that TEX expects to be in some state, say horizontal mode, and you have to make sure it is when you start feeding back something from LUA into TEX. Normally a user will not run into issues but when you start writing tokens or nodes or have a nested run there can be situations that you need to run `forcehmode`. There is no recipe for this and intercepting possible cases would weaken LUATEX's flexibility.

11.3.14.12 hashtokens

```
for i,v in pairs (tex.hashtokens()) do ... end
```

Returns a list of names. This can be useful for debugging, but note that this also reports control sequences that may be unreachable at this moment due to local redefinitions: it is strictly a dump of the hash table. You can use `token.create` to inspect properties, for instance when the `command` key in a created table equals 123, you have the `cmdname` value `undefined_cs`.

11.3.14.13 definefont

```
tex.definefont(<string> csname, <number> fontid)
tex.definefont(<boolean> global, <string> csname, <number> fontid)
```

Associates `csname` with the internal font number `fontid`. The definition is global if (and only if) `global` is specified and true (the setting of `globaldefs` is not taken into account).

11.3.15 Functions for dealing with primitives

11.3.15.1 enableprimitives

```
tex.enableprimitives(<string> prefix, <table> primitive names)
```

This function accepts a prefix string and an array of primitive names. For each combination of 'prefix' and 'name', the `tex.enableprimitives` first verifies that 'name' is an actual primitive (it must be returned by one of the `tex.extraprimitives` calls explained below, or part of TEX82, or `\directlua`). If it is not, `tex.enableprimitives` does nothing and skips to the next pair.

But if it is, then it will construct a csname variable by concatenating the 'prefix' and 'name', unless the 'prefix' is already the actual prefix of 'name'. In the latter case, it will discard the 'prefix', and just use 'name'.

Then it will check for the existence of the constructed csname. If the csname is currently undefined (note: that is not the same as `\relax`), it will globally define the csname to have the meaning: run code belonging to the primitive 'name'. If for some reason the csname is already defined, it does nothing and tries the next pair.

An example:

```
tex.enableprimitives('LuaTeX', {'formatname'})
```

will define `\LuaTeXformatname` with the same intrinsic meaning as the documented primitive `\formatname`, provided that the control sequences `\LuaTeXformatname` is currently undefined.

When LuaTeX is run with `--ini` only the TeX82 primitives and `\directlua` are available, so no extra primitives **at all**.

If you want to have all the new functionality available using their default names, as it is now, you will have to add

```
\ifx\directlua\undefined \else
    \directlua {tex.enableprimitives('',tex.extraprimitives ())}
\fi
```

near the beginning of your format generation file. Or you can choose different prefixes for different subsets, as you see fit.

Calling some form of `tex.enableprimitives` is highly important though, because if you do not, you will end up with a TeX82-lookalike that can run Lua code but not do much else. The defined csnames are (of course) saved in the format and will be available at runtime.

11.3.15.2 extraprimitives

```
<table> t = tex.extraprimitives(<string> s, ...)
```

This function returns a list of the primitives that originate from the engine(s) given by the requested string value(s). The possible values and their (current) return values are given in the following table. In addition the somewhat special primitives '`\ `', '`\/`' and '`-`' are defined.

NAME	VALUES
tex	above abovedisplayshortskip abovedisplayskip abovewithdelims accent adjdemerits advance afterassignment aftergroup atop atopwithdelims badness baselineskip batchmode begingroup belowdisplayshortskip belowdisplayskip binoppenalty botmark box boxmaxdepth brokenpenalty catcode char chardef cleaders clubpenalty copy count countdef cr crcr csname day deadcycles def defaulthyphenchar defaultskewchar delcode delimiter delimiterfactor delimitershortfall dimen dimendef discretionary displayindent displaylimits displaystyle displaywidowpenalty displaywidth divide doublehyphendemerits dp dump edef else emergencystretch end endcsname endgroup endinput endlinechar eqno er-

rhelp errmessage errorcontextlines errorstopmode escapechar everycr everydisplay everyhbox everyjob everymath everypar everyvbox exhyphenchar exhyphenpenalty expandafter fam fi finalhyphendemerits firstmark floatingpenalty font fontdimen fontname fontspecifiedname futurelet gdef global globaldefs halign hangafter hangindent hbadness hbox hfil hfill hfilneg hfuzz holdinginserts hrule hsize hskip hss ht hyphenation hyphenchar hyphenpenalty if ifcase ifcat ifdim iffalse ifhbox ifhmode ifinner ifmmode ifnum ifodd iftrue ifvbox ifvmode ifvoid ifx ignorespaces indent input inputlineno insert insertpenalties interlinepenalty jobname kern language lastbox lastkern lastpenalty lastskip lccode leaders left lefthyphenmin leftskip leqno let limits linepenalty lineskip lineskiplimit long looseness lower lowercase mark mathaccent mathbin mathchar mathchardef mathchoice mathclose mathcode mathinner mathop mathopen mathord mathpunct mathrel mathsurround maxdeadcycles maxdepth meaning meaningfull meaningless medmuskip message middle mkern month moveleft moveright mskip multiply muskip muskipdef newlinechar noalign noexpand noindent nolimits nonscript nonstopmode nulldelimiterspace nullfont number omit or ordlimits outer output outputpenalty over overfullrule overline overshoot overwithdelims pagedepth pagefilllstretch pagefillstretch pagefilstretch pagegoal pageshrink pagestretch pagetotal par parfillleftskip parfillskip parindent parshape parskip patterns pausing penalty postdisplaypenalty predisplaypenalty predisplaysize pretolerance prevdepth prevgraf radical raise relax relpenalty right righthyphenmin rightskip romannumeral scriptfont scriptscriptfont scriptscriptstyle scriptspace scriptstyle scrollmode setbox setlanguage sfcode shipout show showbox showboxbreadth showboxdepth showlists shownodedetails showthe skewchar skip skipdef spacefactor spaceskip span splitbotmark splitfirstmark splitmaxdepth splittopskip string tabskip textfont textstyle the thickmuskip thinmuskip time todimension tointeger toks toksdef tolerance topmark topskip toscaled tracingcommands tracinglostchars tracingmacros tracingonline tracingoutput tracingpages tracingparagraphs tracingrestores tracingstats uccode uchyph underline unhbox unhcopy unhpack unkern unpenalty unskip unvbox unvcopy unvpack uppercase vadjust valign vbadness vbox vcenter vfil vfill vfilneg vfuzz vrule vsize vskip vsplit vss vtop wd widowpenalty xdef xleaders xspaceskip year

core

etex botmarks clubpenalties currentgrouplevel currentgrouptype currentifbranch currentiflevel currentiftype detokenize dimexpr displaywidowpenalties everyeof firstmarks fontchardp fontcharht fontcharic fontcharwd glueexpr glueshrink glueshrinkorder gluestretch gluestretchorder gluetomu ifcsname ifdefined iffontchar interactionmode interlinepenalties lastlinefit lastnodetype marks muexpr mutoglue numexpr pagediscards parshapedimen parshapeindent parshapelength predisplaydirection protected savinghyphcodes savingvdiscards scantokens showgroups showifs showtokens splitbotmarks splitdiscards splitfirstmarks topmarks tracingalignments tracingassigns tracinggroups tracingifs tracingnesting unexpanded unless widowpenalties

luatex UUskewed UUskewedwithdelims Uabove Uabovewithdelims Uatop Uatopwithdelims Uchar Udelcode Udelcodenum Udelimiter Udelimiterover Udelimiterunder

Uhextensible Uleft Umathaccent Umathaxis Umathbinbinspacing Umathbinclos-
espacing Umathbininnerspacing Umathbinopenspacing Umathbinopspacing Umath-
binordspacing Umathbinpunctspacing Umathbinrelspacing Umathchar Umath-
charclass Umathchardef Umathcharfam Umathcharnum Umathcharnumdef Umath-
charslot Umathclass Umathclosebinspacing Umathcloseclosespacing Umath-
closeinnerspacing Umathcloseopenspacing Umathcloseopspacing Umathclose-
ordspacing Umathclosepunctspacing Umathcloserelspacing Umathcode Umathco-
denum Umathconnectoroverlapmin Umathfractiondelsize Umathfractiondenom-
down Umathfractiondenomvgap Umathfractionnumup Umathfractionnumvgap Umath-
fractionrule Umathinnerbinspacing Umathinnerclosespacing Umathinnerin-
nerspacing Umathinneropenspacing Umathinneropspacing Umathinnerordspac-
ing Umathinnerpunctspacing Umathinnerrelspacing Umathlimitabovebgap Umath-
limitabovekern Umathlimitabovevgap Umathlimitbelowbgap Umathlimitbe-
lowkern Umathlimitbelowvgap Umathnolimitsubfactor Umathnolimitsupfactor
Umathopbinspacing Umathopclosespacing Umathopenbinspacing Umathopenclos-
espacing Umathopeninnerspacing Umathopenopenspacing Umathopenopspacing
Umathopenordspacing Umathopenpunctspacing Umathopenrelspacing Umathoper-
atorsize Umathopinnerspacing Umathopopenspacing Umathopopspacing Umatho-
pordspacing Umathoppunctspacing Umathoprelspacing Umathordbinspacing Umath-
ordclosespacing Umathordinnerspacing Umathordopenspacing Umathordopspacing
Umathordordspacing Umathordpunctspacing Umathordrelspacing Umathoverbark-
ern Umathoverbarrule Umathoverbarvgap Umathoverdelimiterbgap Umathoverde-
limitervgap Umathpunctbinspacing Umathpunctclosespacing Umathpunctin-
nerspacing Umathpunctopenspacing Umathpunctopspacing Umathpunctordspac-
ing Umathpunctpunctspacing Umathpunctrelspacing Umathquad Umathradicalde-
greeafter Umathradicaldegreebefore Umathradicaldegreeraise Umathradicalk-
ern Umathradicalrule Umathradicalvgap Umathrelbinspacing Umathrelclos-
espacing Umathrelinnerspacing Umathrelopenspacing Umathrelopspacing Umath-
relordspacing Umathrelpunctspacing Umathrelrelspacing Umathskewedfrac-
tionhgap Umathskewedfractionvgap Umathspaceafterscript Umathspacebefor-
escript Umathspacingmode Umathstackdenomdown Umathstacknumup Umathstack-
vgap Umathsubshiftdown Umathsubshiftdrop Umathsubsupshiftdown Umathsub-
supvgap Umathsubtopmax Umathsupbottommin Umathsupshiftdrop Umathsupshiftup
Umathsupsubbottommax Umathunderbarkern Umathunderbarrule Umathunderbarv-
gap Umathunderdelimiterbgap Umathunderdelimitervgap Umiddle Unosubpre-
script Unosubscript Unosuperprescript Unosuperscript Uover Uoverdelimiter
Uoverwithdelims Uradical Uright Uroot Uskewed Uskewedwithdelims Ustack Us-
tartdisplaymath Ustartmath Ustopdisplaymath Ustopmath Ustyle Usubprescript
Usubscript Usuperprescript Usuperscript Uunderdelimiter Uvextensible ad-
justspacing adjustspacingshrink adjustspacingstep adjustspacingstretch
afterassigned aftergrouped aliased alignmark aligntab atendofgroup atend-
ofgrouped attribute attributedef automaticdiscretionary automatichyphen-
penalty automigrationmode begincsname beginlocalcontrol boundary boxat-
tribute boxdirection boxorientation boxtotal boxxmove boxxoffset boxymove
boxyoffset catcodetable clearmarks crampeddisplaystyle crampedscriptscript-
style crampedscriptstyle crampedtextstyle csstring defcsname directlua ede-

fcsname efcode endlocalcontrol enforced etoksapp etokspre everytab exceptionpenalty expand expandafterpars expandafterspaces expandcstoken expanded expandtoken explicitdiscretionary explicithyphenpenalty firstvalidlanguage fontid fontspecifiedsize formatname frozen futuredef futureexpand futureexpandis futureexpandisap gleaders glet glyphdatafield glyphdimensionsmode glyphoptions glyphscriptfield glyphstatefield gtoksapp gtokspre hjcode hpack hyphenationmin hyphenationmode ifabsdim ifabsnum ifarguments ifboolean ifchkdim ifchknum ifcmpdim ifcmpnum ifcondition ifcstok ifdimval ifempty ifflags ifhastok ifhastoks ifhasxtoks ifincsname ifmathparameter ifmathstyle ifnumval ifparameter iftok ignorearguments ignorepars immediate immutable initcatcodetable insertht instance integerdef lastarguments lastnamedcs lastnodesubtype leftmarginkern letcharcode letcsname letfrozen letprotected linedirection linepar localbrokenpenalty localcontrol localcontrolled localinterlinepenalty localleftbox localrightbox lpcode luabytecode luabytecodecall luacopyinputnodes luadef luaescapestring luafunction luafunctioncall luatexbanner luatexrevision luatexversion mathdelimitersmode mathdirection mathdisplayskipmode matheqnogapstep mathflattenmode mathitalicsmode mathnolimitsmode matholdmode mathpenaltiesmode mathrulesfam mathrulesmode mathrulethicknessmode mathscriptboxmode mathscriptcharmode mathscriptsmode mathstyle mathsurroundmode mathsurroundskip mutable noaligned noboundary nohrule normalizelinemode nospaces novrule orelse orunless outputbox overloaded overloadmode parattr pardirection permanent postexhyphenchar posthyphenchar prebinoppenalty predisplaygapfactor preexhyphenchar prehyphenchar prerelpenalty protrudechars protrusionboundary pxdimen quitvmode rightmarginkern rpcode savecatcodetable scantextokens setfontid snapshotpar supmarkmode swapcsvalues textdirection thewithoutunit tokenized toksapp tokspre tolerant tpack tracingfonts tracingmath unletfrozen unletprotected vpack wordboundary wrapuppar xtoksapp xtokspre

---

Note that `luatex` does not contain `directlua`, as that is considered to be a core primitive, along with all the TEX82 primitives, so it is part of the list that is returned from `'core'`.

Running `tex.extraprimitives` will give you the complete list of primitives `-ini` startup. It is exactly equivalent to `tex.extraprimitives("etex","luatex")`.

11.3.15.3 primitives

```
<table> t = tex.primitives()
```

This function returns a list of all primitives that LUATEX knows about.

11.3.16 Core functionality interfaces

11.3.16.1 badness

```
<number> b = tex.badness(<number> t, <number> s)
```

This helper function is useful during linebreak calculations. `t` and `s` are scaled values; the function returns the badness for when total `t` is supposed to be made from amounts that sum to `s`. The returned number is a reasonable approximation of $100(t/s)^3$;

11.3.16.2 tex.resetparagraph

This function resets the parameters that TEX normally resets when a new paragraph is seen.

11.3.16.3 linebreak

```
local <node> nodelist, <table> info =
    tex.linebreak(<node> listhead, <table> parameters)
```

The understood parameters are as follows:

NAME	TYPE	EXPLANATION
pardir	string	
pretolerance	number	
tracingparagraphs	number	
tolerance	number	
looseness	number	
hyphenpenalty	number	
exhyphenpenalty	number	
pdfadjustspacing	number	
adjdemerits	number	
pdfprotrudechars	number	
linepenalty	number	
lastlinefit	number	
doublehyphendemerits	number	
finalhyphendemerits	number	
hangafter	number	
interlinepenalty	number or table	if a table, then it is an array like \interlinepenalties
clubpenalty	number or table	if a table, then it is an array like \clubpenalties
widowpenalty	number or table	if a table, then it is an array like \widowpenalties
brokenpenalty	number	
emergencystretch	number	in scaled points
hangindent	number	in scaled points
hsize	number	in scaled points
leftskip	glue_spec node	
rightskip	glue_spec node	
parshape	table	

Note that there is no interface for `\displaywidowpenalties`, you have to pass the right choice for `widowpenalties` yourself.

It is your own job to make sure that `listhead` is a proper paragraph list: this function does not add any nodes to it. To be exact, if you want to replace the core line breaking, you may

have to do the following (when you are not actually working in the `pre_linebreak_filter` or `linebreak_filter` callbacks, or when the original list starting at listhead was generated in horizontal mode):

add an 'indent box' and perhaps a `par` node at the start (only if you need them)
replace any found final glue by an infinite penalty (or add such a penalty, if the last node is not a glue)
add a glue node for the `\parfillskip` after that penalty node
make sure all the `prev` pointers are OK

The result is a node list, it still needs to be vpacked if you want to assign it to a `\vbox`. The returned `info` table contains four values that are all numbers:

NAME	EXPLANATION
prevdepth	depth of the last line in the broken paragraph
prevgraf	number of lines in the broken paragraph
looseness	the actual looseness value in the broken paragraph
demerits	the total demerits of the chosen solution

Note there are a few things you cannot interface using this function: You cannot influence font expansion other than via `pdfadjustspacing`, because the settings for that take place elsewhere. The same is true for hbadness and hfuzz etc. All these are in the `hpack` routine, and that fetches its own variables via globals.

11.3.16.4 shipout

`tex.shipout(<number> n)`

Ships out box number `n` to the output file, and clears the box register.

11.3.16.5 getpagestate

This helper reports the current page state: `empty`, `box_there` or `inserts_only` as integer value.

11.3.16.6 getlocallevel

This integer reports the current level of the local loop. It's only useful for debugging and the (relative state) numbers can change with the implementation.

11.3.17 Randomizers

For practical reasons LuaTeX has its own random number generator. The original Lua random function is available as `tex.lua_math_random`. You can initialize with a new seed with `init_rand` (`lua_math_randomseed` is equivalent to this one).

There are three generators: `normal_rand` (no argument is used), `uniform_rand` (takes a number that will get rounded before being used) and `uniformdeviate` which behaves like the primitive and expects a scaled integer, so

```
tex.print(tex.uniformdeviate(65536)/65536)
```

will give a random number between zero and one.

11.3.18 Functions related to synctex

The next helpers only make sense when you implement your own synctex logic. Keep in mind that the library used in editors assumes a certain logic and is geared for plain and LaTeX, so after a decade users expect a certain behaviour.

NAME	EXPLANATION
set_synctex_mode	0 is the default and used normal synctex logic, 1 uses the values set by the next helpers while 2 also sets these for glyph nodes; 3 sets glyphs and glue and 4 sets only glyphs
set_synctex_tag	set the current tag (file) value (obeys save stack)
set_synctex_line	set the current line value (obeys save stack)
set_synctex_no_files	disable synctex file logging
get_synctex_mode	returns the current mode (for values see above)
get_synctex_tag	get the currently set value of tag (file)
get_synctex_line	get the currently set value of line
force_synctex_tag	overload the tag (file) value (0 resets)
force_synctex_line	overload the line value (0 resets)

The last one is somewhat special. Due to the way files are registered in SyncTeX we need to explicitly disable that feature if we provide our own alternative if we want to avoid that overhead. Passing a value of 1 disables registering.

11.4 The texconfig table

This is a table that is created empty. A startup Lua script could fill this table with a number of settings that are read out by the executable after loading and executing the startup file.

KEY	TYPE	DEFAULT
max_strings	number	100000
strings_free	number	100
nest_size	number	50
max_in_open	number	100
param_size	number	60
save_size	number	5000
stack_size	number	500
expand_depth	number	1000
function_size	number	0
error_line	number	79
half_error_line	number	50
hash_extra	number	0
formatname	string	
jobname	string	

If no format name or jobname is given on the command line, the related keys will be tested first instead of simply quitting.

11.5 The texio library

This library takes care of the low-level I/O interface: writing to the log file and/or console.

11.5.1 write

```
texio.write(<string> target, <string> s, ...)
texio.write(<string> s, ...)
```

Without the `target` argument, writes all given strings to the same location(s) TEX writes messages to at this moment. If `\batchmode` is in effect, it writes only to the log, otherwise it writes to the log and the terminal. The optional `target` can be one of `terminal`, `logfile` or `terminal_and_logfile`.

Note: If several strings are given, and if the first of these strings is or might be one of the targets above, the `target` must be specified explicitly to prevent LUA from interpreting the first string as the target.

11.5.2 write_nl

```
texio.write_nl(<string> target, <string> s, ...)
texio.write_nl(<string> s, ...)
```

This function behaves like `texio.write`, but makes sure that the given strings will appear at the beginning of a new line. You can pass a single empty string if you only want to move to the next line.

11.5.3 setescape

You can disable ^^ escaping of control characters by passing a value of zero.

11.5.4 closeinput

This function should be used with care. It acts as `\endinput` but at the LUA end. You can use it to (sort of) force a jump back to TEX. Normally a LUA call will just collect prints and at the end bump an input level and flush these prints. This function can help you stay at the current level but you need to know what you're doing (or more precise: what TEX is doing with input).

11.6 The token library

11.6.1 The scanner

The token library provides means to intercept the input and deal with it at the LUA level. The library provides a basic scanner infrastructure that can be used to write macros that accept a

wide range of arguments. This interface is on purpose kept general and as performance is quite okay so one can build additional parsers without too much overhead. It's up to macro package writers to see how they can benefit from this as the main principle behind LuaTeX is to provide a minimal set of tools and no solutions. The scanner functions are probably the most intriguing.

FUNCTION	ARGUMENT	RESULT
`scan_keyword`	string	returns true if the given keyword is gobbled; as with the regular TeX keyword scanner this is case insensitive (and ASCII based)
`scan_keywordcs`	string	returns true if the given keyword is gobbled; this variant is case sensitive and also suitable for UTF8
`scan_int`		returns an integer
`scan_real`		returns a number from e.g. `1`, `1.1`, `.1` with optional collapsed signs
`scan_float`		returns a number from e.g. `1`, `1.1`, `.1`, `1.1E10`, , `.1e-10` with optional collapsed signs
`scan_dimen`	infinity, mu-units	returns a number representing a dimension or two numbers being the filler and order
`scan_glue`	mu-units	returns a glue spec node
`scan_toks`	definer, expand	returns a table of tokens
`scan_code`	bitset	returns a character if its category is in the given bitset (representing catcodes)
`scan_string`		returns a string given between `{}`, as `\macro` or as sequence of characters with catcode 11 or 12
`scan_argument`		this one is simular to `scanstring` but also accepts a `\cs` (which then get expanded)
`scan_word`		returns a sequence of characters with catcode 11 or 12 as string
`scan_csname`		returns `foo` after scanning `\foo`
`scan_list`		picks up a box specification and returns a `[h|v]list` node

The integer, dimension and glue scanners take an extra optional argument that signals that en optional equal is permitted.

The scanners can be considered stable apart from the one scanning for a token. The `scan_code` function takes an optional number, the `scan_keyword` function a normal Lua string. The `infinity` boolean signals that we also permit `fill` as dimension and the `mu-units` flags the scanner that we expect math units. When scanning tokens we can indicate that we are defining a macro, in which case the result will also provide information about what arguments are expected and in the result this is separated from the meaning by a separator token. The `expand` flag determines if the list will be expanded.

The `scan_argument` function expands the given argument. When a braced argument is scanned, expansion can be prohibited by passing `false` (default is `true`). In case of a control sequence passing `false` will result in a one-level expansion (the meaning of the macro).

The string scanner scans for something between curly braces and expands on the way, or when it sees a control sequence it will return its meaning. Otherwise it will scan characters with catcode `letter` or `other`. So, given the following definition:

```
\def\bar{bar}
\def\foo{foo-\bar}
```

we get:

NAME	RESULT
`\directlua{token.scan_string()}{foo}`	full expansion
`\directlua{token.scan_string()}foo`	letters and others
`\directlua{token.scan_string()}\foo`	meaning

The `\foo` case only gives the meaning, but one can pass an already expanded definition (`\edef`'d). In the case of the braced variant one can of course use the `\detokenize` and `\unexpanded` primitives since there we do expand.

The `scan_word` scanner can be used to implement for instance a number scanner. An optional boolean argument can signal that a trailing space or `\relax` should be gobbled:

```
function token.scan_number(base)
    return tonumber(token.scan_word(),base)
end
```

This scanner accepts any valid LUA number so it is a way to pick up floats in the input.

You can use the LUA interface as follows:

```
\directlua {
    function mymacro(n)
        ...
    end
}

\def\mymacro#1{%
    \directlua {
        mymacro(\number\dimexpr#1)
    }%
}

\mymacro{12pt}
\mymacro{\dimen0}
```

You can also do this:

```
\directlua {
    function mymacro()
        local d = token.scan_dimen()
        ...
    end
}

\def\mymacro{%
```

```
    \directlua {
        mymacro()
    }%
}

\mymacro 12pt
\mymacro \dimen0
```

It is quite clear from looking at the code what the first method needs as argument(s). For the second method you need to look at the LUA code to see what gets picked up. Instead of passing from TEX to LUA we let LUA fetch from the input stream.

In the first case the input is tokenized and then turned into a string, then it is passed to LUA where it gets interpreted. In the second case only a function call gets interpreted but then the input is picked up by explicitly calling the scanner functions. These return proper LUA variables so no further conversion has to be done. This is more efficient but in practice (given what TEX has to do) this effect should not be overestimated. For numbers and dimensions it saves a bit but for passing strings conversion to and from tokens has to be done anyway (although we can probably speed up the process in later versions if needed).

11.6.2 Picking up one token

The scanners look for a sequence. When you want to pick up one token from the input you use `scan_next`. This creates a token with the (low level) properties as discussed next. This token is just the next one. If you want to enforce expansion first you can use `scan_token` or the `_expanded` variants. Internally tokens are characterized by a number that packs a lot of information. In order to access the bits of information a token is wrapped in a userdata object.

The `expand` function will trigger expansion of the next token in the input. This can be quite unpredictable but when you call it you probably know enough about TEX not to be too worried about that. It basically is a call to the internal expand related function.

NAME	EXPLANATION
`scan_next`	get the next token
`scan_next_expanded`	get the next expanded token
`skip_next`	skip the next token
`skip_next_expanded`	skip the next expanded token
`peek_next`	get the next token and put it back in the input
`peek_next_expanded`	get the next expanded token and put it back in the input

The peek function accept a boolean argument that triggers skipping spaces and alike.

11.6.3 Creating tokens

The creator function can be used as follows:

```
local t = token.create("relax")
```


This gives back a token object that has the properties of the `\relax` primitive. The possible properties of tokens are:

NAME	EXPLANATION
`command`	a number representing the internal command number
`cmdname`	the type of the command (for instance the catcode in case of a character or the classifier that determines the internal treatment)
`csname`	the associated control sequence (if applicable)
`id`	the unique id of the token
`tok`	the full token number as stored in TEX
`active`	a boolean indicating the active state of the token
`expandable`	a boolean indicating if the token (macro) is expandable
`protected`	a boolean indicating if the token (macro) is protected
`frozen`	a boolean indicating if the token is a frozen command
`user`	a boolean indicating if the token is a user defined command
`index`	a number that indicated the subcommand; differs per command

Alternatively you can use a getter `get_<fieldname>` to access a property of a token.

The numbers that represent a catcode are the same as in TEX itself, so using this information assumes that you know a bit about TEX's internals. The other numbers and names are used consistently but are not frozen. So, when you use them for comparing you can best query a known primitive or character first to see the values.

You can ask for a list of commands:

```
local t = token.commands()
```

The id of a token class can be queried as follows:

```
local id = token.command_id("math_shift")
```

If you really know what you're doing you can create character tokens by not passing a string but a number:

```
local letter_x = token.create(string.byte("x"))
local other_x  = token.create(string.byte("x"),12)
```

Passing weird numbers can give side effects so don't expect too much help with that. As said, you need to know what you're doing. The best way to explore the way these internals work is to just look at how primitives or macros or `\chardef`'d commands are tokenized. Just create a known one and inspect its fields. A variant that ignores the current catcode table is:

```
local whatever = token.new(123,12)
```

You can test if a control sequence is defined with `is_defined`, which accepts a string and returns a boolean:

```
local okay = token.is_defined("foo")
```

The largest character possible is returned by `biggest_char`, just in case you need to know that boundary condition.

11.6.4 Macros

The `set_macro` function can get upto 4 arguments:

```
set_macro("csname","content")
set_macro("csname","content","global")
set_macro("csname")
```

You can pass a catcodetable identifier as first argument:

```
set_macro(catcodetable,"csname","content")
set_macro(catcodetable,"csname","content","global")
set_macro(catcodetable,"csname")
```

The results are like:

```
 \def\csname{content}
\gdef\csname{content}
 \def\csname{}
```

The `get_macro` function can be used to get the content of a macro while the `get_meaning` function gives the meaning including the argument specification (as usual in TEX separated by `->`).

The `set_char` function can be used to do a `\chardef` at the LUA end, where invalid assignments are silently ignored:

```
set_char("csname",number)
set_char("csname",number,"global")
```

A special one is the following:

```
set_lua("mycode",id)
set_lua("mycode",id,"global","protected")
```

This creates a token that refers to a LUA function with an entry in the table that you can access with `lua.get_functions_table`. It is the companion to `\luadef`. When the first (and only) argument is true the size will preset to the value of `texconfig.function_size`.

The `push_macro` and `pop_macro` function are very experimental and can be used to get and set an existing macro. The push call returns a user data object and the pop takes such a userdata object. These object have no accessors and are to be seen as abstractions.

11.6.5 Pushing back

There is a (for now) experimental putter:

```
local t1 = token.scan_next()
```


```
local t2 = token.scan_next()
local t3 = token.scan_next()
local t4 = token.scan_next()
-- watch out, we flush in sequence
token.put_next { t1, t2 }
-- but this one gets pushed in front
token.put_next ( t3, t4 )
```

When we scan `wxyz!` we get `yzwx!` back. The argument is either a table with tokens or a list of tokens. The `token.expand` function will trigger expansion but what happens really depends on what you're doing where.

This putter is actually a bit more flexible because the following input also works out okay:

```
\def\foo#1{[#1]}

\directlua {
    local list = { 101, 102, 103, token.create("foo"), "{abracadabra}" }
    token.put_next("(the)")
    token.put_next(list)
    token.put_next("(order)")
    token.put_next(unpack(list))
    token.put_next("(is reversed)")
}
```

We get this:

```
(is reversed)efg[abracadabra](order)efg[abracadabra](the)
```

So, strings get converted to individual tokens according to the current catcode regime and numbers become characters also according to this regime.

11.6.6 Nota bene

When scanning for the next token you need to keep in mind that we're not scanning like TEX does: expanding, changing modes and doing things as it goes. When we scan with LUA we just pick up tokens. Say that we have:

```
\bar
```

but `\bar` is undefined. Normally TEX will then issue an error message. However, when we have:

```
\def\foo{\bar}
```

We get no error, unless we expand `\foo` while `\bar` is still undefined. What happens is that as soon as TEX sees an undefined macro it will create a hash entry and when later it gets defined that entry will be reused. So, `\bar` really exists but can be in an undefined state.

bar : bar

foo : foo
myfirstbar :

This was entered as:

```
bar        : \directlua{tex.print(token.scan_csname())}\bar
foo        : \directlua{tex.print(token.scan_csname())}\foo
myfirstbar : \directlua{tex.print(token.scan_csname())}\myfirstbar
```

The reason that you see `bar` reported and not `myfirstbar` is that `\bar` was already used in a previous paragraph.

If we now say:

```
\def\foo{}
```

we get:

bar : bar
foo : foo
myfirstbar :

And if we say

```
\def\foo{\bar}
```

we get:

bar : bar
foo : foo
myfirstbar :

When scanning from LUA we are not in a mode that defines (undefined) macros at all. There we just get the real primitive undefined macro token.

723587 537472629
721910 536968956
722526 536985885

This was generated with:

```
\directlua{local t = token.scan_next() tex.print(t.id.." "..t.tok)}\myfirstbar
\directlua{local t = token.scan_next() tex.print(t.id.." "..t.tok)}\mysecondbar
\directlua{local t = token.scan_next() tex.print(t.id.." "..t.tok)}\mythirdbar
```

So, we do get a unique token because after all we need some kind of LUA object that can be used and garbage collected, but it is basically the same one, representing an undefined control sequence.

12 The METAPOST library mplib

12.1 Process management

The METAPOST library interface registers itself in the table mplib. It is based on MPLIB version 3.03.

12.1.1 new

To create a new METAPOST instance, call

```
<mpinstance> mp = mplib.new({...})
```

This creates the mp instance object. The argument is a hash table that can have a number of different fields, as follows:

NAME	TYPE	DESCRIPTION	DEFAULT
error_line	number	error line width	79
print_line	number	line length in ps output	100
random_seed	number	the initial random seed	variable
math_mode	string	the number system to use: scaled, double or decimal	scaled
interaction	string	the interaction mode: batch, nonstop, scroll or errorstop	errorstop
job_name	string	a compatibility value	mpout
find_file	function	a function to find files	only local files
utf8_mode	boolean	permit characters in the range 128 upto 255 to be part of names	false
text_mode	boolean	permit characters 2 and 3 as fencing string literals	false

The binary mode is no longer available in the LUATEX version of MPLIB. It offers no real advantage and brings a ton of extra libraries with platform specific properties that we can now avoid. We might introduce a high resolution scaled variant at some point but only when it pays of performance wise.

The find_file function should be of this form:

```
<string> found = finder (<string> name, <string> mode, <string> type)
```

with:

NAME	THE REQUESTED FILE
mode	the file mode: r or w
type	the kind of file, one of: mp, tfm, map, pfb, enc

Return either the full path name of the found file, or nil if the file cannot be found.

Note that the new version of MPLIB no longer uses binary mem files, so the way to preload a set of macros is simply to start off with an `input` command in the first `execute` call.

When you are processing a snippet of text starting with `btex` or `verbatimtex` and ending with `etex`, the METAPOST `texscriptmode` parameter controls how spaces and newlines get honoured. The default value is 1. Possible values are:

NAME	MEANING
0	no newlines
1	newlines in `verbatimtex`
2	newlines in `verbatimtex` and `etex`
3	no leading and trailing strip in `verbatimtex`
4	no leading and trailing strip in `verbatimtex` and `btex`

That way the LUA handler (assigned to `make_text`) can do what it likes. An `etex` has to be followed by a space or `;` or be at the end of a line and preceded by a space or at the beginning of a line.

12.1.2 statistics

You can request statistics with:

```
<table> stats = mp:statistics()
```

This function returns the vital statistics for an MPLIB instance. There are four fields, giving the maximum number of used items in each of four allocated object classes:

FIELD	TYPE	EXPLANATION
`main_memory`	number	memory size
`hash_size`	number	hash size
`param_size`	number	simultaneous macro parameters
`max_in_open`	number	input file nesting levels

Note that in the new version of MPLIB, this is informational only. The objects are all allocated dynamically, so there is no chance of running out of space unless the available system memory is exhausted.

12.1.3 execute

You can ask the METAPOST interpreter to run a chunk of code by calling

```
<table> rettable = execute(mp,"metapost code")
```

for various bits of METAPOST language input. Be sure to check the `rettable.status` (see below) because when a fatal METAPOST error occurs the MPLIB instance will become unusable thereafter.

Generally speaking, it is best to keep your chunks small, but beware that all chunks have to obey proper syntax, like each of them is a small file. For instance, you cannot split a single statement over multiple chunks.

In contrast with the normal stand alone `mpost` command, there is *no* implied 'input' at the start of the first chunk.

12.1.4 `finish`

```
<table> rettable = finish(mp)
```

If for some reason you want to stop using an MPLIB instance while processing is not yet actually done, you can call `finish`. Eventually, used memory will be freed and open files will be closed by the LUA garbage collector, but an explicit `finish` is the only way to capture the final part of the output streams.

12.2 The end result

The return value of `execute` and `finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

FIELD	TYPE	EXPLANATION
`log`	string	output to the 'log' stream
`term`	string	output to the 'term' stream
`error`	string	output to the 'error' stream (only used for 'out of memory')
`status`	number	the return value: `0` = good, `1` = warning, `2` = errors, `3` = fatal error
`fig`	table	an array of generated figures (if any)

When `status` equals 3, you should stop using this MPLIB instance immediately, it is no longer capable of processing input.

If it is present, each of the entries in the `fig` array is a userdata representing a figure object, and each of those has a number of object methods you can call:

FIELD	TYPE	EXPLANATION
`boundingbox`	function	returns the bounding box, as an array of 4 values
`postscript`	function	returns a string that is the ps output of the `fig`. this function accepts two optional integer arguments for specifying the values of `prologues` (first argument) and `procset` (second argument)
`svg`	function	returns a string that is the svg output of the `fig`. This function accepts an optional integer argument for specifying the value of `prologues`
`objects`	function	returns the actual array of graphic objects in this `fig`
`copy_objects`	function	returns a deep copy of the array of graphic objects in this `fig`
`filename`	function	the filename this `fig`'s POSTSCRIPT output would have written to in stand alone mode
`width`	function	the `fontcharwd` value
`height`	function	the `fontcharht` value
`depth`	function	the `fontchardp` value
`italcorr`	function	the `fontcharit` value
`charcode`	function	the (rounded) `charcode` value

Note: you can call `fig:objects()` only once for any one `fig` object!

When the boundingbox represents a 'negated rectangle', i.e. when the first set of coordinates is larger than the second set, the picture is empty.

Graphical objects come in various types: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `special`. Each type has a different list of accessible values.

There is a helper function (`mplib.fields(obj)`) to get the list of accessible values for a particular object, but you can just as easily use the tables given below.

All graphical objects have a field `type` that gives the object type as a string value; it is not explicit mentioned in the following tables. In the following, `numbers` are POSTSCRIPT points represented as a floating point number, unless stated otherwise. Field values that are of type `table` are explained in the next section.

12.2.1 fill

FIELD	TYPE	EXPLANATION
`path`	table	the list of knots
`htap`	table	the list of knots for the reversed trajectory
`pen`	table	knots of the pen
`color`	table	the object's color
`linejoin`	number	line join style (bare number)
`miterlimit`	number	miterlimit
`prescript`	string	the prescript text
`postscript`	string	the postscript text

The entries `htap` and `pen` are optional.

12.2.2 outline

FIELD	TYPE	EXPLANATION
`path`	table	the list of knots
`pen`	table	knots of the pen
`color`	table	the object's color
`linejoin`	number	line join style (bare number)
`miterlimit`	number	miterlimit
`linecap`	number	line cap style (bare number)
`dash`	table	representation of a dash list
`prescript`	string	the prescript text
`postscript`	string	the postscript text

The entry `dash` is optional.

12.2.3 text

FIELD	TYPE	EXPLANATION
`text`	string	the text

font	string	font tfm name
dsize	number	font size
color	table	the object's color
width	number	
height	number	
depth	number	
transform	table	a text transformation
prescript	string	the prescript text
postscript	string	the postscript text

12.2.4 special

FIELD	TYPE	EXPLANATION
prescript	string	special text

12.2.5 start_bounds, start_clip

FIELD	TYPE	EXPLANATION
path	table	the list of knots

12.2.5.1 stop_bounds, stop_clip

Here are no fields available.

12.3 Subsidiary table formats

12.3.1 Paths and pens

Paths and pens (that are really just a special type of paths as far as MPLIB is concerned) are represented by an array where each entry is a table that represents a knot.

FIELD	TYPE	EXPLANATION
left_type	string	when present: endpoint, but usually absent
right_type	string	like left_type
x_coord	number	X coordinate of this knot
y_coord	number	Y coordinate of this knot
left_x	number	X coordinate of the precontrol point of this knot
left_y	number	Y coordinate of the precontrol point of this knot
right_x	number	X coordinate of the postcontrol point of this knot
right_y	number	Y coordinate of the postcontrol point of this knot

There is one special case: pens that are (possibly transformed) ellipses have an extra key type with value elliptical besides the array part containing the knot list.

12.3.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

FIELD	TYPE	EXPLANATION
`0`	marking only	no values
`1`	greyscale	one value in the range $(0, 1)$, 'black' is 0
`3`	RGB	three values in the range $(0, 1)$, 'black' is $0, 0, 0$
`4`	CMYK	four values in the range $(0, 1)$, 'black' is $0, 0, 0, 1$

If the color model of the internal object was `uninitialized`, then it was initialized to the values representing 'black' in the colorspace `defaultcolormodel` that was in effect at the time of the `shipout`.

12.3.3 Transforms

Each transform is a six-item array.

INDEX	TYPE	EXPLANATION
`1`	number	represents x
`2`	number	represents y
`3`	number	represents xx
`4`	number	represents yx
`5`	number	represents xy
`6`	number	represents yy

Note that the translation (index 1 and 2) comes first. This differs from the ordering in POSTSCRIPT, where the translation comes last.

12.3.4 Dashes

Each `dash` is a hash with two items. We use the same model as POSTSCRIPT for the representation of the dashlist. `dashes` is an array of 'on' and 'off', values, and `offset` is the phase of the pattern.

FIELD	TYPE	EXPLANATION
`dashes`	hash	an array of on-off numbers
`offset`	number	the starting offset value

12.3.5 Pens and `pen_info`

There is helper function (`pen_info(obj)`) that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

FIELD	TYPE	EXPLANATION
`width`	number	width of the pen
`sx`	number	x scale

rx	number	$x\,y$ multiplier
ry	number	$y\,x$ multiplier
sy	number	y scale
tx	number	x offset
ty	number	y offset

12.3.6 Character size information

These functions find the size of a glyph in a defined font. The `fontname` is the same name as the argument to `infont`; the `char` is a glyph id in the range 0 to 255; the returned `w` is in AFM units.

```
<number> w = char_width(mp,<string> fontname, <number> char)
<number> h = char_height(mp,<string> fontname, <number> char)
<number> d = char_depth(mp,<string> fontname, <number> char)
```

12.4 Scanners

After a relative long period of testing the scanners are now part of the interface. That doesn't mean that there will be no changes: depending on the needs and experiences details might evolve. The summary below is there still preliminary and mostly provided as reminder.

SCANNER	ARGUMENT	RETURNS
`scan_next`	instance, keep	token, mode, type
`scan_expression`	instance, keep	type
`scan_token`	instance, keep	token, mode, kind
`scan_symbol`	instance, keep, expand	string
`scan_numeric`	instance, type	number
`scan_integer`	instance, type	integer
`scan_boolean`	instance, type	boolean
`scan_string`	instance, type	string
`scan_pair`	instance, hashed, type	table or two numbers
`scan_color`	instance, hashed, type	table or three numbers
`scan_cmykcolor`	instance, hashed, type	table or four numbers
`scan_transform`	instance, hashed, type	table or six numbers
`scan_path`	instance, hashed, type	table with hashes or arrays
`scan_pen`	instance, hashed, type	table with hashes or arrays

The types and token codes are numbers but they actually depend on the implementation (although changes are unlikely). The types of data structures can be queried with `mplib.gettypes()`:

0: undefined, 1: vacuous, 2: boolean, 3: unknownboolean, 4: string, 5: unknownstring, 6: pen, 7: unknownpen, 8: path, 9: unknownpath, 10: picture, 11: unknownpicture, 12: transform, 13: color, 14: cmykcolor, 15: pair, 16: numeric, 17: known, 18: dependent, 19: protodependent, 20: independent, 21: tokenlist, 22: structured, 23: unsuffixedmacro, 24: suffixedmacro.

The command codes are available with `mplib.getcodes()`:

0: undefined, 1: btex, 2: etex, 3: if, 4: fi_or_else, 5: input, 6: iteration, 7: repeat_loop, 8: exitif, 9: relax, 10: scantokens, 11: runscript, 12: maketext, 13: expandafter, 14: definedmacro, 15: save, 16: interim, 17: let, 18: newinternal, 19: def, 20: shipout, 21: addto, 22: setbounds, 23: scope, 24: show, 25: mode, 26: randomseed, 27: message, 28: everyjob, 29: delimiters, 30: special, 31: write, 32: declare, 33: leftdelimiter, 34: begingroup, 35: nullary, 36: unary, 37: str, 38: void, 39: cycle, 40: ofbinary, 41: capsule, 42: string, 43: internal, 44: tag, 45: numeric, 46: plus_or_minus, 47: secondarydef, 48: tertiarybinary, 49: leftbrace, 50: join, 51: ampersand, 52: tertiarydef, 53: primarybinary, 54: equals, 55: and, 56: primarydef, 57: slash, 58: secondarybinary, 59: parametertype, 60: controls, 61: tension, 62: atleast, 63: curl, 64: macrospecial, 65: rightdelimiter, 66: leftbracket, 67: rightbracket, 68: rightbrace, 69: with, 70: thingstoadd, 71: of, 72: to, 73: step, 74: until, 75: within, 76: assignment, 77: skip, 78: colon, 79: comma, 80: semicolon, 81: endgroup, 82: stop, 83: outertag, 84: undefinedcs.

Now, if you really want to use these, keep in mind that the internals of METAPOST are not trivial, especially because expression scanning can be complex. So you need to experiment a bit. In CONTEXT all is (and will be) hidden below an abstraction layer so users are not bothered by all these look-ahead and push-back issues that originate in the way METAPOST scans its input.

12.5 Injectors

It is important to know that piping code into the library is pretty fast and efficient. Most processing time relates to memory management, calculations and generation of output can not be neglected either. Out of curiousity I added some functions that directly push data into the library but the gain is not that large.[7]

SCANNER	ARGUMENT
`inject_numeric`	instance, number
`inject_integer`	instance, number
`inject_boolean`	instance, boolean
`inject_string`	instance, string
`inject_pair`	instance, (table with) two numbers
`inject_color`	instance, (table with) three numbers
`inject_cmykcolor`	instance, (table with) four numbers
`inject_transform`	instance, (table with) six numbers
`inject_path`	instance, table with hashes or arrays, cycle, variant

The path injector takes a table with subtables that are either hashed (like the path solver) or arrays with two, four or six entries. When the third argument has the value `true` the path is closed. When the fourth argument is `true` the path is constructed out of straight lines (as with `--`) by setting the `curl` values to 1 automatically.[8]

[7] The main motivation was checking of huge paths could be optimized. The other data structures were then added for completeness.

[8] This is all experimental so future versions might provide more control.

13 The PDF related libraries

13.1 The pdfe library

13.1.1 Introduction

The pdfe library replaces the epdf library and provides an interface to PDF files. It uses the same code as is used for PDF image inclusion. The pplib library by Paweł Jackowski replaces the poppler (derived from xpdf) library.

A PDF file is basically a tree of objects and one descends into the tree via dictionaries (key/value) and arrays (index/value). There are a few topmost dictionaries that start at root that are accessed more directly.

Although everything in PDF is basically an object we only wrap a few in so called userdata LUA objects.

TYPE	MAPPING
null	nil
boolean	boolean
integer	integer
float	number
name	string
string	string
array	array userdatum
dictionary	dictionary userdatum
stream	stream userdatum (with related dictionary)
reference	reference userdatum

The regular getters return these LUA data types but one can also get more detailed information.

13.1.2 open, openfile, new, getstatus, close, unencrypt

A document is loaded from a file (by name or handle) or string:

```
<pdfe document> = pdfe.open(filename)
<pdfe document> = pdfe.openfile(filehandle)
<pdfe document> = pdfe.new(somestring,somelength)
```

Such a document is closed with:

```
pdfe.close(<pdfe document>)
```

You can check if a document opened well by:

```
pdfe.getstatus(<pdfe document>)
```


The returned codes are:

VALUE	EXPLANATION
-2	the document failed to open
-1	the document is (still) protected
0	the document is not encrypted
2	the document has been unencrypted

An encrypted document can be unencrypted by the next command where instead of either password you can give `nil`:

```
pdfe.unencrypt(<pdfe document>,userpassword,ownerpassword)
```

13.1.3 getsize, getversion, getnofobjects, getnofpages

A successfully opened document can provide some information:

```
bytes = getsize(<pdfe document>)
major, minor = getversion(<pdfe document>)
n = getnofobjects(<pdfe document>)
n = getnofpages(<pdfe document>)
bytes, waste = getnofpages(<pdfe document>)
```

13.1.4 get[catalog|trailer|info]

For accessing the document structure you start with the so called catalog, a dictionary:

```
<pdfe dictionary> = pdfe.getcatalog(<pdfe document>)
```

The other two root dictionaries are accessed with:

```
<pdfe dictionary> = pdfe.gettrailer(<pdfe document>)
<pdfe dictionary> = pdfe.getinfo(<pdfe document>)
```

13.1.5 getpage, getbox

A specific page can conveniently be reached with the next command, which returns a dictionary.

```
<pdfe dictionary> = pdfe.getpage(<pdfe document>,pagenumber)
```

Another convenience command gives you the (bounding) box of a (normally page) which can be inherited from the document itself. An example of a valid box name is `MediaBox`.

```
pages = pdfe.getbox(<pdfe dictionary>,boxname)
```

13.1.6 get[string|integer|number|boolean|name]

Common values in dictionaries and arrays are strings, integers, floats, booleans and names (which are also strings) and these are also normal LUA objects:

```
s = getstring (<pdfe array|dictionary>,index|key)
i = getinteger(<pdfe array|dictionary>,index|key)
n = getnumber (<pdfe array|dictionary>,index|key)
b = getboolean(<pdfe array|dictionary>,index|key)
n = getname   (<pdfe array|dictionary>,index|key)
```

The `getstring` function has two extra variants:

```
s, h = getstring (<pdfe array|dictionary>,index|key,false)
s    = getstring (<pdfe array|dictionary>,index|key,true)
```

The first call returns the original string plus a boolean indicating if the string is hex encoded. The second call returns the unencoded string.

13.1.7 get[dictionary|array|stream]

Normally you will use an index in an array and key in a dictionary but dictionaries also accept an index. The size of an array or dictionary is available with the usual # operator.

```
<pdfe dictionary>   = getdictionary(<pdfe array|dictionary>,index|key)
<pdfe array>        = getarray     (<pdfe array|dictionary>,index|key)
<pdfe stream>,
<pdfe dictionary>   = getstream    (<pdfe array|dictionary>,index|key)
```

These commands return dictionaries, arrays and streams, which are dictionaries with a blob of data attached.

Before we come to an alternative access mode, we mention that the objects provide access in a different way too, for instance this is valid:

```
print(pdfe.open("foo.pdf").Catalog.Type)
```

At the topmost level there are `Catalog`, `Info`, `Trailer` and `Pages`, so this is also okay:

```
print(pdfe.open("foo.pdf").Pages[1])
```

13.1.8 [open|close|readfrom|whole|]stream

Streams are sort of special. When your index or key hits a stream you get back a stream object and dictionary object. The dictionary you can access in the usual way and for the stream there are the following methods:

```
okay   = openstream(<pdfe stream>,[decode])
         closestream(<pdfe stream>)
str, n = readfromstream(<pdfe stream>)
str, n = readwholestream(<pdfe stream>,[decode])
```

You either read in chunks, or you ask for the whole. When reading in chunks, you need to open and close the stream yourself. The `n` value indicates the length read. The `decode` parameter controls if the stream data gets uncompressed.

As with dictionaries, you can access fields in a stream dictionary in the usual LUA way too. You get the content when you 'call' the stream. You can pass a boolean that indicates if the stream has to be decompressed.

13.1.9 getfrom[dictionary|array]

In addition to the interface described before, there is also a bit lower level interface available.

```
key, type, value, detail = getfromdictionary(<pdfe dictionary>,index)
type, value, detail = getfromarray(<pdfe array>,index)
```

TYPE	MEANING	VALUE	DETAIL
0	none	nil	
1	null	nil	
2	boolean	boolean	
3	integer	integer	
4	number	float	
5	name	string	
6	string	string	hex
7	array	arrayobject	size
8	dictionary	dictionaryobject	size
9	stream	streamobject	dictionary size
10	reference	integer	

A hex string is (in the PDF file) surrounded by <> while plain strings are bounded by <>.

13.1.10 [dictionary|array]totable

All entries in a dictionary or table can be fetched with the following commands where the return values are a hashed or indexed table.

```
hash = dictionarytotable(<pdfe dictionary>)
list = arraytotable(<pdfe array>)
```

You can get a list of pages with:

```
{ { <pdfe dictionary>, size, objnum }, ... } = pagestotable(<pdfe document>)
```

13.1.11 getfromreference

Because you can have unresolved references, a reference object can be resolved with:

```
type, <pdfe dictionary|array|stream>, detail = getfromreference(<pdfe refer-
ence>)
```

So, as second value you get back a new pdfe userdata object that you can query.

13.2 Memory streams

The `pdfe.new` function takes three arguments:

VALUE	EXPLANATION
`stream`	this is a (in low level LUA speak) light userdata object, i.e. a pointer to a sequence of bytes
`length`	this is the length of the stream in bytes (the stream can have embedded zeros)
`name`	optional, this is a unique identifier that is used for hashing the stream

The third argument is optional. When it is not given the function will return a `pdfe` document object as with a regular file, otherwise it will return a filename that can be used elsewhere (e.g. in the image library) to reference the stream as pseudo file.

Instead of a light userdata stream (which is actually fragile but handy when you come from a library) you can also pass a LUA string, in which case the given length is (at most) the string length.

The function returns a `pdfe` object and a string. The string can be used in the `img` library instead of a filename. You need to prevent garbage collection of the object when you use it as image (for instance by storing it somewhere).

Both the memory stream and it's use in the image library is experimental and can change. In case you wonder where this can be used: when you use the swiglib library for `graphicmagick`, it can return such a userdata object. This permits conversion in memory and passing the result directly to the backend. This might save some runtime in one-pass workflows. This feature is currently not meant for production and we might come up with a better implementation.

13.3 The `pdfscanner` library

This library is not available in LUAMETATEX.

14 Extra libraries

14.1 Introduction

The libraries can be grouped in categories like fonts, languages, TEX, METAPOST, PDF, etc. There are however also some that are more general purpose and these are discussed here.

14.2 File and string readers: `fio` and type sio

This library provides a set of functions for reading numbers from a file and in addition to the regular `io` library functions. The following work on normal LUA file handles.

NAME	ARGUMENTS	RESULTS
readcardinal1	(f)	a 1 byte unsigned integer
readcardinal2	(f)	a 2 byte unsigned integer
readcardinal3	(f)	a 3 byte unsigned integer
readcardinal4	(f)	a 4 byte unsigned integer
readcardinaltable	(f,n,b)	n cardinals of b bytes
readinteger1	(f)	a 1 byte signed integer
readinteger2	(f)	a 2 byte signed integer
readinteger3	(f)	a 3 byte signed integer
readinteger4	(f)	a 4 byte signed integer
readintegertable	(f,n,b)	n integers of b bytes
readfixed2	(f)	a float made from a 2 byte fixed format
readfixed4	(f)	a float made from a 4 byte fixed format
read2dot14	(f)	a float made from a 2 byte in 2dot4 format
setposition	(f,p)	goto position p
getposition	(f)	get the current position
skipposition	(f,n)	skip n positions
readbytes	(f,n)	n bytes
readbytetable	(f,n)	n bytes

When relevant there are also variants that end with `le` that do it the little endian way. The fixed and dot floating points formats are found in font files and return LUA doubles.

A similar set of function as in the `fio` library is available in the `sio` library: `sio.readcardinal1`, `sio.readcardinal2`, `sio.readcardinal3`, `sio.readcardinal4`, `sio.readcardinaltable`, `sio.readinteger1`, `sio.readinteger2`, `sio.readinteger3`, `sio.readinteger4`, `sio.readintegertable`, `sio.readfixed2`, `sio.readfixed4`, `sio.read2dot14`, `sio.setposition`, `sio.getposition`, `sio.skipposition`, `sio.readbytes` and `sio.readbytetable`. Here the first argument is a string instead of a file handle.

14.3 `md5`

NAME	ARGUMENTS	RESULTS
sum		

hex
HEX

14.4 sha2

NAME	ARGUMENTS	RESULTS
digest256		
digest384		
digest512		

14.5 xzip

NAME	ARGUMENTS	RESULTS
compress		
decompress		
adler32		
crc32		

14.6 xmath

This library just opens up standard C math library and the main reason for it being there is that it permits advanced graphics in METAPOST (via the LUA interface). There are three constant values:

NAME	ARGUMENTS	RESULTS
inf	–	inf
nan	–	nan
pi	–	3.1415926535898

and a lot of functions:

NAME	ARGUMENTS	RESULTS
acos	(a)	
acosh	(a)	
asin	(a)	
asinh	(a)	
atan	(a[,b])	
atan2	(a[,b])	
atanh	(a)	
cbrt	(a)	
ceil	(a)	
copysign	(a,b)	
cos	(a)	
cosh	(a)	

```
deg                 (a)
erf                 (a)
erfc                (a)
exp                 (a)
exp2                (a)
expm1               (a)
fabs                (a)
fdim                (a,b)
floor               (a)
fma                 (a,b,c)
fmax                (...)
fmin                (...)
fmod                (a,b)
frexp               (a,b)
gamma               (a)
hypot               (a,b)
isfinite            (a)
isinf               (a)
isnan               (a)
isnormal            (a)
j0                  (a)
j1                  (a)
jn                  (a,b)
ldexp               (a,b)
lgamma              (a)
l0                  (a)
l1                  (a)
ln                  (a,b)
log                 (a[,b])
log10               (a)
log1p               (a)
log2                (a)
logb                (a)
modf                (a,b)
nearbyint           (a)
nextafter           (a,b)
pow                 (a,b)
rad                 (a)
remainder           (a,b)
remquo              (a,b)
round               (a)
scalbn              (a,b)
sin                 (a)
sinh                (a)
sqrt                (a)
tan                 (a)
```


tanh	(a)
tgamma	(a)
trunc	(a)
y0	(a)
y1	(a)
yn	(a)

14.7 xcomplex

LUAMETATEX also provides a complex library `xcomplex`. The complex number is a userdatum:

NAME	ARGUMENTS	RESULTS
new	(r,i)	a complex userdata type
tostring	(z)	a string representation
topair	(z)	two numbers

There is a bunch of functions that take a complex number:

NAME	ARGUMENTS
abs	(a)
arg	(a)
imag	(a)
real	(a)
onj	(a)
proj	(a)
exp"	(a)
log	(a)
sqrt	(a)
pow	(a,b)
sin	(a)
cos	(a)
tan	(a)
asin	(a)
acos	(a)
atan	(a)
sinh	(a)
cosh	(a)
tanh	(a)
asinh	(a)
acosh	(a)
atanh	(a)

These are accompanied by `libcerf` functions:

NAME	ARGUMENTS	RESULTS
erf	(a)	The complex error function erf(z)

erfc	(a)	The complex complementary error function erfc(z) = 1 - erf
erfcx	(a)	The underflow-compensating function erfcx(z) = exp(z^2) er
erfi	(a)	The imaginary error function erfi(z) = -i erf(iz)
dawson	(a)	Dawson's integral D(z) = sqrt(pi)/2 * exp(-z^2) * erfi(z)
voigt	(a,b,c)	The convolution of a Gaussian and a Lorentzian
voigt_hwhm	(a,b)	The half width at half maximum of the Voigt profile

14.8 xdecimal

As an experiment LuaMetaTeX provides an interface to the decNumber library that we have on board for MetaPost anyway. Apart from the usual support for operators there are some functions.

NAME	ARGUMENTS	RESULTS
abs	(a)	
new	([n or s])	
copy	(a)	
trim	(a)	
tostring	(a)	
tonumber	(a)	
setprecision	(n)	
getprecision	()	
conj	(a)	
abs	(a)	
pow	(a,b)	
sqrt	(a)	
ln	(a)	
log	(a)	
exp	(a)	
bor	(a,b)	
bxor	(a,b)	
band	(a,b)	
shift	(a,b)	
rotate	(a,b)	
minus	(a)	
plus	(a)	
min	(a,b)	
max	(a,b)	

14.9 lfs

The original lfs module has been adapted a bit to our needs but for practical reasons we kept the namespace. This module will probably evolve a bit over time.

NAME	ARGUMENTS	RESULTS
attributes	(name)	

chdir	(name)	
currentdir	()	
dir	(name)	name, mode, size and mtime
mkdir	(name)	
rmdir	(name)	
touch	(name)	
link	(name)	
symlinkattributes	(name)	
isdir	(name)	
isfile	(name)	
iswriteabledir	(name)	
iswriteablefile	(name)	
isreadabledir	(name)	
isreadablefile	(name)	

The dir function is a traverser which in addition to the name returns some more properties. Keep in mind that the traverser loops over a directory and that it doesn't run well when used nested. This is a side effect of the operating system. It is also the reason why we return some properties because querying them via attributes would interfere badly.

The following attributes are returned by attributes:

NAME	VALUE
mode	
size	
modification	
access	
change	
permissions	
nlink	

14.10 pngdecode

This module is experimental and used in image inclusion. It is not some general purpose module and is supposed to be used in a very controlled way. The interfaces might evolve.

NAME	ARGUMENTS	RESULTS
applyfilter	(str,nx,ny,slice)	string
splitmask	(str,nx,ny,bpp,bytes)	string
interlace	(str,nx,ny,slice,pass)	string
expand	(str,nx,ny,parts,xline,factor)	string

14.11 basexx

Some more experimental helpers:

NAME	ARGUMENTS	RESULTS
encode16	(str[,newline])	string
decode16	(str)	string
encode64	(str[,newline])	string
decode64	(str)	string
encode85	(str[,newline])	string
decode85	(str)	string
encodeRL	(str)	string
decodeRL	(str)	string
encodeLZW	(str[,defaults])	string
decodeLZW	(str[,defaults])	string

14.12 Multibyte `string` functions

The `string` library has a few extra functions, for example `string.explode`. This function takes upto two arguments: `string.explode(s[,m])` and returns an array containing the string argument `s` split into sub-strings based on the value of the string argument `m`. The second argument is a string that is either empty (this splits the string into characters), a single character (this splits on each occurrence of that character, possibly introducing empty strings), or a single character followed by the plus sign `+` (this special version does not create empty sub-strings). The default value for `m` is ‘ +’ (multiple spaces). Note: `m` is not hidden by surrounding braces as it would be if this function was written in TEX macros.

The `string` library also has six extra iterators that return strings piecemeal: `string.utfvalues`, `string.utfcharacters`, `string.characters`, `string.characterpairs`, `string.bytes` and `string.bytepairs`.

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

The `string.characterpairs()` and `string.bytepairs()` iterators are useful especially in the conversion of UTF16 encoded data into UTF8.

There is also a two-argument form of `string.dump()`. The second argument is a boolean which, if true, strips the symbols from the dumped data. This matches an extension made in `luajit`. This is typically a function that gets adapted as LUA itself progresses.

The `string` library functions `len`, `lower`, `sub` etc. are not UNICODE-aware. For strings in the UTF8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the `slnunicode` library can be used, e.g., `unicode.utf8.len`, `unicode.utf8.lower` etc. The exceptions are `unicode.utf8.find`, that always returns byte positions in a string, and `unicode.utf8.match` and `unicode.utf8.gmatch`. While the latter two functions in general *are*

UNICODE-aware, they fall-back to non-UNICODE-aware behavior when using the empty capture `()` but other captures work as expected. For the interpretation of character classes in `unicode.utf8` functions refer to the library sources at http://luaforge.net/projects/sln.

Version 5.3 of LUA provides some native UTF8 support but we have added a few similar helpers too: `string.utfvalue`, `string.utfcharacter` and `string.utflength`.

`string.utfvalue(s)`: returns the codepoints of the characters in the given string
`string.utfcharacter(c,...)`: returns a string with the characters of the given code points
`string.utflength(s)`: returns the length of the given string

These three functions are relative fast and don't do much checking. They can be used as building blocks for other helpers.

14.13 Extra os library functions

The `os` library has a few extra functions and variables: `os.selfdir`, `os.selfarg`, `os.setenv`, `os.env`, `os.gettimeofday`, `os.type`, `os.name` and `os.uname`, that we will discuss here. There are also some time related helpers in the `lua` namespace.

`os.selfdir` is a variable that holds the directory path of the actual executable. For example: `\directlua{tex.sprint(os.selfdir)}`.
`os.selfarg` is a table with the command line arguments.
`os.setenv(key,value)` sets a variable in the environment. Passing `nil` instead of a value string will remove the variable.
`os.env` is a hash table containing a dump of the variables and values in the process environment at the start of the run. It is writeable, but the actual environment is *not* updated automatically.
`os.gettimeofday` returns the current 'UNIX time', but as a float. Keep in mind that there might be platforms where this function is not available.
`os.type` is a string that gives a global indication of the class of operating system. The possible values are currently `windows`, `unix`, and `msdos` (you are unlikely to find this value 'in the wild').
`os.name` is a string that gives a more precise indication of the operating system. These possible values are not yet fixed, and for `os.type` values `windows` and `msdos`, the `os.name` values are simply `windows` and `msdos`
The list for the type `unix` is more precise: `linux`, `freebsd`, `kfreebsd`, `cygwin`, `openbsd`, `solaris`, `sunos` (pre-solaris), `hpux`, `irix`, `macosx`, `gnu` (hurd), `bsd` (unknown, but BSD-like), `sysv`, `generic` (unknown). But ... we only provide LUAMETATEX binaries for the mainstream variants.
Officially we only support mainstream systems: MS WINDOWS, LINUX, FREEBSD and OS-X. Of course one can build LUAMETATEX for other systems, in which case on has to check the above.
`os.uname` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string values, and their names are: `sysname`, `machine`, `release`, `version`, and `nodename`.

14.14 The lua library functions

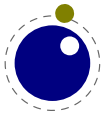
The `lua` library provides some general helpers.

The `newtable` and `newindex` functions can be used to create tables with space reserved beforehand for the given amount of entries.
The `getstacktop` function returns a number that can be used for diagnostic purposes.
The functions `getruntime`, `getcurrenttime`, `getpreciseticks` and `getpreciseseconds` return what their name suggests.
On MS WINDOWS the `getcodepage` function returns two numbers, one for the command handler and one for the graphical user interface.
The name of the startup file is reported by `getstartupfile`.
The LUA version is reported by `getversion`.
The `lua.openfile` function can be used instead of `io.open`. On MS WINDOWS it will convert the filename to a so called wide one which means that filenames in UTF8 encoding will work ok. On the other hand, names given in the codepage won't.

Topics

Primitives

This register contains the primitives that are mentioned in the manual. There are of course many more primitives. The LuaTeX primitives are typeset in bold. The primitives from pdfTeX are not supported that way but mentioned anyway.



Callbacks

Nodes

This register contains the nodes that are known to LuaTeX. The primary nodes are in bold, whatsits that are determined by their subtype are normal. The names prefixed by `pdf_` are backend specific.

236 Nodes

Libraries

This register contains the functions available in libraries. Not all functions are documented, for instance because they can be experimental or obsolete.

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` `\openout` `\write` `\closeout` `\openin` `\read` `\readline` `\closein` `\ifeof`
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. More extensive status information is provided in the overhauled status library.

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_format_file open_data_file
```

The callbacks related to errors are changed:

```
intercept_tex_error intercept_lua_error
show_error_message show_warning_message
```

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

```
trace_memory
```

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:

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

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

```
define_font
```

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

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.

```
automatichyphenmode
bodydir
bodydirection
boxdir
breakafterdirmode
closein
closeout
compoundhyphenmode
copyfont
draftmode
dviextension
dvifeedback
dvivariable
eTeXVersion
eTeXglueshrinkorder
eTeXgluestretchorder
eTeXminorversion
eTeXrevision
eTeXversion
expandglyphsinfont
fixupboxesmode
hoffset
hyphenationbounds
hyphenpenaltymode
ifeof
ifprimitive
ignoreligaturesinfont
immediateassigned
immediateassignment
lastsavedboxresourceindex
lastsavedimageresourceindex
lastsavedimageresourcepages
lastxpos
lastypos
latelua
lateluafunction
leftghost
letterspacefont
```


linedir
mag
mathdir
mathoption
nokerns
noligs
nolocaldirs
nolocalwhatsits
normaldeviate
openin
openout
outputmode
pagebottomoffset
pagedir
pagedirection
pageheight
pageleftoffset
pagerightoffset
pagetopoffset
pagewidth
pardir
pdfextension
pdffeedback
pdfvariable
primitive
randomseed
read
readline
rightghost
saveboxresource
saveimageresource
savepos
setrandomseed
shapemode
special
suppressfontnotfounderror
suppressifcsnameerror
suppresslongerror
suppressmathparerror
suppressoutererror
suppressprimitiveerror
synctex
tagcode
textdir
tracingscantokens
uniformdeviate
useboxresource
useimageresource
voffset
write

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

UUskewed
UUskewedwithdelims
Uabove
Uabovewithdelims
Uatop
Uatopwithdelims
Umathclass
Umathspacebeforescript
Umathspacingmode
Unosubprescript
Unosuperprescript
Uover
Uoverwithdelims
Ustyle
Usubprescript
Usuperprescript
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
afterassigned
aftergrouped
aliased
atendofgroup
atendofgrouped
automigrationmode
beginlocalcontrol
boxattribute
boxorientation
boxtotal
boxxmove
boxxoffset
boxymove
boxyoffset
defcsname

edefcsname
enforced
everytab
expand
expandafterpars
expandafterspaces
expandcstoken
expandtoken
fontspecifiedname
fontspecifiedsize
frozen
futuredef
futureexpand
futureexpandis
futureexpandisap
glyphdatafield
glyphoptions
glyphscriptfield
glyphstatefield
hyphenationmode
ifarguments
ifboolean
ifchkdim
ifchknum
ifcmpdim
ifcmpnum
ifcstok
ifdimval
ifempty
ifflags
ifhastok
ifhastoks
ifhasxtoks
ifmathparameter
ifmathstyle
ifnumval
ifparameter
iftok
ignorearguments
ignorepars
immutable
instance
integerdef
lastarguments
lastnodesubtype
letcsname
letfrozen
letprotected
linepar
localcontrol
localcontrolled
matholdmode
meaningfull
meaningless
mutable
noaligned
normalizelinemode
ordlimits
orelse
orunless
overloaded
overloadmode
overshoot
parattr
parfillleftskip
permanent
shownodedetails
snapshotpar
supmarkmode
swapcsvalues
thewithoutunit
todimension
tointeger
tokenized
tolerant
toscaled
tracingalignments
tracingmath
unhpack
unletfrozen
unletprotected
unvpack
wrapuppar

Statistics

The following fonts are used in this document:

used	filesize	version	filename
22	988.684	5.000	cambmath.ttf
4	927.280	5.020	cambria.ttf
11	163.452	1.802	LucidaBrightMathOT-Demi.otf
11	348.296	1.802	LucidaBrightMathOT.otf
4	73.284	1.801	LucidaBrightOT.otf
22	733.500	1.958	latinmodern-math.otf
1	64.684	2.004	lmmono10-regular.otf
1	64.160	2.004	lmmonoltcond10-regular.otf
4	111.536	2.004	lmroman10-regular.otf
18	525.008	1.106	texgyredejavu-math.otf
22	601.220	1.632	texgyrepagella-math.otf
4	218.100	2.501	texgyrepagella-regular.otf
1	693.876	2.340	DejaVuSans-Bold.ttf
1	741.536	2.340	DejaVuSans.ttf
5	318.392	2.340	DejaVuSansMono-Bold.ttf
1	245.948	2.340	DejaVuSansMono-Oblique.ttf
4	335.068	2.340	DejaVuSansMono.ttf
13	345.364	2.340	DejaVuSerif-Bold.ttf
1	336.884	2.340	DejaVuSerif-BoldItalic.ttf
2	343.388	2.340	DejaVuSerif-Italic.ttf
6	367.260	2.340	DejaVuSerif.ttf
158	**8.546.920**		**21 files loaded**