MKII

CONTEXT

CONTEXT

The history of luaTEX

2006–2009 / v 0.50

MKIV

Contents

Introduction

In this document I will keep track of the transition of ConTEXt from MkII to MkIV, the latter being the Lua aware version.

The development of LuaTEX started with a few email exchanges between me and Hartmut Henkel. I had played a bit with Lua in SciTE and somehow felt that it would fit into TEX quite well. Hartmut made me a version of pdfTEX which provided a `\lua` command. After exploring this road a bit Taco Hoekwater took over and we quickly reached a point where the pdfTEX development team could agree on following this road to the future.

The development was boosted by a substantial grant from Colorado State University in the context of the Oriental TEX Project of Idris Samawi Hamid. This project aims at bringing features into TEX that will permit ConTEXt to do high quality Arabic typesetting. Due to this grant Taco could spent substantial time on development, which in turn meant that I could start playing with more advanced features.

This document is not so much a users manual as a history of the development. Consider it a collection of articles, and some chapters indeed have ended up in the journals of user groups. Things may evolve and the way things are done may change, but it felt right to keep track of the process this way. Keep in mind that some features may have changed while LuaTEX matured.

Just for the record: development in the LuaTEX project is done by Taco Hoekwater, Hartmut Henkel and Hans Hagen. Eventually, the stable versions will become pdfTEX version 2 and other members of the pdfTEX team will be involved in development and maintenance. In order to prevent problems due to new and maybe even slightly incompatible features, pdfTEX version 1 will be kept around as well, but no fundamentally new features will be added to it. For practical reasons we use LuaTEX as the name of the development version but also for pdfTEX 2. That way we can use both engines side by side.

This document is also one of our test cases. Here we use traditional TEX fonts (for math), Type1 and OpenType fonts. We use color and include test code. Taco and I always test new versions of LuaTEX (the program) and MkIV (the macros and Lua code) with this document before a new version is released. It also means that there can be temporary flaws in the rendering. Keep tuned,

Hans Hagen, Hasselt NL,
August 2006–2016

`http://www.luatex.org`

I From MkII to MkIV

Sometime in 2005 the development of LuaTeX started, a further development of pdfTeX and a precursor to pdfTeX version 2. This TeX variant will provide:

- 21–32 bit internals plus a code cleanup
- flexible support for OpenType fonts
- an internal UTF data flow
- the bidirectional typesetting of Aleph
- Lua callbacks to the most relevant TeX internals
- some extensions to TeX (for instance math)
- an efficient way to communicate with MetaPost

In the tradition of TeX this successor will be downward compatible in most essential parts and in the end, there is still pdfTeX version 1 as fall back.

In the mean time we have seen another unicode variant show up, XeTeX which is under active development, uses external libraries, provides access to the fonts on the operating system, etc.

From the beginning, ConTeXt always worked with all engines. This was achieved by conditional code blocks: depending on what engine was used, different code was put in the format and/or used at runtime. Users normally were unaware of this. Examples of engines are ε-TeX, Aleph, and XeTeX. Because nowadays all engines provide the ε-TeX features, in August 2006 we decided to consider those features to be present and drop providing the standard TeX compatible variants. This is a small effort because all code that is sensitive for optimization already has ε-TeX code branches for many years.

However, with the arrival of LuaTeX, we need a more drastic approach. Quite some existing code can go away and will be replaced by different solutions. Where TeX code ends up in the format file, along with its state, Lua code will be initiated at run time, after a Lua instance is started. ConTeXt reserves its own instance of Lua.

Most of this will go unnoticed for the users because the user interface will not change. For developers however, we need to provide a mechanism to deal with these issues. This is why, for the first time in ConTeXt's history we will officially use a kind of version tag. When we changed the low level interface from Dutch to English we jokingly talked of version 2. So, it makes sense to follow this lead.

- **ConTeXt MkI** At that moment we still had a low level Dutch interface, invisible for users but not for developers.
- **ConTeXt MkII** We now have a low level English interface, which (as we indeed saw happen) triggers more development by users.
- **ConTeXt MkIV** This is the next generation of ConTeXt, with parts re–implemented. It's an at some points drastic system overhaul.

Keep in mind that the functionality does not change, although in some places, for instance fonts, MkIV may provide additional functionality. The reason why most users will not notice the difference (maybe apart from performance and convenience) is that at the user interface level nothing changes (most of it deals with typesetting, not with low level details).

The hole in the numbering permits us to provide a MkIII version as well. Once XeTeX is stable, we may use that slot for XeTeX specific implementations.

As per August 2006 the banner is adapted to this distinction:

```
...  ver: 2006.09.06 22:46 MK II  fmt: 2006.9.6  ...
...  ver: 2006.09.06 22:47 MK IV  fmt: 2006.9.6  ...
```

This numbering system is reflected at the file level in such a way that we can keep developing the way we do, i.e. no files all over the place, in subdirectories, etc.

Most of the system's core files are not affected, but some may be, like those dealing with fonts, input- and output encodings, file handling, etc. Those files may come with different suffixes:

- `somefile.tex`: the main file, implementing the interface and common code
- `somefile.mkii`: mostly existing code, suitable for good old TeX (ε-TeX, pdfTeX, Aleph).
- `somefile.mkiv`: code optimized for use with LuaTeX, which could follow completely different approaches
- `somefile.lua`: Lua code, loaded at format generation time and/or runtime

As said, some day `somefile.mkiii` code may show up. Which variant is loaded is determined automatically at format generation time as well as at run time.

II How Lua fits in

introduction

Here I will discuss a few of the experiments that drove the development of LuaTeX. It describes the state of affairs around the time that we were preparing for TUG 2006. This development was pretty demanding for Taco and me but also much fun. We were in a kind of permanent Skype chat session, with binaries flowing in one direction and TeX and Lua code the other way. By gradually replacing (even critical) components of ConTeXt we had a real test bed and torture tests helped us to explore and debug at the same time. Because Taco uses LINUX as platform and I mostly use MS Windows, we could investigate platform dependent issues conveniently. While reading this text, keep in mind that this is just the beginning of the game.

I will not provide sample code here. When possible, the MkIV code transparantly replaces MkII code and users will seldom notices that something happens in different way. Of course the potential is there and future extensions may be unique to MkIV.

compatibility

The first experiments, already conducted with the experimental versions involved runtime conversion of one type of input into another. An example of this is the (TI) calculator math input handler that converts a rather natural math sequence into TeX and feeds that back into TeX. This mechanism eventually will evolve into a configurable math input handler. Such applications are unique to MkIV code and will not be backported to MkII. The question is where downward compatibility will become a problem. We don't expect many problems, apart from occasional bugs that result from splitting the code base, mostly because new features will not affect older functionality. Because we have to reorganize the code base a bit, we also use this opportunity to start making a variant of ConTeXt which consists of building blocks: MetaTeX. This is less interesting for the average user, but may be of interest for those using ConTeXt in workflows where only part of the functionality is needed.

metapost

Of course, when I experiment with such new things, I cannot let MetaPost leave untouched. And so, in the early stage of LuaTeX development I decided to play with two MetaPost related features: conversion and runtime processing.

Conversion from MetaPost output to PDF is currently done in pure TeX code. Apart from convenience, this has the advantage that we can let TeX take care of font inclusions. The tricky part of this conversion is that MetaPost output has some weird aspects, like dvips specific linewidth snapping. Another nasty element in the conversion is that we need to transform paths when pens are used. Anyhow, the converter has reached a rather stable state by now.

One of the ideas with MetaPost version 1^{+} is that we will have an alternative output mode. In the perspective of LuaTeX it makes sense to have a Lua output mode. Whatever converter we use, it needs to deal with MetaFun specials. These are responsible for special features like transparency, graphic inclusion, shading, and more. Currently we misuse colors to signal such features, but the new pre/post path hooks permit more advanced implementations. Experimenting with such new features is easier in Lua than in TeX.

The MkIV converter is a multi–pass converter. First we clean up the MetaPost output, next we convert the PostScript code into Lua calls. We assume that this Lua code eventually can be output directly from MetaPost. We then evaluate this converted Lua blob, which results in TeX commands. Think of:

```
1.2 setlinejoin
```

turned into:

```
mp.setlinejoin(1.2)
```

becoming:

```
\PDFcode{1.2 j}
```

which is, when the pdfTeX driver is active, equivalent to:

```
\pdfliteral{1.2 j}
```

Of course, when paths are involved, more things happen behind the scenes, but in the end an `mp.path` enters the Lua machinery.

When the MkIV converter reached a stable state, tests demonstrated then the code was upto 20% slower that the pure TeX alternative on average graphics, and but faster when many complex path transformations (due to penshapes) need to be done. This slowdown was due to the cleanup (using expressions) and intermediate conversion. Because Taco develops LuaTeX as well as maintains and extends MetaPost, we conducted experiments that combine features of these programs. As a result of this, shortcuts found their way into the MetaPost output.

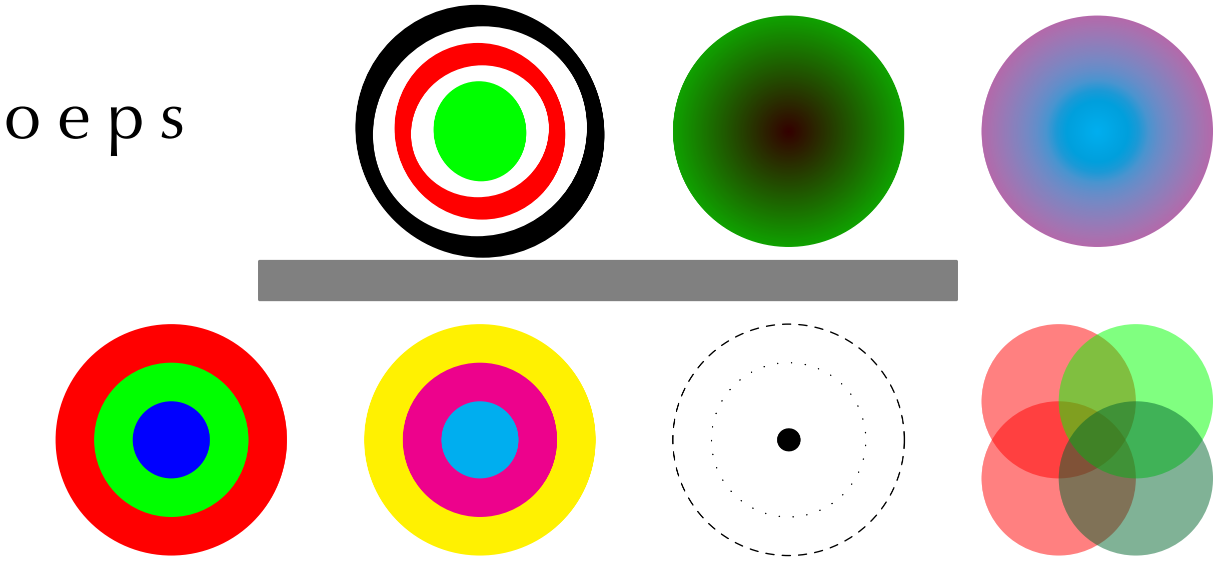


Figure II.1 converter test figure

Cleaning up the MetaPost output using Lua expressions takes relatively much time. However, starting with version 0.970 MetaPost uses a preamble, which permits not only short commands, but also gets rid of the weird linewidth and filldraw related PostScript constructs. The moderately complex graphic that we use for testing (figure II.1) takes over 16 seconds when converted 250 times. When we enable shortcuts we can avoid part of the cleanup and runtime goes down to under 7.5 seconds. This is significantly faster than the MkII code. We did experiments with simulated Lua output from MetaPost and then the MkIV converter really flies. The values on Taco's system are given between parenthesis.

prologues/mpprocset	**1/0**	**1/1**	**2/02/1**
MkII	8.5 (5.7)	8.0 (5.5)	8.8 8.5
MkIV	16.1 (10.6)	7.2 (4.5)	16.3 7.4

The main reason for the huge difference in the MkIV times is that we do a rigourous cleanup of the older MetaPost output in order avoid messy the messy (but fast) code that we use in the MkII converter. Think of:

```
0 0.5 dtransform truncate idtransform setlinewidth pop
closepath gsave fill grestore stroke
```

In the MkII converter, we push every number or keyword on a stack and use keywords as trigger points. In the MkIV code we convert the stack based PostScript calls to Lua function calls. Lines as shown are converted to single calls first. When `prologues` is set to 2, such line no longer show up and are replaced by simple calls accompanied by definitions in the preamble. Not only that, instead of verbose keywords, one or two character shortcuts are used. This means that the MkII code can be faster when procsets are used because shorter strings end up in the stack and comparison happens faster. On the other hand, when no procsets are used, the runtime is longer because of the larger preamble.

Because the converter is used outside ConTEXt as well, we support all combinations in order not to get error messages, but the converter is supposed to work with the following settings:

```
prologues := 1 ;
mpprocset := 1 ;
```

We don't need to set `prologues` to 2 (font encodings in file) or 3 (also font resources in file). So, in the end, the comparison in speed comes down to 8.0 seconds for MkII code and 7.2 seconds for the MkIV code when using the latest greatest MetaPost. When we simulate Lua output from MetaPost, we end up with 4.2 seconds runtime and when MetaPost could produce the converter's TEX commands, we need only 0.3 seconds for embedding the 250 instances. This includes TEX taking care of handling the specials, some of which demand building moderately complex PDF data structures.

But, conversion is not the only factor in convenient MetaPost usage. First of all, runtime MetaPost processing takes time. The actual time spent on handling embedded MetaPost graphics is also dependent on the speed of starting up MetaPost, which in turn depends on the size of the TEX trees used: the bigger these are, the more time KPSE spends on loading the `ls-R` databases. Eventually this bottleneck may go away when we have MetaPost as a library. (In ConTEXt one can also run MetaPost between runs. Which method is faster, depends on the amount and complexity of the graphics.)

Another factor in dealing with MetaPost, is the usage of text in a graphic (`btex`, `textext`, etc.). Taco Hoekwater, Fabrice Popineau and I did some experiments with a persistent MetaPost session in the background in order to simulate a library. The results look very promising: the overhead of embedded MetaPost graphics goes to nearly zero, especially when we also let the parent TEX job handle the typesetting of texts. A side effect of these experiments was a new mechanism in ConTEXt (and MetaFun) where TEX did all typesetting of labels, and MetaPost only worked with an abstract representation of the result. This way we can completely avoid nested TEX runs (the ones triggered by MetaPost). This also works ok in MkII mode.

Using a persistent MetaPost run and piping data into it is not the final solution if only because the terminal log becomes messed up too much, and also because intercepting errors is real messy. In the end we need a proper library approach, but the experiments demonstrated that we needed to go this way: handling hundreds of complex graphics that hold typeset paragraphs (being slanted and rotated and more by MetaPost), tooks mere seconds compared to minutes when using independent MetaPost runs for each job.

characters

Because LuaTEX is UTF based, we need a different way to deal with input encoding. For this purpose there are callbacks that intercept the input and convert it as needed. For context this means that the regime related modules get a Lua based counterparts. As a prelude to advanced character manipulations, we already load extensive unicode and conversion tables, with the benefit of being able to handle case handling with Lua.

The character tables are derived from unicode tables and MkII ConTEXt data files and generated using mtxtools. The main character table is pretty large, and this made us experiment a bit with efficiency. It was in this stage that we realized that it made sense to use precompiled Lua code (using `luac`). During format generation we let ConTEXt keep track of used Lua files and compiled them on the fly. For a production run, the compiled files were loaded instead.

Because at that stage LuaTEX was already a merge between pdfTEX and Aleph, we had to deal with pretty large format files. About that moment the ConTEXt format with the english user interface amounted to:

date	luatex	pdftex	xetex	aleph
2006-09-18	9 552 042	7 068 643	8 374 996	7 942 044

One reason for the large size of the format file is that the memory footprint of a 32 bit TEX is larger than that of good old TEX, even with some of the clever memory allocation techniques as used in LuaTEX. After some experiments where size and speed were measured Taco decided to compress the format using a level 3 zip compression. This brilliant move lead to the following size:

date	luatex	pdftex	xetex	aleph
2006-10-23	3 135 568	7 095 775	8 405 764	7 973 940

The first zipped versions were smaller (around 2.3 meg), but in the meantime we moved the Lua code into the format and the character related tables take some space.

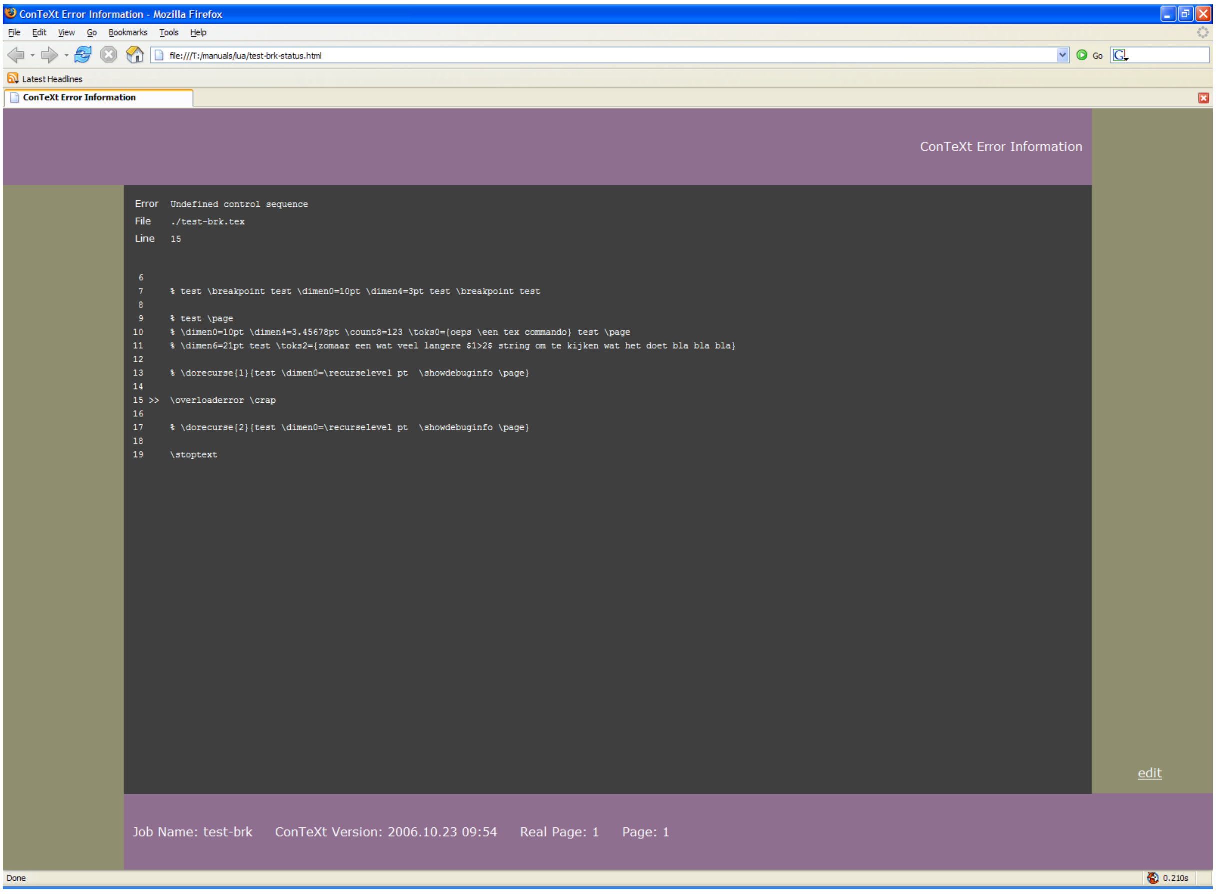
How stable are the mentioned numbers? Ten months after writing the previous text we get the following numbers:

date	luatex	pdftex	xetex	aleph
2007-08-16	5 603 676	7 505 925	8 838 538	8 369 206

They are all some 400K larger, which is probably the result of changes in hyphenation patterns (we now load them all, some several times depending on the font encodings used). Also, some extra math support has been brought in the kernel and we predefine a few more things. However, LuaTEX's format has become much larger! Partly this is the result of more Lua code, especially OpenType font handling and attributes related code. The extra TEX code is probably compensated by the removal of obsolete (at least for MkIV) code. However, the significantly larger number is mostly there because a different compression algorithm is used: speed is now favoured over efficiency.

debugging

In the process of experimenting with callbacks I played a bit with handling TEX error information. An option is to generate an HTML page instead of spitting out the usual blob of into on the terminal. In figure II.2 and figure II.3 you can see an example of this.


ConTeXt Error Information - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///T:/manuals/lua/test-brk-status.html
Go
Latest Headlines
ConTeXt Error Information
ConTeXt Error Information
edit
Done
0.210s

```
Error  Undefined control sequence
File   ./test-brk.tex
Line   15


 6
 7     % test \breakpoint test \dimen0=10pt \dimen4=3pt test \breakpoint test
 8
 9     % test \page
10     % \dimen0=10pt \dimen4=3.45678pt \count8=123 \toks0={oeps \een tex commando} test \page
11     % \dimen6=21pt test \toks2={zomaar een wat veel langere $1>2$ string om te kijken wat het doet bla bla bla}
12
13     % \dorecurse{1}{test \dimen0=\recurselevel pt  \showdebuginfo \page}
14
15 >>  \overloaderror \crap
16
17     % \dorecurse{2}{test \dimen0=\recurselevel pt  \showdebuginfo \page}
18
19     \stoptext
```

Job Name: test-brk   ConTeXt Version: 2006.10.23 09:54   Real Page: 1   Page: 1

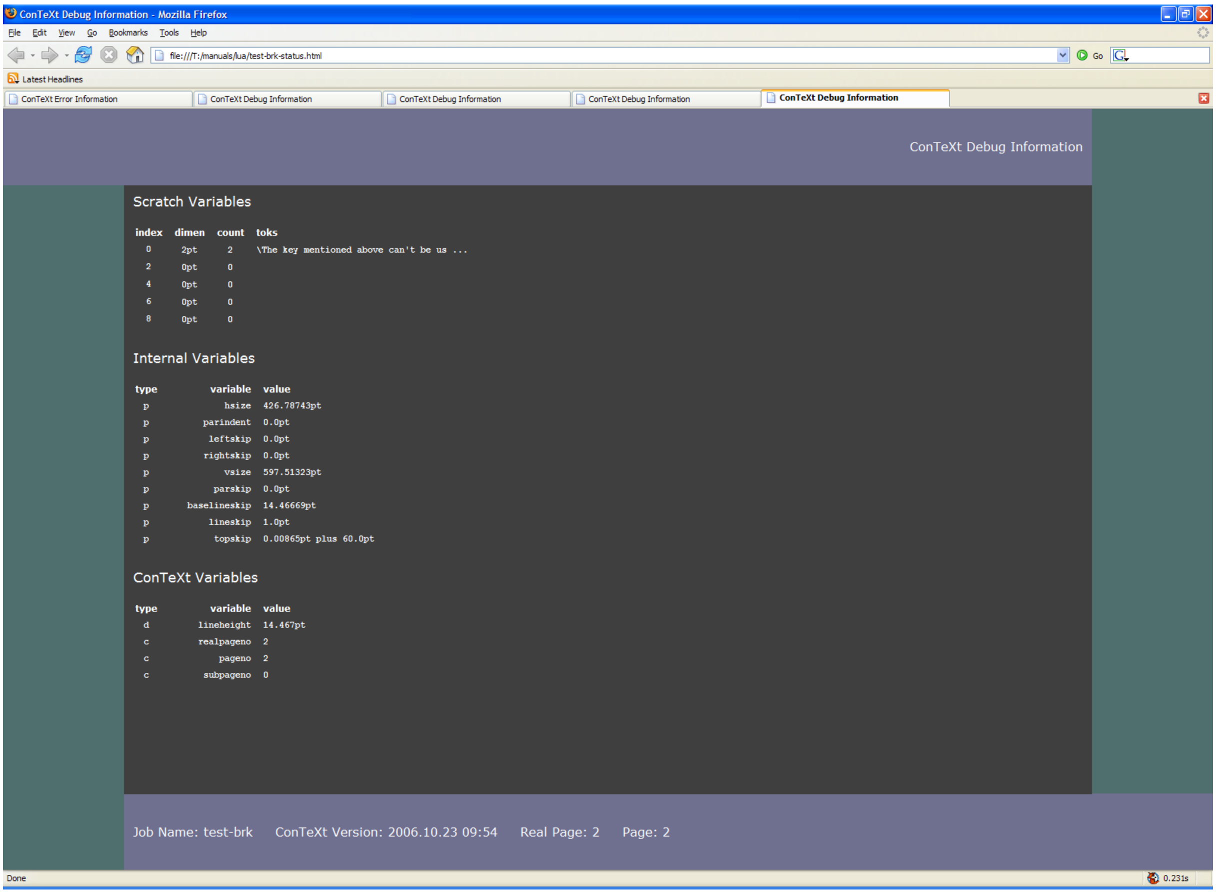
Figure II.2 An example error screen.

Playing with such features gives us an impression of what kind of access we need to TEX's internals. It also formed a starting point for conversion routines and a mechanism for embedding LUA code in HTML pages generated by CONTEXT.

file io

Replacing TEX's in- and output handling is non–trival. Not only is the code quite interwoven in the WEB2C source, but there is also the KPSE library to deal with. This means that quite some callbacks are needed to handle the different types of files. Also, there is output to the log and terminal to take care of.

Getting this done took us quite some time and testing and debugging was good for some headaches. The mechanisms changed a few times, and TEX and LUA code was thrown


ConTeXt Debug Information - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///T:/manuals/lua/test-brk-status.html
Go
Latest Headlines
ConTeXt Error Information
ConTeXt Debug Information
ConTeXt Debug Information
ConTeXt Debug Information
ConTeXt Debug Information
Done
0.231s

ConTeXt Debug Information

Scratch Variables

index	dimen	count	toks
0	2pt	2	\The key mentioned above can't be us ...
2	0pt	0	
4	0pt	0	
6	0pt	0	
8	0pt	0	

Internal Variables

type	variable	value
p	hsize	426.78743pt
p	parindent	0.0pt
p	leftskip	0.0pt
p	rightskip	0.0pt
p	vsize	597.51323pt
p	parskip	0.0pt
p	baselineskip	14.46669pt
p	lineskip	1.0pt
p	topskip	0.00865pt plus 60.0pt

ConTeXt Variables

type	variable	value
d	lineheight	14.467pt
c	realpageno	2
c	pageno	2
c	subpageno	0

Job Name: test-brk ConTeXt Version: 2006.10.23 09:54 Real Page: 2 Page: 2

Figure II.3 An example debug screen.

away as soon as better solutions came around. Because we were testing on real documents, using a fully loaded ConTEXt we could converge to a stable version after a while.

Getting this IO stuff done is tightly related to generating the format and starting up LuaTEX. If you want to overload the file searching and IO handling, you need overload as soon as possible. Because LuaTEX is also supposed to work with the existing KPSE library, we still have that as fallback, but in principle one could think of a KPSE free version, in which case the default file searching is limited to the local path and memory initialization also reverts to the hard coded defaults. A complication is that the soure code has KPSE calls and references to KPSE variables all over the place, so occasionally we run into interesting bugs.

Anyhow, while Taco hacked his way around the code, I converted my existing Ruby based KPSE variant into Lua and started working from that point. The advantage of having our own IO handler is that we can go beyond KPSE. For instance, since LuaTEX has, among a few others, the ZIP libraries linked in, we can read from ZIP files, and keep all TEX related files in TDS compliant ZIP files as well. This means that one can say:

```
\input zip:///somezipfile.zip?name=/somepath/somefile.tex
```

and use similar references to access files. Of course we had to make sure that KPSE like searching in the TDS (standardized TEX trees) works smoothly. There are plans to link the curl library into LUATEX, so that we can go beyong this and access repositories.

Of course, in order to be more or less KPSE and WEB2C compliant, we also need to support this paranoid file handling, so we provide mechanisms for that as well. In addition, we provide ways to create sandboxes for system calls.

Getting to intercept all log output (well, most log output) was a problem in itself. For this I used a (preliminary) XML based log format, which will make log parsing easier. Because we have full control over file searching, opening and closing, we can also provide more information about what files are loaded. For instance we can now easily trace what TFM files TEX reads.

Implementing additional methods for locating and opening files is not that complex because the library that ships with CONTEXT is already prepared for this. For instance, implementing support for:

```
\input http://www.someplace.org/somepath/somefile.tex
```

involved a few lines of code, most of which deals with caching the files. Because we overload the whole IO handling, this means that the following works ok:

```
\placefigure
  [] []
  {http handling}
  {\externalfigure
     [http://www.pragma-ade.com/show-gra.pdf]
     [page=1,width=\textwidth]}
```

Other protocols, like FTP are also supported, so one can say:

```
\typefile {ftp://anonymous:@ctan.org/tex-archive/systems\
    /knuth/lib/plain.tex}
```

On the agenda is playing with database, but by the time that we enter that stage linking the `curl` libraries into LUATEX should have taken place.

verbatim

The advance of LUATEX also permitted us to play with a long standing wish of catcode tables, a mechanism to quickly switch between different ways of treating input characters. An example of a place where such changes take place is verbatim (and in CONTEXT also when dealing with XML input).

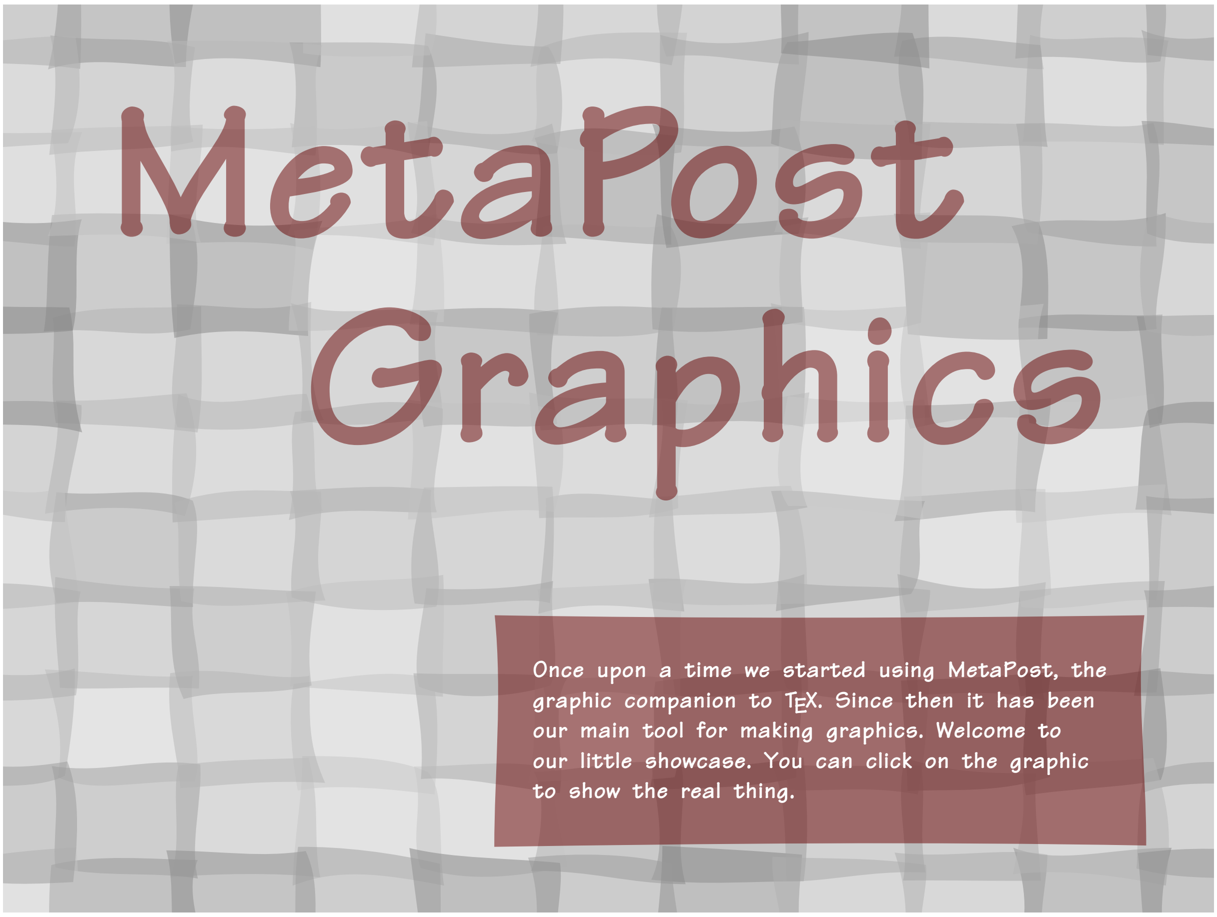

MetaPost
Graphics
Once upon a time we started using MetaPost, the graphic companion to TEX. Since then it has been our main tool for making graphics. Welcome to our little showcase. You can click on the graphic to show the real thing.

Figure II.4 http handling

We already had encountered the phenomena that when piping back results from LUA to TEX, we needed to take care of catcodes so that TEX would see the input as we wished. Earlier experiments with applying `\scantokens` to a result and thereby interpreting the result conforming the current catcode regime was not sufficient or at least not handy enough, especially in the perspective of fully expandable LUA results. To be honest, the `\scantokens` command was rather useless for this purposes due to its pseudo file nature and its end–of–file handling but in LUATEX we now have a convenient `\scantextokens` which has no side effects.

Once catcode tables were in place, and the relevant CONTEXT code adapted, I could start playing with one of the trickier parts of TEX programming: typesetting TEX using TEX, or verbatim. Because in CONTEXT verbatim is also related to buffering and pretty printing, all these mechanism were handled at once. It proved to be a pretty good testcase for writing LUA results back to TEX, because anything you can imagine can and will interfere (line endings, catcode changes, looking ahead for arguments, etc). This is one of the areas where MKIV code will make things look more clean and understandable, especially because we could move all kind of postprocessing (needed for pretty printing, i.e. syntax highlighting) to LUA. Interesting is that the resulting code is not beforehand faster.

Pretty printing 1000 small (one line) buffers and 5000 simple `\type` commands perform as follows:

	TEX normal	TEX pretty	LUA normal	LUA pretty
buffer	2.5 (2.35)	4.5 (3.05)	2.2 (1.8)	2.5 (2.0)
inline	7.7 (4.90)	11.5 (7.25)	9.1 (6.3)	10.9 (7.5)

Between braces the runtime on Taco's more modern machine is shown. It's not that easy to draw conclusions from this because TEX uses files for buffers and with LUA we store buffers in memory. For inline verbatim, LUA call's bring some overhead, but with more complex content, this becomes less noticable. Also, the LUA code is probably less optimized than the TEX code, and we don't know yet what benefits a Just In Time LUA compiler will bring.

xml

Interesting is that the first experiments with XML processing don't show the expected gain in speed. This is due to the fact that the CONTEXT XML parser is highly optimized. However, if we want to load a whole XML file, for instance the formal CONTEXT interface specification `cont-en.xml`, then we can bring down loading time (as well as TEX memory usage) down from multiple seconds to a blink of the eyes. Experiments with internal mappings and manipulations demonstrated that we may not so much need an alternative for the current parser, but can add additional, special purpose ones.

We may consider linking XSLTPROC into LUATEX, but this is yet undecided. After all, the problem of typesetting does not really change, so we may as well keep the process of manipulating and typesetting separated.

multipass data

Those who know CONTEXT a bit will know that it may need multiple passes to typeset a document. CONTEXT not only keeps track of index entries, list entries, cross references, but also optimizes some of the output based on information gathered in previous passes. Especially so called two–pass data and positional information puts some demands on memory and runtime. Two–pass data is collapsed in lists because otherwise we would run out of memory (at least this was true years ago when these mechanisms were introduced). Positional information is stored in hashes and has always put a bit of a burden on the size of a so called utility file (CONTEXT stores all information in one auxiliary file).

These two datatypes were the first we moved to a LUA auxiliary file and eventually all information will move there. The advantage is that we can use efficient hashes (without limitations) and only need to run over the file once. And LUA is incredibly fast in loading the tables where we keep track of these things. For instance, a test file storing and reading

10.000 complex positions takes 3.2 seconds runtime with LuaTeX but 8.7 seconds with traditional pdfTeX. Imagine what this will save when dealing with huge files (400 page 300 Meg files) that need three or more passes to be typeset. And, now we can without problems bump position tracking to milions of positions.

resources

Finding files is somewhat tricky and has a history in the TeX community and its distributions. For reasons of packaging and searching files are organized in a tree and there are rules for locating files of given types in this tree. When we say

```
\input blabla.tex
```

TeX will look for this file by consulting the path specification associated with the filetype. When we say

```
\input blabla
```

TeX will add the `.tex` suffix itself. Most other filetypes are not seen by users but are dealt with in a similar way internally.

As mentioned before, we support reading from other resources than the standard file system, for instance we can input files from websites or read from zip archives. Although this works quite well, we need to keep in mind that there are some conflicting interests: structured search based on type related specifications versus more or less explicit requests.

```
\input zip:///archive.zip?name=blabla.tex
\input zip:///archive.zip?name=/somepath/blabla.tex
```

Here we need to be rather precise in defining the file location. We can of course build rather complex mechanisms for locating files here, but at some point that may backfire and result in unwanted matches.

If you want to treat a zip archive as a TeX tree, then you need to register the file:

```
\usezipfile[archive.zip]
\usezipfile[tex.zip][texmf-local]
\usezipfile[tex.zip?tree=texmf-local]
```

The first variant registers all files in the archive, but the next two are equivalent and only register a subtree. The registered tree is prepended to the `TEXMF` specification and thereby may overload existing trees.

If an acrhive is not a real TeX tree, you can access files anywhere in the tree by using wildcards

```
\input */blabla.tex
\input */somepath/blabla.tex
```

These mechanisms evolve over time and it may take a while before they stabelize. For instance, the syntax for the ZIP inclusion has been adapted more than a year after this chapter was written (which is why this section is added).

III Initialization revised

Initializing LuaTeX in such a way that it does what you want it to do your way can be tricky. This has to do with the fact that if we want to overload certain features (using callbacks) we need to do that before the orginals start doing their work. For instance, if we want to install our own file handling, we must make sure that the built–in file searching does not get initialized. This is particularly important when the built in search engine is based on the kpse library. In that case the first serious file access will result in loading the `ls-R` filename databases, which will take an amount of time more or less linear with the size of the TeX trees. Among the reasons why we want to replace kpse are the facts that we want to access zip files, do more specific file searches, use http, ftp and whatever comes around, integrate ConTeXt specific methods, etc.

Although modern operating systems will cache files in memory, creating the internal data structures (hashes) from the rather dumb files take some time. On the machine where I was developing the first experimental LuaTeX code, we're talking about 0.3 seconds for pdfTeX. One would expect a Lua based alternative to be slower, but it is not. This may be due to the different implementation, but for sure the more efficient file cache plays a role as well. So, by completely disabling kpse, we can have more advanced io related features (like reading from zip files) at about the same speed (or even faster). In due time we will also support progname (and format) specific caches, which speeds up loading. In case one wonders why we bother about a mere few hundreds of milliseconds: imagine frequent runs from an editor or sub–runs during a job. In such situation every speed up matters.

So, back to initialization: how do we initialize LuaTeX. The method described here is developed for ConTeXt but is not limited to this macro package; when one tells TeXexec to generate formats using the `--luatex` directive, it will generate the ConTeXt formats as well as mptopdf using this engine.

For practical reasons, the Lua based io handler is kpse compliant. This means that the normal `texmf.cnf` and `ls-R` files can be used. However, their content is converted in a more Lua friendly way. Although this can be done at runtime, it makes more sense to to this in advance using luatools. The files involved are:

input	**raw input**	**runtime input**	**runtime fallback**
	`ls-R`	`files.luc`	`files.lua`
`texmf.lua`	`temxf.cnf`	`configuration.luc`	`configuration.lua`

In due time luatools will generate the directory listing itself (for this some extra libraries need to be linked in). The configuration file(s) eventually will move to a Lua table format, and when a `texmf.lua` file is present, that one will be used.

```
luatools --generate
```

This command will generate the relevant databases. Optionally you can provide `--minimize` which will generate a leaner database, which in turn will bring down loading time to (on my machine) about 0.1 sec instead of 0.2 seconds. The `--sort` option will give nicer intermediate (`.lua`) files that are more handy for debugging.

When done, you can use LUATOOLS roughly in the same manner as KPSEWHICH, for instance to locate files:

```
luatools texnansi-lmr10.tfm
luatools --all tufte.tex
```

You can also inspect its internal state, for instance with:

```
luatools --variables  --pattern=TEXMF
luatools --expansions --pattern=context
```

This will show you the (expanded) variables from the configuration files. Normally you don't need to go that deep into the belly.

The LUATOOLS script can also generate a format and run LuaTEX. For ConTEXt this is normally done with the TEXexec wrapper, for instance:

```
texexec --make --all --luatex
```

When dealing with this process we need to keep several things in mind:

- LuaTEX needs a Lua startup file in both ini and runtime mode
- these files may be the same but may also be different
- here we use the same files but a compiled one in runtime mode
- we cannot yet use a file location mechanism

A `.luc` file is a precompiled Lua chunk. In order to guard consistency between Lua code and tex code, ConTEXt will preload all Lua code and store them in the bytecode table provided by LuaTEX. How this is done, is another story. Contrary to these tables, the initialization code can not be put into the format, if only because at that stage we still need to set up memory and other parameters.

In our case, especially because we want to overload the IO handler, we want to store the startup file in the same path as the format file. This means that scripts that deal with format generation also need to take care of (relocating) the startup file. Normally we will use TEXexec but we can also use LUATOOLS.

Say that we want to make a plain format. We can call LUATOOLS as follows:

```
luatools --ini plain
```

This will give us (in the current path):

```
120,808 plain.fmt
  2,650 plain.log
 80,767 plain.lua
 64,807 plain.luc
```

From now on, only the `plain.fmt` and `plain.luc` file are important. Processing a file

```
test \end
```

can be done with:

```
luatools --fmt=./plain.fmt test
```

This returns:

```
This is luaTeX, Version 3.141592-0.1-alpha-20061018 (Web2C 7.5.5)
(./test.tex [1] )
Output written on test.dvi (1 page, 260 bytes).
Transcript written on test.log.
```

which looks rather familiar. Keep in mind that at this stage we still run good old Plain TEX. In due time we will provide a few files that will making work with LUA more convenient in Plain TEX, but at this moment you can already use for instance `\directlua`.

In case you wonder how this is related to CONTEXT, well only to the extend that it uses a couple of rather generic CONTEXT related LUA files.

CONTEXT users can best use TEXEXEC which will relocate the format related files to the regular engine path. In LUATOOLS terms we have two choices:

```
luatools --ini cont-en
luatools --ini --compile cont-en
```

The difference is that in the first case `context.lua` is used as startup file. This LUA file creates the `cont-en.luc` runtime file. In the second call LUATOOLS will create a `cont-en.lua` file and compile that one. An even more specific call would be:

```
luatools --ini --compile --luafile=blabla.lua          cont-en
luatools --ini --compile --lualibs=bla-1.lua,bla-2.lua cont-en
```

This call does not make much sense for CONTEXT. Keep in mind that LUATOOLS does not set up user specific configurations, for instance the `--all` switch in TEXEXEC will set up all patterns.

I know that it sounds a bit messy, but till we have a more clear picture of where LuaTEX is heading this is the way to proceed. The average ConTEXt user won't notice those details, because TEXexec will take care of things.

Currently we follow the TDS and WEB2C conventions, but in the future we may follow different or additional approaches. This may as well be driven by more complex IO models. For the moment extensions still fit in. For instance, in order to support access to remote resources and related caching, we have added to the configuration file the variable:

```
TEXMFCACHE = $TMP;$TEMP;$TMPDIR;$HOME;$TEXMFVAR;$VARTEXMF;.
```

IV An example: CalcMath

introduction

For a long time TeX's way of coding math has dominated the typesetting world. However, this kind of coding is not that well suited for non academics, like schoolkids. Often kids do know how to key in math because they use advanced calculators. So, when a couple of years ago we were implementing a workflow where kids could fill in their math workbooks (with exercises) on–line, it made sense to support so called Texas Instruments math input. Because we had to parse the form data anyway, we could use a `[[` and `]]` as math delimiters instead of `$`. The conversion too place right after the form was received by the web server.

`sin(x) + x^2 + x^(1+x) + 1/x^2`	$\sin(x) + x^2 + x^{1+x} + \frac{1}{x^2}$
`mean(x+mean(y))`	$\overline{x + \overline{y}}$
`int(a,b,c)`	$\int_b^a c$
`(1+x)/(1+x) + (1+x)/(1+(1+x)/(1+x))`	$\frac{1+x}{1+x} + \frac{1+x}{1+\frac{1+x}{1+x}}$
`10E-2`	10×10^{-2}
`(1+x)/x`	$\frac{1+x}{x}$
`(1+x)/12`	$\frac{1+x}{12}$
`(1+x)/-12`	$\frac{1+x}{-12}$
`1/-12`	$\frac{1}{-12}$
`12x/(1+x)`	$\frac{12x}{1+x}$
`exp(x+exp(x+1))`	$e^{x+e^{x+1}}$
`abs(x+abs(x+1)) + pi + inf`	$\|x + \|x+1\|\| + \pi + \mathrm{inf}$
`Dx Dy`	$\frac{\mathrm{d}x}{\mathrm{d}x}\frac{\mathrm{d}y}{\mathrm{d}x}$
`D(x+D(y))`	$\frac{\mathrm{d}}{\mathrm{d}x}(x + \frac{\mathrm{d}}{\mathrm{d}x}(y))$
`Df(x)`	$\mathrm{f}'(x)$
`g(x)`	$\mathrm{g}(x)$
`sqrt(sin^2(x)+cos^2(x))`	$\sqrt{\sin^2(x) + \cos^2(x)}$

By combining Lua with TeX, we can do the conversion from calculator math to TeX immediately, without auxiliary programs or complex parsing using TeX macros.

tex

In a ConTEXt source one can use the `\calcmath` command, as in:

```
The strange formula \calcmath {sqrt(sin^2(x)+cos^2(x))} boils
down to ...
```

One needs to load the module first, using:

```
\usemodule[calcmath]
```

Because the amount of code involved is rather small, eventually we may decide to add this support to the MkIV kernel.

xml

Coding math in TEX is rather efficient. In XML one needs way more code. Presentation MathML provides a few basic constructs and boils down to combining those building blocks. Content MathML is better, especially from the perspective of applications that need to do interpret the formulas. It permits for instance the ConTEXt content MathML handler to adapt the rendering to cultural driven needs. The OpenMath way of coding is like content MathML, but more verbose with less tags. Calculator math is more restrictive than TEX math and less verbose than any of the XML variants. It looks like:

```
<icm>sqrt(sin^2(x)+cos^2(x))</icm> test
```

And in display mode:

```
<dcm>sqrt(sin^2(x)+cos^2(x))</dcm> test
```

speed

This script (which you can find in the ConTEXt distribution as soon as the MkIV code variants are added) is the first real TEX related Lua code that I wrote; so far I had only written some wrapping and spell checking code for the SciTE editor. It also made a nice demo for a couple of talks that I held at usergroup meetings. The script has a lot of expressions. These convert one string into another. They are less powerful than regular expressions, but pretty fast and adequate. The feature I miss most is alternation like `(l|st)uck` but it's a small price to pay. As the Lua manual explains: adding a POSIX compliant regexp parser would take more lines of code than Lua currently does.

On my machine, running this first version took 3.5 seconds for 2500 times typesetting the previously shown square root of sine and cosine. Of this, 2.1 seconds were spent on typesetting and 1.4 seconds on converting. After optimizing the code, 0.8 seconds were used for conversion. A stand alone LUA takes .65 seconds, which includes loading the interpreter. On a test of 25.000 sample conversions, we could gain some 20% conversion time using the LUAJIT just in time compiler.

V Going UTF

LuaTeX only understands input codes in the Universal Character Set Transformation Format, aka UCS Transformation Format, better known as: UTF. There is a good reason for this universal view on characters: whatever support gets hard coded into the programs, it's never enough, as 25 years of TeX history have clearly demonstrated. Macro packages often support more or less standard input encodings, as well as local standards, user adapted ones, etc.

There is enough information on the Internet and in books about what exactly is UTF. If you don't know the details yet: UTF is a multi–byte encoding. The characters with a bytecode up to 127 map onto their normal ASCII representation. A larger number indicates that the following bytes are part of the character code. Up to 4 bytes make an UTF-8 code, while UTF-16 always uses two pairs of bytes.

byte 1	byte 2	byte 3	byte 4	unicode
192–223	128–191			0x80–0x7ff
224–239	128–191	128–191		0x800–0xffff
240–247	128–191	128–191	128–191	0x10000–0x1ffff

In UTF-8 the characters in the range 128–191 are illegal as first characters. The characters 254 and 255 are completely illegal and should not appear at all since they are related to UTF-16.

Instead of providing a never-complete truckload of other input formats, LuaTeX sticks to one input encoding but at the same time provides hooks that permits users to write filters that preprocess their input into UTF.

While writing the LuaTeX code as well as the ConTeXt input handling, we experimented a lot. Right from the beginning we had a pretty clear picture of what we wanted to achieve and how it could be done, but in the end arrived at solutions that permitted fast and efficient Lua scripting as well as a simple interface.

What is involved in handling any input encoding and especially UTF?. First of all, we wanted to support UTF-8 as well as UTF-16. LuaTeX implements UTF-8 rather straightforward: it just assumes that the input is usable UTF. This means that it does not combine characters. There is a good reason for this: any automation needs to be configurable (on/off) and the more is done in the core, the slower it gets.

In Unicode, when a character is followed by an 'accent', the standard may prescribe that these two characters are replaced by one. Of course, when characters turn into glyphs, and when no matching glyph is present, we may need to decompose any character into components and paste them together from glyphs in fonts. Therefore, as a first step, a collapser was written. In the (pre)loaded Lua tables we have stored information about what combination of characters need to be combined into another character.

So, an a followed by an ` becomes à and an e followed by " becomes ë. This process is repeated till no more sequences combine. After a few alternatives we arrived at a solution that is acceptably fast: mere milliseconds per average page. Experiments demonstrated that we can not gain much by implementing this in pure C, but we did gain some speed by using a dedicated loop–over–utf–string function.

A second utf related issue is utf-16. This coding scheme comes in two endian variants. We wanted to do the conversion in Lua, but decided to play a bit with a multi–byte file read function. After some experiments we quickly learned that hard coding such methods in TeX was doomed to be complex, and the whole idea behind LuaTeX is to make things less complex. The complexity has to do with the fact that we need some control over the different linebreak triggers, that is, (combinations of) character 10 and/or 13. In the end, the multi–byte readers were removed from the code and we ended up with a pure Lua solution, which could be sped up by using a multi–byte loop–over–string function.

Instead of hard coding solutions in LuaTeX a couple of fast loop–over–string functions were added to the Lua string function repertoire and the solutions were coded in Lua. We did extensive timing with huge utf-16 encoded files, and are confident that fast solutions can be found. Keep in mind that reading files is never the bottleneck anyway. The only drawback of an efficient utf-16 reader is that the file is loaded into memory, but this is hardly a problem.

Concerning arbitrary input encodings, we can be brief. It's rather easy to loop over a string and replace characters in the 0–255 range by their utf counterparts. All one needs is to maintain conversion tables and TeX macro packages have always done that.

Yet another (more obscure) kind of remapping concerns those special TeX characters. If we use a traditional TeX auxiliary file, then we must make sure that for instance percent signs, hashes, dollars and other characters are handled right. If we set the catcode of the percent sign to 'letter', then we get into trouble when such a percent sign ends up in the table of contents and is read in under a different catcode regime (and becomes for instance a comment symbol). One way to deal with such situations is to temporarily move the problematic characters into a private Unicode area and deal with them accordingly. In that case they no longer can interfere.

Where do we handle such conversions? There are two places where we can hook converters into the input.

1. each time when we read a line from a file, i.e. we can hook conversion code into the read callbacks

2. using the special `process_input_buffer` callback which is called whenever TeX needs a new line of input

Because we can overload the standard file open and read functions, we can easily hook the UTF collapse function into the readers. The same is true for the UTF-16 handler. In ConTeXt, for performance reasons we load such files into memory, which means that we also need to provide a special reader to TeX. When handling UTF-16, we don't need to combine characters so that stage is skipped then.

So, to summarize this, here is what we do in ConTeXt. Keep in mind that we overload the standard input methods and therefore have complete control over how LuaTeX locates and opens files.

1. When we have a UTF file, we will read from that file line by line, and combine characters when collapsing is enabled.
2. When LuaTeX wants to open a file, we look into the first bytes to see if it is a UTF-16 file, in either big or little endian format. When this is the case, we load the file into memory, convert the data to UTF-8, identify lines, and provide a reader that will give back the file linewise.
3. When we have been told to recode the input (i.e. when we have enabled an input regime) we use the normal line–by–line reader and convert those lines on the fly into valid UTF. No collapsing is needed.

Because we conduct our experiments in ConTeXt MkIV the code that we provide may look a bit messy and more complex than the previous description may suggest. But keep in mind that a mature macro package needs to adapt to what users are accustomed to. The fact that LuaTeX moved on to UTF input does not mean that all the tools that users use and the files that they have produced over decades automagically convert as well.

Because we are now living in a UTF world, we need to keep that in mind when we do tricky things with sequences of characters, for instance in processing verbatim. When we implement verbatim in pure TeX we can do as before, but when we let Lua kick in, we need to use string methods that are UTF-aware. In addition to the linked-in Unicode library, there are dedicated iterator functions added to the `string` namespace; think of:

```
for c in string.utfcharacters(str) do
    something_with(c)
end
```

Occasionally we need to output raw 8-bit code, for instance to DVI or PDF backends (specials and literals). Of course we could have cooked up a truckload of conversion functions for this, but during one of our travels to a TeX conference, we came up with the following trick.

We reserve the top 256 values of the Unicode range, starting at hexadecimal value 0x110000, for byte output. When writing to an output stream, that offset will be subtracted. So, 0x1100A9 is written out as hexadecimal byte value A9, which is the decimal value 169, which in the Latin 1 encoding is the slot for the copyright sign.

VI A fresh look at fonts

readers

Now that we have the file system, LUA script integration, input encoding and basic logging in place, we have arrived at fonts. Although today OPENTYPE fonts are the fashion, we still need to deal with TEX's native font machinery. Although Latin Modern and the TEX Gyre collection will bring us many free OPENTYPE fonts, we can be sure that for a long time TYPE1 variants will be used as well, and when one has lots of bought fonts, replacing them with OPENTYPE updates is not always an option. And so, reimplementing the readers for TEX Font Metrics (`tfm` files) and Virtual Fonts (`vf` files), was the first step.

Because ALEPH font handling was integrated already, Taco decided to combine the TFM and OFM readers into a new one. The combined loader is written in C and produces tables that are accessible from within LUA. A problem is that once a font is used, one cannot simply change its metrics. So, we have to make sure that we apply changes before a font is actually used:

```
\font\test=texnansi-lmr at 31.415 pt
\test Yet another nice Kate Bush song: Pi
```

In this example, any change to the fontmetrics has to be done before `test` is invoked. For this purpose the `define_font` callback is provided. Below you see an experimental overload:

```
callback.register("define_font", function (name,area,size)
    return fonts.patches.process(font.read_tfm(name,size))
end )
```

The `fonts.patched.process` function (currently in CONTEXT MKIV) implements a mechanism for tweaking the font parameters in between. In order to get an idea of further features we played a bit with ligature replacement, character spacing, kern tweaking etc. Think of such a function (or a chain of functions) doing things similar to:

```
callback.register("define_font", function (name,area,size)
    local tfmblob = font.read_tfm(name,size) -- build in loader
    tfmblob.characters[string.byte("f")].ligatures = nil
    return tfmblob -- datastructure that TeX will use internally
end )
```

Of course the above definition is not complete, if only because we need to handle chained ligatures as well (fl followed by i).

In practice we prefer a more abstract interface (at the macro level) but the idea stays the same. Interesting is that having access to the internals this way already makes our TEX Live more interesting. (We cannot demonstrate this trickery here because when this document is processed you cannot be sure if the experimental interface is still in place.)

When playing with this we ran into problems with file searching. When performing the backend role, LUATEX will look in the TEX tree if there is a corresponding virtual file. It took a while and a bit of tracing (which is not that hard in the LUA based reader) to figure out that the omega related path definitions in `texmf.cnf` files were not correct, something that went unnoticed because omega never had a backend integrated and the DVI processors did multiple searches to get around this.

Currently, if you want to enable extensive tracing of file searching and loading, you can set an environment variable:

```
MTX.INPUT.TRACE=3
```

This will produce a lot of information about what file is asked for, what types (tex, font, etc) determines the search, along what paths is being searched, what readers and locators are used (file, zip, protocol), etc.

AFM

While Taco implemented the virtual font reader —eventually its data will be merged with the TFM table— I started playing with constructing TFM tables directly. Because CONTEXT has a rather systematic naming scheme, we can rather easily see which encoding we are dealing with. This means that in principle we can throw all encoded TFM files out of our tree and construct the tables using the AFM file and an encoding vector.

It took us a good day to figure out the details, but in the end we were able to trick LUATEX into using AFM files. With a bit of internal caching it was even reasonable fast. When the basic conversion mechanism was written we tried to compare the results with existing TFM metrics as generated by `afm2tfm` and `afm2pl`. Doing so was less trivial than we first thought. To mention a few aspects:

- heights and depths have a limited number of values in TEX
- we need to convert to TEX's scaled points
- rounding errors of one scaled point occur
- `afm2tfm` can only add kerns when virtual fonts are used
- `afm2tfm` adds some extra ligatures and also does some kern magic
- `afm2pl` adds even more kerns
- the tools remove kern pars between digits

In this perspective we need not be too picky on what exactly a ligature is. An example of a ligature is `fi` and such a character can be in the font. In the TFM file, the definition of `f` contains information about what to do when it's followed by an `i`: it has to insert a reference (character number) pointing to the fi glyph.

However, because TEX was written in ASCII time space, there was a problem of how to get access to for instance the Spanish quotation and exclamation marks. Here the ligature mechanism available in the TFM format was misused in the sense that a combination of `exclam` and `quoteleft` becomes `exclamdown`. In a similar fashion will two single quotes become a double quote. And every TEXie knows that multiple hyphens combine into – (endash) and — (emdash), where the later one is achieved by defining a ligature between an endash and a hyphen.

Of course we have to deal with conversions from AFM units (1000 per em) to TEX's scaled points. Such conversions may be sensitive for rounding errors. Because we noticed differences of one scaled point, I tried several strategies to get the results consistent but so far I didn't manage to find out where these differences come from. Rounding errors seem to be rather random and I have no clue what strategy the regular converters follow. Another fuzzy area are the font parameters (visible as font dimensions for users): I wonder how many users really know what values are used and why.

You may wonder to what extend this rounding problem will influence consistent typesetting. We have no reason to assume that the rounding error is operating system dependent. This leaves the different methods used and personally I have no problems with the direct reader being not 100% compatible with the regular tools. First of all it's an illusion to think that TEX distributions are stable over the years. Fonts and conversion tools are being updated every now and then, and metrics change over time (apart from Computer Modern which is stable by definition). Also, pattern file are updated, so paragraphs may be broken into lines different anyway. If you really want stability, then you need to store the fonts and patterns with your document.

As we already mentioned, the regular converter programs add kerns as well. Treating common glyph shapes similar is not uncommon in ConTEXt so I decided to provide methods for adding 'missing' kerns. For example, with regards to kerning, we can treat `eacute` the same way as an `e`. Some ligatures, like `ae` or `fi`, need to be seen from two sides: when looked at from the left side they resemble an `a` and `f`, but when kerned at their right, they are to be treated as `e` and `i`.

So, when all this is taken care of, we will have a reasonable robust and compatible way to deal with AFM files and when this variant is enabled, we can prune our TEX trees pretty well. Also, now that we have font related tables, we can start moving tables built out of TEX macros (think of protruding and hz) to Lua, which will not only save us much hash entries but also permits us faster implementations.

The question may arise why there is no hard coded AFM reader. Although some speed up can be achieved by reading the table with AFM data directly, there would still be the issue of making that table accessible for manipulations as described (costs time too). The AFM format is human readable contrary to the TFM format and therefore they can conveniently be processed by LUA. Also, the possible manipulations may differ per macro package, user, and even documents. The changes of users and developers reaching an agreement about such issues is near zero. By writing the reader in LUA, a macro package writer can also implement caching mechanisms that suits the package. Also, keep in mind that we often only need to load about four AFM files or a few more when we mix fonts.

In my main tree (regular distributions) there are some 350 files in `texnansi` encoding that take over 2 MByte. My personal font tree has over a thousand such entries which means that we can prune the tree considerably when we use the AFM loader. Why bother about TFM when AFM can do the job.

In order to reduce the overhead in reading the AFM file, we now use external caching, which (in ConTEXt MkIV) boils down to serializing the internal AFM tables and compiling them to bytecode. As a result, the runtime becomes comparable to a run using regular TFM files. On this document usign the AFM reader (cached) takes some .3 seconds more on 8 seconds total (28 pages in Optima Nova with a couple of graphics).

While we were playing with this, Hermann Zapf surprised me by sending me a CD with his marvelous new Palatino Sans. So, instead of generating TFM metrics, I decided to use `ttf2afm` to generate me an AFM file from the TRUETYPE files and use these metrics. It worked right out of the box which means that one can copy a set of font files directly from the source to the tree. In a demo document the Palatino Sans came out quite well and so we will use this font to explore the upcoming Open Type features.

Because we now have less font resources (only two files per font) we decided to get away from the spread–all–over–the–tree paradigm. For this we introduced

```
../fonts/data/vendor/collection
```

like:

```
../fonts/data/tex/latin-modern
../fonts/data/tex-gyre/bonum
../fonts/data/linotype/optima-nova
../fonts/data/linotype/palatino-nova
../fonts/data/linotype/palatino-sans
```

Of course one needs to adapt the related font paths in the configuration files but getting that done in tex distributions is another story.

map files

Reading an AFM file is only part of the game. Because we bypass the regular TFM reader we may internally end up with different names of fonts (and/or files). This also means that the map files that map an internal name onto an font (outline) file may be of no use. The map file also specifies the encoding file which maps character numbers onto names used in font files.

The map file maps a font name to a (preferable outline) font resource file. This can be a file with suffix `pfb`, `ttf`, `otf` or alike. When we convert am AFM file into a more suitable format, we also store the associated (outline) filename, that we use later when we assemble the map line data (we use `\pdfmapline` to tell LuaTEX how to prepare and embed a file.

Eventually LuaTEX will take care of all these issues itself thereby rendering map files and encoding files kind of useless. When loading an AFM file we already have to read encoding files, so we have all the information available that normally goes into the map file. While conducting experiments with reading AFM files, we therefore could use the `\pdfmapline` primitive to push the right entries into font inclusion machinery. Because ConTEXt already handles map data itself we could easily hook this into the normal handlers for that. (There are some nasty synchronization issues involved in handling map entries in general but we will not bother you with that now).

Although eventually we may get rid of map files, we also used the general map file handling in ConTEXt as a playground for the XML handler that we wrote in Lua. Playing with many map files (a few KBytes) coded in XML format, or with one big map file (easily 800 MBytes) makes a good test case for loading and dumping

But why bother too much about map files in LuaTEX . . . they will go away anyway.

OTF & TTF

One of the reasons for starting the LuaTEX development was that we wanted to be able to use OpenType (and TrueType) fonts in pdfTEX. As a prelude (and kind of transition) we first dealt with Type1 using either TFM or AFM. For TEX it does not really matter what font is used, it only deals with dimensions and generic characteristics. Of course, when fonts offer more advanced possibilities, we may need more features in the TEX kernel, but think of HZ or protruding as provided by pdfTEX: it's not part of the font (specification) but of the engine. The same is actually true for kerning and ligature building, although here the font (data) may provide the information needed to deal with it properly.

OpenType fonts come with features. Examples of features are using oldstyle figures or tabular digits instead of the default ones. Dealing with such issues boils down to replacing one character representation by another or treating combinations of character in the input differently depending on the circumstances. There can be relationships between languages and scripts, but, as TEXies know, other relationships exist as well, for instance between content and visualization.

Therefore, it will be no surprise that LUATEX does not simply implement the OPENTYPE specification as such. On the one hand it implements a way to load information stored in the font, on the other hand it implements mechanisms to fullfil the demands of such fonts and more. The glue between both is done with LUA. In the simple case of ligatures and kerns this goes as follows. A user (or macropackage) specified a font, and this call can be intercepted using a callback. This callback can use a built in function that loads an OTF or TTF font. From this table, a font table is constructed that is passed on to TEX. The construction may involve building ligature and kerning tables using the information present in the font file, but it may as well mean more. So, given a bare LUATEX system, OPENTYPE font support is not giving you automatically handling of features, or more precisely, there is no hard coded support for features.

This may sound as a disadvantage but as soon as you start looking at how TEX users use their system (in most cases by using a macro package) you may understand that flexibility is larger this way. Instead of adding more and more control and exceptions, and thereby making the kernel more instable and complex, we delegate control to the macro package. The advantage is that there are no (everlasting) discussions on how to deal with things and in the end the user will use a high level interface anyway. Of course the macro package needs proper access to the font's internals, but this is provided: the code used for reading in the data comes from FontForge (an advanced font editor) and is presented via LUA tables in a well organized way.

Given that users expect OPENTYPE features to be supported, how do we provide an interface. In CONTEXT the user interface has always be an important aspect and consistency is a priority. On the other hand, there has been the tradition of specifying the size explicity and a new custom introduced by XETEX to enhance fontname with directives. Traditional TEX provides:

```
\font \name filename [optional size]
```

XETEX accepts

```
\font \name "fontname[:optional features]" [optional size]
\font \name  fontname[:optional features]  [optional size]
```

Instead of a fontname one can pass a filename between square brackets. LUATEX handles:

```
\font \name  anything  [optional size]
\font \name {anything} [optional size]
```

where anything as well as the size are passed on to the callback.

This permits us to implement a traditional specification, support XƎTEX like definitions, and easily pass information from a macro package down to the callback as well. Interpreting anything is done in LUA.

While implementing the LUA side of the loader we took a similar approach as the AFM reader and cached intermediate tables as well as keep track of font names (in addition to filenames). In order to be able to quickly determine the (internal) font name of an OPENTYPE font, special loader functions are provided.

The size is kind of special, because we can have specifications like

```
at 10pt
at 3ex
at \dimexpr\bodyfontsize+1pt\relax
```

This means that we need to handle that on the TEX side and pass the calculated value to the callback.

Virtual fonts have a rather special nature. They permit you to define variations of fonts using other fonts and special (DVI related) operators. However, from the perspective of TEX itself they don't exist at all. When you create a virtual font you also end up with a TFM file and TEX only needs this file, which defined characters in terms of a width, height, depth and italic correction as well as associates characters with kerning pairs and ligatures. TEX leaves it to the backend to deal the actual glyphs and therefore the backend will be confronted by the internals of a virtual font. Because PDFTEX and therefore LUATEX has the backend built in, it is capable of handling virtual fonts information.

In LUATEX you can build your own virtual font and this will suit us well. It permits us for instance to complete fonts that lack certain characters (glyphs) and thereby let us get rid of ugly macro based fallback trickery. Although in CONTEXT we will provide a high level interface, we will give you a taste of LUA here.

```
callback.register("define_font", function(name,size)
    if name == "demo" then
        local f = font.read_tfm('texnansi-lmr10',size)
        if f then
            local capscale, digscale = 0.85, 0.75
            f.name, f.type = name, 'virtual'
            f.fonts = {
                { name="texnansi-lmr10" , size=size },
                { name="texnansi-lmss10", size=size*capscale },
                { name="texnansi-lmtt10", size=size*digscale }
            }
            for k,v in pairs(f.characters) do
               local chr = utf.char(k)
               if chr:find("[A-Z]") then
                    v.width = capscale*v.width
                    v.commands = {
                        {"special","pdf: 1 0 0 rg"},
                        {"font",2}, {"char",k},
                        {"special","pdf: 0 g"}
                    }
                elseif chr:find("[0-9]") then
                    v.width  = digscale*v.width
                    v.commands = {
                        {"special","pdf: 0 0 1 rg"},
                        {"font",3}, {"char",k},
                        {"special","pdf: 0 g"}
                    }
                else
                    v.commands = {
                        {"font",1}, {"char",k}
                    }
                end
            end
            return f
        end
    end
    return font.read_tfm(name,size)
end)
```

Here we define a virtual font that uses three real fonts and which font is used depends on the kind of character we're dealing with (inreal world situations we can best use the MkIV function that tells what class a character belongs to). The `commands` table determines what glyphs comes out in what way. We use a bit of literal pdf code to color the special characters but generally color is not handled at the font level.

This example can be used like:

```
\font\test=demo \test
Hi there, this is the first (number 1) example of playing with
Virtual Fonts, some neat feature of \TeX, once you have access
to it. For instance, we can misuse it to fill in gaps in fonts.
```

During development of this mechanism, we decided to save some redundant loading by permitting id's in the fonts array:

```
callback.register("define_font", function(name,size)
    if name == "demo" then
        local f = font.read_tfm('texnansi-lmr10',size)
        if f then
            local id = font.define(f)
            local capscale, digscale = 0.85, 0.75
            f.name, f.type = name, 'virtual'
            f.fonts = {
                { id=id },
                { name="texnansi-lmss10", size=size*capscale },
                { name="texnansi-lmtt10", size=size*digscale }
            }
            for k,v in pairs(f.characters) do
               local chr = utf.char(k)
               if chr:find("[A-Z]") then
                    v.width = capscale*v.width
                    v.commands = {
                        {"special","pdf: 1 0 0 rg"},
                        {"slot",2,k},
                        {"special","pdf: 0 g"}
                    }
                elseif chr:find("[0-9]") then
                    v.width  = digscale*v.width
                    v.commands = {
                        {"special","pdf: 0 0 1 rg"},
                        {"slot",3,k},
                        {"special","pdf: 0 g"}
                    }
                else
                    v.commands = {
                        {"slot",1,k}
                    }
                end
            end
            return f
        end
    end
    return font.read_tfm(name,size)
end)
```

Hardwiring fontnames in callbacks this way does not deserve a price and when possible we will provide better extension interfaces. Anyhow, in the experimental ConTeXt code we used calls like this, where `demo` is an installed feature.

```
\font\myfont = special@demo-1 at 12pt \myfont
Hi there, this is the first (number 1) example of playing with Virtual
Fonts,
some neat feature of \TeX, once you have access to it. For instance,
we can
misuse it to fill in gaps in fonts.
```

Hi there, this is the first (number 1) example of playing with Virtual Fonts, some neat feature of TEX, once you have access to it. For instance, we can misuse it to fill in gaps in fonts.

Keep in mind that this is just an example. In practice we will not do such things at the font level but by manipulating TEX's internals.

While developing this functionality and especially when Taco was programming the backend functionality, we used more sane MkIV code. Think of (still Lua) definitions like:

```
\ctxlua {
    fonts.definers.methods.install("weird", {
        { "copy-range",     "lmroman10-regular"                      }
,
        { "copy-char",      "lmroman10-regular",          65,     66
} ,
        { "copy-range",     "lmsans10-regular",       0x0100, 0x01FF
} ,
        { "copy-range",     "lmtypewriter10-regular", 0x0200, 0xFF00
} ,
        { "fallback-range", "lmtypewriter10-regular", 0x0000, 0x0200
}
    })
}
```

Again, this is not the final user interface, but it shows the direction we're heading. The result looks like:

```
\font\test={myfont@weird} at 12pt \test
\eacute \rcaron \adoublegrave \char65
```

This shows up as:

éřȁB

Here the `@` tells the (new) ConTEXt font handler what constructor should be used.

Because some testers already have XƎTEX font support files, we also support a XƎTEX like definition syntax.

```
\font\test={lmroman10-regular:dlig;liga}\test
f i fi ffi \crlf
f i f\kern0pti f\kern0ptf\kern0pti \crlf
\char64259 \space\char64256 \char105 \space \char102\char102\char105
```

This gives:

f i fi ffi
f i fi ffi
ffi ffi ffi

We are quite tolerant with regards to this specification and will provide less dense methods as well. Of course we need to implement a whole bunch of features but we will do this in such a way that we give users full control.

encodings

By now we've reached a stage where we can get rid of font encodings. We now have the full unicode range available and no longer depend on the font encoding when we hyphenate. In a previous chapter we discussed the difference in size between formats.

date	luatex	pdftex
2006-10-23	3 135 568	7 095 775
2007-02-18	3 373 206	7 426 451
2007-02-19	3 060 103	7 426 451

The size of the formats has grown a bit due to a few more patterns and a extra preloaded encoding. But the LuaTeX format shrinks some 10% now that we can get rid of encoding support. Some support for encodings is still present, so that one can keep using the metric files that are installed (for instance in project related trees that have special fonts) although AFM/Type1 files or OpenType fonts will be used when available.

A couple of years from now, we may throw away some Lua code related to encodings.

files

TeX distributions tend to be rather large, both in terms of files and bytes. Fonts take most of the space. The merged TeXLive 2007 trees contain some 60.000 files that take 1.123 MBytes. Of this, 25.000 files concern fonts totaling to 431 MBytes. A recent ConTeXt distribution spans 1200 files and 20 MBytes and a bit more when third party modules are taken into account. The fonts in TeXLive are distributed as follows:

format	files	bytes

AFM	1.769	123.068.970	443	22.290.132
TFM	10.613	44.915.448	2.346	8.028.920
VF	3.798	6.322.343	861	1.391.684
TYPE1	2.904	180.567.337	456	18.375.045
TRUETYPE	22	1.494.943		
OPENTYPE	144	17.571.732		
ENC	268	782.680		
MAP	406	6.098.982	110	129.135
OFM	39	10.309.792		
OVF	39	413.352		
OVP	22	2.698.027		
SOURCE	4.736	25.932.413		

We omitted the more obscure file types. The last two columns show the numbers for one of my local font trees.

In due time we will see a shift from Type1 to OpenType and TrueType files and because these fonts are more complete, they may take some more space. More important is that the TeX specific font metric files will phase out and the less Type1 fonts we have, the less afm companions we need (afm files are not compressed and therefore relatively large). Mapping and encoding files can also go away.

In LuaTeX we can do with less files, but the number of bytes may grow a bit depending on how much is catched (especially fonts). Anyhow, we can safely assume that a LuaTeX based distributions will carry less files and less bytes around.

fallbacks

Do we need virtual fonts? Currently in ConTeXt, when a font encoding is chosen, a fallback mechanism steps in as soon as a character is not in the encoding. So far, so good. But occasionally we run into a font that does not (completely) fits an encoding and we end up with defining a non standard one. In traditional TeX a side effects of font encodings is that they relate to hyphenation. ConTeXt can deal with that comfortably and multiple instances of the same set of hyphenation patterns can be loaded, but for custom encodings this is kind of cumbersome.

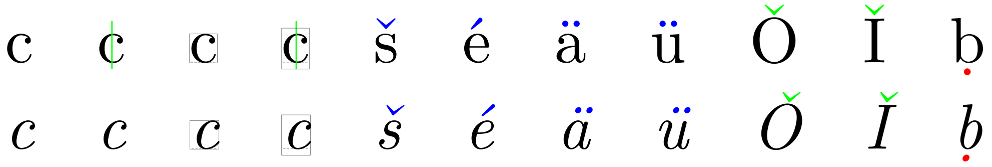
In LuaTeX we have just one font encoding: Unicode. When OpenType fonts are used, we don't expect many problems related to missing glyphs, but you can bet on it that they will occur. This is where in ConTeXt MkIV fallbacks will be used and this will be implemented using vitual fonts. The advantage of using virtual fonts is that we still deal with proper characters and hyphenation will take place as expected. And since virtual fonts can be defined on the fly, we can be flexible in our implementation. We can think of generic fallbacks, not much different than macro based representations, or font specific ones, where we even may rely on METAPOST for generating the glyph data.

How do we define a fall back character. When building this mechanism I used the '¢' as an example. A cent symbol is roughly defined as follows:

```
local t = table.fastcopy(g.characters[0x0063]) -- mkiv function
local s = fonts.constructors.scaled(g.fonts[1].size)    -- mkiv function
t.commands = {
    {"push"},
    {"slot", 1, c},
    {"pop"},
    {"right", .5*t.width},
    {"down",  .2*t.height},
    {"rule", 1.4*t.height, .02*s}
}
t.height = 1.2*t.height
t.depth  = 0.2*t.height
```

Here, `g` is a loaded font (table) which has type `virtual`. The first font in the `fonts` array is the main font. What happens here is the following: we assign the characteristics of 'c' to the cent symbol (this includes kerning and dimensions) and then define a command sequence that draws the 'c' and a vertical rule through it.

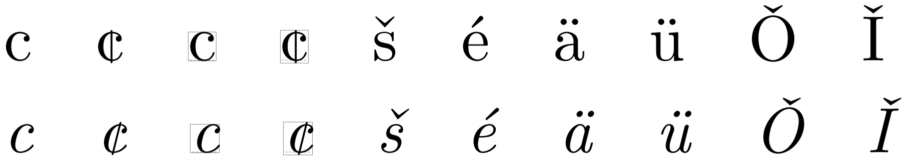
The real code is slightly more complicated because we need to take care of italic properties when applicable and because we have added some tracing too. While playing with this kind of things, it becomes clear what features are handy, and the reason that we now have a virtual command `comment` is that it permits us to implement tracing (using for instance color specials).



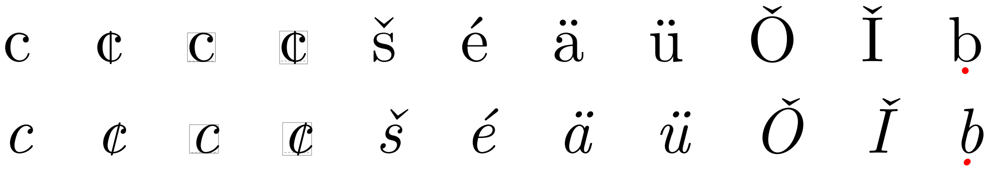
The previous lines are typeset using a similar specification as mentioned before:

```
\font\test=lmroman10-regular@demo-2
```

Without the fallbacks we get:



And with normal (non forced fallbacks) it looks as follows. As it happens, this font has a cent symbol so no fallback is needed.



The font definition callback intercepts the `demo-2` and a couple of chained lua functions make sure that characters missing in the font are replaced by fallbacks. In the case of missing composed characters, they are constructed from their components. In this particular example we have told the handler to assume that all composed characters are missing.

memory

Traditional TeX has been designed for speed and a small memory footprint. Todays implementations are considerably more generous with the amount of memory that you can use (hash, fonts, main memory, patterns, backend, etc). Depending on how complicated a document layout it, memory may run into tens of megabytes.

Because LuaTeX is not only suitable for wide fonts, but also does away with some of the optimizations in the TeX code that complicate extensions, it has a larger footprint that pdfTeX. When implementing the OpenType font basics, we did quite some tests with respect to memory usage. Getting the numbers right is non trivial because the Lua garbage collector is interfering. For instance, on my machine a test file with the regular ConTeXt setup of of Latin Modern fonts made Lua allocate 130 MB, while the same run on Taco's machine took 100 MB.

When a font data table is constructed, it is handled over to TeX, and turned into the internal font data structures. During the construction of that TABLE at the Lua end, ConTeXt MkIV disables the garbage collector. By doing this, the time needed to construct and scale a font can be halved. Curious to the amount of memory involved in passing such a table, I added the following piece of code:

```
if type(fontdata) == "table" then
    local s = statistics.luastate_bytes
    local t = table.copy(fontdata)
    local d = statistics.luastate_bytes-s
    texio.write_nl(string.format("table memory footprint: %s",d))
end
```

It turned out that a Regular Latin Modern font (OpenType) takes around 800 KB. However, more interesting was that by adding this snippet of testcode which duplicted the table in order to measure its size, the total memory footprint dropped to 100 MB (about the amount used on Taco's machine). This demonstrates that one should be very careful with drawing conclusions.

Because fonts are rather important in TEX and because there can be lots of them used, it makes sense to keep an eye on memory as well as performance. Because many manipulations now take place in LUA, it no longer makes sense to let TEX buffer fonts. In plain TEX one finds these magic

```
\font\preloaded=cmr10
\font\preloaded=cmr12
```

lines. The second definitions obscures the first, but the `cmr10` stays loaded.

```
\font\one=cmr10 at 10pt
\font\two=cmr10 at 10pt
```

These two definitions make TEX load the font only once. However, since we can now delegate loading to LUA, TEX no longer helps us there. For instance, TEX has no knowledge to what extend this `cmr10` font has been manipulated and therefore both instances may actually differ.

When you use a callback to define the font, TEX passes a font id number. You can use this number as a reference to a loaded font (that is, passed to TEX). If instead of a table, you return a number, TEX will reuse the already loaded font. This feature can save you a lot of time, especially when a macro package (like CONTEXT) defines fonts dynamically which means that when grouping is used, fonts get (re)defined a lot. Of course additional caching can take place at the LUA end, but there one needs to take into account more than just the scaled instance. Think of OPENTYPE features or virtual font properties. The following are quite certainly different setups, in spite of the common size.

```
\font\one=lmr10@demo-1 at 10pt
\font\two=lmr10@demo-2 at 10pt
```

When scaling a font, one not only needs to handle the regular glyph dimensions, but also the kerning tables. We found out that dealing with such issues takes some 25% of the time spent on loading Latin Modern fonts that have rather extensive kerning tables. When creating a virtual font, copying glyph tables may happen a lot. Deep copying tables takes a bit of time. This is one of the reasons why we discussed (and consider) some dedicated support functions so that copying and recalculating tables happens faster (less costly hash lookups and such). On the other hand, the time wasted on calculations (including rounding to scaled points) can be neglected.

The following table shows what happens when we enforce a different garbage collecting scheme. This test was triggered by another experiment where at regular time, for instance after a pag eis shipped out, say

```
collectgarbage("collect")
```

However, such a complete sweep has drastic consequences for the runtime. But, since the memory footprint becomes 10–15% less by doing so, we played a bit with

```
collectgarbage("setstepmul", somenumber)
```

When processing a not so large file but one that loads a bunch of open type fonts, we get the following values. The left set is on linux (Taco's machine) and the right set in mine.

stepmul	**run (s)**	**mem (MB)**	**run (s)**	**mem (MB)**
200	1.58	69.14	5.6	84.17
1000	1.63	69.14	6.5	72.32
2000	1.64	60.66	6.8	73.53
10000	1.71	59.94	7.0	72.30

Since I use an old laptop running Windows with a probably different TEX configuration (fonts), and under some load, both columns don't compare well, but the general idea is the same. For practical usage a value of 1000 is probably best, especially because memory intensive font and script loading only happens at the first couple of pages.

VII Token speak

tokenization

Most TeX users only deal with (keyed in) characters and (produced) output. Some will play with boxes, skips and kerns or maybe even leaders (repeated sequences of the former). Others will be grateful that macro package writers take care of such things.

Macro writers on the other hand deal properties of characters, like catcodes and a truckload of other codes, with lists made out of boxes, skips, kerns and penalties but even they cannot look much deeper into TeX's internals. Their deeper understanding comes from reading the TeXbook or even looking at the source code.

When someone enters the magic world of TeX and starts asking around on a bit, he or she will at some point get confronted with the concept of 'tokens'. A token is what ends up in TeX after characters have entered its machinery. Sometimes it even seems that one is only considered a qualified macro writer if one can talk the right token–speak. So what are those magic tokens and how can LuaTeX shed light on this.

In a moment we will show examples of how LuaTeX turns characters into tokens, but when looking at those sequences, you need to keep a few things in mind:

- A sequence of characters that starts with an escape symbol (normally this is the backslash) is looked up in the hash table (which relates those names to meanings) and replaced by its reference. Such a reference is much faster than looking up the sequence each time.
- Characters can have special meanings, for instance a dollar is often used to enter and exit math mode, and a percent symbol starts a comment and hides everything following it on the same line. These meanings are determined by the character's catcode.
- All the characters that will end up actually typeset have catcode 'letter' or 'other' assigned. A sequence of items with catcode 'letter' is considered a word and can potentially become hyphenated.

examples

We will now provide a few examples of how TeX sees your input.

```
Hi there!
```

cmd	meaning	properties
letter	H	
letter	i	

spacer	
letter	t
letter	h
letter	e
letter	r
letter	e
other	!

Here we see three kind ot tokens. At this stage a space is still recognizable as such but later this will become a skip. In our current setup, the exclamation mark is not a letter.

```
Hans \& Taco use Lua\TeX \char 33\relax
```

cmd	meaning	properties
letter	H	
letter	a	
letter	n	
letter	s	
spacer		
char_given	&	
spacer		
letter	T	
letter	a	
letter	c	
letter	o	
spacer		
letter	u	
letter	s	
letter	e	
spacer		
letter	L	
letter	u	
letter	a	
call	TeX	expandable protected
char_num	char	
other	3	
other	3	
relax	relax	

Here we see a few new tokens, a 'char_given' and a 'call'. The first represents a `\chardef` i.e. a reference to a character slot in a font, and the second one a macro that will expand to the TEX logo. Watch how the space after a control sequence is eaten up. The exclamation mark is a direct reference to character slot 33.

```
\noindent {\bf Hans} \par \hbox{Taco} \endgraf
```

cmd	meaning	properties
start_par	noindent	
left_brace		
call	bf	expandable protected
letter	H	
letter	a	
letter	n	
letter	s	
right_brace		
spacer		
par_end	par	
make_box	hbox	
left_brace		
letter	T	
letter	a	
letter	c	
letter	o	
right_brace		
spacer		
par_end	endgraf	

As you can see, some primitives and macro's that are bound to them (like `\endgraf`) have an internal representation on top of their name.

```
before \dimen2=10pt after \the\dimen2
```

cmd	meaning	properties
letter	b	
letter	e	
letter	f	
letter	o	
letter	r	
letter	e	
spacer		
register	dimen	
other	2	
other	=	
other	1	
other	0	
letter	p	
letter	t	

```
spacer
letter       a
letter       f
letter       t
letter       e
letter       r
spacer
the         the    expandable
register   dimen
other        2
```

As you can see, registers are not explicitly named, one needs the associated register code to determine it's character (a dimension in our case).

```
before \inframed[width=3cm]{whatever} after
```

```
cmd          meaning  properties
letter          b
letter          e
letter          f
letter          o
letter          r
letter          e
spacer
call         inframed  expandable protected
other           [
letter          w
letter          i
letter          d
letter          t
letter          h
other           =
other           3
letter          c
letter          m
other           ]
left_brace
letter          w
letter          h
letter          a
letter          t
letter          e
letter          v
```

letter	e
letter	r
right_brace	
spacer	
letter	a
letter	f
letter	t
letter	e
letter	r

As you can see, even when control sequences are collapsed into a reference, we still end up with many tokens, and because each token has three properties (cmd, chr and id) in practice we end up with more memory used after tokenization.

```
compound|-|word
```

cmd	meaning	properties
letter	c	
letter	o	
letter	m	
letter	p	
letter	o	
letter	u	
letter	n	
letter	d	
call	|	active expandable protected
other	-	
call	|	active expandable protected
letter	w	
letter	o	
letter	r	
letter	d	

This example uses an active character to handle compound words (a ConTeXt feature).

```
hm, \directlua 0 { tex.sprint("Hello World") }
```

cmd	meaning	properties
letter	h	
letter	m	
other	,	
spacer		
convert	directlua	expandable

other	0
spacer	
left_brace	
spacer	
letter	t
letter	e
letter	x
other	.
letter	s
letter	p
letter	r
letter	i
letter	n
letter	t
other	(
other	"
letter	H
letter	e
letter	l
letter	l
letter	o
spacer	
letter	W
letter	o
letter	r
letter	l
letter	d
other	!
other	"
other	)
spacer	
right_brace	

The previous example shows what happens when we include a bit of LUA code … it is just seen as regular input, but when the string is passed to LUA, only the chr property is passed, so we no longer can distinguish between letters and other characters.

A macro definition converts to tokens as follows.

cmd	**meaning**	**properties**
def	def	
undefined_cs		expandable
mac_param		

```
other            1
mac_param
other            2
left_brace
other            [
mac_param
other            2
other            ]
other            [
mac_param
other            1
other            ]
right_brace
spacer
undefined_cs           expandable
left_brace
letter           A
right_brace
left_brace
letter           B
right_brace
```

As we already mentioned, a token has three properties. More details can be found in the reference manual so we will not go into much detail here.

The original interceptor for tokens but that one has been replaced by a more powerful scanning mechanism. The following text is no longer applicable but kept as historic reference. The new token scanner is discussed in later articles.

```
A most simple callback is:

\starttyping
callback.register('token_filter', token.get_next)
\stoptyping

In principle you can call \type {token.get_next} anytime you want
to intercept a token. In that case you can feed back tokens into
\TEX\ by using a trick like:

\starttyping
function tex.printlist(data)
   callback.register('token_filter', function ()
       callback.register('token_filter', nil)
```

```
        return data
    end)
end
\stoptyping

Another example of usage is:

\starttyping
callback.register('token_filter', function ()
    local t = token.get_next
    local cmd, chr, id = t[1], t[2], t[3]
    -- do something with cmd, chr, id
    return { cmd, chr, id }
end)
\stoptyping

There is a whole repertoire of related functions, one is \type
{token.create}, which can be used as:

\starttyping
tex.printlist{
    token.create("hbox"),
    token.create(utf.byte("{"),  1),
    token.create(utf.byte("?"), 12),
    token.create(utf.byte("}"),  2),
}
\stoptyping

This results in: \ctxlua {
    tex.printlist{
        token.create("hbox"),
        token.create(utf.byte("{"),  1),
        token.create(utf.byte("?"), 12),
        token.create(utf.byte("}"),  2),
    }
}

While playing with this we made a few auxiliary functions that
permit things like:

\starttyping
tex.printlist ( table.unnest ( {
    tokens.hbox,
    tokens.bgroup,
    tokens.letters("12345"),
    tokens.egroup,
} ) )
\stoptyping

Unnesting is needed because the result of the \type {letters} call
is a table, and the \type {printlist} function wants a flattened
table.

The result looks like: \ctxlua {
    local t = table.unnest {
        tokens.hbox,
        tokens.bgroup,
        tokens.letters("12345"),
        tokens.egroup,
    }
    tex.printlist (t)
    tokens.collectors.show(t)
}

In practice, manipulating tokens or constructing lists of tokens
this way is rather cumbersome, but at least we now have some
kind of access, if only for illustrative purposes.

\starttyping
\hbox{12345\hbox{54321}}
\stoptyping

can also be done by saying:

\starttyping
tex.sprint("\\hbox{12345\\hbox{54321}}")
\stoptyping

or under \CONTEXT's basic catcode regime:

\starttyping
tex.sprint(tex.ctxcatcodes, "\\hbox{12345\\hbox{54321}}")
\stoptyping

If you like it the hard way:
```

```
\starttyping
tex.printlist ( table.unnest ( {
    tokens.hbox,
        tokens.bgroup,
            tokens.letters("12345"),
            tokens.hbox,
                tokens.bgroup,
                    tokens.letters(string.reverse("12345")),
                tokens.egroup,
        tokens.egroup
} ) )
\stoptyping

This method may attract those who dislike the traditional \TEX\
syntax for doing the same thing. Okay, a careful reader will
notice that reversing the string in \TEX\ takes a bit more
trickery, so \unknown
```

The `tokens` etc. examples shows here make no sense anyway as we have a more extensive interface to the macro language: `context`.

VIII How about performance

remark

The previous chapters already spent some words on performance and memory usage. By the time that Taco and I were implementing, discussing and testing the callbacks related to node lists, we were already convinced that in all areas covered so far (file management, handling input characters, dealing with fonts, conversion to tokens, string and table manipulation, enz.) the TEX–LUA pair was up to the task And so we were quite confident that processing nodes was not only an important aspect of LUATEX but also quite feasable in terms of performance (after all we needed it in order to deal with advanced typesetting of Arab). When Taco was dealing with the TEX side of the story, I was experimenting with possible mechanisms at the LUA end.

At the same time I got the opportunity to speed up the METAPOST to PDF converter and both activities involved some timing. Here I report some of the observations that we made in this process.

parsing

Expressions in LUA are powerful and definitely faster than regular expressions found in other languages, but they have some limits. Most noticeably is the lack of alternation. In RUBY one can say:

```
str = "there is no gamma in here, just an beta"

if str =~ /(alph|bet|delt)a/ then
    print($1)
end
```

but in LUA you need a few more lines:

```
str = "there is no gamma in here, just an beta"

for _, v in pairs({'alpha','beta','delta'}) do
    local s = str:match(v)
    if s then
        print(s)
        break
    end
end
```

Interesting is that upto now I didn't really miss alternation but it may as well be that the lack of it drove me to come up with different solutions. For ConTeXt MkIV the MetaPost to pdf converter has been rewritten in Lua. This is a prelude to direct Lua output from MetaPost but I needed the exercise. It was among the first Lua code in MkIV.

Progressive (sequential) parsing of the data is an option, and is done in MkII using pure TeX. We collect words and compare them to PostScript directives and act accordingly. The messy parts are scanning the preamble, which has specials to be dealt with as well as lots of unpredictable code to skip, and the `fshow` command which adds text to a graphic. But real dirty are the code fragments that deal with setting the line width and penshapes so the cleanup of this takes some time.

In Lua a different approach is taken. There is an `mp` table which collects a lot of functions that more or less reflect the output of MetaPost. The functions take care of generating the right pdf code and also handle the transformations needed because of the differences between PostScript and pdf.

The sequential PostScript that comes from MetaPost is collected in one string and converted using `gsub` into a sequence of Lua function calls. Before this can be done, some cleanup takes place. The resulting string is then executed as Lua code.

As an example:

```
1 0 0 2 0 0 curveto
```

becomes

```
mp.curveto(1,0,0,2,0,0)
```

which results in:

```
\pdfliteral{1 0 0 2 0 0 c}
```

In between, the path is stored and transformed which is needed in the case of penshapes, where some PostScript feature is used that is not available in pdf.

During the development of LuaTeX a new feature was added to Lua: `lpeg`. With `lpeg` you can define text scanners. In fact, you can build parsers for languages quite conveniently so without doubt we will see it show up all over MkIV.

Since I needed an exercise to get accustomed with `lpeg`, I rewrote the mentioned converter. I'm sure that a better implementation is possible than I did (after all, PostScript is a language) but I went for a speedy solution. The following table shows some timings.

gsub	lpeg

2.5	0.5	100 times test graphic
9.2	1.9	100 times big graphic

The test graphic has about everything that MetaPost can output, including special tricks that deal with transparency and shading. The big one is just four copies of the test graphic.

So, the `lpeg` based variant is about 5 times faster than the original variant. I'm not saying that the original implementation is that brilliant, but a 5 time improvement is rather nice especially when you consider that `lpeg` is still experimental and each version performs better. The tests were done with `lpeg` version 0.5 which performs slightly faster than its predecessor.

It's worth mentioning that the original `gsub` based variant was already a bit improved compared to its first implementation. There we collected the TeX (PDF) code in a table and passed it in its concatenated form to TeX. Because the Lua to TeX interface is by now quite efficient we can just pass the intermediate results directly to TeX.

file io

The repertore of functions that deal with individual characters in Lua is small. This does not bother us too much because the individual character is not what TeX is mostly dealing with. A character or sequence of characters becomes a token (internally represented by a table) and tokens result in nodes (again tables, but larger). There are many more tokens involved than nodes: in ConTeXt a ratio of 200 tokens on 1 node are not uncommon. A letter like `x` become a token, but the control sequence `\command` also ends up as one token. Later on, this `x` may become a character node, possibly surrounded by some kerning. The input characters `width` result in 5 tokens, but may not end up as nodes at all, for instance when they are part of a key/value pair in the argument to a command.

Just as there is no guaranteed one–to–one relationship between input characters and tokens, there is no straight relation between tokens and nodes. When dealing with input it is good to keep in mind that because of these interpretation stages one can never say that 1 megabyte of input characters ends up as 1 million something in memory. Just think of how many megabytes of macros get stored in a format file much smaller than the sum of bytes.

We only deal with characters or sequences of bytes when reading from an input medium. There are many ways to deal with the input. For instance one can process the input lines as TeX sees them, in which case TeX takes care of the UTF input. When we're dealing with other input encodings we can hook code into the file openers and readers and convert the raw data ourselves. We can for instance read in a file as a whole, convert it using the normal expression handlers or the byte(pair) iterators that LuaTeX provides, or we can go real low level using native Lua code, as in:

```
do
    local function nextbyte(f)
        return f:read(1)
    end

    function io.bytes(f)
        return nextbyte, f
    end
end

f = io.open("somefile.dat")
for b in io.bytes(f) do
    do_something(b)
end
f:close()
```

Of course in practice one will need to integrate this into one of the reader callback, but the principle stays the same. In case you wonder if calling functions for each byte is fast enough . . . it's more than fast enough for normal purposes, especially if we keep in mind that other tasks like reading of, preparing of and dealing with fonts of processing token lists take way more time. You can be sore that when half a second runtime is spent on reading a file, processing may take minutes. If one wants to sqeeze more performance out of this part, it's always an option to write special libraries for that, but this is beyond standard LuaTeX. We found out that the speed of loading data from files in Lua is mostly related to the small size of Lua's file buffer. Reading data stored in tables is extremely fast, and even faster when precompiled into bytecode.

tables

When Taco and I were experimenting with the callbacks that intercept tokens and nodes, we wondered what the impact would be on performance. Although in MkIV we allocate quite some memory due to font handling, we were pretty sure that handling TeX's internal lists also could have their impact. Data related to fonts is not always subjected to garbage collection, simply because it's to be available permanently. List processing on the other hand involves a lot of temporary allocated tables. During a run a real huge amount of tokens passes the machinery. When digested, they become nodes. For testing we normally use this document (with the name `mk.tex`) and at the time of writing this, it has some 48 pages.

This document is of moderately complexity, but not as complex as the documents that I normally process; they have with lots of graphics, layers, structural elements, maybe a bit of xml parsing, etc. Nevertheless, we're talking of some 24 million tokens entering the engine for 50 pages of text. Contrary to this the number of nodes is small: only 120 thousand but the tables making up the nodes are more complex than token tables (with three numbers per token). When all tokens are intercepted and returned unchanged, on my machine the run is progressively slow and memory usage grows from 75M to 112M. There is room for improvement there, especially in the garbage collector.

Side note: quite some of these tokens result from macro expansion. Also, when in the input a `\command` is used, the callback passes it as one token. A command stores in a format is already tokenized, but a command read from the input is tokenized when read, so behind each token reported there can be a few more input characters, but their number can be neglected compared to tokens originating from the macro package.

The token callback is rather slow when used for a whole document. However, this is typically a callback that will only be used in very special situations and for a controlled number of tokens. The node callback on the other hand can be set permanently. Fortunately the number of nodes is relatively small. The overhead of a simple token handler that just counts nodes is around 5% but most common manipulations with token lists don't take much more time. For instance, experiments with adding kerns around punctuation (a French speciality) hardly takes time, resolving ligatures is not really noticeable and applying inter–character spacing to a whole document is not that slow either. Actually, the last example is kind of special because it more than doubles the size of the node lists. Inserting or removing table elements in relatively slow when tables are large but there are some ways around this.

One of the reasons of whole–document token handling being slow is that each token is a three–element table and so the garbage collector has to work rather hard. The efficiency of this process is also platform dependent (or maybe compiler specific). Manipulating the garbage collector parameters does not improve performance, unless this forces the collector to be inefficient at the cost of a lot of memory.

However, when we started dealing with nodes, I gave tuning the collector another try and on the mentioned test document the following observations were made when manipulating the step multiplier:

step	**runtime**	**memory**
200	24.0	80.5M
175	21.0	78.2M
150	22.0	74.6M
160	22.0	74.6M
165	21.0	77.6M
125	21.5	89.2M
100	21.5	88.4M

As a result, I decided to set the `stepmul` variable to 165.

```
\ctxlua{collectgarbage("setstepmul", 165)}
```

However, when we were testing thenew `lpeg` based MetaPost converter, we ran into problems. For table intensive operations, temporary disabling the garbage collector gave a significant boost in speed. While testing performance we used the following loop:

```
\dorecurse {2000} {
    \setbox \scratchbox \hbox \bgroup
        \convertMPtoPDF{test-mps-procset.mps}{1}{1}
    \egroup
}
```

In such a loop, turning the garbage collector on and off is disasterous. Because no other Lua calls happen between these calls, the garbage collector is never invoked at all. As a result, memory growed from the baseline of 45M to 120MB and processing became incrementally slow. I found out that restarting the collector before each conversion kept memory usage low and the speed also remained okay.

```
\ctxlua{collectgarbage("restart")}
```

Further experiments learned that it makes sense to restart the collector at each shipout and before table intense operations. On `mk.tex` this results in a memory usage of 74M (at the end of the run) and a runtime of 21 seconds.

Concerning nodes and speed/allocation issues, we need to be aware of the fact that this was still somewhat experimental and in the final version of LuaTeX callbacks may occur at different places and lists may be subjected to parsing multiple times at different moments and locations (for instance when we start dealing with attributes, an upcoming new feature).

Back to tokens. The reason why applying the callback to every token takes a while has to do with the fact that each token goes through the associated function. If you want to have an idea of what this means for 24 million tokens, just run the following Lua code:

```
for i=1,24 do
    print(i)
    for j=1,1000*1000 do
        local t = { 1, 2, 3 }
    end
end
print(os.clock())
```

This takes some 60 seconds on my machine. The following code runs about three times faster because the table has not to be allocated each time.

```
t = { 1, 2, 3 }
for i=1,24 do
    print(i)
    for j=1,1000*1000 do
        t[1]=4 t[2]=5 t[3]=6
    end
end
print(os.clock())
```

Imagine this code to be interwoven with other code and TEX doing things with the tokens it gets back. The memory pool will be scattered and garbage collecting will become more difficult.

However, in practice one will only apply token handling to a marked piece of the input data. It is for this reason that the callback is not:

```
callback.register('token_filter', function(t)
    return t
end )
```

but instead

```
callback.register('token_filter', function()
    return token.get_next()
end )
```

This gives the opportunity to fetch more than one token and keep fetching till a criterium is met (for instance a sentinel).

Because `token.get_next` is not bound to the callback you can fetch tokens anytime you want and only use the callback to feed back tokens into TEX. In ConTEXt MkIV there is some collect and flush tokens present. Here is a trivial example:

```
\def\SwapChars{\directlua 0 {
    do
        local t = { token.get_next(), token.get_next() }
        callback.register('token_filter', function()
            callback.register('token_filter', nil)
            return { t[2], t[1] }
        end )
    end
}}

\SwapChars HH \SwapChars TH
```

Collecting tokens can take place inside the callback but also outside. This also gives you the opportunity to collect them in efficient ways and keep an eye on the memory demands.

Of course using TEX directly takes less code:

```
\def\SwapChars#1#2{#2#1}
```

The example shown here involves so little tokens that running it takes no noticeable time. Here we show this definition in tokenized form:

cmd	meaning	properties
def	def	
undefined_cs		expandable
mac_param		
other	1	
mac_param		
other	2	
left_brace		
mac_param		
other	2	
mac_param		
other	1	
right_brace		

IX Nodes and attributes

introduction

Here we will tell a bit about the development of node access in LuaTEX. We will also introduce attributes, a feature closely related to nodes. We assume that you are somewhat familiar with TEX's nodes: glyphs, kerns, glue, penalties, whatsits and friends.

tables

Access to node lists has been implemented rather early in the development because we needed it to fulfil the objectives of the Oriental TEX project. The first implementation used nested tables, indexed by number. In that approach, the first entry in each node indicated the type in string format. At that time a horizontal list looked as follows:

```
list = {
    [1] = "hlist",
    [2] = 0,
    ...
    [8] = {
        [1] = {
            [1] = "glyph",
            ...
        },
        [2] = {
            ...
        }
}
```

Processing such lists is rather convenient since we can use the normal table iterators. Because in practice only a few entries of a node are accessed, working with numbers is no real problem: in slot 1 we have the type, en in the case of a horizontal or vertical list, we know that slot 8 is either empty or a table. Looping over the list is done with:

```
for i, node in ipairs(list) do
    if node[1] == "glyph" then
        list[i][5] = string.byte(string.upper(string.char(node[5])))
    end
end
```

Node processing code hooks into the box packagers and paragraph builder and a few more places. This means that when using the table approach a lot of callbacks take place where TEX has to convert to and from LUA. Apart from processing time, we also have to deal with garbage collection then and on an older machine with insufficient memory interesting bottlenecks show up. Therefore some following optimizations were implemented at the TEX end of the game.

Side note concerning speed: when memory of processing speed is low, runtime can increase five to tenfold compared to PDFTEX when one does intensive node manipulations. This is due to garbage collection at the LUA end and memory (de)allocation at the TEX end. There is not much we can do about that. Interfacing has a price and hardware is more powerful than when TEX was written. Processing the TEX book using no callbacks is not that much slower than using a traditional TEX engine. However, nowadays fonts are more extensive, demands for special features more pressing and that comes at a price.

When the list is not changed, the callback function can return the value `true`. This signals TEX that it can keep the original list. When the list is empty, the callback function can return the value `false`. This signals TEX that the list can be discarded.

In order to minimize conversions and redundant processing, nested lists were not passed as table but as a reference. One could expand such a list when needed. For instance, when one hooks the same function in the `hpack_filter` and `pre_linebreak_filter` callbacks, this way one can be pretty sure that each node is only processed once. Boxed material that is part of the paragraph stream first enters the box packers and then already is processed before it enters the paragraph callback. Of course one can decide the expand the referred sublist and process it again. Keep in mind that we're still talking of a table approach, but we're slowly moving away from big conversions.

In principle one can insert and delete nodes in such a list but given that the average length of a list representing a page is around 4000, you can imagine that moving around a large amount of data is not that efficient. In order to cope with this, we experimented a lot and came to solutions which will be discussed later on.

At the LUA end some tricks were used to avoid the mentioned insertion and deletion penalty. When a node was deleted, we simply set its value to `false`. Deleting all glyphs then became:

```
for i, node in ipairs(list) do
    if node[1] == "glyph" then
        list[i] = false
    end
end
```

When TEX converted a LUA table back into its internal representation, it ignored such false nodes.

For insertion a dummy node was introduced at the Lua end. The next code duplicates the glyphs.

```
for i, node in ipairs(list) do
    if node[1] == "glyph" then
        list[i] = { 'inline', 0, nil, { node, node } }
    end
end
```

Just before we passed the resulting list back to TEX we collapsed these inline pseudo nodes. This was a rather fast operation.

So far so good. But then we introduced attributes and keeping track of them as well as processing them takes quite some processing power. Nodes with attributes then looked like:

```
someglyph = {
    [1] = "glyph",                 -- type
    [2] = 0,                       -- subtype
    [3] = { [1] = 5, [4] = 10 }, -- attributes
    [4] = 88,                      -- slot
    [5] = 32                       -- font
}
```

Constructing attribute tables for each node is costly in terms of memory usage and processing time and we found out that the garbage collector was becoming a bottleneck, especially when resources are thin. We will go into more detail about attributes elsewhere.

lists

At the same time that we discussed these issues, new Dutch word lists (adapted spelling) were published and we started wondering if we could use such lists directly for hyphenation purposes instead of relying on traditional patterns. Here the first observation was that handling these really huge lists is no problem at all. Okay, it costs some memory but we only need to load one of maybe a few of these lists. Hyphenating a paragraph using tables with hyphenated words and processing the paragraph related node list is not only fast, it also gives us the opportunity to cross font boundaries. Of course there are kerns and ligatures to deal with but this is no big deal. At least it can be an alternative or addendum to the current hyphenator. Some languages have very small pattern files or a very systematic approach to hyphenation so there is no reason to abandon the traditional ways in all cases. Take your choice.

When experimenting with the new implementation we tested the performance by letting LUA take care of hyphenation, spell checking (marking words) and adding inter–character kerns. When playing with big lists of words we found out that the caching mechanism could not be used due to some limitations in the LUA byte code interpreter, so eventually we ended up with a dedicated loader.

However, again we ran into performance problems when lists became more complex. And so, instead of converting TEX datastructures into LUA tables userdata types came into view. Taco already had reimplemented the node memory management, so a logical next step was to reimplement the callbacks and box related code to deal with nodes as linked lists. Since this is now the fashion in LUATEX, you may forget the previous examples, although it is not that hard to introduce table representations again once we need them.

Of course this resulted in an adaption to the regular TEX code but a nice side effect was that we could now use fields instead of indexes into the node data structure. There is a small price to pay in terms of performance, but this can be compensated by clever programming.

```
someglyph = {
    type = 41,
    subtype = 0,
    attributes = <attributes>,
    char = 88,
    font = 32
}
```

Attributes themselves are userdata. The same is true for components that are present when we're for instance dealing with ligatures.

As you can see, in the field variant, a type is a number. In practice, because LUA hashes strings, working with strings is as fast when comparing, but since we now have the more abstract type indicator, we stick with the numbers, which saves a few conversions. When dealing with tables we get code like:

```
function loop_over_nodes(list)
    for i, n in ipairs(list)
        local kind = n[1]
        if kind == "hlist" or kind == "vlist" then
            ...
        end
    end
end
```

But now that we have linked lists, we get the following. Node related methods are available in the `node` namespace.

```
function loop_over_nodes(head)
    local hlist, vlist = node.id('hlist'), node.id('vlist')
    while head do
        local kind = head.type
        if kind == hlist or kind == vlist then
            ...
        end
        head = head.next
    end
end
```

Using an abstraction (i.e. a constant representing `hlist` looks nice here, which is why numbers instead of strings are used. The indexed variant is still supported and there we have strings.

Going from a node list (head node) to a table is not that complex. Sometimes this can be handy because manipulating tables is more convenient that messing around with userdata when it comes down to debugging or tracing.

```
function nodes.totable(n)
    function totable(n)
        local f, tt = node.fields(n.id,n.subtype), { }
        for _,v in ipairs(f) do
            local nv = n[v]
            if nv then
                local tnv = type(nv)
                if tnv == "string" or tnv == "number" then
                    tt[v] = nv
                else -- userdata
                    tt[v] = nodes.totable(nv)
                end
            end
        end
        return tt
    end
    local t = { }
    while n do
        t[#t+1] = totable(n)
        n = n.next
    end
    return t
end
```

It will be clear that here we collect data in LUA while treating nodes as userdata keeps most of it at the TEX side and this is where the gain in speed comes from.

side effects

While experimenting with node lists Taco and I ran into a peculiar side effect. One of the tests involved adding kerns between glyphs (inter character spacing as sometimes uses in titles in a large print). When applied to a whole document we noticed that at some places (words) the added kerning was gone. We used the subtype zero kern (which is most efficient) and in the process of hyphenating TEX removes these kerns and inserts them later (but then based on the information stored in the font.

The reason why TEX removes the font related kerns, is the following. Consider the code:

```
\setbox0=\hbox{some text} the text \unhcopy0 has width \the\wd0
```

While constructing the `\hbox`, TEX will apply kerning as dictated by the font. Otherwise the width of the box would not be correct. This means that the node list entering the linebreak machinery contains such kerns. Because hyphenating works on words TEX will remove these kerns in the process of identifying the words. It creates a string, removes the original sequence of nodes, determines hyphenation points, and add the result to the node list. For efficiency reasons TEX will only look at places where hyphenation makes sense.

Now, imagine that we add those kerns in the callback. This time, all characters are surrounded by kerns (which we gave subtype zero). When TEX is determining feasable breakpoints (hyphenation), it will remove those kerns, but only at certain places. Because our kerns are way larger than the normal interglyph kerns, we suddenly end up with an intercharacter spaced paragraph that has some words without such spacing but the font dictated kerns.

most words are spaced but some words are not

Of course a solution is to use a different kern, but at least this shows that the moment of processing nodes as well as the kind of manipulations need to be chosen with care.

Kerning is a nasty business anyway. Imagine the following word:

```
effe
```

When typeset this turns into three characters, one of them being a ligature.

```
[char e] [liga ff (components f f)] [char e]
```

However, in Dutch, such a word hyphenates as:

```
ef-fe
```

This means that in the node list we eventually find something:

```
[char e] [disc (f-) (f) (skip 1)] [liga ff (components f f)] [char
e]
```

So, eventually we need to kern between the character sequences [e,f-], [e,ff], [ff,e] and [f,e].

attributes

We now arrive at attributes, a new property of nodes. Before we explain a bit more what can be done with them, we show how to define a new attribute and toggle it. In the following example the `\visualizenextnodes` macro is part of ConTEXt MkIV.

```
\newattribute\aa
\newattribute\ab
\visualizenextnodes \hbox {\aa1 T{\ab3\aa2 E}X}
```

For the sake of this example, we start the allocation at 2000 because we don't want to interfere with attributes already defined in ConTEXt. The node list resulting from the box is shown at the next page. As you can see here, internally attributes become a linked list assigned to the `attr` field. This means that one has to do some work in order to inspect attributes.

```
function has_attribute(n,a)
    if n and n.attr then
        n = n.attr.next
        while n do
            if n.number == a then
                return n.value
            end
            n = n.next
        end
    else
        return false
    end
end
```

The previous function can be used in tests like:

```
local total = 0
while n do
```

```
t={
 ["attr"]={
  ["next"]={
   ["id"]="attribute",
   ["next"]={
    ["id"]="attribute",
    ["number"]=158,
    ["value"]=44,
   },
   ["number"]=0,
   ["value"]=0,
  },
 },
 ["depth"]=918,
 ["dir"]="TLT",
 ["glue_order"]=0,
 ["glue_set"]=0,
 ["glue_sign"]=0,
 ["head"]={
  ["attr"]={
   ["next"]={
    ["id"]="attribute",
    ["next"]={
     ["id"]="attribute",
     ["number"]=158,
     ["value"]=44,
    },
    ["number"]=0,
    ["value"]=0,
   },
  },
  ["char"]="U+00054",
  ["depth"]=918,
  ["expansion_factor"]=0,
  ["font"]=31,
  ["height"]=312869,
  ["id"]="glyph",
  ["lang"]=2,
  ["left"]=2,
  ["next"]={
   ["attr"]={
    ["next"]={
     ["id"]="attribute",
     ["next"]={
      ["id"]="attribute",
      ["number"]=158,
      ["value"]=44,
     },
     ["number"]=0,
     ["value"]=0,
    },
   },
   ["char"]="U+00045",
   ["depth"]=918,
   ["expansion_factor"]=0,
   ["font"]=31,
   ["height"]=312869,
   ["id"]="glyph",
   ["lang"]=2,
   ["left"]=2,
   ["next"]={
    ["attr"]={
     ["next"]={
      ["id"]="attribute",
      ["next"]={
       ["id"]="attribute",
       ["number"]=158,
       ["value"]=44,
      },
      ["number"]=0,
      ["value"]=0,
     },
    },
    ["char"]="U+00058",
    ["depth"]=918,
    ["expansion_factor"]=0,
    ["font"]=31,
    ["height"]=312869,
    ["id"]="glyph",
    ["lang"]=2,
    ["left"]=2,
    ["prev"]="<node>",
    ["right"]=3,
    ["uchyph"]=1,
    ["width"]=271122,
    ["xoffset"]=0,
    ["yoffset"]=0,
   },
   ["prev"]="<node>",
   ["right"]=3,
   ["uchyph"]=1,
   ["width"]=221577,
   ["xoffset"]=0,
   ["yoffset"]=0,
  },
  ["right"]=3,
  ["uchyph"]=1,
  ["width"]=242221,
  ["xoffset"]=0,
  ["yoffset"]=0,
 },
 ["height"]=312869,
 ["id"]="hlist",
 ["shift"]=0,
 ["subtype"]="box",
 ["width"]=734920,
}
```

Figure IX.1 `\hbox{\aa 1 T{\ab 3\aa 2 E}X \ab 4}`

```
    if has_attribute(n,2000) then
        total = total + 1
    end
    n = n.next
end
texio.write_nl(string.format(
    "attribute 2000 has been seen % times", total
))
```

When implementing nodes and attributes we did rather extensive tests and one of the test documents implemented some preliminary color mechanism based on attributes. When handling the colors the previous function was called some 300.000 times and the total node processing time (which also involved font handling) was some 2.9 seconds. Implementing this function as a helper brought down node processing time to 2.4 seconds. Of course the gain depends on the complexity of the list (nesting) and the number of attributes that are set (upto 5 per node in this test). A few more helper functions are available, some are for convenience, some gain us some speed.

The nice thing about attributes is that they obey grouping. This means that in the following sequence:

```
x {\aa1 x \ab2 x} x
```

the attributes are assigned like:

```
x x(201=1) x(201=1,202=2) x
```

Internally LuaTeX does some optimizations with respect to assigning a sequence of similar attributes, but you should keep in mind that in practice the memory usage will be larger when using many attributes.

We played with color and other properties, hyphenation based on word lists (and tracking languages with attributes) and or special algorithms (url hyphenation), spell checking (marking words as being spelled wrongly), and a few more things. This involved handling attributes in several callbacks resulting in the insertion or deletion of nodes.

When using attributes for color support, we have to insert `pdfliteral` whatsit nodes at some point depending on the current color. This also means that the time spent with color support at the TeX end will be compensated by time spent at the Lua side. It also means that because housekeeping to do with colors spanning pages and columns is gone because from now on color information travels with the nodes. This saves quite some ugly code.

Because most of the things that we want to do with attributes (and we have quite an agenda) are already nicely isolated in ConTeXt, attributes will find their way rather soon in ConTEXt MkIV.

Let's end with an observation. Attributes themselves are not something revolutionary. However, if you had to deal with them in TEX, i.e. associate them with for instance actions in during shipout, quite some time would have been spent on getting things right. Even worse: it would have lead to never ending discussions in the TEX community and as such it's no surprise that something like this never showed up. The fact that we can use Lua and manipulate node lists in many ways frees us from much discussion.

We are even considering in future versions of LuaTEX to turn font, language and direction related information into attributes (in some private range) so this story is far from finished. As a teaser, consider the following line of thinking.

Currently when a character enters the machinery, it becomes a glyph node. Among other characteristics, this node contains information about the font and the slot in that font which is used to represent that character. In a similar fashion, a space becomes glue with a measure probably related to the current font.

However, with access to nodes and attributes, you can imagine the following scenario. Instead of a font (internally represented by a font id), you use an attribute referring to a font. At that time, the font field us just pointing to TEX's null font. In a pass over the node list, you resolve the character and their attributes to a fonts and (maybe) other characters. Spacing can be postponed as well and instead of real glue values we can use multipliers and again attributes point the way to resolve them.

Of course the question is if this is worth the trouble. After all typesetting is about fonts and there is no real reason not to give them a special place.

X Dirty tricks

If you ever laid your hands on the TEXbook, the words ‘dirty tricks’ will forever be associated with an appendix of that book. There is no doubt that you need to know a bit of the internals of TEX in order to master this kind of trickyness.

In this chaper I will show a few dirty LUATEX tricks. It also gives an impression of what kind of discussions Taco and I had when discussing what kind of support should be build in the interface.

afterlua

When we look at LUA from the TEX end, we can do things like:

```
\def\test#1{%
    \setbox0=\hbox{\directlua0{tex.sprint(math.pi*#1)}}%
    pi: \the\wd0\space\the\ht0\space\the\dp0\par
}
```

But what if we are at the LUA end and want to let TEX handle things? Imagine the following call:

```
\setbox0\hbox{} \dimen0=0pt \ctxlua {
    tex.sprint("\string\\setbox0=\string\\hbox{123}")
    tex.sprint("\string\\the\string\\wd0")
}
```

This gives: 16.31999pt. This may give you the impression that TEX kicks in immediately, but the following example demonstrates otherwise:

```
\setbox0\hbox{} \dimen0=0pt \ctxlua {
    tex.sprint("\string\\setbox0=\string\\hbox{123}")
    tex.dimen[0] = tex.box[0].width
    tex.sprint("\string\\the\string\\dimen0")
}
```

This gives: 0.0pt. When still in LUA, we never get to see the width of the box.

A way out of this is the following rather straightforward approach:

```
function test(n)
    function follow_up()
        tex.sprint(tex.box[0].width)
    end
    tex.sprint("\\setbox0=\\hbox{123}\\directlua 0 {follow_up()}")
end
```

We can provide a more convenient solution for this:

```
after_lua = { } -- could also be done with closures

function the_afterlua(...)
    for _, fun in ipairs(after_lua) do
        fun(...)
    end
    after_lua = { }
end

function afterlua(f)
    after_lua[#after_lua+1] = f
end

function theafterlua(...)
    tex.sprint("\\directlua 0 {the_afterlua("
        .. table.concat({...},',') .. ")}")
end
```

If you look closely, you will see that we can (optionally) pass arguments to the function `theafterlua`. Usage now becomes:

```
function test(n)
    afterlua(function(...)
        tex.sprint(string.format("pi: %s %s %s\\par",...     ))
    end)
    afterlua(function(wd,ht,dp)
        tex.sprint(string.format("ip: %s %s %s\\par",dp,ht,wd))
    end)
    tex.sprint(string.format("\\setbox0=\\hbox{%s}",math.pi*n))
    local box_0 = tex.box[0]
    theafterlua(box_0.width,box_0.height,box_0.depth)
end
```

The last call may confuse you but since it does a print to TeX, it is in fact a delayed action. A cleaner implementation is the following:

```
local delayed = { }

local function flushdelayed(...)
```

```
    delayed = { }
    for i=1, #t do
        t[i](...)
    end
end

function lua.delay(f)
    delayed[#delayed+1] = f
end

function lua.flush(...)
    tex.sprint("\\directlua{flushdelayed(" ..
        table.concat({...},',') .. ")}")
end
```

Usage is similar:

```
function test(n)
    lua.delay(function(...)
        tex.sprint(string.format("pi: %s %s %s\\par",...))
    end)
    tex.sprint(string.format("\\setbox0=\\hbox{%s}",math.pi*n))
    local box_0 = tex.box[0]
    lua.flush(box_0.width,box_0.height,box_0.depth)
end
```

XI Going beta

introduction

We're closing in on the day that we will go beta with LuaTEX (end of July 2007). By now we have a rather good picture of its potential and to what extend LuaTEX will solve some of our persistent problems. Let's first summarize our reasons for and objectives with LuaTEX.

- The world has moved from 8 bits to 32 bits and more, and this is quite noticeable in the arena of fonts. Although Type1 fonts could host more than 256 glyphs, the associated technology was limited to 256. The advent of OpenType fonts will make it easier to support multiple languages at the same time without the need to switch fonts at awkward times.

- At the same time Unicode is replacing 8 bit based encoding vectors and code pages (input regimes). The most popular and rather efficient utf8 encoding has become a de factor standard in document encoding and interchange.

- Although we can do real neat tricks with TEX, given some nasty programming, we are touching the limits of its possibilities. In order for it to survive we need to extend the engine but not at the cost of base compatibility.

- Coding solutions in a macro language is fine, but sometimes you long to a more procedural approach. Manipulating text, handling io, interfacing . . . the technology moves on and we need to move along too.

Hence LuaTEX: a merge of the mainstream traditional TEX engines, stripped from broken or incomplete features and opened up to an embedded Lua scripting engine.

We will describe the impact of this new engine by starting from its core components reflected in the specific Lua interface libraries. Missing here is embedded support for MetaPost, because it's not yet there (apart from the fact that we use Lua to convert MetaPost graphics into TEX). Also missing is the interfacing to the pdf backend, which is also on the agenda for later. Special extensions, for instance those dealing with runtime statistics are also not discussed. Since we use ConTEXt as testbed, we will refer to the LuaTEX aware version of this macro package, MkIV, but most conclusions are rather generic.

tex internals

In order to manipulate TEX's data structures, we need access to all those registers. Already early in the development, dimension and counters were accessible and when token and node interfaces were implemented, those registers also were interfaced.

Those who read the previous chapters will have noticed that we hardly discussed this option. The reason is that we didn't yet needed that access much in order to implement font support and list processing. After all, most of the data that we need to access and manipulate is not in the registers at all. Information meant for LUA can be stored in LUA data structures. In fact, the basic call

```
\directlua 0 {some lua code}
```

has shown to be a pretty good starting point and the fact that one can print back to the TEX engine overcomes the need to store results in shared variables.

```
\def\valueofpi{\directlua0{tex.sprint(math.pi()}}
```

The number of such direct calls is not that large anyway. More often a call to LUA will be initiated by a callback, i.e. a hook into the TEX machinery.

What will be the impact of access on CONTEXT MKIV? This is yet hard to tell. In a later stage of the development, when parts of the TEX machinery will be rewritten in order to get rid of the current global nature of many variables, we will gain more control and access to TEX's internals. Core functionality will be isolated, can be extended and/or overloaded and at that moment access to internals is much more needed. But certainly that will be beyond the current registers and variables.

callbacks

These are the spine of LUATEX: here both worlds communicate with each other. A callback is a place in the TEX kernel where some information is passed to LUA and some result is returned that is then used along the road. The reference manual mentions them all and we will not repeat them here. Interesting is that in MKIV most of them are used and for tasks that are rather natural to their place and function.

```
callback.register("tex_wants_to_do_this",
    function but_use_lua_to_do_it_instead(a,b,c)
        -- do whatever you like with a, b and c
        return a, b, c
    end
)
```

The impact of callbacks on MKIV is big. It provides us a way to solve persistent problems or reimplement existing solutions in more convenient ways. Because we tested realistic functionality on real (moderately complex) documents using a pretty large macro package, we can safely conclude that callbacks are quite efficient. Stepwise LUA kicks in in order to:

- influence the input medium so that it provides a sequence of UTF characters
- manipulate the stream of characters that will be turned into a list of tokens
- convert the list of tokens into another list of tokens
- enhance the list of nodes that will be turned into a typeset paragraph
- tweak the mechanisms that come into play when lines are constructed
- finalize the result that will end up in the output medium

Interesting is that manipulating tokens is less useful than it may look at first sight. This has to do with the fact that it's (mostly) an expanded stream and at that time we've lost some information or need to do quite some coding in order to analyze the information and act upon it.

Will ConTEXt users see any of this? Chances are small that they will, although we will provide hooks so that they can add special code themselves. Users activating a callback has some danger, since it may overload already existing functionality. Chaining functionality in a callback also has drawbacks, if only that one may be confronted with already processed results and/or may destroy this result in unpredictable ways. So, as with most low level TEX features, ConTEXt users will work with more abstract interfaces.

in- and output

In MkIV we will no longer use the KPSE library directly. Instead we use a reimplementation in Lua that not only is more efficient, but also more powerful: it can read from ZIP files, use protocols, be more clever in searching, reencodes the input streams when needed, etc. The impact on MkIV is large. Most TEX code that deals with input reencoding has gone away and is replaced by Lua code.

Although it is not directly related with reading from the input medium, in that stage we also replaced verbatim handling code. Such (often messy) catcode related situations are now handled more flexible, thanks to fast catcode table switching (a new LuaTEX feature) and features like syntax highlighting can be made more neat.

Buffers, a quite old but frequently used feature of ConTEXt, are now kept in memory instead of files. This speeds up runs. Auxiliary data, aka multi–pass information, will no longer be stored in TEX files but in Lua files. In ConTEXt we have one such auxiliary file and in MkII this file is selectively filtered, but in MkIV we will be less careful with memory and load all that data once. Such speed improvements compensate the fact that LuaTEX is somewhat slower than it's ancestor PDFTEX. (Actually, the fact that LuaTEX is a bit slower that PDFTEX is mostly due to the fact that it has Aleph code on board.)

Users often wonder why there are so many temporary files, but these mostly relate to MetaPost support. These will go away once we have MetaPost as a library.

In a similar way support for XML will be enriched. We already have experimental loaders, filters and other code, and integration is on the agenda. Since ConTEXt uses XML for some sub systems, this may have some impact.

Other IO related improvements involve debugging, error handling and logging. We can pop up helpers and debug screens (MkIV can produce XHTML output and then launch a browser). Users can choose more verbose logging of IO and ask for log data to be formatted in XML. These parts need some additional work, because in the end we will also reimplement and extend TEX's error handling.

Another consequence of this will be that we will be able to package TEX more conveniently. We can put all the files that are needed into a ZIP file so that we only need to ship that ZIP file and a binary.

font readers

Handling OpenType involves more that just loading yet another font format. Of course loading an OpenType file is a necessity but we need to do more. Such fonts come with features. Features can involve replacing one representation of a character by another one of combining sequences into other sequences and finaly resolving them to one or more glyphs.

Given the numerous options we will have to spend quite some time on extending ConTEXt with new features. Instead of defining more and more font instances (the traditional TEX way of doing things) we will will provides feature switching. In the end this will make the often confusing font mechanisms less complex for the user to understand. Instead of for instance loading an extra font (set) that provides old style numerals, we will decouple this completely from fonts and provide it as yet another property of a piece of text. The good news is that much of the most important machinery is alresady in place (ligature building and such). Here we also have to decide what we let TEX do and what we do by processing node lists. For instance kerning and ligature building can either be done by TEX or by Lua. Given the fact that TEX does some juggling with character kerning while determining hyphenation points, we can as well disable TEX's kerning and let Lua handle it. Thereby TEX only has to deal with paragraph building. (After all, we need to leave TEX some core functionality to deal with.)

Another everlasting burden on macro writers and users is dealing with character representations missing from a font. Of course, since we use named glyphs in ConTEXt MkII already much of this can be hidden, but in MkIV we can create virtual fonts on the fly and keep thinking in terms of characters and glyphs instead of dealing with boxes and other structures that don't go well with for instance hyphenating words.

This brings us to hyphenation, historically bound to fonts in traditional TEX. This dependency will go away. In MkII we already ship UTF8 based patterns fore some time and these can be conveniently used in MkIV too. We experimented with using hyphenated word lists and this looks promising. You may expect more advanced ways of dealing with words, hyphenation and paragraph building in the near future. When we presented the first version of LuaTeX a few years ago, we only had the basic `\directlua` call available and could do a bit of string manipulation on the input. A fancy demo was to color wrongly spelled words. Now we can do that more robustly on the node lists.

Loading and preparing fonts for usage in LuaTeX or actually MkIV because this depends on the macro package takes some runtime. For this reason we introduces caching into MkIV: data that is used frequently is written to a cache and converted to Lua bytecode. Loading the converted files is incredibly fast. Of course there is aprice to pay: disk space, but that comes cheap these days. Also, it may as well be compensated by the fact that we can kick out many redundant files from the core TeX distributions (metric files for instance).

tokens handlers

Do we need to handle tokens? So far in experimental MkIV code we only used these hooks to demonstrate what TeX does with your characters. For a while we also constructed token lists when we wanted to inject `\pdfliteral` code in node lists, but that became obsolete when automatic string to token conversion was introduced in the node conversion code. Now we inject literal whatsit nodes. It may be worth noticing that playing with token lists gave us some good insight in bottlenecks because quite some small table allocation and garbage collections goes on.

nodes and attributes

These are the most promissing new features. In itself, nodes are not new, nor are attributes. In some sense when we use primitives like `\hbox`, `\vskip`, `\lastpenalty` the result is a node, but we can only control and inspect their properties within hard coded bounds. We cannot really look into boxes, and the last penalty may be obscured by a whatsit (a mark, a special, a write, etc.). Attributes could be fakes with marks and macro bases stacks of states. Native attributes are more powerful and each node can cary a truckload of them.

With LuaTeX, out of a sudden we can look into TeX's internals and manipulate them. Although I don't claim to be a real expert on these internals, even after over a decade of TeX programming, I'm sometimes surprised what I found there. When we are playing with these interfaces, we often run into situations where we need to add much print statements to the Lua code in order to find out what TeX is returning. It all has to do with the way TeX collects information and when it decides to act. In regular TeX much goes unnoticed, but when one has for instance a callback that deals with page building there are many places where this gets called and some of these places need special treatment.

Undoubtely this will have a huge impact on ConTEXt MkIV. Instead of parsing an input stream, we can now manipulate node lists in order to achieve (slight) inter–character spacing which is often needed in sectioning titles. The nice thing about this new approach is that we no longer have interference from characters that need multiple tokens (input characters) in order to be constructed, which complicates parsing (needed to split glyphs in MkII).

Signaling where to letterspace is done with the mentioned attributes. There can be many of them and they behave like fonts: they obey grouping, travel with the nodes and are therefore insensitive for box and page splitting. They can be set at the TEX end but needs to be handled at the Lua side. One may wonder what kind of macro packages would be around when TEX has attributes right from its start.

In MkII letterspacing is handled by parsing the input and injecting skips. Another approach would be to use a font where each character has more kerns or space around it (a virtual font can do that). But that would not only demand knowledge of what fonts need that that treatment, but also many more fonts and generating them is no fun for users. In pdfTEX there is a letterspace feature, where virtual fonts are generated on the fly, and with such an approach one has to compensate for the first and last character in a line, in order to get rid of the left- and rightmost added space (being part of the glyph). The solution where nodes are manipulated does put that burden upon the user.

Another example of node processing is adding specific kerns around some punctuation symbols, as is custom in French. You don't want to know what it takes to do that in traditional TEX, but if I mention the fact that colons become active characters you can imagine the nightmare. Hours of hacking and maybe even days of dealing with mechanisms that make these active colons workable in places where colons are used for non text are now even more wasted time if you consider that it takes a few lines of code in MkIV. Currently we let ConTEXt support both good old TEX (represented by pdfTEX), XƎTEX (a Unicode and OpenType aware variant) and LuaTEX by shared and dedicated MkII and MkIV code.

Vertical spacing can be a pain. Okay, currently MkII has a rather sophisticated way to deal with vertical spacing in ways that give documents a consistent look and feel, but every now and then we run into border cases that cannot be dealt with simply because we cannot look back in time. This is needed because TEX adds content to the main vertical list and then it's gone from our view. Take for instance section titles. We don't want them dangling at the bottom of a page. But at the same time we want itemized lists to look well, i.e. keep items together in some situations. Graphics that follow a section title pose similar problems. Adding penalties helps but these may come too late, or even worse, they may obscure previous skips which then cannot be dealt with by successive skips. To simplify the problem: take a skip of 12pt, followed by a penalty, followed by another skip of 24pt. In ConTEXt this has to become a penalty followed by one skip of 24pt.

Dealing with this in the page builder is rather easy. Ok, due to the way TEX adds content to the page stream, we need to collect, treat and flush, but currently this works all right. In ConTEXt MkIV we will have skips with three additional properties: priority over other skips, penalties, and a category (think of: ignore, force, replace, add).

When we experimented with this kind of things we quickly decided that additional experiments with grid snapping also made sense. These mechanisms are among the more complex ones on ConTEXt. A simple snap feature took a few lines of Lua code and hooking it into MkIV was not that complex either. Eventually we will reimplement all vertical spacing and grid snapping code of MkII in Lua. Because one of ConTEXt column mechanism is grid aware, we may as well adath that and/or implement an additional mechanism.

A side effect of being able to do this in LuaTEX is that the code taken from pdfTEX is cleaned up: all (recently added) static kerning code is removed (inter–character spacing, pre- and post character kerning, experimental code that can fix the heights and depths of lines, etc.). The core engine will only deal with dynamic features, like hz and protruding.

So, the impact on MkIV of nodes and attributes is pretty big! Horizontal spacing isues, vertical spacing, grid snapping are just a few of the things we will reimplement. Other things are line numbering, multiple content streams with synchronization, both are already present in MkII but we can do a better job in MkIV.

generic code

In the previous text MkIV was mentioned often, but some of the features are rather generic in nature. So, how generic can interfaces be implemented? When the MkIV code has matured, much of the Lua and glue–to–TEX code will be generic in nature. Eventually ConTEXt will become a top layer on what we internally call MetaTEX, a collection of kernel modules that one can use to build specialized macro packages. To some extent MetaTEX can be for LuaTEX what plain is for TEX. But if and how fast this will be reality depends on the amount of time that we (and other members of the ConTEXt development team) can allocate to this.

XII Zapfing fonts

remark

The actual form of the tables shown here might have changed in the meantime. However, since this document describes the stepwise development of LuaTEX *and* ConTEXt MkIV *we don't update the following information. The rendering might differ from earlier rendering simply because the code used to process this chapter evolves.*

features

In previous chapters we've seen support for OpenType features creep into LuaTEX and ConTEXt MkIV. However, it may not have been clear that so far we were just feeding the traditional TEX machinery with the right data: ligatures and kerns. Here we will show what so called features can do for you. Not much Lua code will be shown, if only because relatively complex code is needed to handle this kind of trickery with acceptable performance.

In order to support features in their full glory more is needed than TEX's ligature and kern mechanisms: we need to manipulate the node list. As a result, we have now a second mechanism built into MkIV and users can choose what method they like most. The first method, called `base`, is less powerful and less complete than the one named `node`. Eventually ConTEXt will use the node method by default.

There are two variants of features: substitutions and positioning. Here we concentrate on substitutions of which there are several. Positioning is for instance used for specialized kerning as needed in for instance typesetting Arab.

One character representation can be replaced by one or more fixed alternatives or alternatives chosen from a list of alternatives (substitutions or alternates). Multiple characters can be replaces by one character (substitutions, alternates or a ligature). The replacements can depend on preceding and/or following glyphs in which case we say that the replacement is driven by rules. Rules can deal with single glyphs, combinations of glyphs, classes (defined in the font) of glyphs and/or ranges of glyphs.

Because the available documentation of OpenType is rather minimalistic and because most fonts are relatively simple, you can imagine that figuring out how to implement support for fonts with advanced features is not entirely trivial and involves some trial and error. What also complicate things is that features can interfere. Yet another complicating factor is that in the order of applying a rule may obscure a later rule. Such fonts don't ship with manuals and examples of correct output are not part of the buy.

We like testing LuaTeX's open type support with Palatino Regular and Palatino Sans and good old Type1 support with Optima Nova. So it makes sense to test advanced features with Zapfino Pro. This font has many features, which happen to be implemented by Adam Twardoch, a well known font expert and familiar with the TeX community. We had the feeling that when LuaTeX can support Zapfino Pro, designed by Hermann Zapf and enhanced by Adam, we have reached a crucial point in the development.

The first thing that you will observe when using this font is that the files are larger than normal, especially the cached versions in MkIV. This made me extend some of the serialization code that we use for caching font data so that it could handle huge tables better but at the cost of some speed. Once we could handle the data conveniently and as a side effect look into the font data with an editor, it became clear that implementing for the `calt` and `clig` features would take a bit of coding.

example

Before some details will be discussed, we will show two of the test texts that ConTeXt users normally use when testing layouts or new features, a quote from E.R. Tufte and one from Hermann Zapf. The TeX code shows how features are set in ConTeXt.

```
\definefontfeature
   [zapfino]
   [language=nld,script=latn,mode=node,
    calt=yes,clig=yes,liga=yes,rlig=yes,tlig=yes]

\definefont
    [Zapfino]
    [ZapfinoExtraLTPro*zapfino at 24pt]
    [line=40pt]
\Zapfino
\input tufte \par
```

We thrive in information--thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, *distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.*

You don’t even have to look too closely in order to notice that characters are represented by different glyphs, depending on the context in which they appear.

```
\definefontsynonym
  [Zapfino]
  [ZapfinoExtraLTPro]
  [features=zapfino]
\definedfont
  [Zapfino at 24pt]
\setupinterlinespace
  [line=40pt]
\input zapf \par
```

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC’s tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.

obeying rules

When we were testing node based feature support, the only way to check this was to identify the rules that lead to certain glyphs. The more unique glyphs are good candidates for this. For instance

- there is s special glyph representing ℅
- in the input stream this is the character sequence `c/o`
- so there most be a rule that tells us that this sequence becomes that ligature

As said, in this case, the replacement glyph is supposed to be a ligature and indeed there is such a ligature: `c_slash_o`. Of course, this replacement will only take place when the sequence is surrounded by spaces.

However, when testing this, we were not looking at this rule but at the (randomly chosen) rule that was meant to intercept the alternative `h.2` followed by `z.4`. Interesting was that this resolved to a ligature indeed, but the shape associated with this ligature was an `h`, which is not right. Actually, a few more of such rules turned out to be wrong. It took a bit of an effort to reach this conclusion because of the mentioned interferences of features and rules. At that time, the rule entry (in raw LuaTeX table format) looks as follows:

```
[44] = {
    ["format"] = "coverage",
    ["rules"] = {
        [1] = {
            ["coverage"] = {
                ["ncovers"] = {
                    [1] = "h.2",
                    [2] = "z.4",
                }
            },
            ["lookups"] = {
                [1] = {
                    ["lookup_tag"] = "L084",
                    ["seq"] = 0,
                }
            }
        }
    }
    ["script_lang_index"] = 1,
    ["tag"] = "calt",
    ["type"] = "chainsub"
}
```

Instead of reinventing the wheel, we used the FontForge libraries for reading the OpenType font files. Therefore the LuaTeX table is resembling the internal FontForge data structures. Currently we show the version 1 format.

Here `ncovers` means that when the current character has shape h (`h.2`) and the next one is z (`z.4`) (a sequence) then we need to apply the lookup internally tagged `L084`. Such a rule can be more extensive, for instance instead of `h.2` one can have a list of characters, and there can be `bcovers` and `fcovers` as well, which means that preceding or following character need to be taken into account.

When this rule matches, it resolves to a specification like:

```
[6] = {
    ["flags"] = 0,
    ["lig"] = {
        ["char"] = "h",
        ["components"] = "h.2 z.4",
    },
    ["script_lang_index"] = 65535,
    ["tag"] = "L084",
    ["type"] = "ligature",
}
```

Here `tag` and `script_lang_index` are kind of special and are part of an private feature system, i.e. they make up the cross reference between rules and glyphs. Watch how the components don't match the character, which is even more peculiar when we realize that these are the initials of the author of the font. It took a couple of Skype sessions and mails before we came to the conclusion that this was probably a glitch in the font. So, what to do when a font has bugs like this? Should one disable the feature? That would be a pitty because a font like Zapfino depends on it. On the other hand, given the number of rules and given the fact that there are different rule sets for some languages, you can imagine that making up the rules and checking them is not trivial.

We should realize that Zapfino is an extraordinary case, because it used the OpenType features extensively. We can also be sure that the problems will be fixed once they are known, if only because Adam Twardoch (who did the job) has exceptionally high standards but it may take a while before the fix reached the user (who then has to update his or her font). As said, it also takes some effort to run into the situation described here so the likelihood of running into this rule is small. This also brings to our attention the fact that fonts can now contain bugs and updating them makes sense but can break existing documents. Since such fonts are copyrighted and not available on line, font vendors need to find ways to communicate these fixes to their customers.

Can we add some additional checks for problems like this? For a while I thought that it was possible by assuming that ligatures have names like `h.2_z.4` but alas, sequences of glyphs are mapped onto ligatures using mappings like the following:

`three fraction four.2`	`threequarters`	¾
`three fraction four`	`threequarters`	¾
`d r`	`d_r`	dr
`e period`	`e_period`	e.
`f i`	`fi`	fi
`f l`	`fl`	fl
`f f i`	`f_f_i`	ffi
`f t`	`f_t`	ft

Some ligature have no `_` in their names and there are also some inconsistencies, compare the `fl` and `f_f_i`. Here font history is painfully reflected in inconsistency and no solution can be found here.

So, in order to get rid of this problem, MkIV implements a method to ignore certain rules but then, this only makes sense if one knows how the rules are tagged internally. So, in practice this is no solution. However, you can imagine that at some point ConTeXt ships with a database of fixes that are applied to known fonts with certain version numbers.

We also found out that the font table that we used was not good enough for our purpose because the exact order in what rules have to be applies was not available. Then we noticed that in the meantime FontForge had moved on to version 2 and after consulting the author we quickly came to the conclusion that it made sense to use the updated representation.

In version 2 the snippet with the previously mentioned rule looks as follows:

```
["ks_latn_l_66_c_19"]={
 ["format"]="coverage",
 ["rules"]={
  [1]={
   ["coverage"]={
    ["current"]={
     [1]="h.2",
     [2]="z.4",
    }
   },
   ["lookups"]={
    [1]={
     ["lookup"]="ls_l_84",
     ["seq"]=0,
    }
   }
  }
 },
 ["type"]="chainsub",
},
```

The main rule table is now indexed by name which is possible because the order of rules is specified somewhere else. The key `ncovers` has been replaced by `current`. As long as LuaTeX is in beta stage, we have the freedom to change such labels as some of them are rather FontForge specific.

This rule is mentioned in a feature specification table. Here specific features are associated with languages and scripts. This is just one of the entries concerning `calt`. You can imagine that it took a while to figure out how best to deal with this, but eventually the MkIV code could do the trick. The cryptic names are replacements for pointers in the FontForge datastructure. In order to be able to use FontForge for font development and analysis, the decision was made to stick closely to its idiom.

```
["gsub"]={
 ...
 [67]={
  ["features"]={
   [1]={
    ["scripts"]={
     [1]={
      ["langs"]={
       [1]="AFK ",
       [2]="DEU ",
       [3]="NLD ",
       [4]="ROM ",
       [5]="TRK ",
       [6]="dflt",
      },
      ["script"]="latn",
     }
    },
    ["tag"]="calt",
   }
  },
  ["name"]="ks_latn_l_66",
  ["subtables"]={
   [1]={
    ["name"]="ks_latn_l_66_c_0",
   },
   ...
   [20]={
    ["name"]="ks_latn_l_66_c_19",
   },
   ...
  },
  ["type"]="gsub_context_chain",
 },
```

practice

The few snapshots of the font table probably don't make much sense if you haven't seen the whole table. Well, it certainly helps to see the whole picture, but we're talking of a 14 MB file (1.5 MB bytecode). When resolving ligatures, we can follow a straightforward approach:

- walk over the nodelist and at each character (glyph node) call a function
- this function inspects the character and takes a look at the following ones
- when a ligature is identified, the sequence of nodes is replaced

Substitutions are not much different but there we look at just one character. However, contextual substitutions (and ligatures) are more complex. Here we need to loop over a list of rules (dependent on script and language) and this involves a sequence as well as preceding and following characters. When we have a hit, the sequence will be replaced by another one, determined by a lookup in the character table. Since this is a rather time consuming operation, especially because many surrounding characters need to be taken into account, you can imagine that we need a bit of trickery to get an acceptable performance. Fortunately Lua is pretty fast when it comes down to manipulating strings and tables, so we can prepare some handy datastructures in advance.

When testing the implementation of features one need to be aware of the fact that some appearance are also implemented using the regular ligature mechanisms. Take the following definitions:

```
\definefontfeature
```

```
   [none]
   [language=dflt,script=latn,mode=node,liga=no]
\definefontfeature
   [calt]
   [language=dflt,script=latn,mode=node,liga=no,calt=yes]
\definefontfeature
   [clig]
   [language=dflt,script=latn,mode=node,liga=no,clig=yes]
\definefontfeature
   [dlig]
   [language=dflt,script=latn,mode=node,liga=no,dlig=yes]
\definefontfeature
   [liga]
   [language=dflt,script=latn,mode=node]
```

This gives:

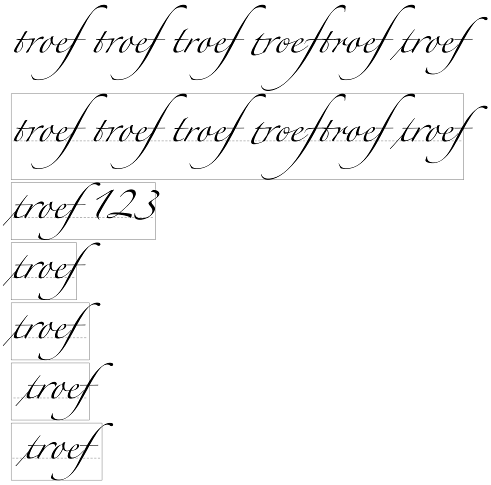
`none`	*on the synthesis*	*winnow the wheat*
`calt`	*on the synthesis*	*winnow the wheat*
`clig`	*on the synthesis*	*winnow the wheat*
`dlig`	*on the synthesis*	*winnow the wheat*
`liga`	*on the synthesis*	*winnow the wheat*

Here are Adam's recommendations with regards to the `dlig` feature: "The `dlig` feature is supposed to by use only upon user's discretion, usually on single runs, words or even pairs. It makes little sense to enable `dlig` for an entire sentence or paragraph. That's how the OpenType specification envisions it."

When testing features it helps to use words that look similar so next we will show some examples that used. When we look at these examples, we need to understand that when a specific character representation is analyzed, the rules can take preceding and following characters into account. The rules take characters as well as their shapes, or more precisely: one of their shapes since Zapfino has many variants, into account. Since different rules are used for languages (okay, this is limited to only a subset of languages that use the latin script) not only shapes but also the way words are constructed are taken into account. Designing te rules is definitely non trivial.

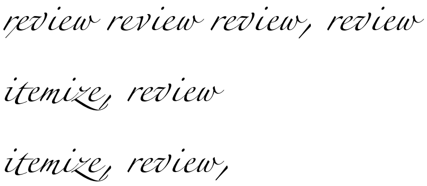
When testing the implementation we ran into cases where the initial `t` showed up wrong, for instance in the the Dutch word `troef`. Because space can be part of the rules, we need to handle the cases where words end and start and boxes are then kind of special.

```
troef troef troef troeftroef troef  \par
\ruledhbox{troef troef troef troeftroef troef} \par
\ruledhbox{troef 123} \par
\ruledhbox{troef} \ruledhbox{troef } \ruledhbox{ troef} \ruledhbox
{ troef } \par
```



Unfortunately, this does not work well with punctuation, which is less prominent in the rules than space. In our favourite test quote of Tufte, we have lots of commas and there it shows up:

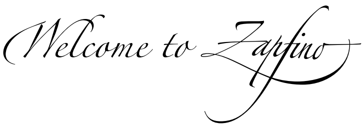
```
review review review, review \par
itemize, review \par
itemize, review, \par
```



Of course we can decide to extend the rule base at runtime and this may well happen when we experiment more with this font.

The next one was one of our first test lines, Watch the initial and the Zapfino ligature.

```
Welcome to Zapfino
```



For a while there was a bug in the rule handler that resulted in the variant of the `y` that has a very large descender. Incidentally the word `synthesize` is also a good test case for the `the` pattern which gets special treatment because there is a ligature available.

```
synopsize versus synthesize versus
synthase versus sympathy versus synonym
```

synopsize versus synthesize versus synthase versus sympathy versus synonym

Here are some examples that use the `g`, `d` and `f` in several places.

```
eggen groet ogen hagen \par
dieren druiven onder aard  donder modder \par
fiets effe flater triest troef \par
```

eggen groet ogen hagen

dieren druiven onder aard donder modder

fiets effe flater triest troef

Let’s see how well Hermann has taken care of the `h`’s representations. There are quite some variants of the lowercase one:

`h`	*h*
`h.2`	*h*
`h.3`	*h*
`h.4`	*h*
`h.5`	*h*
`h.init`	*h*
`h.sups`	*h*
`h.sc`	*H*
`orn.73`	

How about the uppercase variant, as used in his name:

```
M Mr Mr. H He Her Herm Herma Herman Hermann Z Za Zap Zapf \par
Mr. Hermann Zapf
```

M Mr Mr. H He Her Herm Herma Herman Hermann Z Za Zap Zapf

Mr. Hermann Zapf

Of course we have to test another famous name:

```
D Do Don Dona Donal Donald K Kn Knu Knut Knuth \par
Don Knuth Donald Knuth Donald E. Knuth DEK \par
Prof. Dr. Donald E. Knuth \par
```

D Do Don Dona Donal Donald K Kn Knu Knut Knuth

Don Knuth Donald Knuth Donald E. Knuth DEK

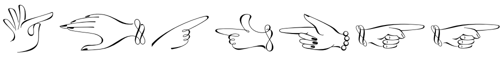
Prof. Dr. Donald E. Knuth

Unfortunately the LUA and TEX logos don't come out that well:

```
L Lu Lua l lu lua t te tex TeX luatex luaTeX LuaTeX
```

L Lu Lua l lu lua t te tex TeX luatex luaTeX LuaTeX

This font has quite some ornaments and there is an `ornm` feature that can be applied. We're still not sure about its usage, but when one keys in text in lowercase, `hermann` comes out as follows:



As said in the beginning, dirty implementation details will be kept away from the reader. Also, you should not be surprised if the current code had some bugs or does some things wrong. Also, if spacing looks a bit weird to you, keep in mind that we're still in the middle of sorting things out.

Taco Hoekwater & Hans Hagen

XIII Arabic

Let's start with admitting that I don't speak or read Arabic, and the sample texts used here are part of what we use in the Oriental TEX project for exploring advanced Arabic typesetting. This chapter will not discuss arab typesetting in much detail, but should be seen as complementing the 'Onthology on Arabic Typesetting' written by Idris. Here I will only show what the consequences are of applying features. Because we see glyphs but often still deal with characters when analyzing what to do, we will use these terms mixed.

The font that we use here is the 'arabtype' font by MicroSoft. This font covers Latin scripts and Arabic and has a rich set of features. It's also a rather big font, so it is a nice torture test for LuaTEX.

First we show what MkIV does with a sequence of characters when no features are enabled by the user. We have turn on color tracing. This gives us some feedback about the how the analyze worked out. Analyzing for Arabic boils down to marking the initial, mid, final and isolated forms. We don't need to explicitly enable analyzing, it's on by default. The `mode` flag is set to `node` because we cannot use TEX's default mechanism. When LuaTEX and MkIV are beyond beta stage, we will use that mode by default.

```
analyze=yes, devanagari=yes, dummies=yes,
extensions=yes, extrafeatures=yes, features=yes,
language=dflt, mathkerns=yes, mode=node,
script=arab, spacekern=yes
```

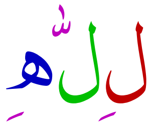
لِلّهِ

اَلْحَمْدُ لِلّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالأَرْوَاحَ وَالأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالأَرْوَاحَ وَالأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

Once these forms are identified, the `init`, `medi`, `fina` and `isol` features can be applied since they need this information. As you can see, different shapes show up. The vowels (marks in OpenType speak) are not affected. It may not be entirely clear here, but these vowels don't have width.

```
analyze=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, language=dflt, mathkerns=yes,
mode=node, script=arab, spacekern=yes
```

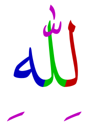


اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

We start with some preparations with regards to combinations of marks. This is really needed in order to get the right output.

```
analyze=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, fina=yes, init=yes, isol=yes,
language=dflt, mathkerns=yes, medi=yes, mode=node,
script=arab, spacekern=yes
```



اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

The order in which features are applied is dictated by the font and users don't need to bother about it. In the next example we enable the `mark` and `mkmk` features. As with other positioning related features, these are normally applied late in the feature chain.

```
analyze=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, fina=yes, init=yes, isol=yes,
language=dflt, mark=yes, mathkerns=yes, medi=yes,
mode=node, script=arab, spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

The `mark` feature positions marks (vowels) relative to characters, also known as mark to base. The `mkmk` feature positions marks to basemarks.

```
analyze=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, fina=yes, init=yes, isol=yes,
language=dflt, mark=yes, mathkerns=yes, medi=yes,
mkmk=yes, mode=node, script=arab, spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِي خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

Kerning depends on the font. Some fonts don't need kerning, others may need extensive relative positioning of characters (by now glyphs).

```
analyze=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, fina=yes, init=yes, isol=yes,
kern=yes, language=dflt, mark=yes, mathkerns=yes,
medi=yes, mkmk=yes, mode=node, script=arab,
spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ
اٰلَائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ،
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ
اٰلَائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ،
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

So far we only had rather straightforward replacements. More sophisticated replacements are those driven by the context. In principle all replacements can be context driven, but the `calt` and `clig` features are normally dedicated to the real complex ones that take preceding and following characters into account.

```
analyze=yes, calt=yes, ccmp=yes, devanagari=yes,
dummies=yes, extensions=yes, extrafeatures=yes,
features=yes, fina=yes, init=yes, isol=yes,
kern=yes, language=dflt, mark=yes, mathkerns=yes,
medi=yes, mkmk=yes, mode=node, script=arab,
spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ
اٰلَائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ،
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ
اٰلَائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ،

وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

Ligatures are often used to beautify Arabic typeset documents. Here we enable the whole lot.

```
analyze=yes, ccmp=yes, clig=yes, devanagari=yes,
dlig=yes, dummies=yes, extensions=yes,
extrafeatures=yes, features=yes, fina=yes,
init=yes, isol=yes, kern=yes, language=dflt,
liga=yes, mark=yes, mathkerns=yes, medi=yes,
mkmk=yes, mode=node, rlig=yes, script=arab,
spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٍ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٍ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

Kerning deals with horizontal displacements, but curs (cursive) goes one step further. As with marks, positioning is based on anchor points and resolving them involves a bit of trickery because one needs to take into account that characters may have vowels attached to them.

```
analyze=yes, ccmp=yes, clig=yes, curs=yes,
devanagari=yes, dlig=yes, dummies=yes,
extensions=yes, extrafeatures=yes, features=yes,
fina=yes, init=yes, isol=yes, kern=yes,
language=dflt, liga=yes, mark=yes, mathkerns=yes,
medi=yes, mkmk=yes, mode=node, rlig=yes,
script=arab, spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٍ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ،

وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

One script can serve multiple languages so let's see what happens when we switch to Urdu.

`analyze=yes, ccmp=yes, clig=yes, curs=yes,`
`devanagari=yes, dlig=yes, dummies=yes,`
`extensions=yes, extrafeatures=yes, features=yes,`
`fina=yes, init=yes, isol=yes, kern=yes,`
`language=urd, liga=yes, mark=yes, mathkerns=yes,`
`medi=yes, mkmk=yes, mode=node, rlig=yes,`
`script=arab, spacekern=yes`
	لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

In practice one will enable most of the features. In MkIV one can define feature sets as follows:

```
\definefontfeature
  [arab-default]
  [mode=node,language=dflt,script=arab,
   init=yes,medi=yes,fina=yes,isol=yes,
   liga=yes,dlig=yes,rlig=yes,clig=yes,
   mark=yes,mkmk=yes,kern=yes,curs=yes]
```

Applying these features to fonts can be done in several ways, with as most basic one:

```
\font\ArabFont=arabtype*arab-default at 18pt
```

Normally one will do something like

```
\definefont[ArabFont][arabtype*arab-default at 18pt]
```

or use typescripts to set up ap proper font collection, in which case we end up with definitions that look like:

```
\definefontsynonym[ArabType][name:arabtype][features=arab-default]
\definefontsynonym[Serif][ArabType]
```

More information about typescripts can be found in manuals and on the ConTeXt wiki.

We end this chapter with showing two arabic fonts so that you can get a taste if the differences: arabtype by MicroSoft and Palatino which is designed by Herman Zapf for Linotype.

analyze=yes, ccmp=yes, clig=yes, curs=yes,
devanagari=yes, dlig=yes, dummies=yes,
extensions=yes, extrafeatures=yes, features=yes,
fina=yes, init=yes, isol=yes, kern=yes,
language=dflt, liga=yes, mark=yes, mathkerns=yes,
medi=yes, mkmk=yes, mode=node, rlig=yes,
script=arab, spacekern=yes
	لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفِ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفِ بِحَمْدِهِ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ آلَائِهِ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

```
analyze=yes, ccmp=yes, clig=yes, curs=yes,
devanagari=yes, dlig=yes, dummies=yes,
extensions=yes, extrafeatures=yes, features=yes,
fina=yes, init=yes, isol=yes, kern=yes,
language=dflt, liga=yes, mark=yes, mathkerns=yes,
medi=yes, mkmk=yes, mode=node, rlig=yes,
script=arab, spacekern=yes
```

لِلّٰهِ

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

اَلْحَمْدُ لِلّٰهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهٖ، مُغْتَرِفٌ مِنْ بِحَارِ مَجْدِهٖ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اٰلائِهٖ نَاشِرًا؛ اَلَّذِيْ خَلَقَ الْمَوْتَ وَالْحَيٰوةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُوْنَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

These fonts are quite different in designsize:

	arabtype	**palatino**
10pt	test	test
12pt	test	test
18pt	test	test
24pt	test	test

XIV Colors redone

introduction

Color support has been present in ConTeXt right from the start and support has been gradualy extended, for instance with transparency and spot colors. About 10 years later we have the first major rewrite of this mechanism using attributes as implemented in LuaTeX.

Because I needed a test file to check if all things still work as expected, I decided to recap the most important commands in this chapter.

color support

The core command is `\definecolor`, so let's define a few colors:

```
\definecolor [red]     [r=1]
\definecolor [green]   [g=1]
\definecolor [blue]    [b=1]
\definecolor [yellow]  [y=1]
\definecolor [magenta] [m=1]
\definecolor [cyan]    [c=1]
```

This gives us the following colors:

color	name	transparency	specification
white black	red		r=1.000,g=0.000,b=0.000
white black	green		r=0.000,g=1.000,b=0.000
white black	blue		r=0.000,g=0.000,b=1.000
white			
white black	yellow		c=0.000,m=0.000,y=1.000,k=0.000
white black	magenta		c=0.000,m=1.000,y=0.000,k=0.000
white black	cyan		c=1.000,m=0.000,y=0.000,k=0.000

As you can see in this table, transparency is part of a color specification, so let's define a few transparent colors:

```
\definecolor [t-red]   [r=1,a=1,t=.5]
\definecolor [t-green] [g=1,a=1,t=.5]
\definecolor [t-blue]  [b=1,a=1,t=.5]
```

color	name	transparency	specification
white black	t-red	a=1.000,t=0.500	r=1.000,g=0.000,b=0.000

white black	t-green	a=1.000,t=0.500	r=0.000,g=1.000,b=0.000
white black	t-blue	a=1.000,t=0.500	r=0.000,g=0.000,b=1.000

Because transparency is now separated from color, we can define transparent behaviour as follows:

```
\definecolor[half-transparent] [a=1,t=.5]
```

Implementing process color spaces was not that complex, but spot and multitone colors took a bit more code.

```
\definecolor      [parentspot]               [r=.5,g=.2,b=.8]
\definespotcolor [childspot-1] [parentspot] [p=.7]
\definespotcolor [childspot-2] [parentspot] [p=.4]
```

The three colors, two of them are spot colors, show up as follows:

color	name	transparency	specification
white black	parentspot		r=0.500,g=0.200,b=0.800
white black	childspot-1		p=0.700
white black	childspot-2		p=0.400

Multitone colors can also be defined:

```
\definespotcolor [spotone]   [red]   [p=1]
\definespotcolor [spottwo]   [green] [p=1]

\definespotcolor [spotone-t] [red]   [a=1,t=.5]
\definespotcolor [spottwo-t] [green] [a=1,t=.5]

\definemultitonecolor
    [whatever]
    [spotone=.5,spottwo=.5]
    [b=.5]
\definemultitonecolor
    [whatever-t]
    [spotone=.5,spottwo=.5]
    [b=.5]
    [a=1,t=.5]
```

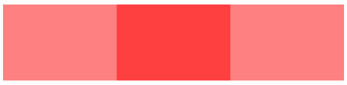
Transparencies don’t carry over:

color	name	transparency	specification
white black	spotone		p=1.000

white black	spottwo		p=1.000
white black	spotone-t	a=1.000,t=0.500	p=1.000
white black	spottwo-t	a=1.000,t=0.500	p=1.000
white black	whatever		p=.5,.5
white black	whatever-t	a=1.000,t=0.500	p=.5,.5

Transparencies combine as follows:

```
\blackrule[width=3cm,height=1cm,color=spotone-t]\hskip-1.5cm
\blackrule[width=3cm,height=1cm,color=spotone-t]
```



We can still clone colors and overload color dynamically. I used the following test code for the MkIV code:

```
{\green green->red}
\definecolor[green] [g=1]
{\green green->green}
\definecolor[green] [blue]
{\green green->blue}
\definecolor[blue] [red]
{\green green->red}
\setupcolors[expansion=yes]%
\definecolor[blue] [red]
\definecolor[green] [blue]
\definecolor[blue] [r=1]
{\green green->blue}
```

green->red green->green green->blue green->red green->blue

Of course palets and color groups are supported too. We seldom use colorgroups, but here is an example:

```
\definecolorgroup
  [redish]
  [1.00:0.90:0.90,1.00:0.80:0.80,1.00:0.70:0.70,1.00:0.55:0.55,
   1.00:0.40:0.40,1.00:0.25:0.25,1.00:0.15:0.15,0.90:0.00:0.00]
```

The redish color is called by number:

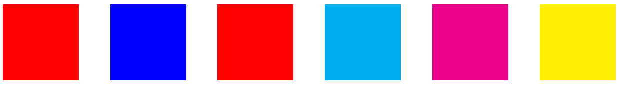
```
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:1]\quad
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:2]\quad
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:3]
```

Palets work with names:

```
\definepalet
  [complement]
  [red=cyan,green=magenta,blue=yellow]
```

This is used as:

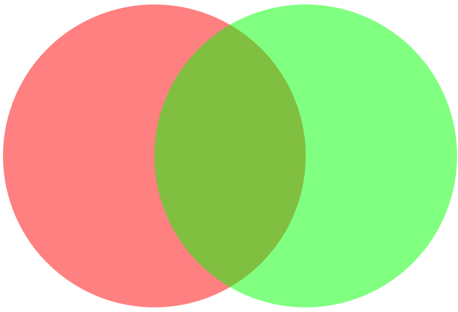
```
\blackrule[width=1cm,height=1cm,depth=0pt,color=red]\quad
\blackrule[width=1cm,height=1cm,depth=0pt,color=green]\quad
\blackrule[width=1cm,height=1cm,depth=0pt,color=blue]\quad
\setuppalet[complement]%
\blackrule[width=1cm,height=1cm,depth=0pt,color=red]\quad
\blackrule[width=1cm,height=1cm,depth=0pt,color=green]\quad
\blackrule[width=1cm,height=1cm,depth=0pt,color=blue]
```



Of course the real torture test is MetaPost inclusion:

```
\startMPcode
    path p ; p := fullcircle scaled 4cm ;
    fill p                     withcolor \MPcolor{spotone-t} ;
    fill p shifted(2cm,0cm) withcolor \MPcolor{spottwo-t} ;
\stopMPcode
```

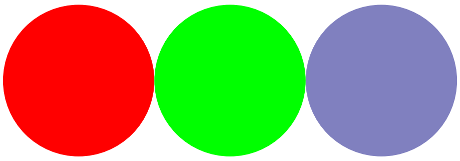
These transparent color circles up as:



Multitone colors also work:

```
\startMPcode
    path p ; p := fullcircle scaled 2cm ;
    fill p                     withcolor \MPcolor{spotone} ;
    fill p shifted(2cm,0cm) withcolor \MPcolor{spottwo} ;
    fill p shifted(4cm,0cm) withcolor \MPcolor{whatever} ;
\stopMPcode
```

This gives:



implementation

The implementation of colors using attributes if quite different from the traditional method. In MkII color support works okay but the associated code is not that clean, if only because:

- we need to keep track of grouped color usage
- and we do that using dedicated marks (using TEX's mark mechanism)
- since this has limitations, we have quite some optimizations
- like local (no marks) and global colors (marks)
- and real dirty code to push and pop color states around pages
- and some messy code to deal with document colors
- and quite some conversion macros (think of TEX not having floats)

Although recent versions of PDFTEX have a color stack mechanism, this is not adequate for our usage, if only because we support more colorspaces than this mechanism is supposed to deal with. (The color stack mechanism is written with a particular macro packag ein mind.)

In MkIV attributes behave like colors and therefore we no longer need to care about what happens at pageboundaries. Also, we no longer have to deal with the limitations of marks. Here:

- we have distributed color spaces, color itself and transparency
- all injection of backend code is postponed to shipout time
- definition and conversion is delegated to LUA

Of course the current implementation is not as nice as we would like it to be. This because:

- support mechanism are under construction
- we need to support both MkII and MkIV in one interface
- backend support is yet limited

Although in principle a mechanism based on attributes is much faster than using marks cum suis, the new implementation is slower. The main reason is that we need to finalize the to be shipped out box. However, since this task involved more than just color, we will gain back some runtime when other mechanisms also use attributes.

complications

This paragraph is somewhat complex, so skip it when you don't feel comfortable with the subject of when you've never seen low level ConTEXt code.

Attributes behave like fonts. This means that they are kind of frozen once material is boxed. Consider that we define a box as follows:

```
\setbox0\hbox{default {\red red \green green} default}
```

What do you expect to come out the next code? In MkII the 'default' inside the box will be colored yellow but the internal red and and green words will keep their color.

```
default {\yellow yellow \box0\ yellow} default
```

When we use fonts switches we don't expect the content of the box to change. So, in the following the 'default' texts will *not* become bold.

```
\setbox0\hbox{default {\sl slanted \bi bold italic} default}
default {\bf bold \box0\ bold} default
```

Future versions of LuaTEX will provide more control over how attributes are applied to boxes, but for the moment we need to fallback on a solution built in MkIV:

```
default {\yellow yellow \attributedbox0\ yellow} default
```

There is also a `\attributedcopy` macro. These macros signal the attribute resolver (that kicks in just before shipout) that this box is to be treated special.

In MkII we had a similar situation which is why we had the option (only used deep down in ConTEXt) to encapsulate a bunch of code with

```
\startregistercolor[foregroundcolor]
some macro code ... here foregroundcolor is applied ... more code
\stopregisteringcode
```

This is for instance used in the `\framed` macro. First we package the content, foregroundcolor is not yet applied because the injected specials of literals can interfere badly, but by registering the colors the nested color calls are tricked into thinking that preceding and following content is colored. When packaged, we apply backgrounds, frames, and foregroundcolor to the whole result. Because nested colors were aware of the foregroundcolor they have properly reverted to this color when needed.

In MkIV the situation is reversed. Here we definitely need to set the foregroundcolor because otherwise attributes are not set and here they don't interfere at all (no extra nodes). For this we use the same registration macros. When the lot is packaged, applying foregroundcolor is ineffective because the attributes are already applied. Instead of registering we could have flushed the framed content using `\attributedbox`, but this way we can keep the MkII and MkIV code base the same.

To summarize, first the naïve approach. Here the nested colors know how to revert, but the color switch can interfere with the content (since color commands inject nodes).

```
\setbox\framed\vbox
  {\color[foregroundcolor]{packaged framed content, can have color
switches}}
```

The MkII approach registers the foreground color so the nested colors know what to do. There is no interfering code:

```
\startregistercolor[foregroundcolor]
\setbox\framed
\stopregisteringcode
\setbox\framed{\color[foregroundcolor]{\box\framed}}
```

The registration actually sets the color, so in fact the final coloring is not needed (does nothing). An alternative MkIV approach is the following:

```
\color
  [foregroundcolor]
  {\setbox\framed{packaged framed content, can have color switches}}
```

This works ok because attributes are applied to the whole content, i.e. the box. In MkII this would be quite ineffective and actually result in weird side effects.

```
< color stack is pushed and marks are set (unless local) >
< color special or literal sets color to foregroundcolor >
\setbox\framed{packaged framed content, can have color switches}
< color special or literal sets color to foregroundcolor >
< color stack is popped and marks are set (unless local) >
```

So, effectively we set a box, and end up with:

```
< whatsits (special, literal and.or mark) >
< whatsits (special, literal and.or mark) >
```

in the main vertical lost and that will interfere badly with spacing and friends.

In MkIV however, a color switch, like a font switch does not leave any traces, it just sets a state. Anyway, keep in mind that there are some rather fundamental conceptual differences between the two appoaches.

Let's end with an example that demonstrates the problem. We fill two boxes:

```
\setbox0\hbox{RED {\blue blue} RED}
\setbox2\hbox{RED {\blue blue} {\attributedcopy0} RED}
```

We will flush these in the following lines:

```
{unset \color[red]{red \CopyMe} unset
    \color[red]{red \hbox{red \CopyMe}} unset}
{unset \color[red]{red \CopyMe} unset
    {\red red \hbox{red \CopyMe}} unset}
{unset \color[red]{red \CopyMe} unset
    {\red red \setbox0\hbox{red \CopyMe}\box0} unset}
{unset \color[red]{red \CopyMe} unset
    {\hbox{\red red \CopyMe}} unset}
{\blue blue \color[red]{red \CopyMe} blue
    \color[red]{red \hbox{red \CopyMe}} blue}
```

First we define `\CopyMe` as follows:

```
\def\CopyMe{\attributedcopy2\ \copy4}
```

This gives:

unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset unset red RED blue RED blue RED RED unset red RED blue RED blue RED RED unset blue red RED blue RED blue RED RED blue red red RED blue RED blue RED RED blue

Compare this with:

```
\def\CopyMe{\copy2\ \copy4}
```

This gives:

unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset

unset red RED blue RED blue RED RED unset red red RED blue RED blue RED RED unset unset red RED blue RED blue RED RED unset red RED blue RED blue RED RED unset blue red RED blue RED blue RED RED blue red red RED blue RED blue RED RED blue

You get the picture? At least in early version of MkIV you need to enable support for inheritance with:

```
\enableattributeinheritance
```

XV Chinese, Japanese and Korean, aka CJK

This aspect of MkIV *is under construction. We use non-realistic examples. We need to reimplement chinese numbering in* Lua*, etc. etc.*

todo: There is no need for checkinf the width if the halfwidth feature is turned on.

introduction

In ConTeXt MkII we support cjk languages. Intercharacter spacing as well as linebreaks are taken care of. Chinese numbering is dealt with and labels and other language specific aspects are supported too. The implementation uses active characters and some special encoding subsystem. Although it works quite okay, in MkIV we follow a different route.

The current implementation is an intermediate one and is used to explore the possibilities and identify needs. One handicap in implementing cjk support is that the wishlist of features and behaviour is somewhat dependent on who you talk to. This means that the implementation will have some default behaviour but can be tuned to specific needs. The current implementation uses the script related analyser and is triggered by fonts but at some point I may decide to provide analysing independent of fonts.

As will all things TeX, we need to find a proper font to get our document typeset and because cjk fonts are normally quite large they are not always available on your system by default.

scripts and languages

I'm no expert on cjk and will never be one so don't expect much insight in the scripts and languages here. Here we only look at the way a sequence of characters in the input turns into a typeset paragraph. For that it is important to keep in mind that in a Korean or Japanese text we might find Chinese characters and that the spacing rules become somewhat fuzzed by that. For instance Korean has spaces between words and words can be broken at any point, while Chinese has no spaces.

Officially Chinese runs from top to bottom but here we focus on the horizontal variant. When turned into glyphs the characters normally are of equal width and in principle we could expect them all to be vertically aligned. However, a font can have characters that take half that space: so called halfwidth characters. And, of course, in practice a font might have shapes that fall into this categrory but happen to have their own width which deviates from this.

This means that a mechanism that deals with cjk has to take care of a few things:

- Spaces at the end of the line (or actually anywhere in the input stream) need to be removed but only for Chinese.
- Opening and closing symbols as well as punctuation needs special treatment especially when they are halfwidth.
- Korean uses proportially spaces punctuation and mixes with other latin fonts, while Chinese often uses built in latin shapes.
- We may break anywhere but not after an opening symbol like (or and not before a closing symbol like).
- We need to deal with mixed Chinese and Korean spacing rules.

Let's start with showing some Korean. We use one of the fonts shipped by Adobe as part of Acrobat but first we define a Korean featureset and a font.

```
\definefontfeature
  [korean]
  [script=hang,language=kor,mode=node,analyze=yes]

\definefont[KoreanSample][adobemyungjostd-medium*korean]
```

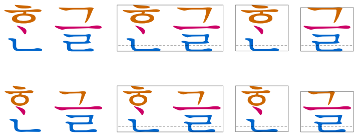
Korean looks like this:

```
\KoreanSample \setscript[hangul]

모든 인간은 태어날 때부터 자유로우며 그 존엄과 권리에 있어 동등하다.
인간은 천부적으로 이성과 양심을 부여받았으며 서로 형제애의 정신으로
행동하여야 한다.
```

모든 인간은 태어날 때부터 자유로우며 그 존엄과 권리에 있어 동등하다. 인간은 천부적으로 이성과 양심을 부여받았으며 서로 형제애의 정신으로 행동하여야 한다.

The Korean script reflect syllabes and is very structured. Although modern fonts contain prebuilt syllabes one can also use the jamo alphabet to build them from components. The following example is provided by Dohyun Kim:

```
\definefontfeature [medievalkorean] [mode=node,script=hang,lang=kor,ccmp=yes,ljmo=ye
\definefontfeature [modernkorean]   [mode=node,script=hang,lang=kor]

\enabletrackers[scripts.analyzing]
\setscript[hangul]
\definedfont [UnBatang*medievalkorean at 20pt]  \ruledhbox{} \ruledhbox{}
\ruledhbox{}\blank
\definedfont [UnBatang*modernkorean   at 20pt]  \ruledhbox{} \ruledhbox{}
\ruledhbox{}\blank
\disabletrackers[scripts.analyzing]
```



There are subtle differences between the medieval and modern shapes. It was this example that lead to more advanced `tounicode` support in MkIV so that copy and paste works out well now for such input.

For Chinese we define a couple of features

```
\definefontfeature
  [chinese-traditional]
  [mode=node,script=hang,lang=zht]
\definefontfeature
  [chinese-simple]
  [mode=node,script=hang,lang=zhs]
\definefontfeature
  [chinese-traditional-hw]
  [mode=node,script=hang,lang=zht,hwid=yes]
\definefontfeature
  [chinese-simple-hw]
  [mode=node,script=hang,lang=zhs,hwid=yes]

\definefont[ChineseSampleFW][adobesongstd-light*chinese-traditional]
\definefont[ChineseSampleHW][adobesongstd-light*chinese-traditional-hw]
\setscript[hanzi]

\ChineseSampleFW
尡也包因沘呡侷柵苗孫孫財崧淫設弼琶跑愍窟榜蒸奭稽
霄瓢館縲擻鬘〈孃魔釁〉佉沎岠狋垚柛肤娭涘罦偟倮牻菂
傒焱菏酡盧滘絺絶堋槢筴踃嬁澕蕁醊獧螗餟燱螬鬌礑錕
瀧鄲瀅騬醹躕鱕。

\ChineseSampleHW
尡也包因沘呡侷柵苗孫孫財崧淫設弼琶跑愍窟榜蒸奭稽
霄瓢館縲擻鬘〈孃魔釁〉佉沎岠狋垚柛肤娭涘罦偟倮牻菂
傒焱菏酡盧滘絺絶堋槢筴踃嬁澕蕁醊獧螗餟燱螬鬌礑錕
瀧鄲瀅騬醹躕鱕。
```

尡也包因沘呡侷柵苗孫孫財崧淫設弼琶跑愍窟榜蒸奭稽霄瓢館縲擻鬘〈孃魔釁〉佉沎岠狋垚柛肤娭涘罦偟倮牻菂傒焱菏酡盧滘絺絶堋槢筴踃嬁澕蕁醊獧螗餟燱螬鬌礑錕瀧鄲瀅騬醹躕鱕。

尵也包因沘氓侷柵苗孫孫財崧淫設弼琶跑愍窟榜蒸奭稽霄瓢館縲擻鼕〈孃魔釁〉佉沘岠狋垚柛胅娭涘罜偟倮牻苭傒焱菏酡盧涾絺艴塴槢筴踃嬁滭蓴醊獧螗餟燱螬騪礑鍡瀧鄞瀯騬醹躕鱕。

A few more samples:

```
\definefont[ChFntAT][name:adobesongstd-light*chinese-traditional-hw at 16pt]
\definefont[ChFntBT][name:songti*chinese-traditional                at 16pt]
\definefont[ChFntCT][name:fangsong*chinese-traditional              at 16pt]

\definefont[ChFntAS][name:adobesongstd-light*chinese-simple-hw      at 16pt]
\definefont[ChFntBS][name:songti*chinese-simple                     at 16pt]
\definefont[ChFntCS][name:fangsong*chinese-simple                   at 16pt]
```

In these fonts traditional comes out as follows:

我〈能吞下玻璃而不傷身〉體。
我〈能吞下玻璃而不傷身〉體。
我〈能吞下玻璃而不傷身〉體。

And simple as:

我〈能吞下玻璃而不伤身〉体。
我〈能吞下玻璃而不伤身〉体。
我〈能吞下玻璃而不伤身〉体。

tracing

As usual in ConTeXt, we have some tracing built in. When you say

You will get the output colored according to the category that the analyser put them in. When you say

some rudimentary information will be written to the log about whet gets inserted in the nodelist.

Analyzed input looks like:

아아, 나는 이제야 도(道)를 알았도다. 마음이 어두운 자는 이목이
누(累)가 되지 않는다. 이목만을 믿는 자는 보고 듣는 것이
더욱 밝혀져서 병이 되는 것이다. 이제 내 마부가 발을 말굽에
밟혀서 뒷차에 실리었으므로, 나는 드디어 혼자 고삐를 늦추어
강에 띄우고, 무릎을 구부려 발을 모으고 안장 위에 앉았다.
한번 떨어지면 강이나 물로 땅을 삼고, 물로 옷을 삼으며,
물로 몸을 삼고, 물로 성정을 삼을 것이다. 이제야 내 마음은

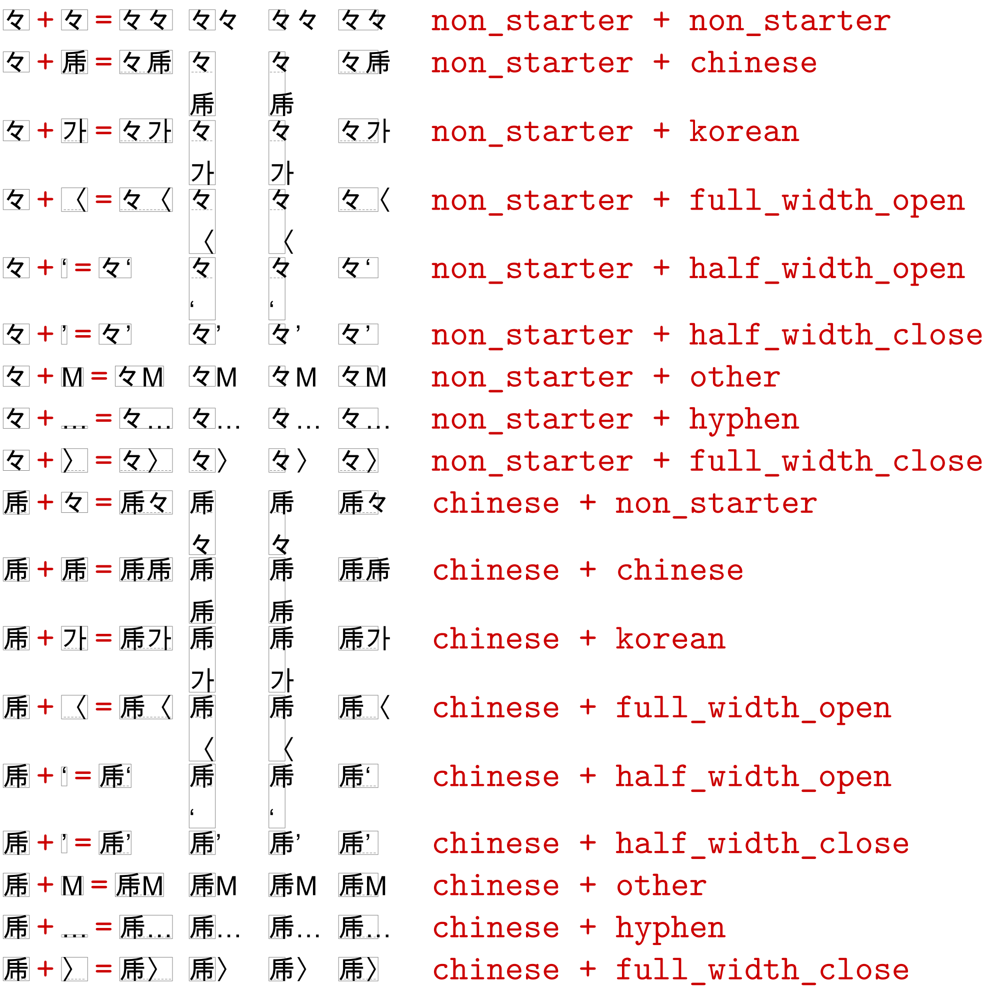
```
한번 떨어질 것을 판단한 터이므로, 내 귓속에 강물 소리가 없어졌다.
무릇 아홉 번 건너는데도 걱정이 없어 의자 위에서 좌와(坐臥)하고
기거(起居)하는 것 같았다.
```

아아, 나는 이제야 도 (道) 를 알았도다. 마음이 어두운 자는 이목이 누 (累) 가 되지 않는다. 이목만을 믿는 자는 보고 듣는 것이 더욱 밝혀져서 병이 되는 것이다. 이제 내 마부가 발을 말굽에 밟혀서 뒷차에 실리었으므로, 나는 드디어 혼자 고삐를 늦추어 강에 띄우고, 무릎을 구부려 발을 모으고 안장 위에 앉았다. 한번 떨어지면 강이나 물로 땅을 삼고, 물로 옷을 삼으며, 물로 몸을 삼고, 물로 성정을 삼을 것이다. 이제야 내 마음은 한번 떨어질 것을 판단한 터이므로, 내 귓속에 강물 소리가 없어졌다. 무릇 아홉 번 건너는데도 걱정이 없어 의자 위에서 좌와 (坐臥) 하고 기거 (起居) 하는 것 같았다.

For developers (and those who provide them with input) we have another tracing

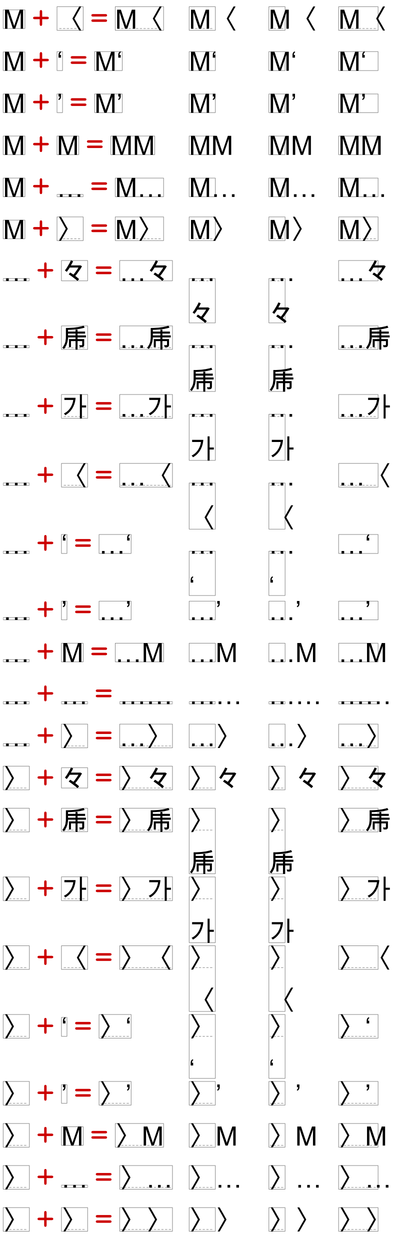
```
\definedfont[arialuni*korean at 10pt] \setscript[hangul] \ShowCombinationsKorean
```

We need to use a font that supports Chinese as well as Korean. This gives quite some output.



```
々 + 々 = 々々    non_starter + non_starter
々 + 帍 = 々帍    non_starter + chinese
々 + 가 = 々가    non_starter + korean
々 + 〈 = 々〈    non_starter + full_width_open
々 + ‘ = 々‘      non_starter + half_width_open
々 + ’ = 々’      non_starter + half_width_close
々 + M = 々M      non_starter + other
々 + … = 々…      non_starter + hyphen
々 + 〉 = 々〉    non_starter + full_width_close
帍 + 々 = 帍々    chinese + non_starter
帍 + 帍 = 帍帍    chinese + chinese
帍 + 가 = 帍가    chinese + korean
帍 + 〈 = 帍〈    chinese + full_width_open
帍 + ‘ = 帍‘      chinese + half_width_open
帍 + ’ = 帍’      chinese + half_width_close
帍 + M = 帍M      chinese + other
帍 + … = 帍…      chinese + hyphen
帍 + 〉 = 帍〉    chinese + full_width_close
```

가 + 々 = 가々	가 々	가 々	가々	korean + non_starter
가 + 乕 = 가乕	가 乕	가 乕	가乕	korean + chinese
가 + 가 = 가가	가 가	가 가	가가	korean + korean
가 + 〈 = 가〈	가 〈	가 〈	가〈	korean + full_width_open
가 + ‘ = 가‘	가 ‘	가 ‘	가‘	korean + half_width_open
가 + ’ = 가’	가’	가’	가’	korean + half_width_close
가 + M = 가M	가M	가M	가M	korean + other
가 + … = 가…	가…	가…	가…	korean + hyphen
가 + 〉 = 가〉	가〉	가〉	가〉	korean + full_width_close
〈 + 々 = 〈々	〈々	〈々	〈々	full_width_open + non_starter
〈 + 乕 = 〈乕	〈乕	〈乕	〈乕	full_width_open + chinese
〈 + 가 = 〈가	〈가	〈가	〈가	full_width_open + korean
〈 + 〈 = 〈〈	〈〈	〈〈	〈〈	full_width_open + full_width_open
〈 + ‘ = 〈‘	〈‘	〈‘	〈‘	full_width_open + half_width_open
〈 + ’ = 〈’	〈’	〈’	〈’	full_width_open + half_width_close
〈 + M = 〈M	〈M	〈M	〈M	full_width_open + other
〈 + … = 〈…	〈…	〈…	〈…	full_width_open + hyphen
〈 + 〉 = 〈〉	〈〉	〈〉	〈〉	full_width_open + full_width_close
‘ + 々 = ‘々	‘々	‘々	‘々	half_width_open + non_starter
‘ + 乕 = ‘乕	‘乕	‘乕	‘乕	half_width_open + chinese
‘ + 가 = ‘가	‘가	‘가	‘가	half_width_open + korean
‘ + 〈 = ‘〈	‘〈	‘〈	‘〈	half_width_open + full_width_open
‘ + ‘ = ‘‘	‘‘	‘‘	‘‘	half_width_open + half_width_open
‘ + ’ = ‘’	‘’	‘’	‘’	half_width_open + half_width_close
‘ + M = ‘M	‘M	‘M	‘M	half_width_open + other
‘ + … = ‘…	‘…	‘…	‘…	half_width_open + hyphen
‘ + 〉 = ‘〉	‘〉	‘〉	‘〉	half_width_open + full_width_close
’ + 々 = ’々	’々	’々	’々	half_width_close + non_starter
’ + 乕 = ’乕	’乕	’乕	’乕	half_width_close + chinese
’ + 가 = ’가	’가	’가	’가	half_width_close + korean
’ + 〈 = ’〈	’〈	’〈	’〈	half_width_close + full_width_open
’ + ‘ = ’‘	’‘	’‘	’‘	half_width_close + half_width_open
’ + ’ = ’’	’’	’’	’’	half_width_close + half_width_close
’ + M = ’M	’M	’M	’M	half_width_close + other
’ + … = ’…	’…	’…	’…	half_width_close + hyphen
’ + 〉 = ’〉	’〉	’〉	’〉	half_width_close + full_width_close
M + 々 = M々	M々	M々	M々	other + non_starter
M + 乕 = M乕	M乕	M乕	M乕	other + chinese
M + 가 = M가	M가	M가	M가	other + korean



other + full_width_open
other + half_width_open
other + half_width_close
other + other
other + hyphen
other + full_width_close
hyphen + non_starter
hyphen + chinese
hyphen + korean
hyphen + full_width_open
hyphen + half_width_open
hyphen + half_width_close
hyphen + other
hyphen + hyphen
hyphen + full_width_close
full_width_close + non_starter
full_width_close + chinese
full_width_close + korean
full_width_close + full_width_open
full_width_close + half_width_open
full_width_close + half_width_close
full_width_close + other
full_width_close + hyphen
full_width_close + full_width_close

XVI Optimization

quality of code

How good is the MkIV code? Well, as good as I can make it. When you browse the code you will probably notice differences in coding style and this is a related to the learning curve. For instance the `luat-inp` module needs some cleanup, for instance hiding local function from users.

Since benchmarking has been done right from the start there is probably not that much to gain, but who knows. When coding in Lua you should be careful with defining global variables, since they may override something. In MkIV we don't guarantee that the name you use for variable will not be used at some point. Therefore, best operate in a dedicated Lua instance, or operate in userspace.

```
do
    -- your code
end
```

If you want to use your data later on, think of working this way (the example is somewhat silly):

```
userdata['your.name'] = userdata['your.name'] or { }

do
    local mydata = userdata['your.name']

    mydata.data = {}

    local function foo() return 'bar' end

    function mydata.dothis()
        mydata[foo] = foo()
    end

end
```

In this case you can always access your user data while temporary variables are hidden. The `userdata` table is predefined. As is `thirddata` for modules that you may write. Of course this assumes that you create a namespace within these global tables.

A nice test for checking global cluttering is the following:

```
for k, v in pairs(_G) do
```

```
    print(k, v)
end
```

When you incidentally define global variables like `n` or `str` they will show up here.

clean or dirty

Processing the first 120 pages of this document (16 chapters) takes some 23.5 seconds on a dell M90 (2.3GHZ, 4GB mem, Windows Vista Ultimate). A rough estimate of where LUA spends its time is:

acticvity	**sec**
input load time	0.114
fonts load time	6.692
mps conversion time	0.004
node processing time	0.832
attribute processing time	3.376

Font loading takes some time, which is nu surprise because we load huge Zapfino, Arabic and CJK fonts and define many instances of them. Some tracing learns that there are some 14.254.041 function calls, of which 13.339.226 concern functions that are called more than 5.000 times. A total of 62.434 function is counted, which is a result of locally defined ones.

A rough indication of this overhead is given by the following test code:

```
local a,b,c,d,e,f = 1,2,3,4,5,6

function one  (a)           local n = 1 end
function three(a,b,c)       local n = 1 end
function six  (a,b,c,d,e,f) local n = 1 end

for i=1,14254041 do one  (a)           end
for i=1,14254041 do three(a,b,c)       end
for i=1,14254041 do six  (a,b,c,d,e,f) end
```

The runtime for these tests (excluding startup) is:

one argument	1.8 seconds
three arguments	2.0 seconds
six arguments	2.3 seconds

So, the of the total runtime for this document we easily spend a couple of seconds on function calls, especially in node processing and attribute resolving. Does this mean that we need to change the code and follow a more inline approach? Eventually we may optimize some code, but for the moment we keep things as readable as possible, and even then much code is still quite complex. Font loading is often constant for a document anyway, and independent of the number of pages. Time spent on node processing depends on the script, and often processing intense scripts are typeset in a larger font and since they are less verbose than latin, this does not really influence the average time spent on typesetting a page. Attribute handling is probably the most time consuming activity, and for large documents the time spent on this is large compared to font loading and node processing. But then, after a few MkIV development cycles the picture may be different.

When we turned on tracing of function calls, if becomes clear where currently the time is spent in a document like this which demands complex Zapfino contextual analysis as well as Arabic analysis and feature application (both fonts demand node insertion and deletion). Of course using color also has a price. Handling weighted and conditional spacing (new in MkIV) involves just over 10.000 calls to the main handler for 120 pages of this document. Glyph related processing of node lists needs 42.000 calls, and contextual analysis of OpenType fonts is good for 11.000 calls. Timing Lua related tasks involves 2 times 37.000 calls to the stopwatch. Collapsing UTF in the input lines equals the number of lines: 7700.

However, at the the top of the charts we find calls to attribute related functions. 97.000 calls for handling special effects, overprint, transparency and alike, and another 24.000 calls for combined color and colorspace handling. These calls result in over 6.000 insertions of PDF literals (this number is large because we show Arabic samples with color based tracing enabled). In case you wonder if the attribute handler can be made more efficient (we're talking seconds here), the answer is "possibly not". This action is needed for each shipped out object and each shipped out page. If we divide the 24.000 (calls) by 120 (pages) we get 200 calls per page for color processing which is okay if you keep in mind that we need to recurse in nested horizontal and vertical lists of the completely made op page.

serialization

When serializing tables, we can end up with very large tables, especially when dealing with big fonts like 'arabtype' or 'zapfino'. When serializing tables one has to find a compromise between speed of writing, effeciency of loading and readability. First we had (sub)tables like:

```
boundingbox = {
    [1] = 0,
    [2] = 0,
    [3] = 100,
    [4] = 200
}
```

I mistakingly assumed that this would generate an indexed table, but at TUG 2007 Roberto Ierusalimschy explained to me that this was not that efficient, since this variant boils down to the following byte code:

```
1       [1]     NEWTABLE        0 0 4
2       [2]     SETTABLE        0 -2 -3 ; 1 0
3       [3]     SETTABLE        0 -4 -3 ; 2 0
4       [4]     SETTABLE        0 -5 -6 ; 3 100
5       [5]     SETTABLE        0 -7 -8 ; 4 200
6       [6]     SETGLOBAL       0 -1    ; boundingbox
7       [6]     RETURN          0 1
```

This creates a hashed table. The following variant is better:

```
boundingbox = { 0, 0, 100, 200 }
```

This results in:

```
1       [1]     NEWTABLE        0 4 0
2       [2]     LOADK           1 -2    ; 0
3       [3]     LOADK           2 -2    ; 0
4       [4]     LOADK           3 -3    ; 100
5       [6]     LOADK           4 -4    ; 200
6       [6]     SETLIST         0 4 1   ; 1
7       [6]     SETGLOBAL       0 -1    ; boundingbox
8       [6]     RETURN          0 1
```

The resulting tables are not only smaller in terms of bytes, but also are less memory hungry when loaded. For readability we write tables with only numbers, strings or boolean values in an inline–format:

```
boundingbox = { 0, 0, 100, 200 }
```

The serialized tables are somewhat smaller, depending on how many subtables are indexed (boundary boxes, lookup sequences, etc.)

normal	**compact**	**filename**
34.055.092	32.403.326	arabtype.tma
1.620.614	1.513.863	lmroman10-italic.tma
1.325.585	1.233.044	lmroman10-regular.tma
1.248.157	1.158.903	lmsans10-regular.tma
194.646	153.120	lmtypewriter10-regular.tma
1.771.678	1.658.461	palatinosanscom-bold.tma

1.695.251	1.584.491	palatinosanscom-regular.tma
13.736.534	13.409.446	zapfinoextraltpro.tma

Since we compile the tables to bytecode, the effects are more spectacular there.

normal	**compact**	**filename**
13.679.038	11.774.106	arabtype.tmc
886.248	754.944	lmroman10-italic.tmc
729.828	466.864	lmroman10-regular.tmc
688.482	441.962	lmsans10-regular.tmc
128.685	95.853	lmtypewriter10-regular.tmc
715.929	582.985	palatinosanscom-bold.tmc
669.942	540.126	palatinosanscom-regular.tmc
1.560.588	1.317.000	zapfinoextraltpro.tmc

Especially when a table is partially indexed and hashed, readability is a bit less than normal but in practice one will seldom consult such tables in its verbose form.

After going beta, users reported problems with scaling of the the Latin Modern and TeX-Gyre fonts. The troubles originate in the fact that the OpenType versions of these fonts lack a design size specification and it happens that the Latin Modern fonts do have design sizes other than 10 points. Here the power of a flexible TeX engine shows . . . we can repair this when we load the font. In MkIV we can now define patches:

```
do
    local function patch(data,filename)
        if data.design_size == 0 then
            local ds = (file.basename(filename)):match("(%d+)")
            if ds then
                logs.report("load otf",string.format("patching design
size (%s)",ds))
                data.design_size = tonumber(ds) * 10
            end
        end
    end

    fonts.otf.enhance.patches["^lmroman"] = patch
    fonts.otf.enhance.patches["^lmsans"]  = patch
    fonts.otf.enhance.patches["^lmmono"]  = patch
end
```

Eventually such code will move to typescripts instead of in the kernel code.

XVII XML revisioned

The code dealing with XML is evolving and the following text might be outdated. So, in case of doubt, check the manual.

the parser

For quite a while ConTEXt has built-in support for XML processing and at Pragma ADE we use this extensively. One of the first things I tried to deal with in Lua was XML, and now that we have LuaTEX up and running it's time to investigate this a bit more. First we'll have a look at the basic functions, the Lua side of the game.

We load an XML file as follows (the `document` namespace is predefined in ConTEXt):

```
\startluacode
    document.xml = document.xml or { } -- define namespace
    document.xml = xml.load("mk-xml.xml") -- load the file
\stopluacode
```

The loader constructs a table representing the document structure, including whitespace, so let's serialize the code and see what shows up:

```
\startluacode
    local prn = xml.newhandlers { handle = tex.sprint }
    tex.sprint("\\starttyping")
    xml.serialize(document.xml, prn)
    tex.sprint("\\stoptyping")
\stopluacode
```

In the first version of the serializer, we could pass extra function arguments that controlled the way content was processed. This method has now been replaced by handlers. In this example we create a simple handler where the `handle` function is responsible for the final print.

```
<?xml version='1.0 standalone='yes' ?>

<one>
    <two>
        <a>alpha</a>
        <b/>
        <c>gamma</c>
        <d/>
        <e>epsilon</e>
    </two>
    <three>
        <some>pdftex</some>
        <some>luatex</some>
        <some>xetex</some>
    </three>
    <four>
        <more:some name="hans"/>
        <more:some name="taco"/>
        <more:some name="hartmut"/>
    </four>
    <five>
        <some>metapost</some>
    </five>
</one>
```

This already gives us a rather basic way to manipulate documents and this method is even not that slow because we bypass TEX reading from file.

```
\startluacode
    local str = "<l> <w>hello</w> <w>world</w> </l>"
    local prn = xml.newhandlers { handle = tex.sprint }
    tex.sprint("\\starttyping")
    xml.serialize(xml.convert(str),prn)
    tex.sprint("\\stoptyping")
\stopluacode
```

Watch the extra print argument, we need this because otherwise the verbatim mode will not work out well.

```
<l> <w>hello</w> <w>world</w> </l>
```

You need to keep in mind that in these examples we print to TEX under the current catcode regime.

You can save a XML table with the command:

```
\startluacode
    xml.save(document.xml,"newfile.xml")
\stopluacode
```

These examples show that you have access to XML files from within your document. If you want to convert the table to just a string, you can use `xml.tostring`. Actually, this method is automatically used for occasions where LUA wants to print an XML table or wants to join string snippets. However, as we are inside TEX, we need to print to TEX instead of the console or file. For this we use specialized handlers.

The reason why I wrote the XML parser is that we need it in the utilities (so it has to provide access to the content of elements) as well as in the text processing (so it needs to provide some manipulation features). To serve both we have implemented a subset of what standard XML tools qualify as path based searching.

```
\startluacode
    xml.sprint(xml.first(document.xml, "/one/three/some"))
\stopluacode
```

The result of this snippet is the content of the first element that matches the specification: '<some>pdftex</some>'. As you can see, this comes out rather verbose. The reason for this is that we need to enter XML mode in order to get such a snippet interpreted.

Below we give a few more variants, this time we use a generic filter:

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some"))
\stopluacode
```

result: `<some>pdftex</some><some>luatex</some><some>xetex</some>`

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some/first()"))
\stopluacode
```

result: `<some>pdftex</some>`

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some[1]"))
\stopluacode
```

result: `<some>pdftex</some>`

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some[-1]"))
\stopluacode
```

result: `<some>luatex</some>`

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some/texts()"))
\stopluacode
```

result:

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some[2]/text()"))
\stopluacode
```

result: `luatex`

The next lines shows some more variants. There are more than these and we will extend the repertoire over time. If needed you can define additional handlers.

performance

Before we continue with more examples, a few remarks about the performance. The first version of the parser was an enhanced version of the one presented in the Lua book: support for namespaces, processing instructions, comments, cdata and doctype, remapping and a few more things. When playing with the parser I was quite satisfied about the performance. However, when I started experimenting with 40 megabyte files, the preprocessing (needed for the special elements) started to become more noticeable. For smaller files its 40% overhead is not that disturbing, but for large files . . .

The current version uses lpeg. We follow the same approach as before, stack and top and such but this time parsing is about twice as fast which is mostly due to the fact that we don't have to prepare the stream for cdata, doctype etc. Loading the mentioned large file took 12.5 seconds (1.5 for file io and the rest for tree building) on my laptop (a 2.3 Ghz Core Duo running Windows Vista). With the lpeg implementation we got that down to less 7.3 seconds. Loading the 14 interface definition files (2.6 meg) went down from 1.05 seconds to 0.55 seconds. Namespace related issues take some 10% of this.

Of course these numbers might change over time. For instance, we now have the second implementation of the filter mechanism which is more advanced and maybe somewhat slower on some tasks.

patterns

We will not implement complete xpath functionality, but only the features that make sense for documents that are well structured and needs to be typeset. In addition we (will) implement text manipulation functions. Of course speed is also a consideration when implementing such mechanisms.

The following list is not complete (after all here we only give an impression of the development) but it gives a good impression.

pattern	**supported**	**comment**
`a`	⋆	not anchored

`!a`	⋆	not anchored,negated
`a/b`	⋆	anchored on preceding
`/a/b`	⋆	anchored (current root)
`^a/c`	⋆	anchored (current root)
`^^/a/c`	todo	anchored (document root)
`a/*/b`	⋆	one wildcard
`a//b`	⋆	many wildcards
`a/**/b`	⋆	many wildcards
`.`	⋆	ignored self
`..`	⋆	parent
`a[5]`	⋆	index upwards
`a[-5]`	⋆	index downwards
`a[position()=5]`	maybe	
`a[first()]`	maybe	
`a[last()]`	maybe	
`(b|c|d)`	⋆	alternates (one of)
`b|c|d`	⋆	alternates (one of)
`!(b|c|d)`	⋆	not one of
`a/(b|c|d)/e/f`	⋆	anchored alternates
`(c/d|e)`	not likely	nested subpaths
`a/b[@bla]`	⋆	any value of
`a/b/@bla`	⋆	any value of
`a/b[@bla='oeps']`	⋆	equals value
`a/b[@bla=='oeps']`	⋆	equals value
`a/b[@bla<>'oeps']`	⋆	different value
`a/b[@bla!='oeps']`	⋆	different value
`...../attribute(id)`	⋆	
`...../attributes()`	⋆	
`...../text()`	⋆	
`...../texts()`	⋆	
`...../first()`	⋆	
`...../last()`	⋆	
`...../index(n)`	⋆	
`...../position(n)`	⋆	
`root::`	⋆	
`parent::`	⋆	
`child::`	⋆	
`ancestor::`	⋆	
`preceding-sibling::`	not soon	
`following-sibling::`	not soon	
`preceding-sibling-of-self::`	not soon	

`following-sibling-or-self::`	not soon
`descendent::`	⋆
`descendent-or-self::`	⋆
`preceding::`	not soon
`following::`	not soon
`self::node()`	not soon
`id("tag")`	not soon
`node()`	not soon

This list shows that it is also possible to ask for more matches at once. Namespaces are supported (including a wildcard) and there are mechanisms for namespace remapping.

```
\startluacode
    lxml.concat(document.xml,"/one/(three|five)/some",", "," and ")
\stopluacode
```

We get: `<some>pdftex</some>, <some>luatex</some>, <some>xetex</some> and <some>metapost</some>` and if we say:

```
\startluacode
    lxml.concat(document.xml,"/one/(three|five)/some",", "," and ",
        true)
\stopluacode
```

We get: ‘pdftex, luatex, xetex and metapost’.

Watch how we use the `lxml` namespace here! Here live the functions that pipe the result to TEX.

There a several helper functions, like `xml.count` which in this case returns 4.

```
\startluacode
    lxml.count(document.xml,"/one/(three|five)/some")
\stopluacode
```

Functions like this gives the opportunity to loop over lists of elements by index.

manipulations

We can manipulate elements too. The next code will add some elements at specific locations.

```
\startluacode
    xml.before(document.xml,"xml:///one/three/some","<be>ok</be>")
    xml.after (document.xml,"xml:///one/three/some","<af>ok</af>")
    tex.sprint("\\starttyping")
    xml.sprint(lxml.filter(document.xml,"/one/three"))
    tex.sprint("\\stoptyping")
\stopluacode
```

And indeed, we suddenly have a couple of 'ok''s there:

```
<three>
<be>ok</be><some>pdftex</some><af>ok</af>
<be>ok</be><some>luatex</some><af>ok</af>
<be>ok</be><some>xetex</some><af>ok</af>
</three>
```

Of course wel can also delete elements:

```
\startluacode
    xml.delete(document.xml,"/one/three/some")
    xml.delete(document.xml,"/one/three/af")
    tex.sprint("\\starttyping")
    xml.sprint(lxml.filter(document.xml,"/one/three"))
    tex.sprint("\\stoptyping")
\stopluacode
```

Now we have:

```
<three>
<be>ok</be><af>ok</af>
<be>ok</be><af>ok</af>
<be>ok</be><af>ok</af>
</three>
```

Replacing an element is also possible. The replacement can be a table (representing elements) or a string which is then converted into a table first.

```
\startluacode
    xml.replace(document.xml,"/one/three/be","<mid>done</mid>")
    tex.sprint("\\starttyping")
    xml.sprint(lxml.filter(document.xml,"/one/three"))
    tex.sprint("\\stoptyping")
\stopluacode
```

And indeed we get:

```
<three>
<be>ok</be><af>ok</af>
```

```
<be>ok</be><af>ok</af>
<be>ok</be><af>ok</af>
</three>
```

These are just a few features of the library. I will add some more (rather) generic manipulaters and extend the functionality of the existing ones. Also, there will be a few manipulation functions that come in handy when preparing texts for processing with TEX (most of the XML that I deal with is rather dirty and needs some cleanup).

streaming trees

Eventually we will provies series of convenient macros that will provide an alternative for most of the MkII code. In MkII we have a streaming parser, which boils down to attaching macros to elements. This includes a mechanism for saving an restoring data, but this is not always convenient because one also has to intercept elements that needs to be hidden.

In MkIV we do things different. First we load the complete document in memory (a Lua table). Then we flush the elements that we want to process. We can associate setups with elements using the filters mentioned before. We can either use TEX or use Lua to manipulate content. Instead if a streaming parser we now have a mixture of streaming and tree manipulation available. Interesting is that the XML loader is pretty fast and piping data to TEX is also efficient. Since we no longer need to manipulate the elements in TEX we gain processing time too, so in practice we have now much faster XML processing available.

To give you an idea we show a few commands:

```
\xmlload {main}{mk-xml.xml}
```

So that we can do things like (there are and will be a few more):

command	**arguments**	**result**
`\xmlfirst`	`{main} {/one/three/some}`	<some>pdftex</some>
`\xmllast`	`{main} {/one/three/some}`	<some>xetex</some>
`\xmlindex`	`{main} {/one/three/some} {2}`	<some>luatex</some>

There is a set of about 30 commands that operates on the tree: loading, flushing, filtering, associating setups and code in modules to elements. For instance when one uses so called cals–tables, the processing is automatically activates when the namespace can be resolved. Processing is collected in setups and those registered are these are processed after loading the tree. In the following example we register a handler for content that needs to end up bold.

```
\startxmlsetups xml:mysetups
```

```
    \xmlsetsetup{\xmldocument}{bold|bf}{xml:handlebold}
\stopxmlsetups

\xmlregistersetup{xml:mysetups}

\startxmlsetups xml:handlebold
    \dontleavehmode
    \bgroup
    \bf
    \xmlflush{#1}
    \egroup
\stopxmlsetups
```

In this example `#1` represents the root of the subtree. Say that we want to process an index entry which is coded as follows:

```
<index>
    <entry>whatever</entry>
    <key>whatever</key>
</index>
```

We register an additional handler (here the `*` is a shortcut for using the element's tag as setup name):

```
\startxmlsetups xml:mysetups
    \xmlsetsetup{\xmldocument}{bold|bf}{xml:handlebold}
    \xmlsetsetup{\xmldocument}{index}{*}
\stopxmlsetups

\xmlregistersetup{xml:mysetups}

\startxmlsetups index
    \index[\xmlfirst{#1}{key}]{\xmlfirst{#1}{entry}}
\stopxmlsetups
```

In practice MkIV definitions are more compact than the comparable MkII ones, especially for more complex constructs (tables and such).

```
\defineXMLenvironment
  [index]
  {\bgroup
   \defineXMLsave[key]%
   \defineXMLsave[entry]}
  {\index[\XMLflush{key}]{\XMLflush{entry}}%
```

```
    \egroup}
```

This looks compact, but keep in mind that we also need to get rid of spurry spaces and when the code grows, we usually use setups to separate the definition from the code. In any case, the MkII solution involves a few definitions as well as saving the content of elements. This is often much more costly than the MkIV method where we only locate and flush content. Of course the document is stored in memory, but that happens pretty fast: storing the 14 files (2 per interface) that define the ConTEXt user interface takes .85 seconds on a 2.3 Ghz Core Duo (Windows Vista) which is not that bad if you take into account that we're talking of 2.7 megabytes of highly structured data (many elements and attributes, not that much text). Loading one of these files using MkII code (for storing elements) takes many more seconds.

I didn't do extensive speed tests yet but for normal streamed processing of simple documents the penalty of loading the tree can be neglected. When comparing traditional MkII code like:

```
\defineXMLargument   [title][id=] {\subject[\XMLop{at}]}
\defineXMLenvironment[p]          {} {\par}

\starttext
    \processXMLfilegrouped{testspeed.xml}
\stoptext
```

with its MkIV counterpart:

```
\startxmlsetups document
    \xmlsetsetup\xmldocument{title|p}{*}
\stopxmlsetups

\xmlregistersetup{document}

\startxmlsetups title
    \section[\xmlatt{#1}{id}]{\xmlcontent{#1}{/}}
\stopxmlsetups

\startxmlsetups p
    \xmlflush{#1}\endgraf
\stopxmlsetups

\starttext
    \processXMLfilegrouped{testspeed.xml}
\stoptext
```

```
I found that processing a one megabyte file with some 400 sections
takes the same runtime for both approaches. However, as soon as more
complex manipulations enter the game the \MKIV\ method starts taking
less time. Think of the manipulations needed for \MATHML\ or converting
tables into something that \CONTEXT\ can handle. Also, when we deal
with documents where we need to ignore large portions of shuffle content
around, the traditional method also has to store data in memory and
in
that case \MKII\ code always loses from \MKIV\ code. Of course any
speed
we gain in handling \XML\ is lost on processing complex fonts and
attributes but there we gain in quality.
```

Another advantage of the MkIV mechanisms is that we suddenly have so called fully expandable XML handling. All manipulations take place in Lua and there is no interfering code at the TEX end.

examples

For the path freaks we now show what patterns lead to. For this we will use the following XML data:

```
<?xml version='1.0' ?>
<a>
    <?what is this?>
    <b>
        <c n='x'>c1</c><d>d1</d>
    </b>
    <b>
        <c n='y'>c2</c><d>d2</d>
    </b>
    <?what is that?>
    <c><d>d3</d></c>
    <c n='y'><d>d4</d></c>
    <c><d>d5</d></c>
</a>
```

Here come the examples:

```
a/b/c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
```

```
/a/b/c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
```

```
b/c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
```

```
c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
<c><d>d3</d></c>
<c n="y"><d>d4</d></c>
<c><d>d5</d></c>
```

```
a/*/c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
```

```
a/**/c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
```

```
a//c
```

```
<c n="x">c1</c>
<c n="y">c2</c>
<c><d>d3</d></c>
<c n="y"><d>d4</d></c>
<c><d>d5</d></c>
```

```
a/*/*/c
```

```
no match
```

```
*/c
```

```
<c><d>d3</d></c>
<c n="y"><d>d4</d></c>
<c><d>d5</d></c>
```

```
**/c
```

```
<c><d>d3</d></c>
<c n="y"><d>d4</d></c>
<c><d>d5</d></c>
<c n="x">c1</c>
<c n="y">c2</c>
```

```
a/../*/c
```

```
<c><d>d3</d></c>
<c n="y"><d>d4</d></c>
<c><d>d5</d></c>
```

```
a/../c
```

```
no match
```

```
c[@n='x']
```

```
<c n="x">c1</c>
```

```
c[@n]
```

```
<c n="x">c1</c>
<c n="y">c2</c>
<c n="y"><d>d4</d></c>
```

```
c[@n='y']
```

```
<c n="y">c2</c>
<c n="y"><d>d4</d></c>
```

```
c[1]
```

```
<c n="x">c1</c>
```

```
b/c[1]
```

```
<c n="x">c1</c>
```

```
a/c[1]
```

```
<c><d>d3</d></c>
```

```
a/c[-1]
```

```
<c n="y"><d>d4</d></c>
```

```
c[1]
```

```
<c n="x">c1</c>
```

```
c[-1]
```

```
<c n="y"><d>d4</d></c>
```

```
pi::
```

```
no match
```

```
pi::what
```

```
no match
```

XVIII Breaking apart

[todo: mention changes to hyphenchar etc]

Because the long term objective is to have control over all aspects of the typesetting, quite some effort went into opening up one of the cornerstones of TEX: breaking paragraphs into lines. And because this is closely related to hyphenating words, this effort also meant that we had to deal with ligature building and kerning.

This is best explained with an example. Imagine that we have the following sentence[1]

> We imagined it was being ground down smaller and smaller, into a kind of powder. And we realized that smaller and smaller could lead to bigger and bigger problems.

With the current language settings for US English this can be hyphenated as follows:

> We imag-ined it was be-ing ground down smaller and smaller, into a kind of pow-der. And we re-al-ized that smaller and smaller could lead to big-ger and big-ger prob-lems.

So, when breaking a paragraph into lines, TEX has a few options, but here actually not that many. If we permits two character snippets, we can get:

> We imag-ined it was be-ing ground down small-er and small-er, in-to a kind of pow-der. And we re-al-ized that small-er and small-er could lead to big-ger and big-ger prob-lems.

If we revert to UK English, we get:

> We ima-gined it was being ground down smal-ler and smal-ler, into a kind of powder. And we real-ized that smal-ler and smal-ler could lead to big-ger and big-ger prob-lems.

or, more tolerant,

> We ima-gined it was being ground down smal-ler and smal-ler, into a kind of powder. And we real-ized that smal-ler and smal-ler could lead to big-ger and big-ger prob-lems.

or with Dutch patterns:

> We ima-gi-ned it was being ground down smal-ler and smal-ler, in-to a kind of pow-der. And we re-a-li-zed that smal-ler and smal-ler could lead to big-ger and big-ger pro-blems.

The code in traditional TEX that deals with hyphenation and linebreaks is rather interwoven. There is a relationship between the font encoding and the way patterns are encodes.

---

[1] The World Without Us, Alan Weisman; a quote from Richard Thomson in chapter: Polymers are Forever.

A few years after TEX was written, support for multiple languages was added, which resulted in a mix of (kind of global) language settings (no nodes) and language nodes in the node lists. Traditionally it roughly works as follows:

- The input `We imagined it` is tokenized and turned into glyph nodes. If non ASCII characters are used (like pre composed accented characters) there may be a translation step: macros or active characters can insert `\char` commands or map onto other characters, for instance input byte 123 can become byte 198 which in turn ends up as a reference in a glyph node to a font slot. Whatever method is used to go from input to glyph node, eventually we have a reference to a position in a font. Unfortunately we had only 256 such slots per font.

- When it's time to break a paragraph into lines, traditional TEX walks over the list, reconstruct words and inserts hyphenation points. In the process, inter-character kerns that are already injected need to be removed and reinserted, and ligatures have to be decomposed and recomposed. The magic of hyphenation is controlled by discretionary nodes. These specify what to do when a word is hyphenated. Take for instance the Dutch word `effe` which hyphenated becomes `ef-fe` so the `ff` either stays, or is split into `f-` and `f`.

- Because a glyph node is bound to a font, there is a relationship with the font encoding. Because there is no one 8-bit encoding that suits all languages, we may end up with several instances of a font in one document (used for different languages) and each when we switch language and/or font, we also have to enable a suitable set of patterns (in a matching encoding).

You can imagine that this may lead to moderately complex mechanisms in macro packages. For instance, in ConTEXt, to each language multiple font encodings can be bound and a switch of fonts (with related encoding) also results in a switch to a suitable set of patterns. But in MkIV things are done different.

First of all, we got rid of font encodings by exclusively using Unicode. We already were using UTF encoded patterns (so that we could load them under different font encodings) so less patterns had to be loaded per language. That happened even before the LuaTEX development arrived at hyphenation.

Before that effort started, Taco and I already played a bit with alternative hyphenation methods. For instance, we took large word lists with hyphenation points inserted. Taco wrote a loader (Lua could not handle the large tables as function return value) and I made some hyphenation code in Lua. Surprisingly we found out that it was pretty efficient, although we didn't have the weighted hyphenation points that patterns may provide. Basically we simulated the `\hyphenation` command.

While we went back to fonts, Taco's college Nanning wrote the first version of a new hyphenation storage mechanism, so when about half a year later we were ready to deal with the linebreak mechanisms, one of the key components was more or less ready. Where fonts forced me to write quite some Lua code (still not finished), the new hyphenation mechanisms could be supported rather easy, if only because the framework was already kind of present (written during the experiments). Even better, when splitting the old code into MkII and new MkIV code, I could do most housekeeping in Lua, and only needed a minimal amount of TEX interfacing (partly redundant because of the shared interface). The new mechanism also was no longer bound to the format, which means that we could postpone loading of the patterns to runtime. Instead of the still supported traditional loading of patterns and exceptions, we load them under Lua control. This gave me yet another nice excercise in using `lpeg` (Lua's string parser).

With a new pattern loader in place, Taco started separating the hyphenation, ligature building and kerning. Each stage now has its own callback and each stage has an associated Lua function, so that one can create a different order of execution or integrate it in other node parsing activities, most noticeably the handling of OpenType features.

When I was trying to integrate this into the already existing node processing sequences, some nasty tricks were needed in order to feed the hyphenation function. At that moment it was still partly modelled after the traditional TEX way, which boiled down to the following. As soon as the hyphenation function is invoked, it needs to know what the current language is. This information is not stored in the node list, only mid paragraph language switched are stored. Due to the fact that much information in TEX is global (well, in LuaTEX less and less) this complicates matters. Because in MkIV hyphenation, ligature building and kerning are done differently (dus to OpenType) we used the hyphenation callback to collect the language parameters so that we could use them when we called the hyphenation function later. This can definetely be qualified as an ugly hack.

Before we discuss how this was solved, we summarize the state of affairs. In LuaTEX we now have a sequence of callbacks related to paragraph building and in between not much happens any more.

- hyphenation
- ligaturing
- kerning
- preparing linebreaking
- linebreaking
- finishing linebreaking

Before we only had:

- preparing linebreaking

and this is where MkIV hooks in ist code. The first three are disabled by associating them with dummy functions. I'm still not sure how the last two will fit it, especially because there is some interplay between OpenType features and linebreaking, like alternative glyphs at the end of the line. Because the hz and protruding mechanisms also will be supported we may as well end up with a mechanism for alternative glyphs built into the linebreak algorithm.

Back to the current situation. What made matters even more complicated was the fact that we need to manipulate node lists while building horizontal material (hpacking) as well as for paragraphs (pre-linebreaking). Compare the following two situations. In the first case the hbox is packaged and hyphenation is not needed.

```
text \hbox {text} text
```

However, when we unbox the content, hyphenation needs to be applied.

```
\setbox0=\hbox{text} text \unhbox0\ text
```

[I need to check the next]

Traditional TEX does not look at all potential hyphenation points, but only around places that have a high probability as line-end. LuaTEX just hyphenates the whole list, although the function can be used selectively over a range, in MkIV we see no reason for this and hyphenate whole lists.

The new hyphenation routine not only operates on the whole list, but also can be made transparent for uppercase characters. Because we assume Unicode lowercase codes are no longer stored with the patterns (an ε-TEX extension). The usual left- and righthyphenmin control is still there. The first word of a paragraph is no longer ignored in the process.

Because the stages are separated now, the opportunity was there to separate between characters and glyphs. As with traditional TEX, only characters are taken into account when hyphenating, so how do we distinguish between the two? The subtype (a property of each node) already registered if we were dealing with a ligature or not. Taco and Nanning had decided to treat the subtype as a bitset and after a bit of testing ans skyping we came to the conclusion that we needed an easy way to tag a glyph node as being 'already processed'. Keep in mind that as in the unhboxed example, the unhboxed content is already treated (hpack callback). If you wonder why we have these two moments of treatment think of this: if you put something in a box and want to know its dimensions, all font related features need to be applied. If the box is inserted as is, it can be recognized (a hlist or vlist node) and safely skipped in the prelinebreak handling. However, when it is unhboxed, we want to avoid reprocessing. Normally reprocessing will be prevented because the glyph nodes are mixed with kerns and ligatures are already built, but we can best play safe. Once we're done with processing a list (which can involve many passes, depending on what treatment is needed) we can tag the glyphs nodes as 'done' by adding 256 to the subtype. We can then test on this property in callbacks while at the same time built-in functions like those responsible for hyphenation ignore this high bit.

The transition from character to glyph is also done by changing bits in the subtype. At some point we need to set the subtype so that it reflects the node being a glyph, ligature or other special type (there are a few more types inherited from omega). I know that this all sounds complicated, but in MkIV we now roughly do the following (of course this may and probably will change):

- attribute driven manipulations (for instance case change)
- language driven manipulations (spell checking, hyphenation)
- font driven treatments, mostly features (ligature building, kerning)
- turn characters into glyphs (so that they will not be hyphenated again)
- normal ligaturing routine (currently still needed for not open type fonts, may become obsolete)
- normal kerning routine (currently still needed for not open type fonts, may become obsolete)
- attribute driven manipulations (special spacing and kerning)

When no callbacks are used, turning characters into glyphs happens automatically behind the screens. When using callbacks (as in MkIV) this needs to be done explicitly (but there is a helper function for this).

So, by now LuaTeX can determine which glyph nodes play a role in hyphenation but still we have this 'what language are we in' problem. As usual in the development of LuaTeX, these fundamental changes took place in a setting where Taco and I are in a persistent state of Skyping, and it did not take much time to decide that in order to make the callbacks usable, it made much sense to moving the language related information to the glyph node as well, i.e. the number of the language object (patterns and exceptions), the left and right min values, and the boolean that tells how to treat uppercase characters. Each is now accessible in the usual way (by key). The penalty in additional memory is zero because it's stored along with the subtype bitset. By going this route, the ugly hack mentioned before could be removed as well.

In the process of finalizing the code, discretionary nodes got a slightly different implementation. Originally they were organized as follows (ﬀ is a ligature):

```
con-text == [c][o](pre=n-,post=,replace=1)[n][t][e][x][t]
effe     == [e](pre=f-,post=f,replace=1)[ff][e]
```

So, a discretionaty node contained information about what to put at the end of the broken line and what to put in front of the next line, as well as the number of following nodes in the list to skip when such a linebreak occured. Because this leads to rather messy code especially when ligatures are involved, so the decision was made to change the replacement counter into a node list holding those (optionally) to be replaced nodes.

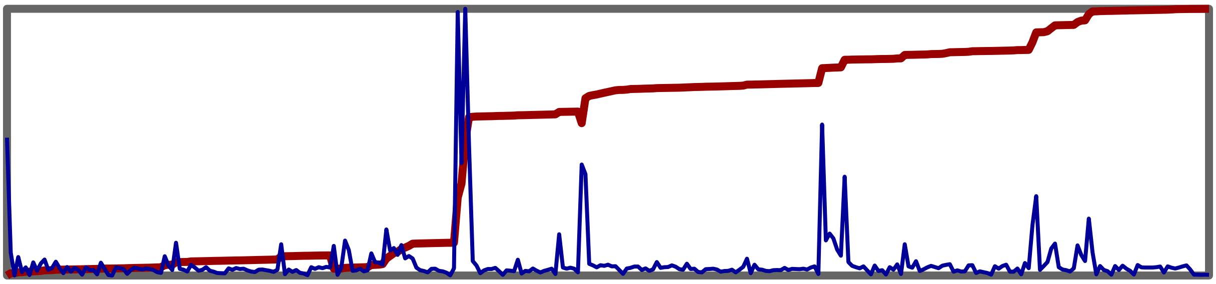
```
con-text == [c][o](pre=n-,post=,replace=n)[t][e][x][t]
effe     == [e](pre=f-,post=f,replace=ff)[e]
```

This is much cleaner, but a consequence of this change was that all MkIV node manipulation code written so far had to be reviewed.

Of course we need to spend a few words on performance. We keep doing performance tests but currently we only remove bottlenecks that bother us. Later in the development optimization will tke place in the code. One reason is that the code changes, another reason is that large portions of Pascal code is turned into C. Because integrating these changes (apart from preparations) took place within a few weeks, we could reasonably well compare the old and the new hyphenation mechanisms using our (evolving) manuals and surprisingly the performance was certainly not worse than before.

XIX Collecting garbage

We use the `mk.tex` document for testing and because it keeps track of how LuaTEX evolves. As a result it has some uncommon characteristics. For instance, you can see increments in memory usage at points where we load fonts: the chapters on Zapfino, Arabic and CJK (unfinished). This memory is not freed because the font memory is used permanently. In the following graphic, the red line is the memory consumption of LuaTEX for the current version of `mk.tex`. The blue line is the runtime per page.

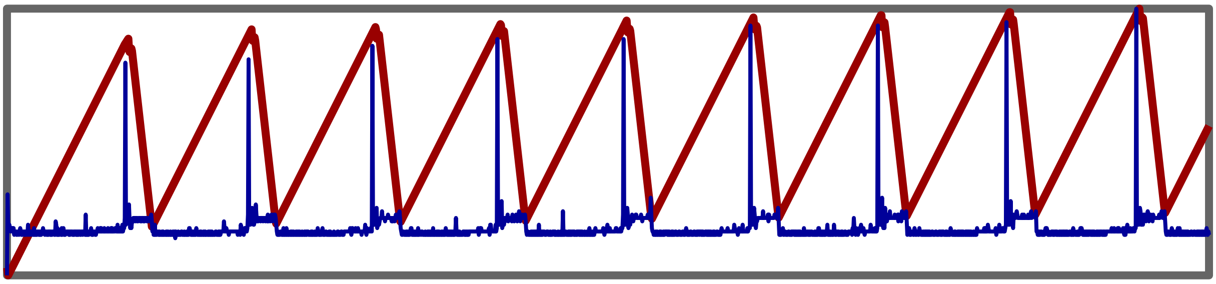


luastate_bytes min:76184711, max:795257742, pages:320

At the moment of writing this Taco has optimized the LuaTEX code base and I have added dynamic feature support to the MkIV and optimized much of the critical Lua code. At the time of writing this (December 23, 2007), `mk.tex` counted 142 pages. Our rather aggressive optimizations brought down runtime from about 29 seconds to under 16 seconds. By sharing as much font data as possible at the Lua end (at the cost of a more complex implementation) the memory consumption of huge fonts was brought down to a level where a somewhat 'older' computer with 512 MB memory could also cope with MkIV. Keep in mind that some fonts are just real big. Eventually we may decide to use a more compact table model for passing OpenType fonts to Lua, but this will not happen in 2007.

The following tests show when Lua's garbage collector becomes active. The blue spike shows that some extra time is spent on this initially. After that garbage more garbage is collected, which makes the time spent per page slightly higher.

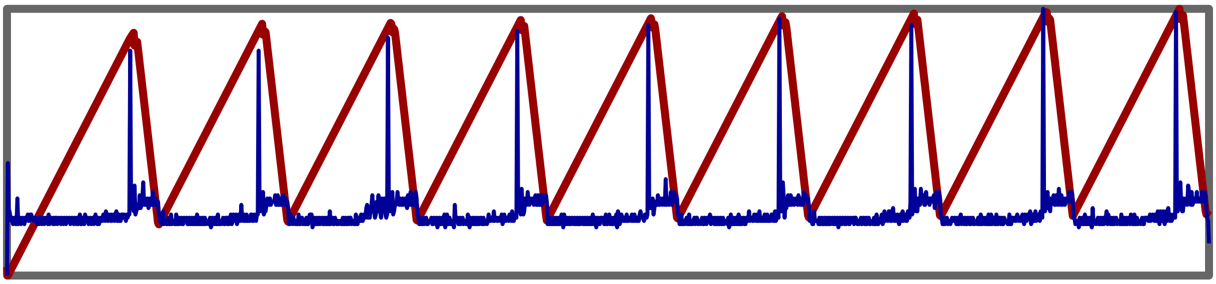
```
\usemodule[timing] \starttext \dorecurse{2000}{
    \input tufte \par \input tufte \par \input tufte \page
} \stoptext
```



luastate_bytes min:37009927, max:87755930, pages:2000

The maximum memory footprint is somewhat misleading because Lua reserves more than needed. As discussed in an earlier chapter, it is possible to tweak to control memory management somewhat, but eventually we decided that it does not make much sense to divert from the default settings.

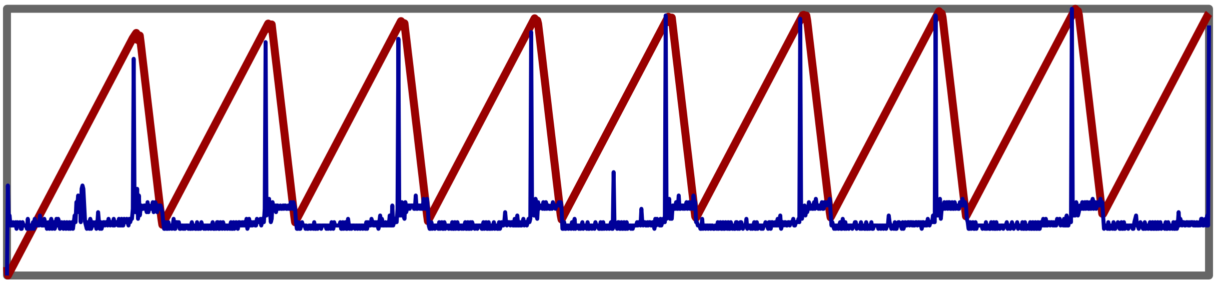
```
\usemodule[timing] \starttext \dorecurse{2000}{
    \input tufte \par \input tufte \par \input tufte \par
} \stoptext
```



luastate_bytes min:36884954, max:86480013, pages:1385

The last example of this set does not load files, but stores the text in a macro. This is faster, although not that mich because the operating system caches the file and there is not UTF collapsing needed for this file.

```
\usemodule[timing] \starttext \dorecurse{2000}{
    \tufte \par \tufte \par \tufte \par
} \stoptext
```

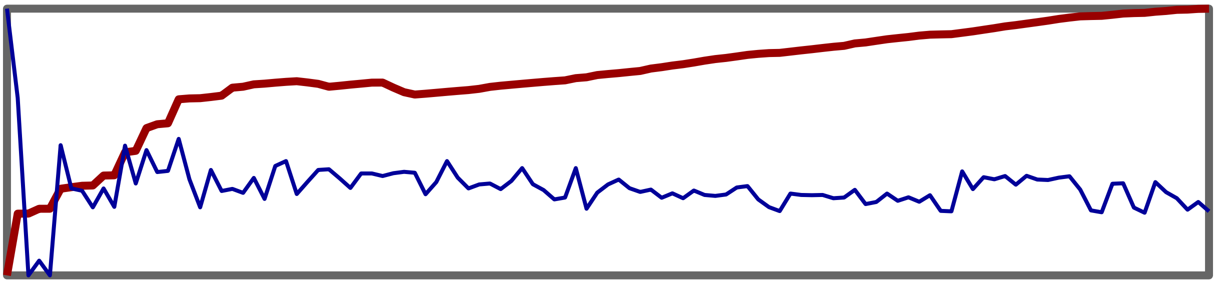


luastate_bytes min:36876892, max:86359763, pages:1385

There are subtle differences in memory usage between the examples and eventually test like these will permit us to optimize the code even further. For the record: the first test runs in 39.5 seconds, the second on in 36.5 seconds and the last one only takes 31.5 seconds (all in batch mode).

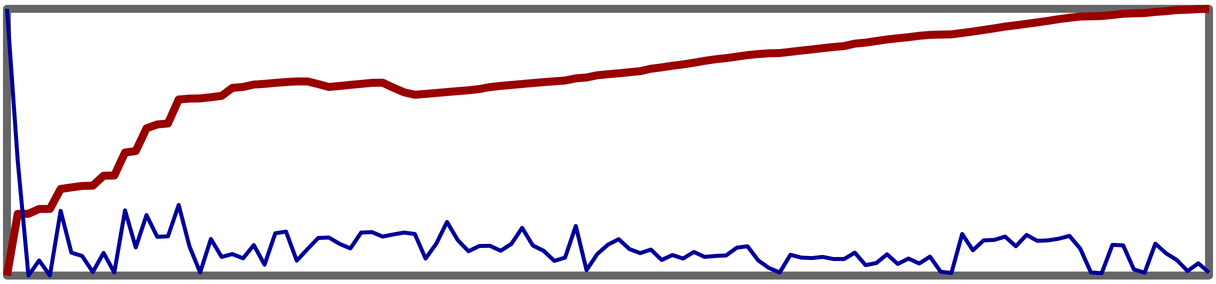
Keep in mind that these quotes in `tufte.tex` are just test samples, and not that realistic in everyday documents. On the other hand, these tests involve the usual font loading, node processing, attribute handling etc. They provide a decent baseline.

Another document that we use for testing functionality and performance is the reference manual. The preliminary beta 2 version gives the following statistics.



`luastate_bytes` `min:59690872, max:155651415, pages:112`

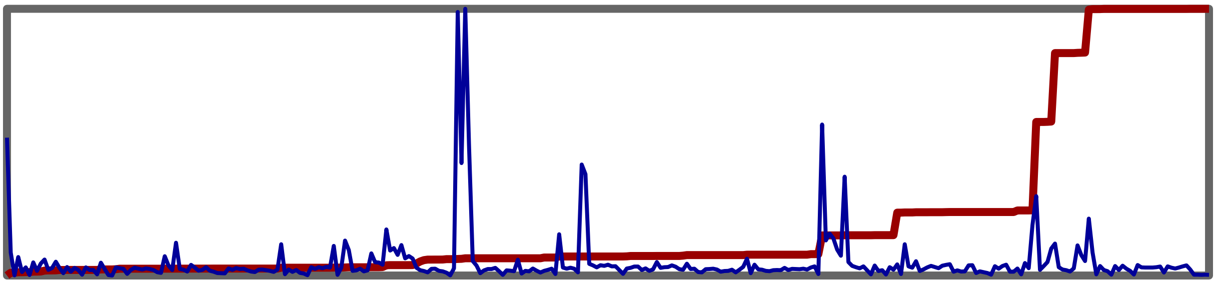
The previous graphic shows the statistics of a run with runtime MetaPost graphics enabled. This means that, because each pagenumber comes with a graphic, for each page MetaPost is called. The speed of this call is heavily influenced by the MetaPost startup time, which in turn (in a windows platform) is influences by the initialization time of the kpse library. Technically the call time can near zero but this demands sharing libraries and databases. Anyhow, we're moving towards an embedded MetaPost library anyway, and the next graphic shows what will happen then. Here we run ConTeXt in delayed MetaPost mode: graphics are collected and processed between runs. Where the runtime variant takes some 45 seconds processing time, the intermediate versions takes 15.



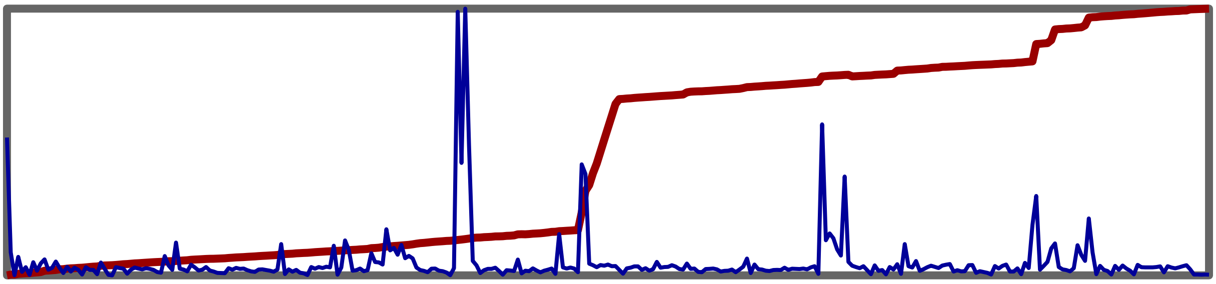
`luastate_bytes` `min:59690749, max:155669371, pages:112`

In the `mk.tex` document we use Type1 fonts for the main body of the text and load some (huge) OpenType fonts later on. Here we use OpenType fonts exclusively and since ConTeXt loads fonts only when needed, you see several spikes in the time per page bars and memory consumption quickly becomes stable. Interesting is that contrary to the `tufte.tex` samples, memory usage is quite stable. Here we don't have a memory sawtooth and no garbage collection spikes.

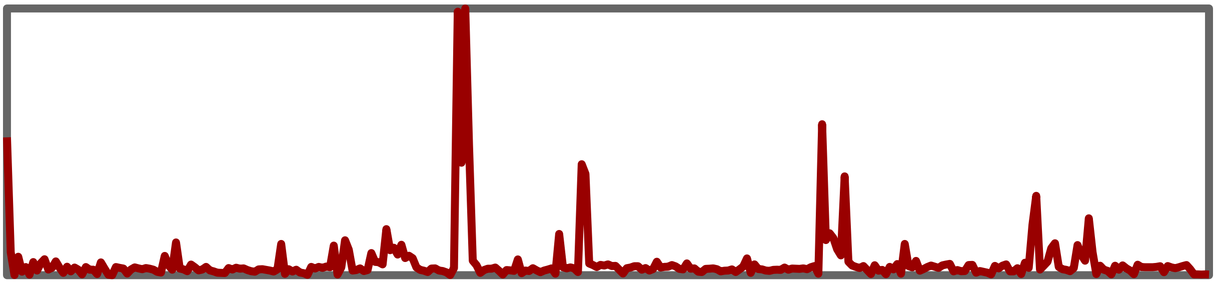
The previous graphics combine Lua memory consumption with time spent per page. The following graphics show variants of this. The graphics concern this document (`mk.tex`). Again, the blue lines represent the runtime per page.



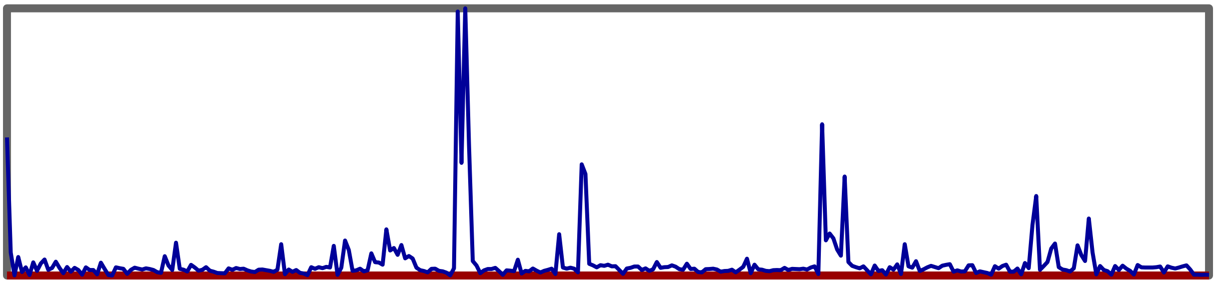
cs_count min:44951, max:52390, pages:320



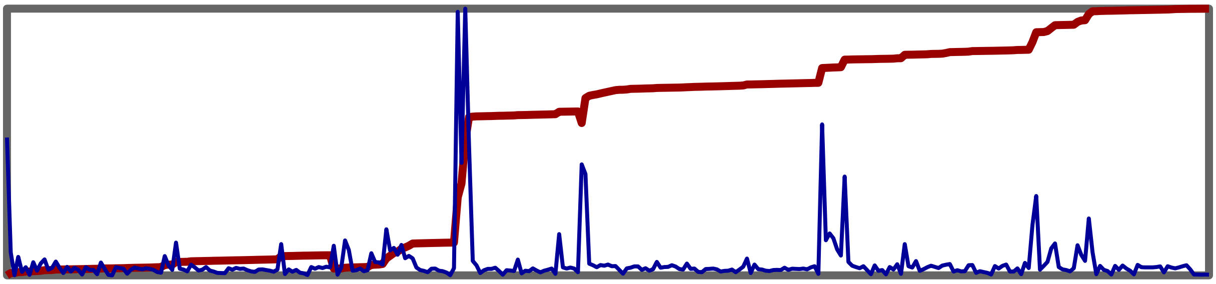
dyn_used min:632849, max:1959607, pages:320



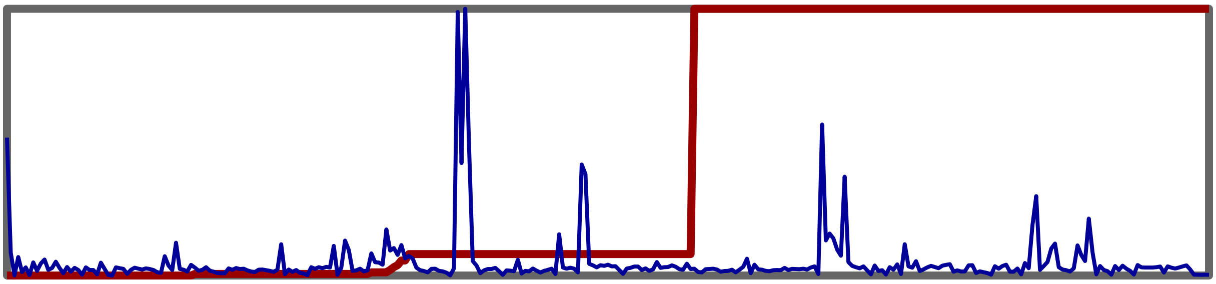
elapsed_time min:0.005, max:0.808, pages:320



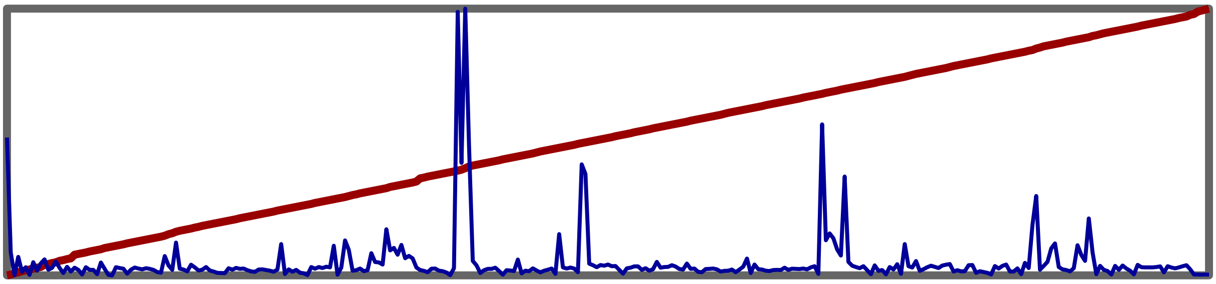
luabytecode_bytes min:21552, max:21552, pages:320



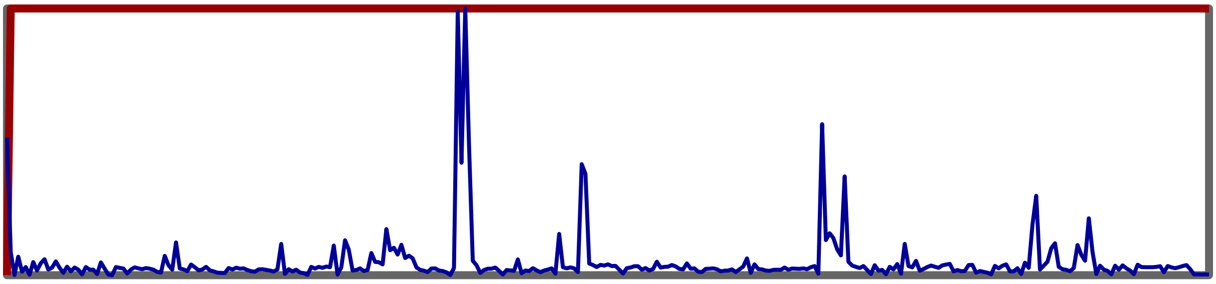
luastate_bytes min:76184711, max:795257742, pages:320



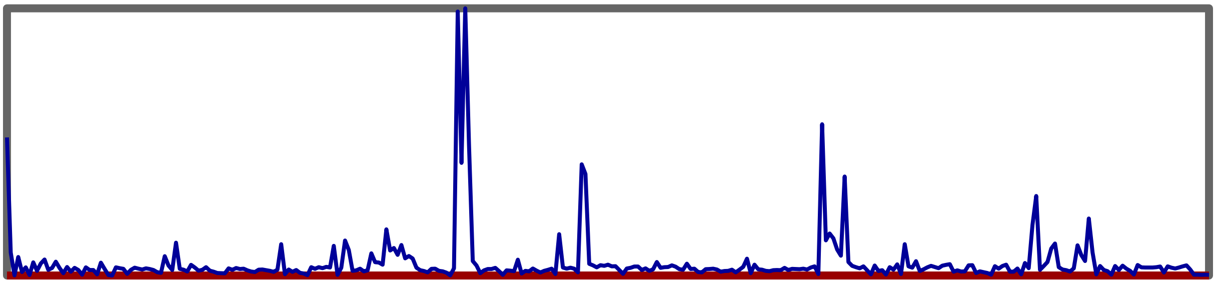
`max_buf_stack` min:302, max:2306, pages:320



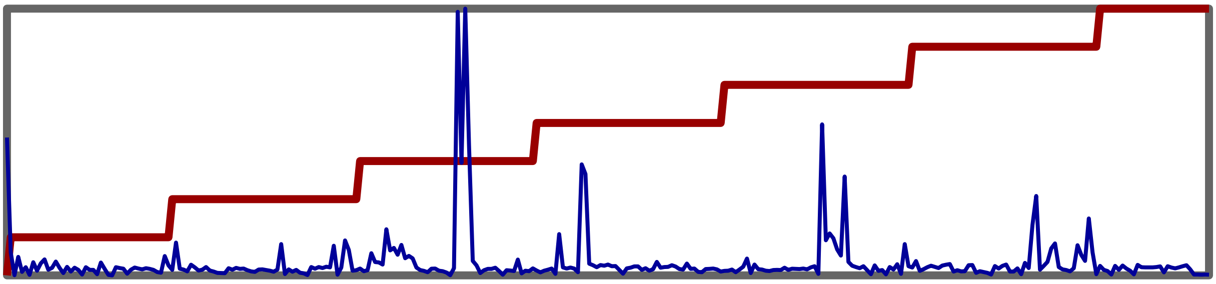
`obj_ptr` min:6, max:1089, pages:320



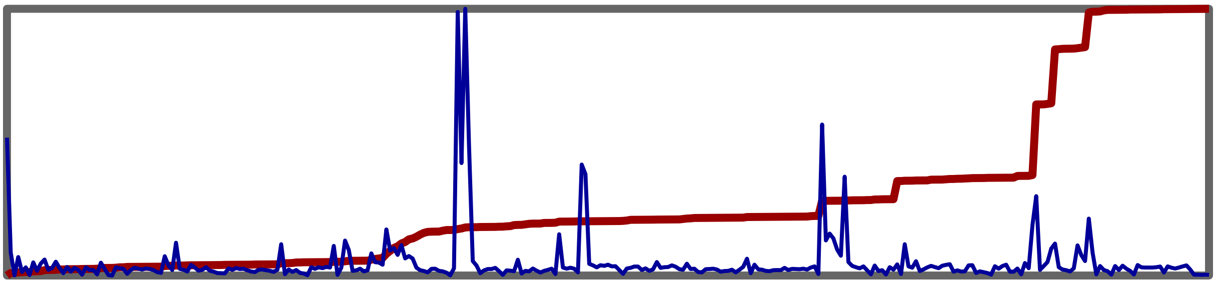
`pdf_mem_ptr` min:1, max:10, pages:320



`pdf_mem_size` min:10000, max:10000, pages:320

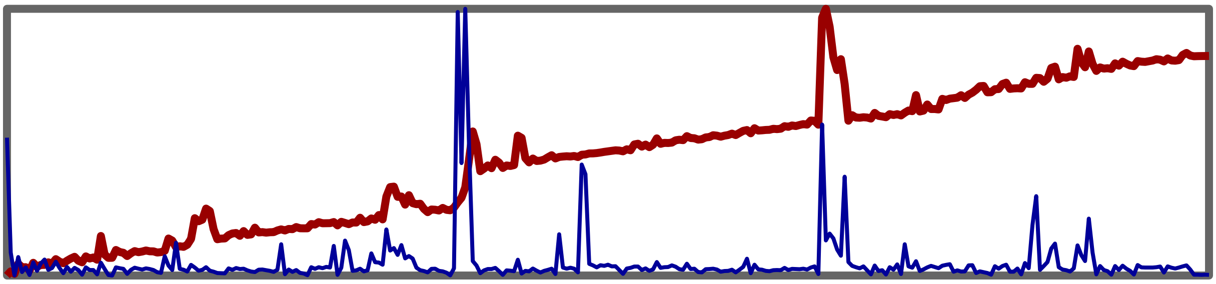


`pdf_os_cntr` min:0, max:7, pages:320

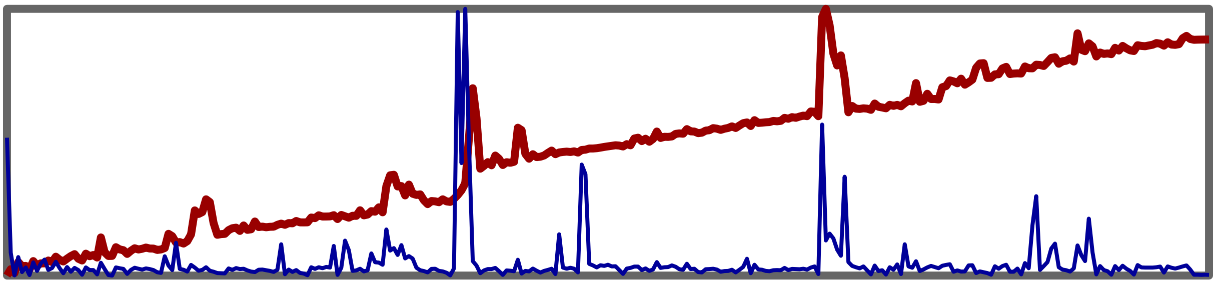


`str_ptr` min:2145595, max:2155120, pages:320

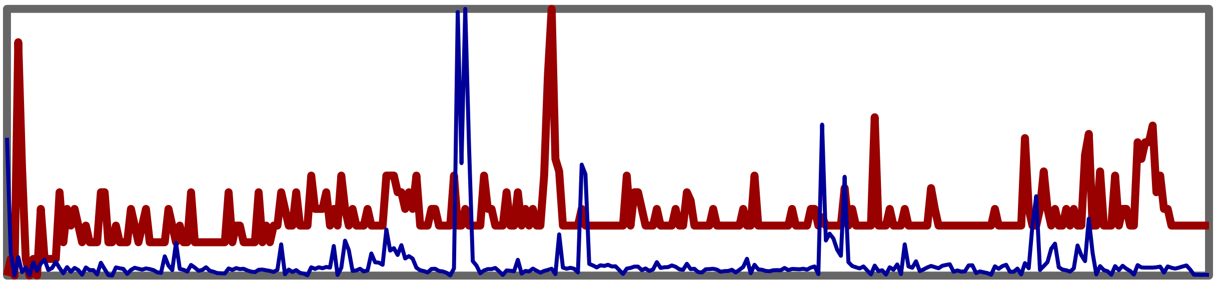
In LuaTeX node memory management is rewritten. Contrary to what you may expect, node memory consumption is not that large. Pages seldom contain more than 5000 nodes, although extensive use of attributes can easily duplicate this. Node usage in this documents is as follows.



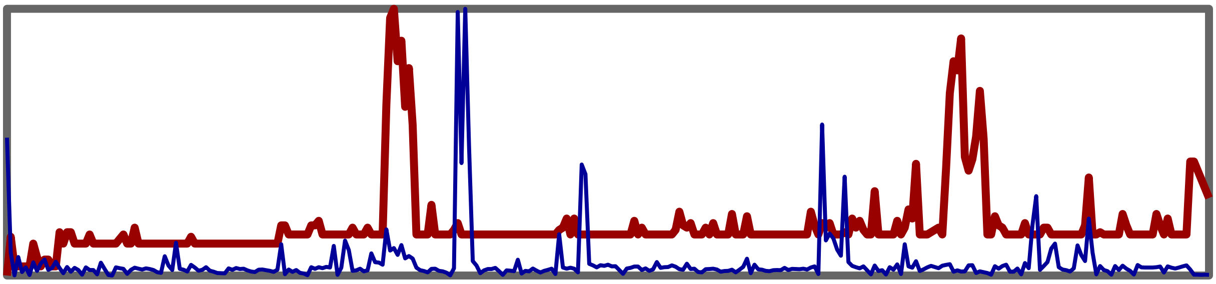
`attribute` min:33, max:33851, pages:320



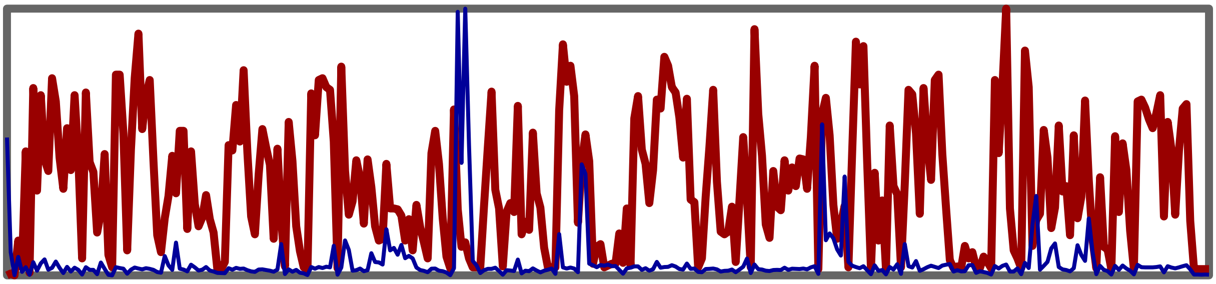
`attribute_list` min:13, max:7309, pages:320



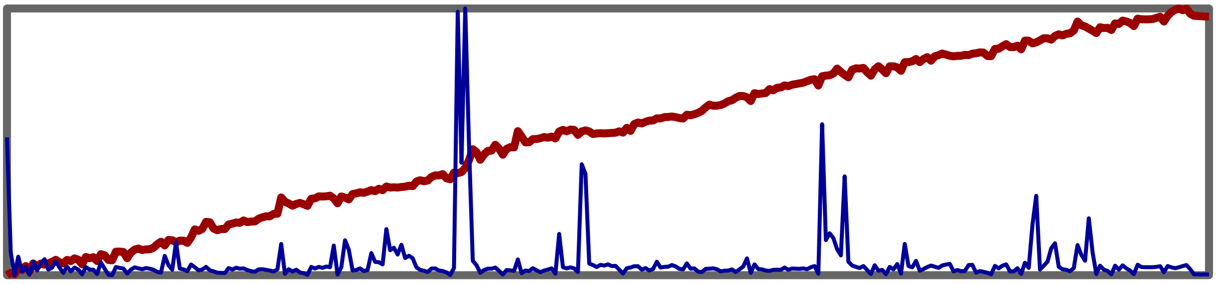
`boundary` min:2, max:66, pages:320



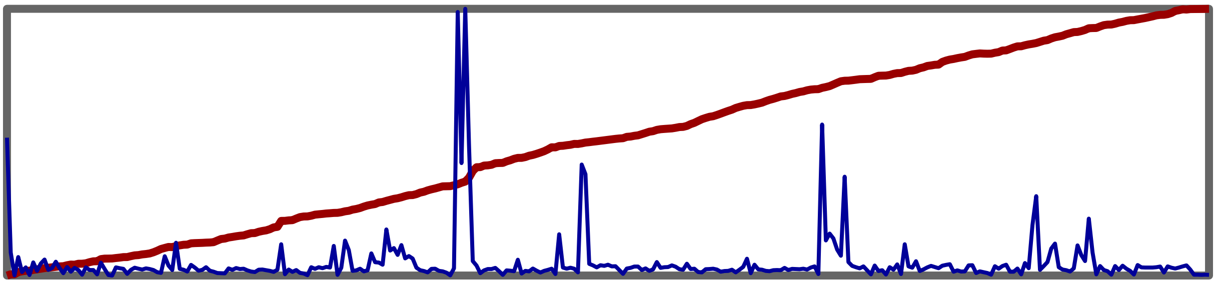
dir min:2, max:119, pages:320



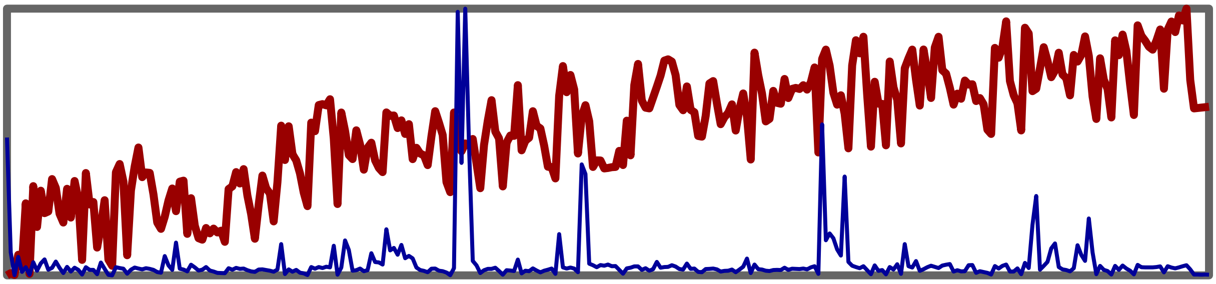
disc min:1, max:300, pages:320



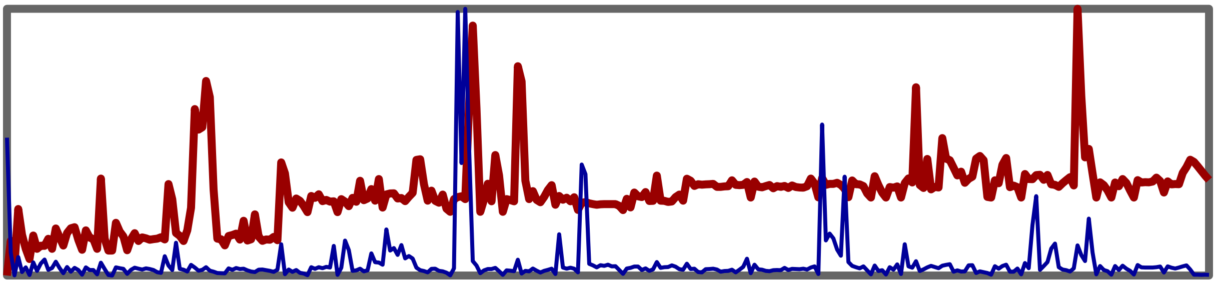
glue min:15, max:23120, pages:320



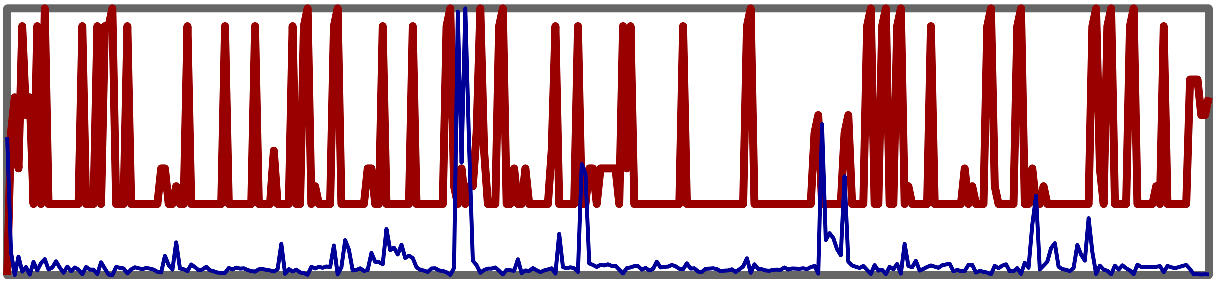
glue_spec min:38, max:6160, pages:320



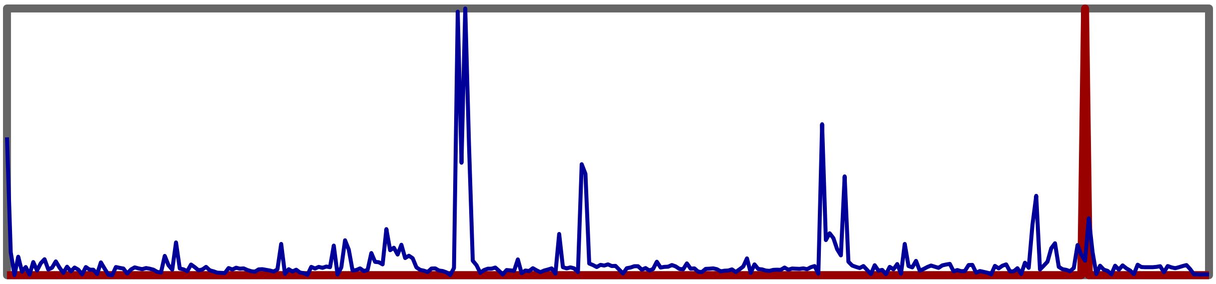
glyph min:2, max:7901, pages:320



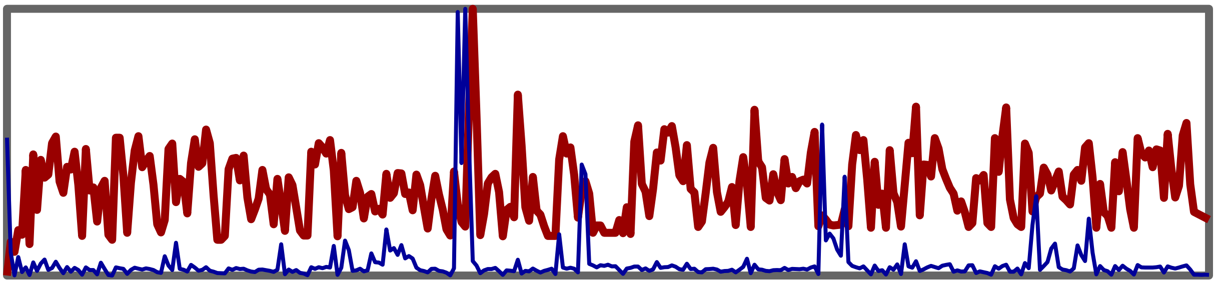
`hlist` min:5, max:1705, pages:320



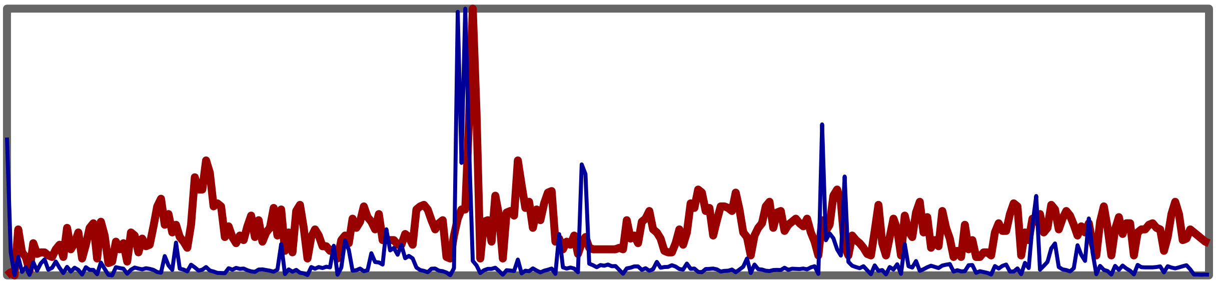
`if_stack` min:0, max:15, pages:320



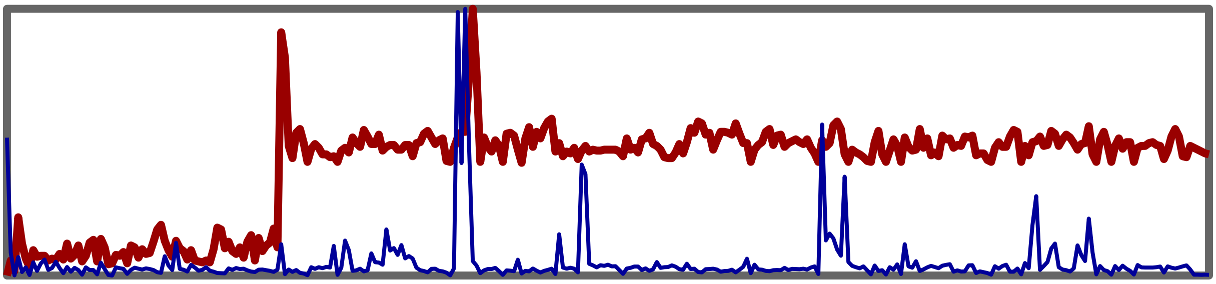
`ins` min:0, max:1, pages:320



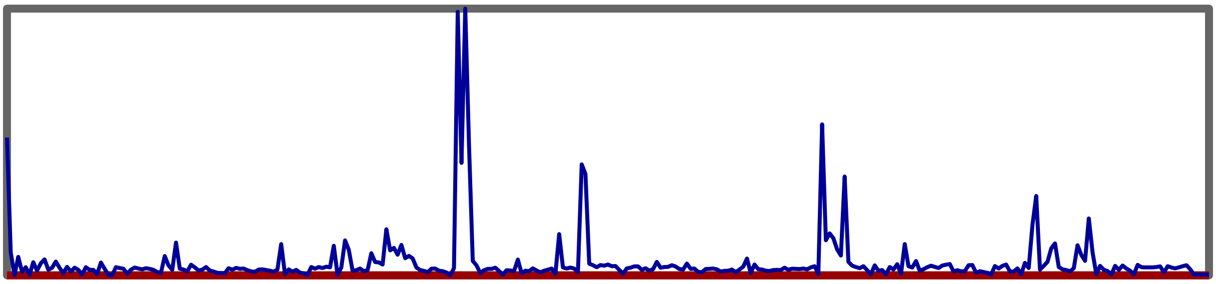
`kern` min:4, max:691, pages:320



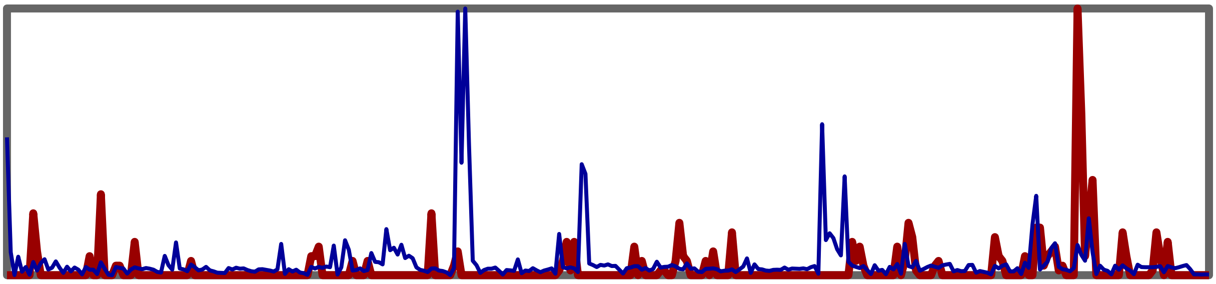
`late_lua` min:2, max:176, pages:320



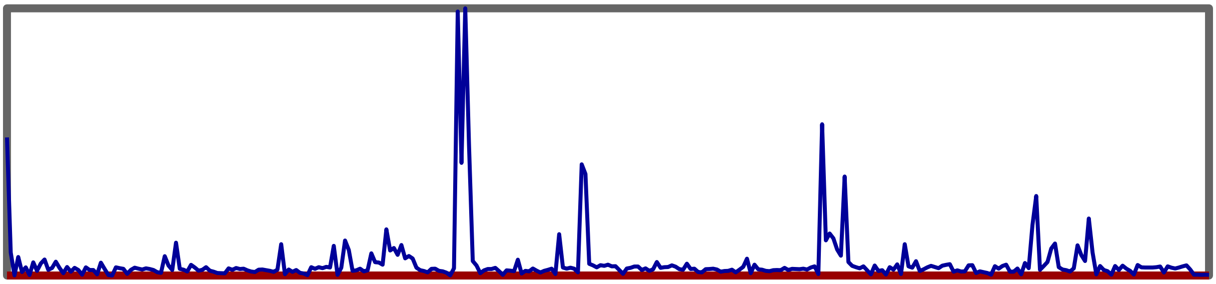
`local_par` min:0, max:284, pages:320



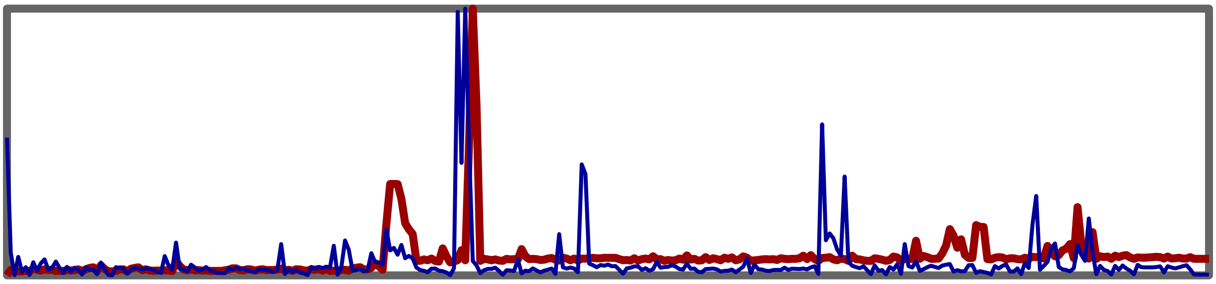
`margin_kern` min:2, max:2, pages:320



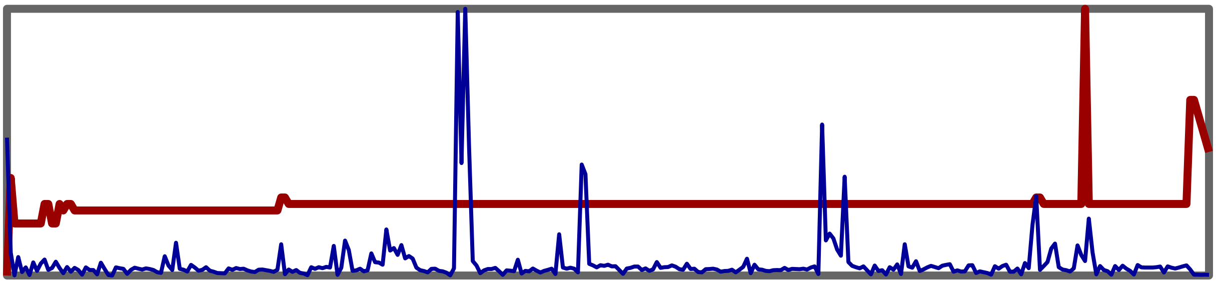
`math` min:0, max:112, pages:320



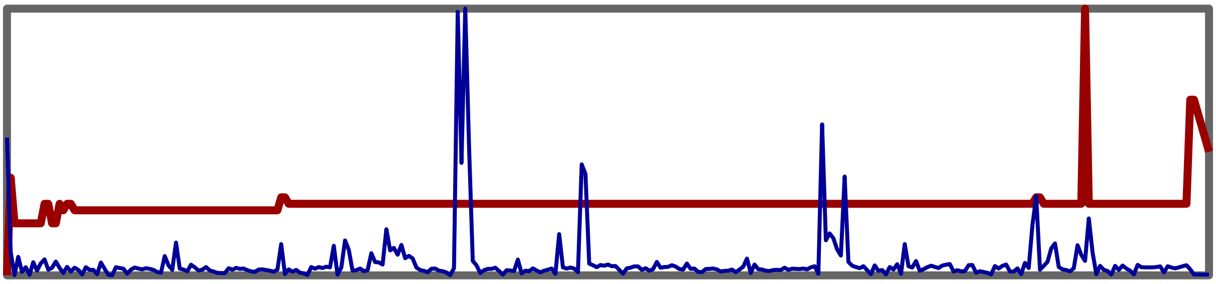
`noad` min:1, max:1, pages:320



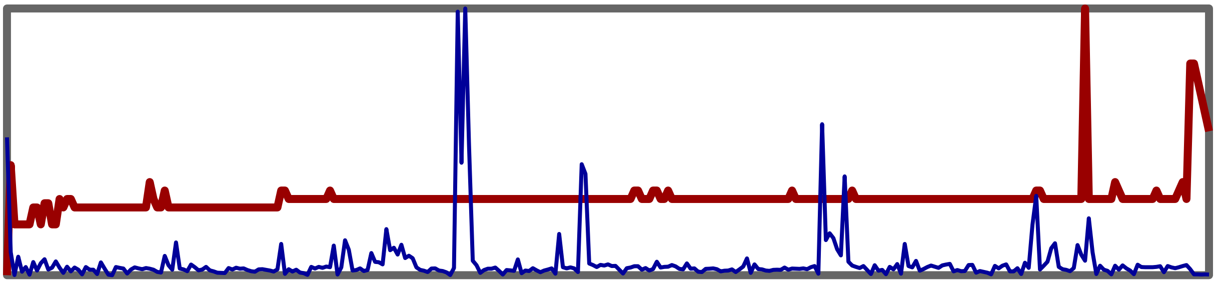
`pdf_literal` min:8, max:1864, pages:320



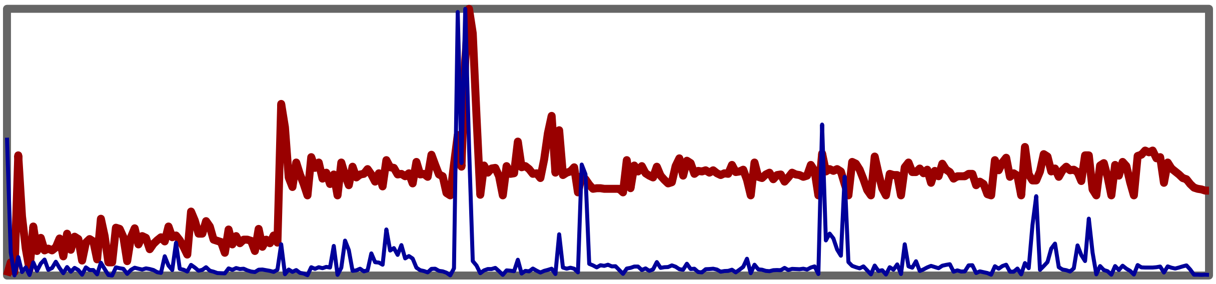
pdf_restore min:1, max:42, pages:320



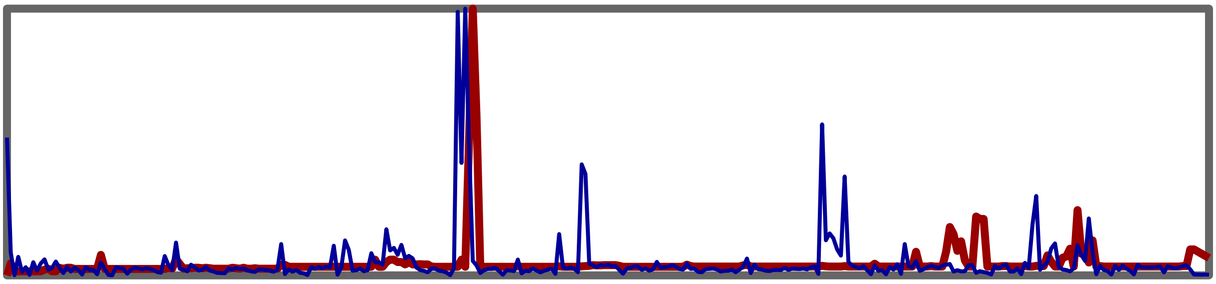
pdf_save min:1, max:42, pages:320



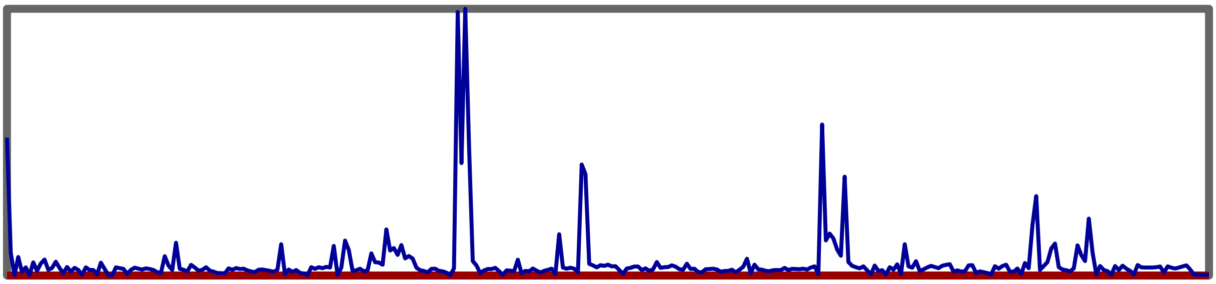
pdf_setmatrix min:1, max:64, pages:320



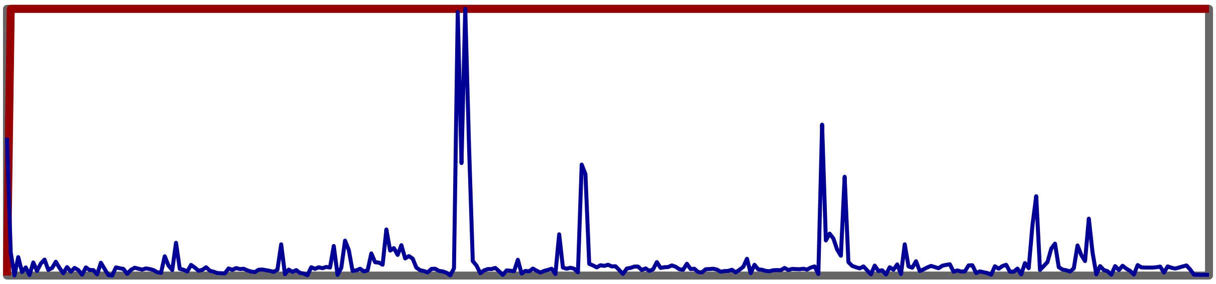
penalty min:1, max:447, pages:320



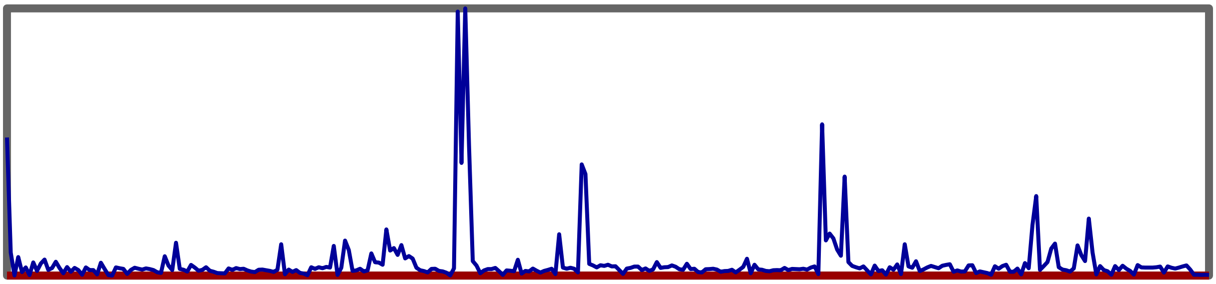
rule min:4, max:1124, pages:320



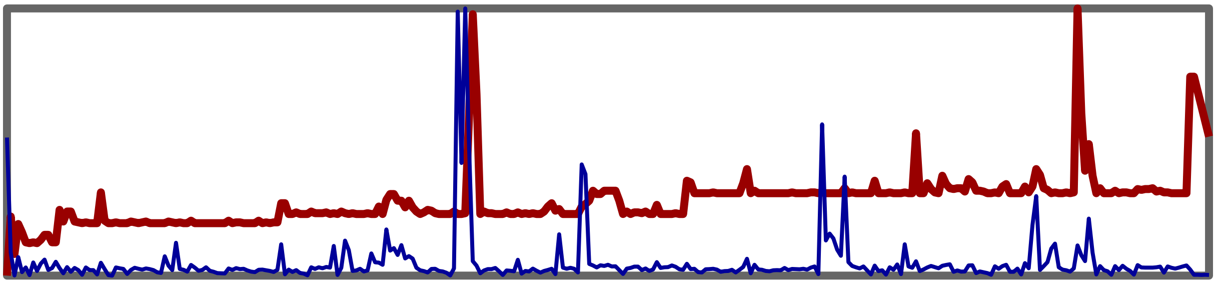
special
min:1, max:1, pages:320



temp
min:3, max:8, pages:320



user_defined
min:7, max:7, pages:320



vlist
min:6, max:327, pages:320

If node memory usage stays high, i.e. is not reclaimed, this can be an indication of a memory leak. In the December 2007 beta version there is such a leak in math subformulas, something that will be resolved when math node processing is opened up. The current MkIV code cleans up most of its temporary data. We do so, because it permits us to keep an eye on unwanted memory leaks. When writing this chapter, some of the peaks in the graphics coincided with peaks in the runtime per page, which is no surprise.

If you want to run such tests yourself, you need to load a module at startup:

```
\usemodule[timing]
```

The graphics can be generated with:

```
\def\ShowUsage      {optional filename}
\def\ShowNamedUsage {optional filename}{red graphic}{blue graphic}
\def\ShowMemoryUsage{optional filename}
\def\ShowNodeUsage  {optional filename}
```

(This interface may change.)

XX Nice to know

XX.I Tricky ligatures

Getting the 1.06 release of Latin Modern out in the wild took some discussion and testing. Not only were the names (internal names as well as file names) changed in such a way that multiple paplications could deal with it, but also some more advanced ligature trickery was added.

```
\definefontfeature
  [ijtest]
  [mode=node,
   script=latn,language=nld,strategy=3,
   liga=yes,kern=yes]

\definefont
  [ijfont]
  [name:lmroman10regular*ijtest at 36pt]

\start \ijfont \setstrut fijn ijsje fiets flink effe\stop
```

This bit of Dutch shows up as:

fijn ijsje fiets flink effe

Do you see the trick? There are both an ij and an fi ligature, but we need to prevent the ij ligature in fijn. Of course not all fonts have this feature, which indicated that you can never depend on it.

XX.II Herds

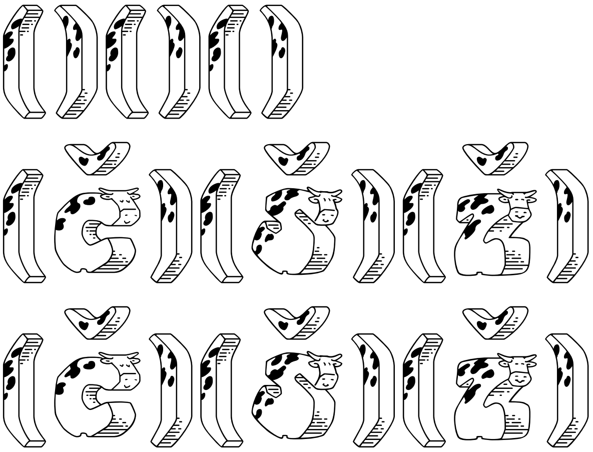
A while ago, Duane, Taco and I published the Cow Font. It's non–trivial to cook up a font made of cows, but of course Mojca Miklavec (who else) wants to typeset something Slovenian in this font. Now, the problem is that in MkIV we don't have fallback characters, or more precisely, we don't make UTF characters active and accent composing commands are mapped onto UTF.

This means that nothing will show up when we have no characters in the defined fonts. For the moment we stick to simple virtual fonts but because we can use node lists in virtual fonts, in the near future we will cook up a way to create arbitrary fallback characters.

The following example demonstrates how to 'complete' a font that misses glyphs.

```
\definefontfeature[coward] [kern=yes,ligatures=yes]
\definefontfeature[cowgirl][kern=yes,ligatures=yes,compose=yes]

\definefontsynonym [cows] [koeieletters.afm*coward]
\definefontsynonym [herd] [koeieletters.afm*cowgirl]

\blank[3*medium]
\dontleavehmode\hbox{\definedfont[cows sa 5](č)(š)(ž)}
\blank[3*medium]
\dontleavehmode\hbox{\definedfont[herd sa 5](č)(š)(ž)}
\blank[3*medium]
\dontleavehmode\hbox{\definedfont[herd sa 5](\v{c})(\v{s})(\v{z})}
```

As expected (at least by me) the first line has no compose characters.



XXI The luafication of TEX and CONTEXT

introduction

Here I will present the current stage of LUATEX around beta stage 2, and discuss the impact so far on CONTEXT MKIV that we use as our testbed. I'm writing this at the end of February 2008 as part of the series of regular updates on LUATEX. As such, this report is part of our more or less standard test document (`mk.tex`). More technical details can be found in the reference manual that comes with LUATEX. More information on MKIV is available in the CONTEXT mailing lists, WIKI, and `mk.pdf`.

For those who never heard of LUATEX: this is a new variant of TEX where several long pending wishes are fulfilled:

- combine the best of all TEX engines
- add scripting capabilities
- open up the internals to the scripting engine
- enhance font support to OPENTYPE
- move on to UNICODE
- integrate METAPOST

There are a few more wishes, like converting the code base to C but these are long term goals.

The project started a few years ago and is conducted by Taco Hoekwater (PASCAL and C coding, code base management, reference manual), Hartmut Henkel (PDF backend, experimental features) and Hans Hagen (general overview, LUA and TEX coding, website). The code development got a boost by a grant of the Oriental TEX project (project lead: Idris Samawi Hamid) and funding via the TUG. The related MPLIB project by the same team is also sponsored by several user groups. The very much needed OPENTYPE fonts are also a user group funded effort: the Latin Modern and TEX Gyre projects (project leads: Jerzy Ludwichowski, Volker RW Schaa and Hans Hagen), with development (the real work) by: Bogusław Jackowski and Janusz Nowacki.

One of our leading principles is that we focus on opening up. This means that we don't implement solutions (which also saves us many unpleasant and everlasting discussions). Implementing solutions is up to the user, or more precisely: the macro package writer, and since there are many solutions possible, each can do it his or her way. In that sense we follow the footsteps of Don Knuth: we make an extensible tool, you are free to like it or not, you can take it and extend it where needed, and there is no need to bother us (unless of course you find bugs or weird side effects). So far this has worked out quite well and we're confident that we can keep our schedule.

We do our tests of a variant of ConTEXt tagged MkIV, especially meant for LuaTEX, but LuaTEX itself is in no way limited to or tuned for ConTEXt. Large chunks of the code written for MkIV are rather generic and may eventually be packaged as a base system (especially font handling) so that one can use LuaTEX in rather plain mode. To a large extent MkIV will be functionally compatible with MkII, the version meant for traditional TEX, although it knows how to profit from XeTEX. Of course the expectation is that certain things can be done better in MkIV than in MkII.

status

By the end of 2007 the second major beta release of LuaTEX was published. In the first quarter of 2008 Taco would concentrate on mplib, Hartmut would come up with the first version of the image library while I could continue working on MkIV and start using LuaTEX in real projects. Of course there is some risk involved in that, but since we have a rather close loop for critical bug fixes, and because I know how to avoid some dark corners, the risk was worth taking.

What did we accomplish so far? I can best describe this in relation to how ConTEXt MkIV evolved and will evolve. Before we do this, it makes sense to spend some words on why we started working on MkIV in the first place.

When the LuaTEX project started, ConTEXt was about 10 years in the field. I can safely say that we were still surprised by the fact that what at first sight seems unsolvable in TEX somehow could always be dealt with. However, some of the solutions were rather tricky. The code evolved towards a more or less stable state, but sometimes depended on controlled processing. Take for instance backgrounds that can span pages and columns, can be nested and can have arbitrary shapes. This feature has been present in ConTEXt for quite a while, but it involves an interplay between TEX and MetaPost. It depends on information collected in a previous run as well as (at runtime or not) processing of graphics.

This means that by now ConTEXt is not just a bunch of TEX macros, but also closely related to MetaPost. It also means that processing itself is by now rather controlled by a wrapper, in the case of MkII called TEXexec. It may sound complicated, but the fact that we have implemented workflows that run unattended for many years and involve pretty complex layouts and graphic manipulations demonstrates that in practice it's not as bad as it may sound.

With the arrival of LuaTEX we not only have a rigourously updated TEX engine, but also get MetaPost integrated. Even better, the scripting language Lua is not only used for opening up TEX, but is also used for all kind of management tasks. As a result, the development of MkIV not only concerns rewriting whole chunks of ConTEXt, but also results in a set of new utilities and a rewrite of existing ones. Since dealing with MkIV will demand some changes in the way users deal with ConTEXt I will discuss some of them first. It also demonstrates that LuaTEX is more than just TEX.

utilities

There are two main scripts: LUATOOLS and MTXRUN. The first one started as a replacement for KPSEWHICH but evolved into a base tool for generating (TDS) file databases and generating formats. In MkIV we replace the regular file searching, and therefore we use a different database model. That's the easy part. More tricky is that we need to bootstrap MkIV into this alternative mode and when doing so we don't want to use the `kpse` library because that would trigger loading of its databases. To discuss the gory details here might cause users to refrain from using LuaTeX so we stick to a general description.

- When generating a format, we also generate a bootstrap Lua file. This file is compiled to bytecode and is put alongside the format file. The libraries of this bootstrap file are also embedded in the format.

- When we process a document, we instruct LuaTeX to load this bootstrap file before loading the format. After the format is loaded, we re-initialize the embedded libraries. This is needed because at that point more information may be available than at loading time. For instance, some functionality is available only after the format is loaded and LuaTeX enters the TeX state.

- File databases, formats, bootstrap files, and runtime-generated cached data is kept in a TDS tree specific cache directory. For instance, OpenType font tables are stored on disk so that next time loading them is faster.

Starting LuaTeX and MkIV is done by LUATOOLS. This tool is generic enough to handle other formats as well, like MPTOPDF or Plain. When you run this script without argument, you will see:

```
version 1.1.1 - 2006+ - PRAGMA ADE / CONTEXT

--generate        generate file database
--variables       show configuration variables
--expansions      show expanded variables
--configurations  show configuration order
--expand-braces   expand complex variable
--expand-path     expand variable (resolve paths)
--expand-var      expand variable (resolve references)
--show-path       show path expansion of ...
--var-value       report value of variable
--find-file       report file location
--find-path       report path of file
--make or --ini   make luatex format
--run or --fmt=   run luatex format
--luafile=str     lua inifile (default is <progname>.lua)
```

```
--lualibs=list    libraries to assemble (optional)
--compile         assemble and compile lua inifile
--verbose         give a bit more info
--minimize        optimize lists for format
--all             show all found files
--sort            sort cached data
--engine=str      target engine
--progname=str    format or backend
--pattern=str     filter variables
--lsr             use lsr and cnf directly
```

For the Lua based file searching, luatools can be seen as a replacement for mktexlsr and kpsewhich and as such it also recognizes some of the kpsewhich flags. The script is self contained in the sense that all needed libraries are embedded. As a result no library paths need to be set and packaged. Of course the script has to be run using LuaTeX itself. The following commands generate the file databases, generate a ConTeXt MkIV format, and process a file:

```
luatools --generate
luatools --make --compile cont-en
luatools --fmt=cont-en somefile.tex
```

There is no need to install Luain order to run this script. This is because LuaTeX can act as such with the advantage that the built-in libraries are available too, for instance the Lua file system `lfs`, the zip file manager `zip`, the Unicode libary `unicode`, `md5`, and of course some of our own.

luatex	a Lua–enhanced TeX engine
texlua	a Lua engine enhanced with some libraries
texluac	a Lua bytecode compiler enhanced with some libraries

In principle `luatex` can perform all tasks but because we need to be downward compatible with respect to the command line and because we want Lua compatible variants, you can copy or symlink the two extra variants to the main binary.

The second script, mtxrun, can be seen as a replacement for the Ruby script texmfstart, a utility whose main task is to launch scripts (or documents or whatever) in a tds tree. The mtxrun script makes it possible to get away from installing Ruby and as a result a regular TeX installation can be made independent of scripting tools.

```
version 1.0.2 - 2007+ - PRAGMA ADE / CONTEXT

--script                run an mtx script
```

```
--execute             run a script or program
--resolve             resolve prefixed arguments
--ctxlua              run internally (using preloaded libs)
--locate              locate given filename

--autotree            use texmf tree cf.\ environment settings
--tree=pathtotree     use given texmf tree (def: 'setuptex.tmf')
--environment=name    use given (tmf) environment file
--path=runpath        go to given path before execution
--ifchanged=filename  only execute when given file has changed
--iftouched=old,new   only execute when given file has changed

--make                create stubs for (context related) scripts
--remove              remove stubs (context related) scripts
--stubpath=binpath    paths where stubs wil be written
--windows             create windows (mswin) stubs
--unix                create unix (linux) stubs

--verbose             give a bit more info
--engine=str          target engine
--progname=str        format or backend

--edit                launch editor with found file
--launch (--all)      launch files (assume os support)

--intern              run script using built-in libraries
```

This help information gives an impression of what the script does: running other scripts, either within a certain TDS tree or not, and either conditionally or not. Users of ConTeXt will probably recognize most of the flags. As with texmfstart, arguments with prefixes like `file:` will be resolved before being passed to the child process.

The first option, `--script` is the most important one and is used like:

```
mtxrun --script fonts --reload
mtxrun --script fonts --pattern=lm
```

In MkIV you can access fonts by filename or by font name, and because we provide several names per font you can use this command to see what is possible. Patterns can be Lua expressions, as demonstrated here:

```
mtxrun --script font  --list --pattern=lmtype.*regular

lmtypewriter10-capsregular   LMTypewriter10-CapsRegular   lmtypewriter10-cap
```

```
lmtypewriter10-regular       LMTypewriter10-Regular       lmtypewriter10-reg
lmtypewriter12-regular       LMTypewriter12-Regular       lmtypewriter12-reg
lmtypewriter8-regular        LMTypewriter8-Regular        lmtypewriter8-regu
lmtypewriter9-regular        LMTypewriter9-Regular        lmtypewriter9-regu
lmtypewritervarwd10-regular  LMTypewriterVarWd10-Regular  lmtypewritervarwd1
```

A simple

```
mtxrun --script fonts
```

gives:

```
version 1.0.2 - 2007+ - PRAGMA ADE / CONTEXT | font tools

--reload              generate new font database
--list                list installed fonts
--save                save open type font in raw table

--pattern=str         filter files
--all                 provide alternatives
```

In MkIV font names can be prefixed by `file:` or `name:` and when they are resolved, several attempts are made, for instance non-characters are ignored. The `--all` flag shows more variants.

Another example is:

```
mtxrun --script context --ctx=somesetup somefile.tex
```

Again, users of TeXexec may recognize part of this and indeed this is its replacement. Instead of TeXexec we use a script named `mtx-context.lua`. Currently we have the following scripts and more will follow:

The `babel` script is made in cooperation with Thomas Schmitz and can be used to convert babelized Greek files into proper utf. More of such conversions may follow. With `cache` you can inspect the content of the MkIV cache and do some cleanup. The `chars` script is used to construct some tables that we need in the process of development. As its name says, `check` is a script that does some checks, and in particular it tries to figure out if TeX files are correct. The already mentioned `context` script is the MkIV replacement of TeXexec, and takes care of multiple runs, preloading project specific files, etc. The `convert` script will replace the Ruby script `pstopdf`.

A rather important script is the already mentioned `fonts`. Use this one for generating font name databases (which then permits a more liberal access to fonts) or identifying installed fonts. The `unzip` script indeed unzips archives. The `update` script is still somewhat experimental and is one of the building blocks of the ConTeXt minimal installer system by Mojca Miklavec and Arthur Reutenauer. This update script synchronizes a local tree with a repository and keeps an installation as small as possible, which for instance means: no OpenType fonts for pdfTeX, and no redundant Type1 fonts for LuaTeX and XeTeX.

The (for the moment) last two scripts are `watch` and `web`. We use them in (either automated or not) remote publishing workflows. They evolved out of the eXaMpLe framework which is currently being reimplemented in Lua.

As you can see, the LuaTeX project and its ConTeXt companion MkIV project not only deal with TeX itself but also facilitates managing the workflows. And the next list is just a start.

context	controls processing of files by MkIV
babel	conversion tools for LaTeX files
cache	utilities for managing the cache
chars	utilities used for MkIV development
check	TeX syntax checker
convert	helper for some basic graphic conversion
fonts	utilities for managing font databases
update	tool for installing minimal ConTeXt trees
watch	hot folder processing tool
web	utilities related to automate workflows

There will be more scripts. These scripts are normally rather small because they hook into mtxrun which provides the libraries. Of course existing tools remain part of the toolkit. Take for instance ctxtools, a Ruby script that converts font encoded pattern files to generic utf encoded files.

Those who have followed the development of ConTeXt will notice that we moved from utilities written in Modula to tools written in Perl. These were later replaced by Ruby scripts and eventually most of them will be rewritten in Lua.

macros

I will not repeat what is said already in the MkIV related documents, but stick to a summary of what the impact on ConTeXt is and will be. From this you can deduce what the possible influence on other macro packages can be.

Opening up TeX started with rewriting all io related activities. Because we wanted to be able to read from zip files, the web and more, we moved away from the traditional kpse based file handling. Instead MkIV uses an extensible variant written in Lua. Because we need to be downward compatible, the code is somewhat messy, but it does the job, and pretty quickly and efficiently too. Some alternative input media are implemented and many more can be added. In the beginning I permitted several ways to specify a resource but recently a more restrictive URL syntax was imposed. Of course the file locating mechanisms provide the same control as provided by the file readers in MkII.

An example of reading from a ZIP file is:

```
\input zip:///archive.zip?name=blabla.tex
\input zip:///archive.zip?name=/somepath/blabla.tex
```

In addition one can register files, like:

```
\usezipfile[archive.zip]
\usezipfile[tex.zip][texmf-local]
\usezipfile[tex.zip?tree=texmf-local]
```

The last two variants register a zip file in the TDS structure where more specific lookup rules apply. The files in a registered file are known to the file searching mechanism so one can give specifications like the following:

```
\input */blabla.tex
\input */somepath/blabla.tex
```

In a similar fashion one can use the `http`, `ftp` and other protocols. For this we use independent fetchers that cache data in the MkIV cache. Of course, in more structured projects, one will seldom use the `\input` command but use a project structure instead.

Handling of files rather quickly reached a stable state, and we seldom need to visit the code for fixes. Already after a few years of developing the first code for LuaTeX we reached a state of 'Hm, when did I write this?'. When we have reached a stable state I foresee that much of the older code will need a cleanup.

Related to reading files is the sometimes messy area of input regimes (file encoding) and font encoding, which itself relates to dealing with languages. Since LuaTeX is UTF-8 based, we need to deal with file encoding issues in the frontend, and this is what Lua based file handling does. In practice users of LuaTeX will swiftly switch to UTF anyway but we provide regime control for historic reasons. This time the recoding tables are Lua based and as a result MkIV has no regime files. In a similar fashion font encoding is gone: there is still some old code that deals with default fallback characters, but most of the files are gone. The same will be true for math encoding. All information is now stored in a character table which is the central point in many subsystems now.

It is interesting to notice that until now users have never asked for support with regards to input encoding. We can safely assume that they just switched to UTF and recoded older documents. It is good to know that LuaTeX is mostly PDFTeX but also incorporates some features of Omega. The main reason for this is that the Oriental TeX project needed bidirectional typesetting and there was a preference for this implementation over the one provided by ε-TEX. As a side effect input translation is also present, but since no one seems to use it, that may as well go away. In MkIV we refrain from input processing as much as possible and focus on processing the node lists. That way there is no interference between user data, macro expansion and whatever may lead to the final data that ends up in the to-be-typeset stream. As said, users seem to be happy to use UTF as input, and so there is hardly any need for manipulations.

Related to processing input is verbatim: a feature that is always somewhat complicated by the fact that one wants to typeset a manual about TEX in TEX and therefore needs flexible escapes from illustrative as well as real TEX code. In MkIV verbatim as well as all buffering of data is dealt with in LUA. It took a while to figure out how LUATEX should deal with the concept of a line ending, but we got there. Right from the start we made sure that LUATEX could deal with collections of catcode settings (those magic states that characters can have). This means that one has complete control at both the TEX and LUA end over the way characters are dealt with.

In MkIV we also have some pretty printing features, but many languages are still missing. Cleaning up the premature verbatim code and extending pretty printing is on the agenda for the end of 2008.

Languages also are handled differently. A major change is that pattern files are no longer preloaded but read in at runtime. There is still some relation between fonts and languages, no longer in the encoding but in dealing with OPENTYPE features. Later we will do a more drastic overhaul (with multiple name schemes and such). There are a few experimental features, like spell checking.

Because we have been using UTF encoded hyphenation patterns for quite some time now, and because CONTEXT ships with its own files, this transition probably went unnoticed, apart maybe from a faster format generation and less startup time.

Most of these features started out as an experiment and provided a convenient way to test the LUATEX extensions. In MkIV we go quite far in replacing TEX code by LUA, and how far one goes is a matter of taste and ambition. An example of a recent replacement is graphic inclusion. This is one of the oldest mechanisms in CONTEXT and it has been extended many times, for instance by plugins that deal with figure databases (selective filtering from PDF files made for this purpose), efficient runtime conversion, color conversion, downsampling and product dependent alternatives.

One can question if a properly working mechanism should be replaced. Not only is there hardly any speed to gain (after all, not that many graphics are included in documents), a LUA–TEX mix may even look more complex. However, when an opened-up TEX keeps evolving at the current pace, this last argument becomes invalid because we can no longer give that TEXie code to LUA. Also, because most of the graphic inclusion code deals with locating files and figuring out the best quality variant, we can benefit much from LUA: file handling is more robust, the code looks cleaner, complex searches are faster, and eventually we can provide way more clever lookup schemes. So, after all, switching to LUA here makes sense. A nice side effect is that some of the mentioned plugins now take a few lines of extra code instead of many lines of TEX. At the time of writing this, the beta version of MKIV has LUA based graphic inclusion.

A disputable area for Luafication is multipass data. Most of that has already been moved to LUA files instead of TEX files, and the rest will follow: only tables of contents still use a TEX auxiliary file. Because at some point we will reimplement the whole section numbering and cross referencing, we postponed that till later. The move is disputable because in the end, most data ends up in TEX again, which involves some conversion. However, in LUA we can store and manipulate information much more easily and so we decided to follow that route. As a start, index information is now kept in LUA tables, sorted on demand, depending on language needs and such. Positional information used to take up much hash space which could deplete the memory pool, but now we can have millions of tracking points at hardly any cost.

Because it is a quite independent task, we could rewrite the METAPOST conversion code in LUA quite early in the development. We got smaller and cleaner code, more flexibility, and also gained some speed. The code involved in this may change as soon as we start experimenting with MPLIB. Our expectations are high because in a bit more modern designs a graphic engine cannot be missed. For instance, in educational material, backgrounds and special shapes are all over the place, and we're talking about many METAPOST runs then. We expect to bring down the processing time of such documents considerably, if only because the METAPOST runtime will be close to zero (as experiments have shown us).

While writing the code involved in the METAPOST conversion a new feature showed up in LUA: `lpeg`, a parsing library. From that moment on `lpeg` was being used all over the place, most noticeably in the code that deals with processing XML. Right from the start I had the feeling that LUA could provide a more convenient way to deal with this input format. Some experiments with rewriting the MKII mechanisms did not show the expected speedup and were abandoned quickly.

Challenged by `lpeg` I then wrote a parser and started playing with a mixture of a tree based and stream approach to XML (MKII is mostly stream based). Not only is loading XML code extremely fast (we used 40 megaByte files for testing), dealing with the tree is also convenient. The additional MKIV methods are currently being tested in real projects and so far they result in an acceptable and pleasant mix of TEX and XML. For instance, we can now selectively process parts of the tree using path expressions, hook in code, manipulate data, etc.

The biggest impact of LUATEX on the CONTEXT code base is not the previously mentioned mechanisms but one not yet mentioned: fonts. Contrary to XETEX, which uses third party libraries, LuaTEX does not implement dealing with font specific issues at all. It can load several font formats and accepts font data in a well-defined table format. It only processes character nodes into glyph nodes and it's up to the user to provide more by manipulating the node lists. Of course there is still basic ligature building and kerning available but one can bypass that with other code.

In MkIV, when we deal with Type1 fonts, we try to get away from traditional TFM files and use AFM files instead (indeed, we parse them using `lpeg`). The fonts are mapped onto Unicode. Awaiting extensions of math we only use TFM files for math fonts. Of course OpenType fonts are dealt with and this is where we find most Lua code in MkIV: implementing features. Much of that is a grey area but as part of the Oriental TEX project we're forced to deal with complex feature support, so that provides a good test bed as well as some pressure for getting it done. Of course there is always the question to what extent we should follow the (maybe faulty) other programs that deal with font features. We're lucky that the Latin Modern and TEX Gyre projects provide real fonts as well as room for discussion and exploring these grey areas.

In parallel to writing this, I made a tracing feature for Oriental TEXer Idris so that he could trace what happened with the Arabic fonts that he is making. This was relatively easy because already in an early stage of MkIV some debugging mechanisms were built. One of its nice features is that on an error, or when one traces something, the results will be shown in a web browser. Unfortunately I have not enough time to explore such aspects in more detail, but at least it demonstrates that we can change some aspects of the traditional interaction with TEX in more radical ways.

Many users may be aware of the existence of so-called virtual fonts, if only because it can be a cause of problems (related to map files and such). Virtual fonts have a lot of potential but because they were related to TEX's own font data format they never got very popular. In LuaTEX we can make virtual fonts at runtime. In MkIV for instance we have a feature (we provide features beyond what OpenType does) that completes a font by composing missing glyphs on the fly. More of this trickery can be expected as soon as we have time and reason to implement it.

In PDFTEX we have a couple of font related goodies, like character expansion (inspired by Hermann Zapf) and character protruding. There are a few more but these had limitations and were suboptimal and therefore have been removed from LuaTEX. After all, they can be implemented more robustly in Lua. The two mentioned extensions have been (of course) kept and have been partially reimplemented so that they are now uniquely bound to fonts (instead of being common to fonts that traditional TEX shares in memory). The character related tables can be filled with Lua and this is what MkIV now does. As a result much TEX code could go away. We still use shape related vectors to set up the values, but we also use information stored in our main character database.

A likely area of change is math and not only as a result of the TEX gyre math project which will result in a bunch of UNICODE compliant math fonts. Currently in MKIV the initialization already partly takes place using the character database, and so again we will end up with less TEX code. A side effect of removing encoding constraints (i.e. moving to UNICODE) is that things get faster. Later this year math will be opened up.

One of the biggest impacts of opening up is the arrival of attributes. In traditional TEX only glyph nodes have an attribute, namely the font id. Now all nodes can have attributes, many of them. We use them to implement a variety of features that already were present in MKII, but used marks instead: color (of course including color spaces and transparency), inter-character spacing, character case manipulation, language dependent pre and post character spacing (for instance after colons in French), special font rendering such as outlines, and much more. An experimental application is a more advanced glue/penalty model with look-back and look-ahead as well as relative weights. This is inspired by the one good thing that XML formatting objects provide: a spacing and pagebreak model.

It does not take much imagination to see that features demanding processing of node lists come with a price: many of the callbacks that LUATEX provides are indeed used and as a result quite some time is spent in LUA. You can add to that the time needed for handling font features, which also boils down to processing node lists. The second half of 2007 Taco and I spent much time on benchmarking and by now the interface between TEX and LUA (passing information and manipulating nodes) has been optimized quite well. Of course there's always a price for flexibility and LUATEX will never be as fast as PDFTEX, but then, PDFTEX does not deal with OPENTYPE and such.

We can safely conclude that the impact of LUATEX on CONTEXT is huge and that fundamental changes take place in all key components: files, fonts, languages, graphics, METAPOST XML, verbatim and color to start with, but more will follow. Of course there are also less prominent areas where we use LUA based approaches: handling URL's, conversions, alternative math input to mention a few. Sometime in 2009 we expect to start working on more fundamental typesetting related issues.

roadmap

On the LUATEX website `www.luatex.org` you can find a roadmap. This roadmap is just an indication of what happened and will happen and it will be updated when we feel the need. Here is a summary.

- merging engines

  Merge some of the ALEPH codebase into PDFTEX (which already has ε-TEX) so that LUATEX in DVI mode behaves like ALEPH, and in PDF mode like PDFTEX. There will be LUA callbacks for file searching. This stage is mostly finished.

- OPENTYPE fonts

  Provide PDF output for ALEPH bidirectional functionality and add support for OPENTYPE fonts. Allow LUA scripts to control all aspects of font loading, font definition and manipulation. Most of this is finished.

- tokenizing and node lists

  Use LUA callbacks for various internals, complete access to tokenizer and provide access to node lists at moments that make sense. This stage is completed.

- paragraph building

  Provide control over various aspects of paragraph building (hyphenation, kerning, ligature building), dynamic loading loading of hyphenation patterns. Apart from some small details these objectives are met.

- METAPOST (MPLIB)

  Incorporate a METAPOST library and investigate options for runtime font generation and manipulation. This activity is on schedule and integration will take place before summer 2008.

- image handling

  Image identification and loading in LUA including scaling and object management. This is nicely on schedule, the first version of the image library showed up in the 0.22 beta and some more features are planned.

- special features

  Cleaning up of HZ optimization and protruding and getting rid of remaining global font properties. This includes some cleanup of the backend. Most of this stage is finished.

- page building

  Control over page building and access to internals that matter. Access to inserts. This is on the agenda for late 2008.

- TEX primitives

  Access to and control over most TEX primitives (and related mechanisms) as well as all registers. Especially box handling has to be reinvented. This is an ongoing effort.

- PDF backend

  Open up most backend related features, like annotations and object management. The first code will show up at the end of 2008.

- math

  Open up the math engine parallel to the development of the TEX Gyre math fonts. Work on this will start during 2008 and we hope that it will be finished by early 2009.

- CWEB

  Convert the TEX Pascal source into CWEB and start using LUA as glue language for components. This will be tested on MPLIB first. This is on the long term agenda, so maybe around 2010 you will see the first signs.

In addition to the mentioned functionality we have a couple of ideas that we will implement along the road. The first formal beta was released at TUG 2007 in San Diego (USA). The first formal release will be at TUG 2008 in Cork (Ireland). The production version will be released at EuroTEX in the Netherlands (2009).

Eventually LUATEX will be the successor to PDFTEX (informally we talk of PDFTEX version 2). It can already be used as a drop-in for ALEPH (the stable variant of OMEGA). It provides a scripting engine without the need to install a specific scripting environment. These factors are among the reasons why distributors have added the binaries to the collections. Norbert Preining maintains the LINUX packages, Akira Kakuto provides WINDOWS binaries as part of his distribution, Arthur Reutenauer takes care of MACOSX and Christian Schenk recently added LUATEX to MIKTEX. The LUATEX and MPLIB projects are hosted at Supelec by Fabrice Popineau (one of our technical consultants). And with Karl Berry being one of our motivating supporters, you can be sure that the binaries will end up someplace in TEXLIVE this year.

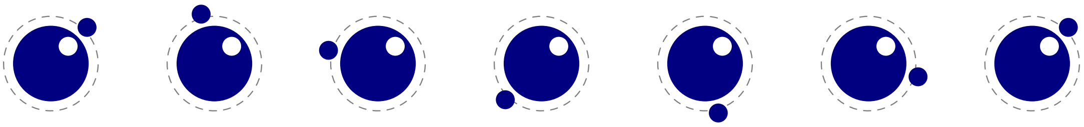
XXII The MetaPost Library

This chapter is written by Taco and Hans around the time that MPLIB was integrated into LUATEX. It is part of our torture test.

introduction

If METAPOST support had not been as tightly integrated into CONTEXT as it is, at least half of the projects PRAGMA ADE has been doing in the last decade could not have been done at all. Take for instance backgrounds behind text or graphic markers alongside text. These are probably the most complex mechanisms in CONTEXT: positions are stored, and positional information is passed on to METAPOST, where intersections between the text areas and the running text are converted into graphics that are then positioned in the background of the text. Underlining of text (sometimes used in the educational documents that we typeset) and change bars (in the margins) are implemented using the same mechanism because those are basically a background with only one of the frame sides drawn.

You can probably imagine that a 300 page document with several such graphics per page takes a while to process. A nice example of such integrated graphics is the LUATEX reference manual, that has an unique graphic at each page: a stylized image of a revolving moon.



Most of the running time integrating such graphics seemed to be caused by the mechanics of the process: starting the separate METAPOST interpreter and having to deal with a number of temporary files. Therefore our expectations were high with regards to integrating METAPOST more tightly into LUATEX. Besides the speed gain, it also true that the simpler the process of using such use of graphics becomes, the more modern a TEX runs looks and the less problems new users will have with understanding how all the processes cooperate.

This article will not discuss the application interface of the MPLIB library in detail, for that there is the LUATEX manual. In short, using the embedded METAPOST interpreter in LUATEX boils down to the following:

- Open an instance using `mplib.new`, either to process images with a format to be loaded, or to create such a format. This function returns a library object.
- Execute sequences of METAPOST commands, using the object's `execute` method. This returns a result.
- Check if the result is valid and (if it is okay) request the list of objects. Do whatever you want with them, most probably convert them to some output format. You can

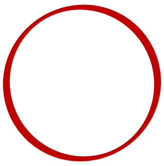
also request a string representation of a graphic in PostScript format.

There is no need to close the library object. As long as you didn't make any fatal errors, the library recovers well and can stay alive during the entire LuaTeX run.

Support for mplib depends on a few components: integration, conversion and extensions. This article shows some of the code involved in supporting the library. Let's start with the conversion.

conversion

The result of a MetaPost run traditionally is a PostScript language description of the generated graphic(s). When pdf is needed, that PostScript code has to be converted to the target format. This includes embedded text as well as penshapes used for drawing. To demonstrate, here is a simple example graphic:



```
draw fullcircle
  scaled 2cm
  withpen pencircle xscaled 1mm yscaled .5mm rotated 30
  withcolor .75red ;
```

Figure XXII.1 Notice how the pen is not a circle but a rotated ellipse. Later on it will become clear what the consequences of that are for the conversion.

How does this output look in PostScript? If the preamble is left out it looks like this:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -30 -30 30 30
%%HiResBoundingBox: -29.624 -29.28394 29.624 29.28394
%%Creator: MetaPost 1.9991
%%CreationDate: 2016.07.29:0935
%%Pages: 1
%%DocumentResources: procset mpost
%%DocumentSuppliedResources: procset mpost
%%EndComments
%% <<stripped preamble was here>>
%%BeginSetup
%%EndSetup
%%Page: 1 1
 0.75 0 0 R 2.55511 hlw rd 1 lj 10 ml
q n 28.34645 0 m
28.34645 7.51828 25.35938 14.72774 20.04356 20.04356 c
14.72774 25.35938 7.51828 28.34645 0 28.34645 c
-7.51828 28.34645 -14.72774 25.35938 -20.04356 20.04356 c
-25.35938 14.72774 -28.34645 7.51828 -28.34645 0 c
```

```
-28.34645 -7.51828 -25.35938 -14.72774 -20.04356 -20.04356 c
-14.72774 -25.35938 -7.51828 -28.34645 0 -28.34645 c
7.51828 -28.34645 14.72774 -25.35938 20.04356 -20.04356 c
25.35938 -14.72774 28.34645 -7.51828 28.34645 0 c p
 [0.96075 0.55469 -0.27734 0.48038 0 0] t S Q
P
%%EOF
```

The most prominent code here concerns the path. The numbers in brackets define the transformation matrix for the pen we used. The PDF variant looks as follows:

The operators don't look much different from the PostScript, which is mostly due to the fact that in the PostScript code, the preamble defines shortcuts like `c` for `curveto`. Again, most code involves the path. However, this time the numbers are different and the transformation comes before the path.

In the case of PDF output, we could use TeX itself to do the conversion: a generic converter is implemented in `supp-pdf.tex`, while a converter optimized for ConTeXt MkII is defined in the files whose names start with `meta-pdf`. But in ConTeXt MkIV we use Lua code for the conversion instead. Thanks to Lua's powerful LPEG parsing library, this gives cleaner code and is also faster. This converter currently lives in `mlib-pdf.lua`.

Now, with the embedded MetaPost library, conversion goes different still because now it is possible to request the drawn result and associated information in the form of Lua tables.

```
figure={
 ["boundingbox"]={
  ["llx"]=-29.623992919922,
  ["lly"]=-29.283935546875,
  ["urx"]=29.623992919922,
  ["ury"]=29.283935546875,
 },
 ["objects"]={
  {
   ["color"]={ 0.75, 0, 0 },
   ["linecap"]=1,
   ["linejoin"]=1,
   ["miterlimit"]=10,
   ["path"]={
    {
     ["left_x"]=28.346450805664,
     ["left_y"]=-7.5182800292969,
     ["right_x"]=28.346450805664,
     ["right_y"]=7.5182800292969,
     ["x_coord"]=28.346450805664,
     ["y_coord"]=0,
    },
    {
     ["left_x"]=25.359375,
     ["left_y"]=14.727737426758,
     ["right_x"]=14.727737426758,
     ["right_y"]=25.359375,
     ["x_coord"]=20.043563842773,
     ["y_coord"]=20.043563842773,
    },
    {
     ["left_x"]=7.5182800292969,
     ["left_y"]=28.346450805664,
     ["right_x"]=-7.5182800292969,
     ["right_y"]=28.346450805664,
     ["x_coord"]=0,
     ["y_coord"]=28.346450805664,
    },
    {
     ["left_x"]=-14.727737426758,
     ["left_y"]=25.359375,
     ["right_x"]=-25.359375,
     ["right_y"]=14.727737426758,
     ["x_coord"]=-20.043563842773,
     ["y_coord"]=20.043563842773,
    },
    {
     ["left_x"]=-28.346450805664,
     ["left_y"]=7.5182800292969,
     ["right_x"]=-28.346450805664,
     ["right_y"]=-7.5182800292969,
     ["x_coord"]=-28.346450805664,
     ["y_coord"]=0,
    },
    {
     ["left_x"]=-25.359375,
     ["left_y"]=-14.727737426758,
     ["right_x"]=-14.727737426758,
     ["right_y"]=-25.359375,
     ["x_coord"]=-20.043563842773,
     ["y_coord"]=-20.043563842773,
    },
    {
     ["left_x"]=-7.5182800292969,
     ["left_y"]=-28.346450805664,
     ["right_x"]=7.5182800292969,
     ["right_y"]=-28.346450805664,
     ["x_coord"]=0,
     ["y_coord"]=-28.346450805664,
    },
    {
     ["left_x"]=14.727737426758,
     ["left_y"]=-25.359375,
     ["right_x"]=25.359375,
     ["right_y"]=-14.727737426758,
     ["x_coord"]=20.043563842773,
     ["y_coord"]=-20.043563842773,
    },
   },
   ["pen"]={
    {
     ["left_x"]=2.4548797607422,
     ["left_y"]=1.4173278808594,
     ["right_x"]=-0.70866394042969,
     ["right_y"]=1.2274475097656,
     ["x_coord"]=0,
     ["y_coord"]=0,
    },
    ["type"]="elliptical",
   },
   ["type"]="outline",
  },
 },
}
```

This means that instead of parsing PostScript output, we now can operate on a proper datastructure and get code like the following:

```
function convertgraphic(result)
  if result then
    local figures = result.fig
    if figures then
      for fig in ipairs(figures) do
```

```
        local llx, lly, urx, ury = unpack(fig:boundingbox())
        if urx > llx then
          startgraphic(llx, lly, urx, ury)
          for object in ipairs(fig:objects()) do
            if object.type == "..." then
              ...
              flushgraphic(...)
              ...
            else
              ...
            end
          end
          finishgraphic()
        end
      end
    end
  end
end
```

Here `result` is what the library returns when one or more graphics are processed. As you can deduce from this snippet, a result can contain multiple figures. Each figure corresponds with a `beginfig ... endfig`. The graphic operators that the converter generates (so called PDF literals) have to be encapsulated in a proper box so this is why we have:

- `startgraphic`: start packaging the graphic
- `flushgraphic`: pipe literals to TEX
- `finishgraphic`: finish packaging the graphic

It does not matter what number you passed to `beginfig`, the graphics come out in the natural order.

Little over half a dozen different object types are possible. The example METAPOST `draw` command from above results in an `outline` object. This object contains not only path information but also carries rendering data, like the color and the pen. So, in the end we will flush code like `1 M` which sets the `miterlimit` to one or `.5 g` which sets the color to 50% gray, in addition to a path.

Because objects are returned in a way that closely resembles a METAPOST's internals, some extra work needs to be done in order to calculate paths with elliptical pens. An example of a helper function in somewhat simplified form is shown next:

```
function pen_characteristics(object)
  local p = object.pen[1]
  local wx, wy, width
  if p.right_x == p.x_coord and p.left_y == p.y_coord then
    wx = abs(p.left_x  - p.x_coord)
    wy = abs(p.right_y - p.y_coord)
  else -- pyth: sqrt(a^2 +b^2)
    wx = pyth(p.left_x - p.x_coord, p.right_x - p.x_coord)
    wy = pyth(p.left_y - p.y_coord, p.right_y - p.y_coord)
  end
  if wy/coord_range_x(object.path, wx) >=
                    wx/coord_range_y(object.path, wy) then
    width = wy
  else
    width = wx
  end
  local sx, sy = p.left_x, p.right_y
  local rx, ry = p.left_y, p.right_x
  local tx, ty = p.x_coord, p.y_coord
  if width ~= 1 then
    if width == 0 then
      sx, sy = 1, 1
    else
      rx, ry, sx, sy = rx/width, ry/width, sx/width, sy/width
    end
  end
  if abs(sx) < eps then sx = eps end
  if abs(sy) < eps then sy = eps end
  return sx, rx, ry, sy, tx, ty, width
end
```

If `sx` and `sy` are 1, there is no need to transform the path, otherwise a suitable transformation matrix is calculated and returned. The function itself uses a few helpers that make the calculations even more obscure. This kind of code does not fall in the category trivial and as already mentioned, these basic algorithms were derived from the MetaPost sources. Even so, these snippets demonstrate that interfacing using Lua does not look that bad.

In the actual MkIV code things look a bit different because it does a bit more and uses optimized code. There you will also find the code dealing with the actual transformation, of which these helpers are just a portion.

If you compare the PostScript and the PDF code you will notice that the paths looks different. This is because the use and application of a transformation matrix in PDF is different from how it is handled in PostScript. In PDF more work is assumed to be done by the PDF generating application. This is why in both the TeX and the Lua based converters you will find transformation code and the library follows the same pattern. In that respect PDF differs fundamentally from POSTSCRIPT.

Within the TEX based converter there was the problem of keeping the needed calculations within TEX's accuracy, which fortunately permits larger values that METAPOST can produce. This plus the parsing code resulted in a not-that-easy to follow bunch of TEX code. The LUA based parser is more readable, but since it also operates on POSTSCRIPT code it is kind of unnatural too, but at least there are less problems with keeping the calculations sane. The MPLIB based converter is definitely the cleanest and least sensitive to future changes in the POSTSCRIPT output. Does this mean that there is no ugly code left? Alas, as we will see in the next section, dealing with extensions is still somewhat messy. In practice users will not be bothered with such issues, because writing a converter is a one time job by macro package writers.

extensions

In METAFUN, which is the METAPOST format used with CONTEXT, a few extensions are provided, like:

- cmyk, spot and multitone colors
- including external graphics
- lineair and circulair shades
- texts converted to outlines
- inserting arbitrary texts

Until now, most of these extensions have been implemented by using specially coded colors and by injecting so called specials (think of them as comments) into the output. On one of our trips to a TEX conference, we discussed ways to pass information along with paths and eventually we arrived at associating text strings with paths as a simple and efficient solution. As a result, recently METAPOST was extended by `withprescript` and `withpostscript` directives. For those who are unfamiliar with these new scripts, they are used as follows:

```
draw fullcircle withprescript "hello" withpostscript "world" ;
```

In the POSTSCRIPT output these scripts end up before and after the path, but in the PDF converter they can be overloaded to implement extensions, and that works reasonably well. However, at the moment there cannot be multiple pre- and postscripts associated with a single path inside the METAPOST internals. This means that for the moment, the scripts mechanism is only used for a few of the extensions. Future versions of MPLIB may provide more sophisticated methods for carrying information around.

The MKIV conversion mechanism uses scripts for graphic inclusion, shading and text processing but unfortunately cannot use them for more advanced color support.

A nasty complication is that the color spaces in METAPOST don't cast, which means that one cannot assign any color to a color variables: each colorspace has it's own type of variable.

```
color     one ; one := (1,1,0)   ; % correct
cmykcolor two ; two := (1,0,0,1) ; % correct
one := two ; % error
fill fullcircle scaled 1cm withcolor .5[one,two] ; % error
```

In CONTEXT we use constructs like this:

```
\startreusableMPgraphic{test}
  fill fullcircle scaled 1cm withcolor \MPcolor{mycolor} ;
\stopreusableMPgraphic

\reuseMPgraphic{test}
```

Because `withcolor` is clever enough to understand what color type it receives, this is ok, but how about:

```
\startreusableMPgraphic{test}
  color c ; c := \MPcolor{mycolor} ;
  fill fullcircle scaled 1cm withcolor c ;
\stopreusableMPgraphic
```

Here the color variable only accepts an RGB color and because in CONTEXT there is mixed color space support combined with automatic colorspace conversions, it doesn't know in advance what type it is going to get. By implementing color spaces other than RGB using special colors (as before) such type mismatches can be avoided.

The two techniques (coding specials in colors and pre/postscripts) cannot be combined because a script is associated with a path and cannot be bound to a variable like `c`. So this again is an argument for using special colors that remap onto CMYK spot or multi-tone colors.

Another area of extensions is text. In previous versions of CONTEXT the text processing was already isolated: text ended up in a separate file and was processed in an separate run. More recent versions of CONTEXT use a more abstract model of boxes that are preprocessed before a run, which avoids the external run(s). In the new approach everything can be kept internal. The conversion even permits constructs like:

```
for i=1 upto 100 :
  draw btex oeps etex rotated i ;
endfor ;
```

but since this construct is kind of obsolete (at least in the library version of METAPOST) it is better to use:

```
for i=1 upto 100 :
  draw textext("cycle " & decimal i) rotated i ;
endfor ;
```

Internally a trial pass is done so that indeed 100 different texts will be drawn. The throughput of texts is so high that in practice one will not even notice that this happens.

Dealing with text is yet another example of using LPEG. The following snippet of code sheds some light on how text in graphics is dealt with. Actually this is a variation on a previous implementation. That one was slightly faster but looked more complex. It was also not robust for complex texts defined in macros in a format.

```
local P, S, V, Cs = lpeg.P, lpeg.S, lpeg.V, lpeg.Cs

local btex    = P("btex")
local etex    = P(" etex")
local vtex    = P("verbatimtex")
local ttex    = P("textext")
local gtex    = P("graphictext")
local spacing = S(" \n\r\t\v")^0
local dquote  = P('"')

local found = false

local function convert(str)
  found = true
  return "textext(\"" .. str .. "\")"
end
local function ditto(str)
  return "\" & ditto & \""
end
local function register()
  found = true
end

local parser = P {
    [1] = Cs((V(2)/register + V(3)/convert + 1)^0),
    [2] = ttex + gtex,
    [3] = (btex + vtex) * spacing *
                Cs((dquote/ditto + (1-etex))^0) * etex,
}
```

```
function metapost.check_texts(str)
  found = false
  return parser:match(str), found
end
```

If you are unfamiliar with LPEG it may take a while to see what happens here: we replace the text between `btex` and `etex` by a call to `textext`, a macro. Special care is given to embedded double quotes.

When text is found, the graphic is processed two times. The definition of `textext` is different for each run. The first run we have:

```
vardef textext(expr str) =
    image (
        draw unitsquare
            withprescript "tf"
            withpostscript str ;
    )
enddef ;
```

After the first run the result is not really converted, but just the outlines with the `tf` prescript are filtered. In the loop over the object there is code like:

```
local prescript = object.prescript
if prescript then
  local special = metapost.specials[prescript]
  if special then
    special(object.postscript,object)
  end
end
```

Here, `metapost` is just the namespace used by the converter. The prescript tag `tf` triggers a function:

```
function metapost.specials.tf(specification,object)
  tex.sprint(tex.ctxcatcodes,format("\\MPLIBsettext{%s}{%s}",
    metapost.textext_current,specification))
  if metapost.textext_current < metapost.textext_last then
    metapost.textext_current = metapost.textext_current + 1
  end
  ...
end
```

Again, you can forget about the details of this function. Important is that there is a call out to TEX that will process the text. Each snippet gets the number of the box that holds the content. The macro that is called just puts stuff in a box:

```
\def\MPLIBsettext#1#2%
  {\global\setbox#1\hbox{#2}}
```

In the next processing cycle of the MetaPost code, the `textext` macro does something different :

```
vardef textext(expr str) =
    image (
        _tt_n_ := _tt_n_ + 1 ;
        draw unitsquare
            xscaled _tt_w_[_tt_n_]
            yscaled (_tt_h_[_tt_n_] + _tt_d_[_tt_n_])
            withprescript "ts"
            withpostscript decimal _tt_n_ ;
    )
enddef ;
```

This time the by then known dimensions of the box that is used to store the snippet are used. These are stored in the `_tt_w_`, `_tt_h_` and `_tt_d_` arrays. The arrays are defined by Lua using information about the boxes, and passed to the library before the second run. The result from the second MetaPost run is converted, and again the prescript is used as trigger:

```
function metapost.specials.ts(specification,object,result)
    local op = object.path
    local first, second, fourth  = op[1], op[2], op[4]
    local tx, ty = first.x_coord      , first.y_coord
    local sx, sy = second.x_coord - tx, fourth.y_coord - ty
    local rx, ry = second.y_coord - ty, fourth.x_coord - tx
    if sx == 0 then sx = 0.00001 end
    if sy == 0 then sy = 0.00001 end
    metapost.flushfigure(result)
    tex.sprint(tex.ctxcatcodes,format(
        "\\MPLIBgettext{%f}{%f}{%f}{%f}{%f}{%f}{%s}",
        sx,rx,ry,sy,tx,ty,metapost.textext_current))
    ...
end
```

At this point the converter is actually converting the graphic and passing PDF literals to TeX. As soon as it encounters a text, it flushes the PDF code collected so far and injects some TeX code. The TeX macro looks like:

```
\def\MPLIBgettext#1#2#3#4#5#6#7%
  {\ctxlua{metapost.sxsy(\number\wd#7,\number\ht#7,\number\dp#7)}%
   \pdfliteral{q #1 #2 #3 #4 #5 #6 cm}%
   \vbox to \zeropoint{\vss\hbox to \zeropoint
     {\scale[sx=\sx,sy=\sy]{\raise\dp#7\box#7}\hss}}%
   \pdfliteral{Q}}
```

Because text can be transformed, it needs to be scale back to the right dimensions, using both the original box dimensions and the transformation of the unitqaure associated with the text.

```
local factor = 65536*(7200/7227)

function metapost.sxsy(wd,ht,dp) -- helper for text
  commands.edef("sx",(wd ~= 0 and 1/( wd    /(factor))) or 0)
  commands.edef("sy",(wd ~= 0 and 1/((ht+dp)/(factor))) or 0)
end
```

So, in fact there are the following two processing alternatives:

- tex: calls a Lua function that processed the graphic
- lua: parse the MetaPost code for texts and decide if two runs are needed

Now, if there was no text to be found, the continuation is:

- lua: process the code using the library
- lua: convert the resulting graphic (if needed) and check if texts are used

Otherwise, the next steps are:

- lua: process the code using the library
- lua: parse the resulting graphic for texts (in the postscripts) and signal TEX to process these texts afterwards
- tex: process the collected text and put the result in boxes
- lua: process the code again using the library but this time let the unitsquare be transformed using the text dimensions
- lua: convert the resulting graphic and replace the transformed unitsquare by the boxes with text

The processor itself is used in the MkIV graphic function that takes care of the multiple passes mentioned before. To give you an idea of how it works, here is how the main graphic processing function roughly looks.

```
local current_format, current_graphic
```

```
function metapost.graphic_base_pass(mpsformat,str,preamble)
    local prepared, done = metapost.check_texts(str)
    metapost.textext_current = metapost.first_box
    if done then
        current_format, current_graphic = mpsformat, prepared
        metapost.process(mpsformat, {
            preamble or "",
            "beginfig(1); ",
            "_trial_run_ := true ;",
            prepared,
            "endfig ;"
        }, true ) -- true means: trialrun
        tex.sprint(tex.ctxcatcodes,
            "\\ctxlua{metapost.graphic_extra_pass()}")
    else
        metapost.process(mpsformat, {
            preamble or "",
            "beginfig(1); ",
            "_trial_run_ := false ;",
            str,
            "endfig ;"
        } )
    end
end

function metapost.graphic_extra_pass()
    metapost.textext_current = metapost.first_box
    metapost.process(current_format, {
        "beginfig(0); ",
        "_trial_run_ := false ;",
        table.concat(metapost.texttextsdata()," ;\n"),
        current_graphic,
        "endfig ;"
    })
end
```

The box information is generated as follows:

```
function metapost.texttextsdata()
    local t, n = { }, 0
    for i = metapost.first_box, metapost.last_box do
        n = n + 1
        local box_i = tex.box[i]
        if box_i then
            t[#t+1] = format(
                "_tt_w_[%i]:=%f;_tt_h_[%i]:=%f;_tt_d_[%i]:=%f;",
                n, box_i.width /factor,
                n, box_i.height/factor,
                n, box_i.depth /factor
            )
        else
            break
        end
    end
    return t
end
```

This is a typical example of accessing information available inside TEX from LUA, in this case information about boxes.

The `trial_run` flag is used at the METAPOST end, in fact the `textext` macro looks as follows:

```
vardef textext(expr str) =
    if _trial_run_ :
        % see first variant above
    else :
        % see second variant above
    fi
enddef ;
```

This trickery is not new. We used it already in CONTEXT for some time, but until now the multiple runs took way more time and from the perspective of the user this all looked much more complex.

It may not be that obvious, but: in the case of a trial run (for instance when texts are found), after the first processing stage, and during the parsing of the result, the commands that typeset the content will be printed to TEX. After processing, the command to do an extra pass is printed to TEX also. So, once control is passed back to TEX, at some point TEX itself will pass control back to LUA and do the extra pass.

The base function is called in:

```
function metapost.graphic(mpsformat,str,preamble)
    local mpx = metapost.format(mpsformat or "metafun")
    metapost.graphic_base_pass(mpx,str,preamble)
end
```

The `metapost.format` function is part of `mlib-run`. It loads the `metafun` format, possibly after (re)generating it.

Now, admittedly all this looks a bit messy, but in pure TEX macros it would be even more so. Sometime in the future, the postponed calls to `\ctxlua` and the explicit `\pdfliterals` can and will be replaced by using direct node generation, but that requires a rewrite of the internal LUATEX support for PDF literals.

The snippets are part of the `mlib-*` files of MKIV. These files are tagged as experimental and will stay that way for a while yet. This is proved by the fact that by now we use a slightly different approach.

Summarizing the impact of MPLIB on extensions, we can conclude that some are done better and some more or less the same. There are some conceptual problems that prohibit using pre- and postscripts for everything (at least currently).

integrating

The largest impact of MPLIB is processing graphics at runtime. In MKII there are two methods: real runtime processing (each graphic triggered a call to METAPOST) and collective processing (between TEX runs). The first method slows down the TEX run, the second method generates a whole lot of intermediate POSTSCRIPT files. In both cases there is a lot of file IO involved.

In MKIV, the integrated library is capable of processing thousands of graphics per second, including conversion. The preliminary tests (which involved no extensions) involved graphics with 10 random circles drawn with penshapes in random colors, and the thoughput was around 2000 such graphics per second on a 2.3 MHz Core Duo:

In practice there will be some more overhead involved than in the tests. For instance, in CONTEXT information about the current state of TEX has to be passed on also: page dimensions, font information, typesetting related parameters, preamble code, etc.

The whole TEX interface is written around one process function:

```
metapost.graphic(metapost.format("metafun"),"mp code")
```

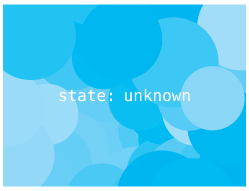
optionally a preamble can be passed as the third argument. This one function is used in several other macros, like:

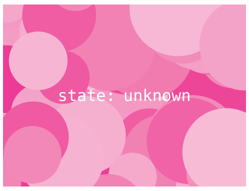
```
\startMPcode                         ... \stopMPcode
\startMPpage                         ... \stopMPpage
\startuseMPgraphic      {name} ... \stopuseMPgraphic
\startreusableMPgraphic{name} ... \stopreusableMPgraphic
\startuniqueMPgraphic   {name} ... \stopuniqueMPgraphic

\useMPgraphic{name}
\reuseMPgraphic{name}
\uniqueMPgraphic{name}
```

The user interface is downward compatible: in MkIV the same top-level commands are provided as in MkII. However, the (previously required) configuration macros and flags are obsolete.

This time, the conclusion is that the impact on ConTeXt is immense: The code for embedding graphics is very clean, and the running time for graphics inclusion is now negligible. Support for text in graphics is more natural now, and takes no runtime either (in MkII some parsing in TeX takes place, and if needed long lines are split; all this takes time).

In the styles that Pragma ADE uses internally, there is support for the generation of placeholders for missing graphics. These placeholders are MetaPost graphics that have some 60 randomly scaled circles with randomized colors. The time involved in generating 50 such graphics is (on Hans' machine) some 14 seconds, while in LuaTeX only half a second is needed.





Because LuaTeX needs more startup time and deals with larger fonts resources, pdfTeX is generally faster, but now that we have mplib, LuaTeX suddenly is the winner.

XXIII The LuaTEX Mix

introduction

The idea of embedding LUA into TEX originates in some experiments with LUA embedded in the SciTE editor. You can add functionality to this editor by loading LUA scripts. This is accomplished by a library that gives access to the internals of the editing component.

The first integration of LUA in PDFTEX was relatively simple: from TEX one could call out to LUA and from LUA one could print to TEX. My first application was converting math encoded a calculator syntax to TEX. Following experiments dealt with METAPOST. At this point integration meant as little as: having some scripting language as addition to the macro language. But, even in this early stage further possibilities were explored, for instance in manipulating the final output (i.e. the PDF code). The first versions of what by then was already called LUATEX provided access to some internals, like counter and dimension registers and the dimensions of boxes.

Boosted by the oriental TEX project, the team started exploring more fundamental possibilities: hooks in the input/output, tokenization, fonts and nodelists. This was followed by opening up hyphenation, breaking lines into paragraphs and building ligatures. At that point we not only had access to some internals but also could influence the way TEX operates.

After that, an excursion was made to MPLIB, which fulfilled a long standing wish for a more natural integration of METAPOST into TEX. At that point we ended up with mixtures of TEX, LUA and METAPOST code.

Medio 2008 we still need to open up more of TEX, like page building, math, alignments and the backend. Eventually LUATEX will be nicely split up in components, rewritten in C, and we may even end up with LUA glueing together the components that make up the TEX engine. At that point the interoperation between TEX and LUA may be more rich that it is now.

In the next sections I will discuss some of the ideas behind LUATEX and the relationship between LUA and TEX and how it presents itself to users. I will not discuss the interface itself, which consists of quite some functions (organized in pseudo libraries) and the mechanisms used to access and replace internals (we call them callbacks).

tex vs. lua

TEX is a macro language. Everything boils down to either allowing stepwise expansion or explicitly preventing it. There are no real control features, like loops; tail recursion is a key concept. There are few accessible data-structures like numbers, dimensions, glue, token lists and boxes. What happens inside TEX is controlled by variables, mostly hidden from view, and optimized within the constraints of 30 years ago.

The original idea behind TEX was that an author would write a specific collection of macros for each publication, but increasing popularity among non-programmers quickly resulted in distributed collections of macros, called macro packages. They started small but grew and grew and by now have become pretty large. In these packages there are macros dealing with fonts, structure, page layout, graphic inclusion, etc. There is also code dealing with user interfaces, process control, conversion and much of that code looks out of place: the lack of control features and string manipulation is solved by mimicking other languages, the unavailability of a float datatype is compensated by misusing dimension registers, and you can find provisions to force or inhibit expansion all over the place.

TEX is a powerful typographical programming language but lacks some of the handy features of scripting languages. Handy in the sense that you will need them when you want to go beyond the original purpose of the system. LUA is a powerful scripting language, but knows nothing of typesetting. To some extent it resembles the language that TEX was written in: PASCAL. And, since LUA is meant for embedding and extending existing systems, it makes sense to bring LUA into TEX. How do they compare? Let's give some examples.

About the simplest example of using LUA in TEX is the following:

```
\directlua { tex.print(math.sqrt(10)) }
```

This kind of application is probably what most users will want and use, if they use LUA at all. However, we can go further than that.

In TEX a loop can be implemented as in the plain format (copied with comment):

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next\iterate\else\let\next\relax\fi\next}
\let\repeat=\fi % this makes \loop...\if...\repeat skippable
```

This is then used as:

```
\newcount \mycounter \mycounter=1
\loop
    ...
    \advance\mycounter 1
    \ifnum\mycounter < 11
\repeat
```

The definition shows a bit how TEX programming works. Of course such definitions can be wrapped in macros, like:

```
\forloop{1}{10}{1}{some action}
```

and this is what often happens in more complex macro packages. In order to use such control loops without side effects, the macro writer needs to take measures that permit for instance nested usage and avoids clashes between local variables (counters or macros) and user defined ones. Here we use a counter in the condition, but in practice expressions will be more complex and this is not that trivial to implement.

The original definition of the iterator can be written a bit more efficient:

```
\def\iterate{\body \expandafter\iterate \fi}
```

And indeed, in macro packages you will find many such expansion control primitives being used, which does not make reading macros easier.

Now, get me right, this does not make TEX less powerful, it's just that the language is focused on typesetting and not on general purpose programming, and in principle users can do without: documents can be preprocessed using another language, and document specific styles can be used.

We have to keep in mind that TEX was written in a time when resources in terms of memory and CPU cycles weres less abundant than they are now. The 255 registers per class and the about 3000 hash slots in original TEX were more than enough for typesetting a book, but in huge collections of macros they are not all that much. For that reason many macropackages use obscure names to hide their private registers from users and instead of allocating new ones with meaningful names, existing ones are shared. It is therefore not completely fair to compare TEX code with LUA code: in LUA we have plenty of memory and the only limitations are those imposed by modern computers.

In LUA, a loop looks like this:

```
for i=1,10 do
    ...
end
```

But while in the TEX example, the content directly ends up in the input stream, in LUA we need to do that explicitly, so in fact we will have:

```
for i=1,10 do
    tex.print("...")
end
```

And, in order to execute this code snippet, in LUATEX we will do:

```
\directlua 0 {
    for i=1,10 do
```

```
        tex.print("...")
    end
}
```

So, eventually we will end up with more code than just LUA code, but still the loop itself looks quite readable and more complex loops are possible:

```
\directlua 0 {
    local t, n = { }, 0
    while true do
        local r = math.random(1,10)
        if not t[r] then
            t[r], n = true, n+1
            tex.print(r)
            if n == 10 then break end
        end
    end
}
```

This will typeset the numbers 1 to 10 in randomized order. Implementing a random number generator in pure TEX takes some bit of code and keeping track of already defined numbers in macros can be done with macros, but both are not very efficient.

I already stressed that TEX is a typographical programming language and as such some things in TEX are easier than in LUA, given some access to internals:

```
\setbox0=\hbox{x} \the\wd0
```

In LUA we can do this as follows:

```
\directlua 0 {
    local n = node.new('glyph')
    n.font = font.current()
    n.char = string.byte('x')
    tex.box[0] = node.hpack(n)
    tex.print(tex.box[0].width/65536 .. "pt")
}
```

One pitfall here is that TEX rounds the number differently than LUA. Both implementations can be wrapped in a macro cq. function:

```
\def\measured#1{\setbox0=\hbox{#1}\the\wd0\relax}
```

Now we get:

```
\measured{x}
```

The same macro using LUA looks as follows:

```
\directlua 0 {
    function measure(chr)
        local n = node.new('glyph')
        n.font = font.current()
        n.char = string.byte(chr)
        tex.box[0] = node.hpack(n)
        tex.print(tex.box[0].width/65536 .. "pt")
    end
}
\def\measured#1{\directlua0{measure("#1")}}
```

In both cases, special tricks are needed if you want to pass for instance a `#` to TEX's variant, or a `"` to LUA. In both cases we can use shortcuts like `\#` and in the second case we can pass strings as long strings using double square brackets to LUA.

This example is somewhat misleading. Imagine that we want to pass more than one character. The TEX variant is already suited for that, but the function will now look like:

```
\directlua 0 {
    function measure(str)
        if str == "" then
            tex.print("0pt")
        else
            local head, tail = nil, nil
            for chr in str:gmatch(".") do
                local n = node.new('glyph')
                n.font = font.current()
                n.char = string.byte(chr)
                if not head then
                    head = n
                else
                    tail.next = n
                end
                tail = n
            end
            tex.box[0] = node.hpack(head)
            tex.print(tex.box[0].width/65536 .. "pt")
        end
    end
}
```

And still it's not okay, since TEX inserts kerns between characters (depending on the font) and glue between words, and doing that all in LUA takes more code. So, it will be clear that although we will use LUA to implement advanced features, TEX itself still has quite some work to do.

In the following example we show code, but this is not of production quality. It just demonstrates a new way of dealing with text in TEX.

Occasionally a design demands that at some place the first character of each word should be uppercase, or that the first word of a paragraph should be in small caps, or that each first line of a paragraph has to be in dark blue. When using traditional TEX the user then has to fall back on parsing the data stream, and preferably you should then start such a sentence with a command that can pick up the text. For accentless languages like English this is quite doable but as soon as commands (for instance dealing with accents) enter the stream this process becomes quite hairy.

The next code shows how CONTEXT MKII defines the `\Word` and `\Words` macros that capitalize the first characters of word(s). The spaces are really important here because they signal the end of a word.

```
\def\doWord#1%
  {\bgroup\the\everyuppercase\uppercase{#1}\egroup}

\def\Word#1%
  {\doWord#1}

\def\doprocesswords#1 #2\od
  {\doifsomething{#1}{\processword{#1} \doprocesswords#2 \od}}

\def\processwords#1%
  {\doprocesswords#1 \od\unskip}

\let\processword\relax

\def\Words
  {\let\processword\Word \processwords}
```

Actually, the code is not that complex. We split of words and feed them to a macro that picks up the first token (hopefully a character) which is then fed into the `\uppercase` primitive. This assumes that for each character a corresponding uppercase variant is defined using the `\uccode` primitive. Exceptions can be dealt with by assigning relevant code to the token register `\everyuppercase`. However, such macros are far from robust. What happens if the text is generated and not input as-is? What happens with commands in the stream that do something with the following tokens?

A Lua based solution can look as follows:

```
\def\Words#1{\directlua 0
    for s in unicode.utf8.gmatch("#1", "([^ ])") do
        tex.sprint(string.upper(s:sub(1,1)) .. s:sub(2))
    end
}
```

But there is no real advantage here, apart from the fact that less code is needed. We still operate on the input and therefore we need to look to a different kind of solution: operating on the node list.

```
function CapitalizeWords(head)
    local done = false
    local glyph = node.id("glyph")
    for start in node.traverse_id(glyph,head) do
        local prev, next = start.prev, start.next
        if prev and prev.id == kern and prev.subtype == 0 then
            prev = prev.prev
        end
        if next and next.id == kern and next.subtype == 0 then
            next = next.next
        end
        if (not prev or prev.id ~= glyph) and
                next and next.id == glyph then
            done = upper(start)
        end
    end
    return head, done
end
```

A node list is a forward-linked list. With a helper function in the `node` library we can loop over such lists. Instead of traversing we can use a regular while loop, but it is probably less efficient in this case. But how to apply this function to the relevant part of the input? In LuaTeX there are several callbacks that operate on the horizontal lists and we can use one of them to plug in this function. However, in that case the function is applied to probably more text than we want.

The solution for this is to assign attributes to the range of text that such a function has to take care of. These attributes (there can be many) travel with the nodes. This is also a reason why such code normally is not written by end users, but by macropackage writers: they need to provide the frameworks where you can plug in code. In ConTeXt we have several such mechanisms and therefore in MkIV this function looks (slightly stripped) as follows:

```
function cases.process(namespace,attribute,head)
    local done, actions = false, cases.actions
    for start in node.traverse_id(glyph,head) do
        local attr = has_attribute(start,attribute)
        if attr and attr > 0 then
            unset_attribute(start,attribute)
            local action = actions[attr]
            if action then
                local _, ok = action(start)
                done = done and ok
            end
        end
    end
    return head, done
end
```

Here we check attributes (these are set at the TEX end) and we have all kind of actions that can be applied, depending on the value of the attribute. Here the function that does the actual uppercasing is defined somewhere else. The `cases` table provides us a namespace; such namespaces needs to be coordinated by macro package writers.

This approach means that the macro code looks completely different; in pseudo code we get:

```
\def\Words#1{{<setattribute><cases><somevalue>#1}}
```

Or alternatively:

```
\def\StartWords{\begingroup<setattribute><cases><somevalue>}
\def\StopWords {\endgroup}
```

Because starting a paragraph with a group can have unwanted side effects (like `\everypar` being expanded inside a group) a variant is:

```
\def\StartWords{<setattribute><cases><somevalue>}
\def\StopWords {<resetattribute><cases>}
```

So, what happens here is that the users sets an attribute using some high level command, and at some point during the transformation of the input into node lists, some action takes place. At that point commands, expansion and whatever no longer can interfere.

In addition to some infrastructure, macro packages need to carry some knowledge, just as with the `\uccode` used in `\uppercase`. The `upper` function in the first example looks as follows:

```
local function upper(start)
    local data, char = characters.data, start.char
    if data[char] then
        local uc = data[char].uccode
        if uc and fonts.ids[start.font].characters[uc] then
            start.char = uc
            return true
        end
    end
    return false
end
```

Such code is really macro package dependent: LuaTeX only provides the means, not the solutions. In ConTeXt we have collected information about characters in a `data` table in the `characters` namespace. There we have stored the uppercase codes (`uccode`). The, again ConTeXt specific, `fonts` table keeps track of all defined fonts and before we change the case, we make sure that this character is present in the font. Here `id` is the number by which LuaTeX keeps track of the used fonts. Each glyph node carries such a reference.

In this example, eventually we end up with more code than in TeX, but the solution is much more robust. Just imagine what would happen when in the TeX solution we would have:

```
\Words{\framed[offset=3pt]{hello world}}
```

It simply does not work. On the other hand, the Lua code never sees TeX commands, it only sees the two words represented by glyphs nodes and separated by glue.

Of course, there is a danger when we start opening TeX's core features. Currently macro packages know what to expect, they know what TeX can and cannot do. Of course macro writers have exploited every corner of TeX, even the dark ones. Where dirty tricks in the TeXbook had an educational purpose, those of users sometimes have obscene traits. If we just stick to the trickery introduced for parsing input, converting this into that, doing some calculations, and alike, it will be clear that Lua is more than welcome. It may hurt to throw away thousands of lines of impressive code and replace it by a few lines of Lua but that's the price the user pays for abusing TeX. Eventually ConTeXt MkIV will be a decent mix of Lua and TeX code, and hopefully the solutions programmed in those languages are as clean as possible.

Of course we can discuss until eternity whether Lua is the best choice. Taco, Hartmut and I are pretty confident that it is, and in the couple of years that we are working on LuaTeX nothing has proved us wrong yet. We can fantasize about concepts, only to find out that they are impossible to implement or hard to agree on; we just go ahead using trial and error. We can talk over and over how opening up should be done, which is what the team does in a nicely closed and efficient loop, but at some points decisions have to be made. Nothing is perfect, neither is LuaTeX, but most users won't notice it as long as it extends TeX's live and makes usage more convenient.

Users of TeX and MetaPost will have noticed that both languages have their own grouping (scope) model. In TeX grouping is focused on content: by grouping the macro writer (or author) can limit the scope to a specific part of the text or keep certain macros live within their own world.

```
.1. \bgroup .2. \egroup .1.
```

Everything done at 2 is local unless explicitly told otherwise. This means that users can write (and share) macros with a small chance of clashes. In MetaPost grouping is available too, but variables explicitly need to be saved.

```
.1. begingroup ; save p ; path p ; .2. endgroup .1.
```

After using MetaPost for a while this feels quite natural because an enforced local scope demands multiple return values which is not part of the macro language. Actually, this is another fundamental difference between the languages: MetaPost has (a kind of) functions, which TeX lacks. In MetaPost you can write

```
draw origin for i=1 upto 10 : .. (i,sin(i)) endfor ;
```

but also:

```
draw some(0) for i=1 upto 10 : .. some(i) endfor ;
```

with

```
vardef some (expr i) =
    if i > 4 : i = i - 4 fi ;
    (i,sin(i))
enddef ;
```

The condition and assignment in no way interfere with the loop where this function is called, as long as some value is returned (a pair in this case).

In TeX things work differently. Take this:

```
\count0=1
\message{\advance\count0 by 1 \the\count0}
\the\count0
```

The terminal wil show:

```
\advance \count 0 by 1 1
```

At the end the counter still has the value 1. There are quite some situations like this, for instance when data like a table of contents has to be written to a file. You cannot write macros where such calculations are done and hidden and only the result is seen.

The nice thing about the way LUA is presented to the user is that it permits the following:

```
\count0=1
\message{\directlua0{tex.count[0] = tex.count[0] + 1}\the\count0}
\the\count0
```

This will report 2 to the terminal and typeset a 2 in the document. Of course this does not solve everything, but it is a step forward. Also, compared to TEX and METAPOST, grouping is done differently: there is a `local` prefix that makes variables (and functions are variables too) local in modules, functions, conditions, loops etc. The LUA code in this story contains such locals.

In practice most users will use a macro package and so, if a user sees TEX, he or she sees a user interface, not the code behind it. As such, they will also not encounter the code written in LUA that deals with for instance fonts or node list manipulations. If a user sees LUA, it will most probably be in processing actual data. Therefore, in the next section I will give an example of two ways to deal with XML: one more suitable for traditional TEX, and one inspired by LUA. It demonstrates how the availability of LUA can result in different solutions for the same problem.

an example: xml

In CONTEXT MKII, the version that deals with PDFTEX and XETEX, we use a stream based XML parser, written in TEX. Each < and & triggers a macro that then parses the tag and/or entity. This method is quite efficient in terms of memory but the associated code is not simple because it has to deal with attributes, namespaces and nesting.

The user interface is not that complex, but involves quite some commands. Take for instance the following XML snippet:

```
<document>
    <section>
        <title>Whatever</title>
        <p>some text</p>
        <p>some more</p>
    </section>
</document>
```

When using CONTEXT commands, we can imagine the following definitions:

```
\defineXMLenvironment[document]{\starttext} {\stoptext}
\defineXMLargument   [title]   {\section}
\defineXMLenvironment[p]       {\ignorespaces}{\par}
```

When attributes have to be dealt with, for instance a reference to this section, things quickly start looking more complex. Also, users need to know what definitions to use in situations like this:

```
<table>
    <tr><td>first</td><td>...</td> <td>last</td></tr>
    <tr><td>left</td><td>...</td> <td>right</td></tr>
</table>
```

Here we cannot be sure if a cell does not contain a nested table, which is why we need to define the mapping as follows:

```
\defineXMLnested[table]{\bTABLE} {\eTABLE}
\defineXMLnested[tr]   {\bTR}    {\eTR}
\defineXMLnested[td]   {\bTD}    {\eTD}
```

The `\defineXMLnested` macro is rather messy because it has to collect snippets and keep track of the nesting level, but users don't see that code, they just need to know when to use what macro. Once it works, it keeps working.

Unfortunately mappings from source to style are never that simple in real life. We usually need to collect, filter and relocate data. Of course this can be done before feeding the source to TEX, but MkII provides a few mechanisms for that too. If for instance you want to reverse the order you can do this:

```
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>
```

```
\defineXMLenvironment[article]
    {\defineXMLsave[author]}
    {\blank author: \XMLflush{author}}
```

This will save the content of the `author` element and flush it when the end tag `article` is seen. So, given previous definitions, we will get the title, some text and then the author. You may argue that instead we should use for instance XSLT but even then a mapping is needed from the XML to TEX, and it's a matter of taste where the burden is put.

Because ConTEXt also wants to support standards like MathML, there are some more mechanisms but these are hidden from the user. And although these do a good job in most cases, the code associated with the solutions has never been satisfying.

Supporting xml this way is doable, and ConTEXt has used this method for many years in fairly complex situations. However, now that we have Lua available, it is possible to see if some things can be done simpler (or differently).

After some experimenting I decided to write a full blown xml parser in Lua, but contrary to the stream based approach, this time the whole tree is loaded in memory. Although this uses more memory than a streaming solution, in practice the difference is not significant because often in MkII we also needed to store whole chunks.

Loading xml files in memory is real fast and once it is done we can have access to the elements in a way similar to xpath. We can selectively pipe data to TEX and manipulate content using TEX or Lua. In most cases this is faster than the stream-based method. Interesting is that we can do this without linking to existing xml libraries, and as a result we are pretty independent.

So how does this look from the perspective of the user? Say that we have the simple article definition stored in `demo.xml`.

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some text</p>
</article>
```

This time we associate so called setups with the elements. Each element can have its own setup, and we can use expressions to assign them. Here we have just one such setup:

```
\startxmlsetups xml:document
   \xmlsetsetup{main}{article}{xml:article}
\stopxmlsetups
```

When loading the document it will automatically be associated with the tag `main`. The previous rule associates setup `xml:article` with the `article` element in tree `main`. We need to register this setup so that it will be applied to the document after loading:

```
\xmlregistersetup{xml:document}
```

and the document itself is processed with:

```
\xmlprocessfile{main}{demo.xml}{} % optional setup
```

The setup `xml:article` can look as follows:

```
\startxmlsetups xml:article
    \section{\xmltext{#1}{/title}}
    \xmlall{#1}{!(title|author)}
    \blank author: \xmltext{#1}{/author}
\stopxmlsetups
```

Here `#1` refers to the current node in the XML tree, in this case the root element, `article`. The second argument of `\xmltext` and `\xmlall` is a path expression, comparable with XPATH: `/title` means: the `title` element anchored to the current root (`#1`), and `!(title|author)` is the negation of (complement to) `title` or `author`. Such expressions can be more complex that the one above, like:

```
\xmlfirst{#1}{/one/(alpha|beta)/two/text()}
```

which returns the content of the first element that satisfies one of the paths (nested tree):

```
/one/alpha/two
/one/beta/two
```

There is a whole bunch of commands like `\xmltext` that filter content and pipe it into TEX. These are calling LUA functions. This is no manual, so we will not discuss them here. However, it is important to realize that we have to associate setups (consider them free formatted macros) to at least one element in order to get started. Also, XML inclusions have to be dealt with before assigning the setups. These are simple one-line commands. You can also assign defaults to elements, which saves some work.

Because we can use LUA to access the tree and manipulate content, we can now implement parts of XML handling in LUA. An example of this is dealing with so-called Cals tables. This is done in approximately 150 lines of LUA code, loaded at runtime in a module. This time the association uses functions instead of setups and those functions will pipe data back to TEX. In the module you will find:

```
\startxmlsetups xml:cals:process
    \xmlsetfunction {\xmldocument} {cals:table} {lxml.cals.table}
\stopxmlsetups

\xmlregistersetup{xml:cals:process}

\xmlregisterns{cals}{cals}
```

These commands tell MKIV that elements with a namespace specification that contains `cals` will be remapped to the internal namespace `cals` and the setup associates a function with this internal namespace.

By now it will be clear that from the perspective of the user hardly any LUA is visible. Sure, he or she can deduce that deep down some magic takes place, especially when you run into more complex expressions like this (the `@` denotes an attribute):

```
\xmlsetsetup
    {main} {item[@type='mpctext' or @type='mrtext']}
    {questions:multiple:text}
```

Such expressions resemble XPATH, but can go much further than that, just by adding more functions to the library.

```
b[position() > 2 and position() < 5 and text() == 'ok']
b[position() > 2 and position() < 5 and text() == upper('ok')]
b[@n=='03' or @n=='08']
b[number(@n)>2 and number(@n)<6]
b[find(text(),'ALSO')]
```

Just to give you an idea . . . in the module that implements the parser you will find definitions that match the function calls in the above expressions.

```
xml.functions.find   = string.find
xml.functions.upper  = string.upper
xml.functions.number = tonumber
```

So much for the different approaches. It's up to the user what method to use: stream based MkII, tree based MkIV, or a mixture.

The main reason for taking XML as an example of mixing TEX and LUA is in that it can be a bit mind boggling if you start thinking of what happens behind the screens. Say that we have

```
<?xml version ='1.0'?>
<article>
    <title>Whatever</title>
    <author>Someone</author>
    <p>some <b>bold</b> text</p>
</article>
```

and that we use the setup shown before with `article`.

At some point, we are done with defining setups and load the document. The first thing that happens is that the list of manipulations is applied: file inclusions are processed first, setups and functions are assigned next, maybe some elements are deleted or added, etc. When that is done we serialize the tree to TEX, starting with the root element. When piping data to TeX we use the current catcode regime; linebreaks and spaces are honored as usual.

Each element can have a function (command) associated and when this is the case, control is given to that function. In our case the root element has such a command, one that will trigger a setup. And so, instead of piping content to TeX, a function is called that lets TeX expand the macro that deals with this setup.

However, that setup itself calls Lua code that filters the title and feeds it into the `\section` command, next it flushes everything except the title and author, which again involves calling Lua. Last it flushes the author. The nested sequence of events is as follows:

- lua: Load the document and apply setups and alike.
- lua: Serialize the `article` element, but since there is an associated setup, tell TeX do expand that one instead.
    - tex: Execute the setup, first expand the `\section` macro, but its argument is a call to Lua.
        - lua: Filter `title` from the subtree under `article`, print the content to TeX and return control to TeX.
    - tex: Tell Lua to filter the paragraphs i.e. skip `title` and `author`; since the `b` element has no associated setup (or whatever) it is just serialized.
        - lua: Filter the requested elements and return control to TeX.
    - tex: Ask Lua to filter `author`.
        - lua: Pipe `author`'s content to TeX.
    - tex: We're done.
- lua: We're done.

This is a really simple case. In my daily work I am dealing with rather extensive and complex educational documents where in one source there is text, math, graphics, all kind of fancy stuff, questions and answers in several categories and of different kinds, either or not to be reshuffled, omitted or combined. So there we are talking about many more levels of TeX calling Lua and Lua piping to TeX etc. To stay in TeX speak: we're dealing with one big ongoing nested expansion (because Luacalls expand), and you can imagine that this somewhat stresses TeX's input stack, but so far I have not encountered any problems.

some remarks

Here I discussed several possible applications of Lua in TeX. I didn't mention yet that because LuaTeX contains a scripting engine plus some extra libraries, it can also be used purely for that. This means that support programs can now be written in Lua and that there are no longer dependencies of other scripting engines being present on the system. Consider this a bonus.

Usage in TeX can be organized in four categories:

1. Users can use Lua for generating data, do all kind of data manipulations, maybe read data from file, etc. The only link with TeX is the print function.

2. Users can use information provided by TeX and use this when making decisions. An example is collecting data in boxes and use Lua to do calculations with the dimensions. Another example is a converter from MetaPost output to pdf literals. No real knowledge of TeX's internals is needed. The MkIV xml functionality discussed before demonstrates this: it's mostly data processing and piping to TeX. Other examples are dealing with buffers, defining character mappings, and handling error messages, verbatim . . . the list is long.

3. Users can extend TeX's core functionality. An example is support for OpenType fonts: LuaTeX itself does not support this format directly, but provides ways to feed TeX with the relevant information. Support for OpenType features demands manipulating node lists. Knowledge of internals is a requirement. Advanced spacing and language specific features are made possible by node list manipulations and attributes. The alternative `\Words` macro is an example of this.

4. Users can replace existing TeX functionality. In MkIV there are numerous example of this, for instance all file io is written in Lua, including reading from zip files and remote locations. Loading and defining fonts is also under Lua control. At some point MkIV will provide dedicated splitters for multicolumn typesetting and probably also better display spacing and display math splitting.

The boundaries between these categories are not frozen. For instance, support for image inclusion and mplib in ConTeXt MkIV sits between category 3 and 4. Category 3 and 4, and probably also 2 are normally the domain of macro package writers and more advanced users who contribute to macro packages. Because a macropackage has to provide some stability it is not a good idea to let users mess around with all those internals, because of potential interference. On the other hand, normally users operate on top of a kernel using some kind of api and history has proved that macro packages are stable enough for this.

Sometime around 2010 the team expects LuaTeX to be feature complete and stable. By that time I can probably provide a more detailed categorization.

XXIV How to convince Don and Hermann to use LuaTeX

The code shown here should look a bit different in versions of MkIV after March 2011. This is because the font system was cleaned up and upgraded. The prinicples remain the same. You can have a look at `m-punk.mkiv` *in the ConTeXt distribution.*

Odds are pretty low that Don Knuth will use LuaTeX for typesetting the next update of his opus magnum, and odds are even lower that Hermann Zapf will use mplib for Melior Nova. However, the next example of combining METAFONT and TeX may draw their interest in this new variant: MetaTeX.

The font used here is called 'punk' and is designed by Donald Knuth. There is a note in the file that says: 'Font inspired by Gerard and Marian Unger's lectures, February 1985'. If you didn't notice it yet: punk is a random font.

You may wonder why we started looking into this masterpiece of font design. Well, there are a few reasons:

- We always liked the font, but after the rise of outline fonts it was not a natural candidate for using in documents. Fun is always a good motive.
- For many years we have been suggesting that special glyphs and/or aspects of typesetting could be realized by runtime generation of graphics, and we need this testbed for the Oriental TeX project: Idris needs stretchable inter-glyph connections.
- Taco likes using tricky MetaPost backgrounds for his presentations that demonstrate this programming language.
- Hartmut loves to tweak the backend and runtime font generation will demand some extensions to the font inclusion and literal handlers.
- Because Hans attends many TeX conferences together with Volker Schaa, he has promised him to avoid repeating talk and presentation layouts, and so a new presentation style was needed.

To this we can add an already mentioned motivation: convince Don and Hermann to use LuaTeX . . . who knows. And, if that fails, maybe they can team up for an extensions to this font: more style variants, proper math and the full range of Unicode glyphs.

The punk font is written in METAFONT and there are multiple sources. These are merged into one file which is to be processed using the `mfplain` format. Definitions of characters in this font look like:

```
beginpunkchar("A",13,1,2) ;
    z1 = pp(1.5u,0) ; z2 = (.5w,1.1h) ; z3 = pp(w-1.5u,0) ;
    pd z1 ; pd z3 ; draw z1 -- z2 -- z3 ;
    z4 = pp .3[z1,z2] ; z5 = pp .3[z3,z2] ;
    pd z4 ; pd z5 ; draw z4 -- z5;
endchar ;
```

When TeX needs a font, i.e. when we have something like this:

```
\font\somefont=whatever at 12pt
```

in ConTeXt control is delegated to a font loader written in Lua that is hooked into TeX. This loader interprets the name and if needed filters the specification from it. Think of this:

```
\font\somefont=whatever*smallcaps at 16pt
```

This means: load font whatever and enable the smallcaps features. However this mechanism is mostly geared towards Type1 and OpenType fonts. But punk is neither: it's a METAFONT, and we need to treat it as such. We will use LuaTeX's powerful virtual font technology because that way we can smuggle the proper shapes in the final file. And . . . no bitmaps and no funny encoding.

In ConTeXt MkIV there is a preliminary virtual font definition mechanism. There is no advanced TeX interface yet so we need to do it in Lua. Fortunately we do have access to this from the font mechanism:

```
\font\somefont=mypunk@punk at 20pt
```

This is a rather valid directive to create a font that internally will be called mypunk. For this the virtual font creation command punk will be used, and in a moment we will see what this triggers.

Of course, users will never see such low level definitions. They will use proper typescript, which set up a whole font system. For instance, in this document we use:

Now, using punk in inself is not that much of a challenge, but how about using multiple instances of this font and then typeset the text chosing variants of a glyph at random. Of course this will have some trade-off in terms of runtime. In this document we use punk as the bodyfont and therefore it comes in several sizes. On Hans's laptop generating the glyphs takes a while:

```
7500 glyphs, 12.887 seconds runtime, 581 glyphs/second
```

Fortunately MkIV provides a caching mechanism so once the fonts are generated, a next run will be more comfortable. This time we get reported:

```
0.187 seconds, 60 instances, 320.856 instances/second
```

which is not that bad for loading 60 files of 5 megabytes pdf literals each. The reason why the files are large is that although these glyphs look simple, in fact they are rather complex: each glyph at least one paths and several knots, and since a special pen is used, conversion results in a larger than normal description of a shape.

Since we use the standard converter from MetaPost to pdf, we can gain some generation time by using a dedicated converter for glyphs. Eventually the mplib library may even provide a proper charstring generator so that we can construct real fonts at runtime.

So, how does this work behind the screens? Because we can use some of the mechanisms already present in ConTeXt it is not even that complex.

- The punk directive tells ConTeXt to create a virtual font. Such a font can be made out of real fonts; we use this for instance in the font feature combine, where we add virtually composed characters that are missing by combining characters present. However, here we have no real font.

- And so this virtual font is not build on top of an existing font, but spawns a mplib process that will build the font, unless it is present in the cache on disk. The shapes are converted to pdf literals and for each character a proper definition table is made.

- In total 10 such fonts are made, but only one is returned to the font callback that asked us to provide the font. The list of the alternatives is stored in the Lua table that represents the font and kept at the Lua end. So, for each size used, a unique set of 10 variants is generated.
- The randomizer operates on the node list. Instead of using a dedicated mechanism for this, we hijack one of the attribute values of the case swapper already present in MkIV. After that we can selectively turn on and off the randomizer.
- At some point TeX will hand over the node lists to ConTeXt. At that moment a lot of things can happen to the list, and one of them is a sequence of character handlers, of which the mentioned case handler is one. The handler sweeps over the nodelist and for each glyph node triggers a function that is bound to the attribute value.
- This function is rather trivial: it looks at the font id of the glyph, and resolves it to the font table. If that table has a list of alternatives, it will randomly choose one and assign it to the font attribute of the glyph. That's all.
- Eventually the backend routines will inject the pdf literals that were collected in the commands table of the virtual glyph.

It will not come as a surprise that our resulting file is larger than what we get when using traditional outline fonts or just one instance of Punk. However, this is just an experiment, and eventually a proper font constructor will be provided, so that the glyph drawing is delegated to the font renderer. An intermediate optimization can be to use so called pdf xforms, but a properly runtime generated font is best because then we can search in the file too.

Because by now reading the Punk font should go fluently we can now move on to the code. We already have a `fonts` namespace, which we now extend with an MetaPost sub namespace:

```
fonts.mp = fonts.mp or { }
```

We set a version number and define a cache on disk. When the number changes fonts stored in the cache will be regenerated when needed. The `containers` module provides the relevant function.

```
fonts.mp.version = 1.01
fonts.mp.cache = containers.define("fonts", "mp", fonts.mp.version, true)
```

We already have a `metapost` namespace, and within it we define a sub namespace:

```
metapost.characters = metapost.characters or { }
```

Now we're ready for the real action: we define a dedicated flusher that will be passed to the MetaPost converter. A next version of mplib will provide the tfm font information which gives better glyph dimensions, plus additional kerning information. All this code is defined in a closure (`do ... end`) which nicely hides the local variables.

```
local characters, descriptions = { }, { }
local factor, total, variants = 100, 0, 0
local l, n, w, h, d  = { }, 0, 0, 0, 0
local flusher = {
    startfigure = function(chrnum,llx,lly,urx,ury)
        l, n = { }, chrnum
        w, h, d = urx - llx, ury, -lly
        total = total + 1
    end,
    flushfigure = function(t)
        for i=1, #t do
            l[#l+1] = t[i]
        end
    end,
    stopfigure = function()
        local cd = characters.data[n]
        descriptions[n] = {
            unicode = n,
            name = cd and cd.adobename,
            width = w*100,
            height = h*100,
            depth = d*100,
        }
        characters[n] = {
            commands = {
                { "special", "pdf: " .. table.concat(l," ") },
            }
        }
    end
}
```

In the normal converter, the start and stop function do the packaging in a box. The flush function is called when literals need to be flushed. This threesome does as much as collecting glyph information in the list table. Intermediate literals are stored in the l table. Each glyph has a description and (in this case) one command that defines the virtual shape. The name is picked up from the character data table that is present in MkIV.

As told before we generate multiple instances per requested font and here is how it happens. We initialize the mpplain format and reset it afterwards. The funk definition file is adapted for multiple runs. Scaling happens here because later on the scaler has no knowledge about what is present in the commands. We use a few helpers for processing the MetaPost code and format the final font table in a way ConTeXt MkIV likes. Currently the parameters (font dimensions) are rather hard coded, but this will change when mplib can provide them.

```
function metapost.characters.process(mpxformat, name, instances, scalefactor)
    statistics.starttiming(metapost.characters)
    scalefactor = scalefactor or 1
    instances = instances or 10
    local fontname = file.removesuffix(file.basename(name))
    local hash = file.robustname(string.format(
        "%s %04i %04i", fontname, scalefactor*1000, instances))
    local lists = containers.read(fonts.mp.cache, hash)
    if not lists then
        statistics.starttiming(flusher)
        local data = io.loaddata(resolvers.findfile(name))
        metapost.reset(mpxformat)
        lists = { }
        for i=1,instances do
            characters, descriptions = { }, { }
            metapost.process(
                mpxformat,
                {
                    "randomseed := " .. i*10 .. ";",
                    "scale_factor := " .. scalefactor .. " ;",
                    data
                },
                false,
                flusher
            )
            lists[#lists+1] = {
                designsize = 655360,
                name = string.format("%s-%03i",hash,i),
                parameters = {
                    slant         =   0,
                    space         = 333   * scalefactor,
                    space_stretch = 166.5 * scalefactor,
                    space_shrink  = 111   * scalefactor,
                    x_height      = 431   * scalefactor,
                    quad          = 1000  * scalefactor,
                    extra_space   =   0
                },
                ["type"] = "virtual",
                characters = characters,
                descriptions = descriptions,
            }
        end
        metapost.reset(mpxformat) -- saves memory
        lists = containers.write(fonts.mp.cache, hash, lists)
        statistics.stoptiming(flusher)
    end
    variants = variants + #lists
    statistics.stoptiming(metapost.characters)
    return lists
end
```

We're not yet there. This was just a font generator that returns a list of fonts defined in a format liked by MkIV and not that far from what TeX wants back from us. Next we define the main definition function, the one that is called when the font is defined as virtual font. The special number -1000 tells the scaler to honour the designsize, which boils down to no scaling, but just copying to the final table that is passed to TeX. The define function returns an id which we will use later.

The scaled uses the descriptions to add dimensions (and other data needed) in the characters table. This is something MkIV specific.

```
function fonts.handlers.vf.combiner.commands.metafont(g,v)
    local size = g.specification.size
    local data = metapost.characters.process(v[2],v[3],v[4],size/655360)
    local list, t = { }, { }
    for d=1,#data do
        t = data[d]
        t = fonts.constructors.scale(t, -1000)
        t.id = font.define(t)
        list[#list+1] = t.id
    end
    for k, v in pairs(t) do
        g[k] = v -- kind of replace, when not present, make nil
    end
    g.variants = list
end
```

We hook this into the ConTeXt font handler and from now on the \punk is recognized:

```
fonts.definers.methods.install( "punk", { { "metafont", "mpfplain", "punkfont.mp", 10 } } )
```

Now that we can define the font, we need to deal with the randomizer. This is optional fun. The mentioned case swappers are implemented in the cases namespace:

```
local fontdata = fonts.hashes.identifiers

cases.actions[99] = function(current)
    local c = current.char
    local used = fontdata[current.font].variants
    if used then
        local f = math.random(1,#used)
        current.font = used[f]
        return current, true
    else
        return current, false
    end
end
```

This function is called in one of the passes over the node list. Thanks to this framework we don't need that much code. We didn't show two statistics functions. They are the reason why we keep track of the total number of glyphs defined.

This leaves us defining the interface, so here we go:

```
\def\StartRandomPunk{\begingroup \setcharactercasing[99]}
\def\StopRandomPunk {\endgroup}
```

The set command just sets the attribute that we associated with casing (one of the many attributes). The number 99 is rather arbitrary.

If you follow the development of LuaTeX and MkIV (we do talks at conferences, keep track of the development history in mk.pdf, and report on the ConTeXt mailing list) you will have noticed that we often use somewhat extreme examples to explore and test the functionality and this is no exception. As usual it helped us to improve the code and extend our todo list. Can the previous code convince the grand wizards to start using LuaTeX? Probably not. Let's anyway hope that they will put the addition of punk math to their todo list. In the meantime we've already started adding missing characters:

{ ' \ " } { ' \ " } { ' \ " } { ' \ " } { ' \ " } { ' \ " }

Also, because we can be sure that Mojca Miklavec's first test will be if her favourite characters č, š and ž are supported, we made sure that we composed those accented characters as well.[2]

---

[2] This is accomplished by adding composecharacters(t) at an undisclosed location in the previous code.

XXV OpenType: too open?

In this chapter I will reflect on OpenType from within my limited scope and experience. What I'm writing next is my personal opinion and I may be wrong in many ways.

Until recently installing fonts in a TeX system was not something that a novice user could do easily. First of all, the number of files involved is large:

- If it is a bitmap font, then for each size used there is a PK file, and this is reflected in the suffix, for instance `pk300`.
- If it is an outline font, then there is a Type1 file with suffix `pfb` or sometimes glyphs are taken from OpenType fonts (with `ttf` or `otf` as suffix). In the worst case such wide fonts have to be split into smaller ones.
- Because TeX needs information about the dimensions of the glyphs, a metric file is needed; it has the suffix `tfm`. There is limit of 256 characters per font.
- If the font lacks glyphs it can be turned into a virtual font and borrow glyphs from other fonts; this time the suffix is `vf`.
- If no such metric file is present, one can make one from a file that ships with the fonts; it has the suffix `afm`.
- In order to include the font in the final file, the backend to TeX has to match glyph references to the glyph slots in the font file, and for that it needs an encoding vector, for historical reasons this is a PostScript blob in a file with suffix `enc`.
- This whole lot is associated in a map file, with suffix `map`, which couples metric files to encodings and to font files.

Of course the user also needs TeX code that defines the font, but this differs per macro package. If the user is lucky the distributions ships with files and definitions of his/her favourite fonts, but otherwise work is needed. Font support in TeX systems has been complicated by the facts that the first TeX fonts were not ASCII complete, that a 256 limit does not go well with multilingual typesetting and that most fonts lacked glyphs and demanded drop-ins. Users of ConTeXt could use the `texfont` program to generate metrics and map file for traditional TeX but this didn't limit the number of files.

In modern TeX engines, like XeTeX and LuaTeX, less files are needed, but even then some expertise is needed to use Type1 fonts. However, when OpenType fonts are used in combination with Unicode, things become easy. The (one) fontfile needs to be put in a location that the TeX engine knows and things should work.

In LuaTeX with ConTeXt MkIV support for traditional Type1 fonts is also simplified: only the `pfb` and `afm` files are needed. Currently we only need `tfm` files for math fonts but that will change too. Virtual fonts can be built at runtime and we are experimenting with real time font generation. Of course filenames are still just as messy and inconsistent as ever, so other tools are still needed to figure out the real names of fonts.

So, what is this OpenType and will it really make TeXies life easier? The qualification 'open' in OpenType carries several suggestions:

- the format is defined in an open way, everybody can read the specification and what is said there is clear
- the format is open in the sense that one can add additional features, so there are no limits and/or limits can be shifted
- there is an open community responsible for the advance of this specification and commercial objectives don't interfere and/or lead to conflicts

Is this true or not? Indeed the format is defined in the open although the formal specification is an expensive document. A free variant is available at the Microsoft website but it takes some effort to turn that into a nicely printable document. What is said there is quite certainly clear for the developers, but it takes quite some efforts to get the picture. The format is binary so one cannot look into the file and see what happens.

The key concept is 'features', which boils down to a collection of manipulations of the text stream based on rules laid down in the font. These can be simple rules, like 'replace this character by its smallcaps variant' or more complex, like 'if this character is followed by that character, replace both by yet another'. There are currently two classes of features: substitutions and (relative) positioning. One can indeed add features so there seem to be no limits.

The specification is a hybrid of technologies developed by Microsoft and Adobe with some influence by Apple. These companies may have conflicting interests and therefore this may influence the openness.

So, in practice we're dealing with a semi-open format, crippled by a lack of documentation and mostly controlled by some large companies. These characteristics make that developing support for OpenType is not that trivial. Maybe we should keep in mind that this font format is used for word processors (no focus on typography), desk top publishing (which permits in-situ tweaking) and rendering text in graphical user interfaces (where script and language specific rendering is more important than glyph variants). Depending on the use features can be ignored, or applied selectively, of even compensated.

Anyhow, a font specification is only part of the picture. In order to render it useful we need support in programs that display and typeset text and of course we need fonts. And in order to make fonts, we need programs dedicated to that task too.

Let's go back for a moment to traditional TEX. A letter can be represented by its standard glyph or by a smallcaps variant. A digit can be represented by a shape that sits on the baseline, or one that may go below: an oldstyle numeral. Digits can have the same width, or be spaced proportionally. There can be special small shapes for super- and subscripts. In traditional TEX each such variant demanded a font. So, say that one wants normal shapes, smallcaps and oldstyle, three fonts were needed and this for each of the styles normal, bold, italic, etc. Also a font switch is needed in order to get the desired shapes.

In an OpenType universe normal, smallcaps and oldstyle shapes can be included in one font and they are organized in features. It will be clear that this will make things easier for users: if one buys a font, there is no longer a need to sort out what file has what shapes, there is no longer a reason for reencodings because there is no 256 limit, map files are therefore obsolete, etc. Only the TEX definition part remains, and even that is easier because one file can be used in different combinations of features.

One of the side effects of the already mentioned semi-open character of the standard is that we cannot be completely sure about how features are implemented. Of course one can argue that the specification defines what a feature is and how a font should obey it, but in practice it does not work out that way.

- Nobody forces a font designer (or foundry) to implement features. And if a designer provides variants, they may be incomplete. In the transition from Type1 to OpenType fonts may even have no features at all.

- Some advanced features, like fractions, demand extensive substitution rules in the font. The completeness may depend on the core application the font was made for, or the ambition of the programmer who assists the designer, or on the program that is used to produce the font.

- Many of the features are kind of generic, in the sense that they don't depend on heuristics in the typesetting program: it's just rules that need to be applied. However, the typesetting program may be written in such a way that it only recognized certain features.

- Some features make assumptions, for instance in the sense that they expect the program to figure out what the first character of a word is. Other features only work well if the program implements the dedicated machinery for it.

- Features can originate from different vendors and as a result programs may interpret them differently. Developers of programs may decide only to support certain features, even if similar features can be supported out of the box. In the worst case a symbiosis between bugs in programs and bugs in fonts from the same vendor can lead to pseudo standards.

- Designers (or programmers) may assume that features are applied selectively on a range of input, but in automated workflows this may not be applicable. Style designers may come up with specifications that cannot be matched due to fonts that have only quick and dirty rules.

- Features can be specific for languages and scripts. There are many languages and many scripts and only a few are supported. Some features cover similar aspects (for instance ligatures) and where a specific rendering ends up in the language, script, feature matrix is not beforehand clear.

In some sense OpenType fonts are intelligent, but they are not programs. Take for instance the frac feature. When enabled, and when supported in the font, it *may* result in 1/2 being typeset with small symbols. But what about a/b? or this/that? In principle one can have rules that limit this feature to numerals only or to a simple cases with a few characters. But I have seen fonts that produce garbage when such a feature is applied to the whole text. Okay, so one should apply it selectively. But, if that's the way to go, we could as well have let the typesetting program deal with it and select superior and inferior glyphs from the font. In that case the program can deal with fuzzy situations and we're not dependent on the completeness of rules. In practice, at least for the kind of applications that I have for TEX, I cannot rely on features being implemented correctly.

For ages TEXies have been claiming that their documents can be reprocessed for years and years. Of course there are dependencies on fonts and hyphenation patterns, but these are relatively stable. However, in the case of OpenType we have not only shapes, but also rules built in. And rules can have bugs. Because fonts vendors don't provide automated updating as with programs, your own system can be quite stable. However, chances are that different machines have variants with better or worse rules, or maybe even with variants with features deleted.

I'm sure that at some time Idris Samawi Hamid of the Oriental TEX project (related to LuaTEX) will report on his experiences with font editors, feature editors, and typesetting engines in the process of making an Arabic font that performs the same way in all systems. Trial and error, rereading the specifications again and again, participating in discussions on forums, making special test fonts . . . it's a pretty complex process. If you want to make a font that works okay in many applications you need to test your font with each of them, as the Latin Modern and TEX Gyre font developers can tell you.

This brings me to the main message of this chapter. On the one hand we're better of with OpenType fonts: installation is trivial, definitions are easy, and multi-lingual documents are no problem due to the fact that fonts are relatively complete. However, in traditional TEX the user just used what came with the system and most decisions were already made by package writers. Now, with OpenType, users can choose features and this demands some knowledge about what they are, when they are supposed to be used (!), and what limitations they carry. In traditional TEX the options were limited, but now there are many under user control. This demands some discipline. So, what we see is a shift from technology (installing, defining) to application (typography, quality). In ConTEXt this has resulted in additional interfaces, like for instance dynamic feature switching, which decouples features from font definitions.

It is already clear that OpenType fonts combined with Unicode input will simplify TEX usage considerably. Also, for macro writers things become easier, but they should be prepared to deal with the shortcomings on both Unicode and OpenType. For instance characters that belong together are not always organized logically in Unicode, which results for instance in math characters being (sort of) all over the place, which in turn means that in TEX characters can be either math or text, which in turn relates to the fonts being used, formatting etc. Als, macro package writers now need to take more languages and related interferences into account, but that's mostly a good thing, because it improves the quality of the output.

It will be interesting to see how ten years from now TEX macro packages deal with all the subtleties, exceptions, errors, and user demands. Maybe we will end up with as complex font support as for Type1 with its many encodings. On the other hand, as with all technology, OpenType is not the last word on fonts.

XXVI It works!

One of the more powerful commands in ConTeXt is `\framed`. You can pass quite some parameters that control the spacing, alignment, backgrounds and more. This command is used all over the place (although often hidden for the user) which means that it also has to be quite stable. However, there is one nasty bit of code that is hard to get right. Calculating the height of a box is not that complex: the height that TeX reports is indeed the height. However, the width of box is determined by the value of `\hsize` at the time of typesetting. The actual content can be smaller. In the `\framed` macro by default the width is calculated automatically.

```
\framed
  [align=middle,width=fit]
  {Out beyond the ethernet the spectrum spreads \unknown}
```

this shows up as:[3]

Out beyond the ethernet the spectrum spreads ...

Or take this quote:[4]

```
\hsize=.6\hsize \framed [align=middle,width=fit] {\input weisman }
```

This gives a multi-line paragraph:

Since the mid-1990s, humans have taken an
unprecedented step in Earthly annals by
introducing not just exotic flora or fauna from one
ecosystem into another, but actually inserting
exotic genes into the operating systems
of individual plants and animals, where
they're intended to do exactly the same
thing: copy themselves, over and over.

Here the outer `\hsize` was made a bit smaller. As you can see the frame is determined by the widest line. Because it was one of the first features we needed, the code in ConTeXt that is involved in determining the maximum natural width is pretty old. It boils down to unboxing a `\vbox` and stepwise grabbing the last box, penalty, kern and skip. You unwind the box backwards. However, you cannot grab everything or in TeX speak: there is only a limited number of `\lastsomething` commands. Special nodes, like whatsits

---

[3] Taken from 'Casino Nation' by Jackson Browne.

[4] Taken from 'A World Without Us' by Alan Weisman.

cannot be grabbed and they make the analyzer abort its analysis. There is no way that we can solve this in traditional TEX and in CONTEXT MKII.

So how about LUATEX and CONTEXT MKIV? The macro used in the `\framed` commands is:

```
\doreshapeframedbox{do something with \box\framebox}
```

In LUATEX we can manipulate box content at the LUA level. Instead of providing a truckload of extra primitives (which would also introduce new data types at the TEX end) we just delegate the job to LUA.

```
\def\doreshapeframedbox
  {\ctxlua{commands.doreshapeframedbox(\number\framebox)}}
```

Here `\ctxlua` is our reserved instance and `commands` provides the namespace for commands that we delegate to LUA (so, there are more of them). The amount of LUA code is way less than the TEX code which we will not show here; it's in `supp-box.tex` if you really want to see it.

```
function commands.doreshapeframedbox(n)
    local box_n = tex.box[n]
    if box_n.width ~= 0 then
        local hpack = node.hpack
        local free = node.free
        local copy = node.copy_list
        local noflines, lastlinelength, width = 0, 0, 0
        local list = box_n.list
        local done = false
        for h in node.traverse_id('hlist',list) do
            done = true
            local p = hpack(copy(h.list))
            lastlinelength = p.width
            if lastlinelength > width then
                width = lastlinelength
            end
            free(p)
        end
        if done then
            if width ~= 0 then
                for h in node.traverse_id('hlist',list) do
                    if h.width ~= width then
                        h.list = hpack(h.list,width,'exactly')
                        h.width = width
                    end
                end
            end
            box_n.width = width
        end
        -- we can also do something with lastlinelength
    end
end
```

In the first loop we inspect all lines (nodes with type `hlist`) and repack them to their natural width with `node.hpack`. In the process we keep track of the maximum natural width. In the second loop we repack the content again, but this time permanently. Now we use the maximum encountered width which is forced by the keyword `exactly`. Because all glue is still present we automatically get the desired alignment. We create local shortcuts to some node functions which makes it run faster; keep in mind that this is a core function called many times in a regular ConTEXt job.

In ConTEXt MkIV you will find quite some Lua code and often it looks rather complex, especially if you have no clue why it's needed. Think of OpenType font handling which involves locating fonts, loading and caching them, storing features and later on applying them to node lists, etc. However, once we are beyond the stage of developing all the code that is needed to support the basics, we will start doing the things that more relate to the typesetting process itself, like the previous code. One of the candidates for a similar Lua based solution is for instance column balancing. From the previous example code you can deduce that manipulating the node lists from Lua can make that easier. Of course we're a few more years down the road then.

XXVII Virtual Reality

When a font lacks glyphs we can add new ones by making the font virtual. A virtual font has virtual glyphs: instead of a reference to a slot in the current font, such a glyph refers to a slot in another font, or it combines several glyphs into one, or it just contains code that ends up in the result (for instance a sequence of PDF commands that describes the shape). For TEX a character and its dimensions are what matters and what ends up in the result is mostly a matter for the backend. In LUATEX the backend is integrated but even then during the typesetting process only the characteristics of a glyph are used and not the shape.

In CONTEXT we have a feature called 'compose' which extends the font with extra characters and constructs its representation from those of other characters.

```
\definefontfeature
  [composes]
  [kern=yes,ligatures=yes,compose=yes]
```

When this feature is applied, CONTEXT will try to fill in the gaps in the UNICODE vector of the font based on for instance (de)composition information. Of course this has some limitations. For instance OPENTYPE fonts can ships with features, like smallcaps. Currently we ignore this property when we add composed characters. Technically it is no big deal to add variants but we simply didn't do it yet at the time of this writing. After all, such fallbacks can best be avoided by using proper fonts.

Our CONTEXT MKIV wishlist mentions a mechanism for combining fonts into one font. For this we can use virtual fonts and the machinery for that is in available in LUA code. However such a mechanism will be used for more drastic completion of a font than the compose feature. For instance, often Chinese fonts lack proper Latin glyphs and vise versa. But when we combine such fonts we really do want to keep OPENTYPE features working and so we cannot use virtual fonts (unless we start merging features which can become really messy and runtime consuming).

There is a relative simple solution using real fonts that kind of behave like virtual ones: virtual real fonts. The trick is in the fact that TEX permits access to characters not present in the font. Say that we have

```
<char 123><char 124><char 125>
```

and that slot 124 has no glyph. In that case TEX just inserts a glyph node with a reference to the current font and this character. Of course, when we let TEX carry on, at some point it will need glyph properties like the width, height and/or depth. And in the backend, when writing the result to file, TEX wants to insert the glyph data in the file. In both cases we end up with a message in the log file and a result file with missing data.

In ConTEXt MkIV we intercept the node lists at several points and one of those is directly after the construction. So let's consider the previous example again.

```
<font 32 char 123><font 32 char 124><font 32 char 125>
```

Because the font has no character 124 we need a way to substitute it with another character. All we have to do is to change the font identifier 32 into one that makes sense. Such a replacement loop is kind of trivial.

```
for n in traverse_id(glyph,head) do
    local v = vectors[n.font]
    if v then
        local id = v[n.char]
        if id then
            n.font = id
        end
    end
end
```

We have a table (`vectors`) that can have a subtable (`v`) for font with id (`n.font`) in which there can be a reference from the current character (`n.char`) to another font (`id`) that we use to replace the font reference (`n.font`).

Filling the table is relatively easy but an explanation is beyond this chapter. We only show the high level interface, one that certainly will evolve.

```
\definefontfallback
  [SerifFallback]
  [Mono]
  [0x000-0x3FF]
  [check=yes,force=no]
```

This command registers an entry in the `SerifFallback` namespace. There can be multiple replacement in row (by just using more of these commands), but here we have only one. The range 0x000–0x3FF will be checked and if the main font lacks a glyph in that range, it will be taken from the font with the symbolic name `Mono`. That name will be resolved when the fallback is associated with a font. The `check` option tells the machinery that we need to check for existence and because we don't `force`, we will only replace missing glyphs. There is also an `rscale` option, that permits relative scaling of the fallback font to the main font, something that may be needed when fonts come from different sources.

```
\definefontsynonym
  [SerifPlus]
  [Serif]
  [fallbacks=SerifFallback]
```

This command associates a fallback with a font. There is always a parent font and that is the font that triggers the checking of the node list.

```
\definefont [MySerif] [SerifPlus at 10pt]
```

Here we defines a font called `\MySerif` that refers to a symbolic name `SerifPlus` which in turn refers to the current `Serif` font (these symbolic names are resolved in typescripts, one of the building blocks of ConTeXts font system). The mentioned fallbacks will be initialized when the font is defined. This examples demonstrates that there is a clean separation between font definitions and fallbacks. This makes it possible to share fallback definitions.

So, let's summarize what happens:

- a font is defined in the normal way but has falbacks
- the associated fallback fonts are defined too
- the main font gets a table with fallback id's
- the main font is used in the document stream
- the node list is intercepted and parsed for this font
- references to fallback fonts take care of missing glyphs

We end with an example.

```
\definefontfallback [Demo] [Mono] [0x30-0x39] [force=yes]
\definefontsynonym  [DemoSerif] [Serif] [fallbacks=Demo]

\definefont [MyDemoSerif] [DemoSerif at 20pt]

\MyDemoSerif Here the digits, like 13579, are replaced.
```

Here the digits, like 13579, are replaced.

Beware: the fallback definitions are global, but this is hardly a problem because normal such trickery is taking place at the document level.

XXVIII Everything structure

At the time of this writing, ConTEXt MkIV spends some 50% of its time in Lua. There are several reasons for this.

- All IO goes via Lua, including messages and logging. This includes file searching which happened to be done by the KPSE library.
- Much font handling is done by Lua too, for instance OpenType features are completely handled by Lua.
- Because TEX is highy optimized, its influence on runtime is less prominent. Even if we delegate some tasks to Lua, TEX still has work to do.

Among the reported statistics of a 242 page version of `mk.pdf` (not containing this chapter) we find the following:

```
input load time          - 0.094 seconds
startup time             - 0.905 seconds (including runtime option file processing)
jobdata time             - 0.140 seconds saving, 0.062 seconds loading
fonts load time          - 5.413 seconds
xml load time            - 0.000 seconds, lpath calls: 46, cached calls: 31
lxml load time           - 0.000 seconds preparation, backreferences: 0
mps conversion time      - 0.000 seconds
node processing time     - 1.747 seconds including kernel
kernel processing time   - 0.343 seconds
attribute processing time - 2.075 seconds
language load time       - 0.109 seconds, n=4
graphics processing time  - 0.109 seconds including tex, n=7
metapost processing time  - 0.484 seconds, loading: 0.016 seconds, execution: 0.203 seconds, n: 65
current memory usage     - 332 MB
loaded patterns          - gb:gb:pat:exc:3 nl:nl:pat:exc:4 us:us:pat:exc:2
control sequences        - 34245 of 165536
callbacks                - direct: 235579, indirect: 18665, total: 254244 (1050 per page)
runtime                  - 25.818 seconds, 242 processed pages, 242 shipped pages, 9.373 pages/second
```

The startup time includes initial font loading (we don't store fonts in the format). Jobdata time involves loading and saving multipass data used for tables of contents, references, positioning, etc. The time needed for loading fonts is over 5 seconds due to the fact that we load a couple of real large and complex fonts. Node processing time mostly is related to OpenType feature support. The kernel processing time refers to hyphenation and line breaking, for which (of course) we use TEX. Direct callbacks are implicit calls to Lua, using `\directlua` while the indirect calls concern overloaded TEX functions and callbacks triggered by TEX itself.

Depending on the system load on my laptop, the throughput is around 10 pages per second for this document, which is due to the fact that some font trickery takes place using a few arabic fonts, some chinese, a bunch of metapost punk instances, Zapfino, etc.

The times reported are accumulated times and contain quite some accumulated rounding errors so assuming that the operating system rounds up the times, the totals in practice might be higher. So, looking at the numbers, you might wonder if the load on LUA will become even larger. This is not necessary. Some tasks can be done better in LUA but not always with less code, especially when we want to extend functionality and to provide more robust solutions. Also, even if we win some processing time we might as well waste it in interfacing between TEX and LUA. For instance, we can delegate pretty printing to LUA, but most documents don't contain verbatim at all. We can handle section management by LUA, but how many section headers does a document have?

When the future of TEX is discussed, among the ideas presented is to let TEX stick to typesetting and implement it as a component (or library) on top of a (maybe dedicated) language. This might sound like a nice idea, but eventually we will end up with some kind of user interface and a substantial amount of code dedicated to dealing with fonts, structure, character management, math etc.

In the process of converting CONTEXT to MKIV we try to use each language (TEX, LUA, METAPOST) for what it is best suited for. Instead of starting from scratch, we start with existing code and functionality, because we need a running system. Eventually we might find TEX's role as language being reduced to (or maybe we can better talk of 'focused on') mostly aspects of typesetting, but CONTEXT as a whole will not be much different from the perspective of the user.

So, this is how the transition of CONTEXT takes place:

- We started with replacing isolated bits and pieces of code where LUA is a more natural candidate, like file IO, encoding issues.
- We implement new functionality, for instance OPENTYPE and TYPE1 support.
- We reimplement mechanisms that are not efficient as we want them to be, like buffers and verbatim.
- We add new features, for instance tree based XML processing.
- After evaluating we reimplement again when needed (or when LUATEX evolves).

Yet another transition is the one we will discuss next:

- We replace complex mechanisms by new ones where we separate management and typesetting.

This not so trivial effort because it affects many aspects of CONTEXT and as such we need to adapt a lot of code at the same time: all things related to structure:

- sectioning (chapters, sections, etc)
- numbering (pages, itemize, enumeration, floats, etc)
- marks (used for headers and footers)
- lists (tables of contents, lists of floats, sorted lists)
- registers (including collapsing of page ranges)
- cross referencing (to text as well as pages)
- notes (footnotes, endnotes, etc)

All these mechanisms are somehow related. A section head can occur in a list, can be cross referenced, might be shows in a header and of course can have a number. Such a number can have multiple components (1.A.3) where each component can have its own conversion, rendering (fonts, colors) and selectively have less components. In tables of contents either or not we want to see all components, separators etc. Such a table can be generated at each level, which demands filtering mechanisms. The same is true for registers. There we have page numbers too, and these may be prefixed by section numbers, possibly rendered differently than the original section number.

Much if this is possible in ConTEXt MkII, but the code that deals with this is not always nice and clean and right from the start of the LuaTEX project it has been on the agenda to clean it up. The code evolved over time and functionality was added when needed. But, the projects that we deal with demand more (often local) control over the components of a number.

What makes structure related data complex is that we need to keep track of each aspect in order to be able to reproduce the rendering in for instance a table of contents, where we also may want to change some of the aspects (for instance separators in a different color). Another pending issue is XML and although we could normally deal with this quite well, it started making sense to make all multi-pass data (registers, tables of content, sorted lists, references, etc.) more XML aware. This is a somewhat hairy task, if only because we need to switch between TEX mode and XML mode when needed and at the same time keep an eye on unwanted expansion: do we keep structure in the content or not?

Rewriting the code that deals with these aspects of typesetting is the first step in a separation of code in MkII and MkIV. Until now we tried to share much code, but this no longer makes sense. Also, at the ConTEXt conference in Bohinj (2008) it was decided that given the development of MkIV, it made sense to freeze MkII (apart from bug fixes and minor extensions). This decision opens the road to more drastic changes. We will roll back some of the splits in code that made sharing code possible and just replace whole components of ConTEXt as a whole. This also gives us the opportunity to review code more drastically than until now in the perspective of ε-TEX.

Because this stage in the rewrite of ConTEXt might bring some compatibility issues with it (especially for users who use the more obscure tuning options), I will discuss some of the changes here. A bit of understanding might make users more tolerant.

The core data structure that we need to deal with is a number, which can be constructed in several ways.

sectioning	1.A.2.II some title
pagenumber	page 1.A - 23
reference	in chapter 2.II
marking	A: some title with preceding number
contents	2.II some title with some page number 1.A - 23
index	some word 23, A - 42 — B - 48
itemize	a first item a.1 subitem item
enumerate	example 1.A.2.II . a
floatcaption	figure 1 - 2
footnotes	note ⋆

In this table we see how numbers are composed:

section number	It has several components, separated by symbols and with an optional final symbol
separator	This can be different for each level and can have dedicated rendering options
page number	That can be preceded by a (partial) sectionnumber and separated from the page number by another symbol
counter	It can be preceded by a (partial) sectionnumber and can also have subnumbers with its own separation properties
symbol	Sometimes numbers get represented by symbols in which case we use pagewise restarting symbol sets

Say that at some point we store a section number and/or page number. With the number we need to store information about the conversion (number, character, roman numeral, etc) and the separators, including their rendering. However, when we reuse that stored information we might want to discard some components and/or use a different rendering. In traditional ConTEXt we have control over some aspects but due to the way numbers are stored for later reuse this control is limited.

Say that we have cloned a subsection head as follows:

```
\definehead[MyHead][section]
```

This is used as:

```
\MyHead[example]{Example}
```

In MkII we save a list entry (which has the number, the title and a reference to the page) and a reference to the the number, the title and the page (tagged `example`). Page numbers are stored in such a way that we can filter at specific section levels. This permits local tables of contents.

The entry in the multi pass data file looks as follows (we collect all multi pass data in one file):

```
\mainreference{}{example}{2--0-1-1-0-0-0-0--1}{1}{{I.I}{Example}}%
\listentry{MyHead}{2}{I.I}{Example}{2--0-1-1-0-0-0-0--1}{1}%
```

In MkIV we store more information and use tables for that. Currently the entry looks as follows:

```
structure.lists.collected={
 {
   ...
 },
 {
  metadata={
   catcodes=4,
   coding="tex",
   internal=2,
   kind="section",
   name="MyHead",
   reference="example",
  },
  pagenumber={
   numbers={ 1, 1, 0 },
  },
  sectionnumber={
   conversion="R",
   conversionset="default",
   numbers={ 0, 2 },
   separatorset="default",
  },
  sectiontitle={
   label="MyHead",
   title="Example",
  },
 },
 {
  ...
 },
}
```

There can be much more information in each of the subtables. For instance, the `pagenumber` and `sectionnumber` subtables can have `prefix`, `separatorset`, `conversion`, `conversionset`, `stopper`, `segments` and `connector` fields, and the `metadata` table can contain information about the XML root document so that associated filtering and handling can be reconstructed. With the section title we store information about the preceding label text (seldom used, think of 'Part B').

This entry is used for lists as well as cross referencing. Actually, the stored information is also used for markings (running heads). This means that these mechanisms must be able to distinguish between where and how information is stored.

These tables look rather verbose and indeed they are. We end up with much larger multipass data files but fortunately loading them is quite efficient. Serializing on the other hand might cost some time which is compensated by the fact that we no longer store information in token lists associated with nodes in TEX's lists and in the future we might even move more data handling to the LUA end. Also, in future versions we will share similar data (like page number information) more efficiently.

Storing date at the LUA end also has consequences for the typesetting. When specific data is needed a call to LUA is necessary. In the future we might offer both push and pull methods (LUA pushing information to the typesetting code versus LUA triggering typesetting code). For lists we pull, and for registers we currently push. Depending on our experiences we might change these strategies.

A side effect of the rewrite is that we force more consistency. For instance, you see a `conversion` field in the list. This is the old way of defining the way a number gets converted. The modern approach is to use sets. Because we now have a more stringent inheritance model at the user interface level, this might lead to incompatible conversions at lower levels (when unset). Instead of cooking up some nasty compatibility hacks, we accept some incompatibility, if only because users have to adapt their styles to new font technology anyway. And for older documents there is still MKII.

Instead of introducing many extra configuration variables (for each level of sectioning) we introduce sets. These replace some of the existing parameters and are the follow up on some (undocumented) precursor of sets. Examples of sets are:

```
\definestructureseparatorset  [default] [] [.]
\definestructureconversionset [default] [] [numbers]
\definestructureresetset      [default] [] [0]
\definestructureprefixset     [default] [section-2,section-3] []
\definestructureseparatorset  [appendix] [] [.]
\definestructureconversionset [appendix] [Romannumerals,Characters] []
\definestructureresetset      [appendix] [] [0]
```

The third parameter is the default value. The sets that relate to typesetting can have a rendering specification:

```
\definestructureseparatorset
  [demosep]
  [demo->!,demo->?,demo->*,demo->@]
  [demo->/]
```

Here we apply `demo` to each of the separators as well as to the default. The renderer is defined with:

```
\defineprocessor[demo][style=\bfb,color=red]
```

You can imagine that, although this is quite possible in TeX, dealing with sets, splitting them, handling the rendering, etc. is easier in Lua that in TeX. Of course the code still looks somewhat messy, if only because the problem is messy. Part if this mess is related to the fact that we might have to specify all components that make up a number.

section	section number as part of head
list	section number as part of list entry
	section number as part of page number prefix
	(optionally prefixed) page number
counter	section number as part of counter prefix
	(optionally prefixed) counter value(s)
pagenumber	section number as part of page number
	pagenumber components (realpage, page, subpage)

As a result we have upto 3 sets of parameters:

section	`section*`
list	`section* prefix* page*`
counter	`section* number*`
pagenumber	`prefix* page*`

When reimplementing the structure related commands, we also have to take mechanisms into account that relate to them. For instance, index sorter code is also used for sorted lists, so when we adapt one mechanism we also have to adapt the other. The same is true for cross references, that are used all over the place. It helps that for the moment we can omit the more obscure interaction related mechanism, if only because users will seldom use them. Such mechanisms are also related to the backend and we're not yet in the stage where we upgrade the backend code. In case you wonder why references can be such a problematic areas think of the following:

```
\goto{here}[page(10),StartSound{ping},StartVideo{demo}]
\goto{there}[page(10),VideLayer{example},JS(SomeScript{hi world})]
```

```
\goto{anywhere}[url(mypreviouslydefinedurl)]
```

The ConTEXt cross reference mechanism permits mixed usage of simple hyperlinks (jump to some page) and more advanced viewer actions like showing widgets and runnign JavaScript code. And even a simple reference like:

```
\at{here and there}[somefile::sometarget]
```

involves some code because we need to handle the three words as well as the outer reference.[5] The reason why we need to reimplement referencing along with structure lays in the fact that for some structure components (like section headers and float references) we no longer store cross reference information separately but filter it from the data stored in the list (see example before).

The Lua code involved in dealing with the more complex references shown here is much more flexible and robust than the original TEX code. This is a typical example of where the accumulated time spent on the TEX based solution is large compared to the time spent on the Lua variant. It's like driving 200 km by car through hilly terrain and wondering how one did that in earlier times. Just like today scenery is not by definition better than yestedays, MkIV code is not always better than MkII code.

---

[5] Currently ConTEXt does its own splitting of multiword references, and does so by reusing hyperlink resources in the backend format. This might change in the future.

XXIX Tracking

We entered 2009 with a partial reimplementation of the OpenType feature handler. One of the reasons was an upgrade of the FontForge libraries that LuaTEX uses.

The specification of OpenType is kind of vague. Apart from a lack of a proper free specifications there's also the problem that Microsoft and Adobe may have their own interpretation of how and in what order to apply features. In general the Microsoft website has more detailed specifications and is a better reference. There is also some information in the FontForge help files.

Because there is so much possible, fonts might contain bugs and/or be made to work with certain renderers. These may evolve over time which may have the side effect that suddenly fonts behave differently.

After a lot of experiments (mostly by Taco, me and Idris) we're now at yet another implementation. Of course all errors are mine and of course the code can be improved. There are quite some optimization going on here and processing speed is currently acceptable. Not all functions are implemented yet, often because I lack the fonts for testing. Many scripts are not yet supported either, but I will look into them as soon as ConTEXt users ask for it.

The data provided by the FontForge library has organized lookups (which relate to features) in a certain way. A first implementation of this code was organized featurewise: information related to features was collected and processing boiled down to a run over the features. The current implementation honours the order in the main feature table. Since we can reorder this table as we want, we can eventually support several models of processing. We kept the static as well as dynamic feature processing, because it had proved to be rather useful. The formerly three loop variants have been discarded but might reappear at some time.

One reason for this change is that the interactive version of FontForge now provides a more detailed overview of the way lookups are supposed to be handled. When you consult the information of a font and in particular a glyph in a font, you now get quite some information about what features can be applied and in what order this takes place.

In ConTEXt MkIV we deal with this as follows. Keep in mind that we start with characters but stepwise these can become more abstract representation, named glyphs. For instance a letter a can be represented by a shape (glyph) that is similar to an uppercase A.

- We loop over all lookups. Normally there are only a few lookups but fonts that deal with scripts that resemble handwriting, like arabic of Zapfino, might have hundreds of them. Each lookup has a detailed specification of what language and/or scripts it applies to.

- For each lookup we do a run over the list of glyphs. So, if we have 50 lookups, and a paragraph has 500 glyphs, we do some 25000 loops. Keep in mind that for arab we start with a sequence of characters and vowels, and during a run, these might be replaced by for instance ligatures and combined vowels, so the 500 stepwise becomes less.

- We only process the features that are enabled. Normally the lookups are organized in such a way that features take place in a similar way: (de)composition, replacement of initial, medial, final and isolated forms, specific replacements by one or more variant, composition of ligatures, mark positioning, cursive corrections and kerning. The font itself does not contain information about what features are to be enabled by default. Some applications have built in presets, others might extend their repertoire over time.

- A lookup can be a contextual lookup, which means that treatment takes place on a match of a sequence of characters (glyphs), either of not preceded or followed by specific other characters (glyphs). We we loop over all contexts till we have a match. Some fonts have lots of contextual lookups, which in turn might increase the number of loops over the list of characters (glyphs). If we have a match, we process the associated list of sublookups. Technically it is possible to replace (say) five characters by first a ligature (that replaces the first two by one), then a multiple substitution (resulting in an extra three glyphs replacing one) and then discarding the other rest (being two characters). Because by that time characters (say, unicode points) might have been replaced by glyphs (an index in the font) a contextual lookup can involve quite some match points.

In ConTEXt we do this for each font that is used in a list, so in practice we have quite some nested loops. Each font can have its own set of features enables of features might be applied dynamically, independent of font related settings. So, around the mentioned loops there is another one: a loop over the fonts used in a list (paragraph).

We process the whole list and then consult the glyph nodes. An alternative approach is to collect strings of characters using the same font including spaces (because some lookups involve spaces). However, we then need to reconstruct the list which is no fun. Also, we need to carry quite some information, like attributes, so eventually we don't gain much (if we gain something at all).

Another consideration has been to operate on sublists of font usage (using a subhead and subtail) but again this would complicate matters as we then neext to keep track of a changing subhead and subtail. On the other hand, this might save some runtime. The number of changes in the code needed to do this is not that large but it only makes sense when we have many fonts in a list and don't change fonts to frequently.

This whole treatment is rather extensively optimized and so the process is reasonable fast (you really don't want to know how much time was spent on figuring out fast methods, testing and reimplementing this). While I was implementing the LUA code, Taco made sure that access to the information in nodes was as fast as possible and in our usual chat sessions we compared the output with the one produced by the FONTFORGE preview.

It was for this reason that more and more debugging code was added but even that made tracking of what really happened cumbersome. Therefore a more visual method was written, which will be shown laster on.

You can enable tracing using the designated commands:

```
\enabletracker[otf.ligatures,otf.singles]
```

and disable them for instance with:

```
\disabletracker[otf.*]
```

Or you can pass directives to the command line:

```
context --track=otf.ligatures myfile.tex
```

With regards to OPENTYPE handling we have the following tracker keys available:

`otf.actions`	show all replacements and positioning
`otf.alternatives`	show what glyph is replaced by what alternative
`otf.analyzing`	color glyphs according to script specific analysis
`otf.applied`	applied features per font instance
`otf.bugs`	show diagnostic information
`otf.contexts`	show what contextual lookups take place
`otf.cursive`	show cursive anchoring when applied
`otf.details`	show more details about lookup handling
`otf.dynamics`	show dynamic feature definitions
`otf.features`	show what features are a applied
`otf.kerns`	show kerning between glyphs when applied
`otf.ligatures`	show what glyphs are replaced by one other
`otf.loading`	show more information when loading (caching) a font
`otf.lookups`	keep track of what lookups are consulted
`otf.marks`	show mark anchoring when applied
`otf.multiples`	show what glyph is replaced by multiple others
`otf.positions`	show what glyphs are positioned (combines other trackers)
`otf.preparing`	show what information is collected for later usage in lookups

`otf.replacements`	show what glyphs are replaced (combines other trackers)
`otf.sequences`	
`otf.singles`	show what glyph is replaced by one other

Some other trackers might also come in handy:

`fonts.combining`	show what extra characters are added when forcing combined shapes
`fonts.defining`	show what fonts are defined
`fonts.loading`	show more details when a font is loaded (and cached) for the first time

We now show another way to track what happens with your text. Because this is rather verbose, you should only apply it to words. The second argument can be `-1` (right to left), `0` (default) or `1` (left to right). The third argument can be invisible in the code because the font used for verbatim might lack the shapes. A font has a different ordering than Unicode because after all one character can have multiple representations, one shape can be used for multiple characters, or shapes might not have a Unicode point at all. In MkIV we push all shapes that have no direct relationship with Unicode to the private area so that TEX still sees them (hence the large numbers in the following examples).

The next example uses Latin Modern. Here we apply the following features:

```
\definefontfeature
  [latin-default]
  [mode=node,language=dflt,script=latn,
   liga=yes,calt=yes,clig=yes,
   kern=yes]
```

```
\showotfcomposition
  {name:lmroman12regular*latin-default at 24pt}
  {0}
  {flinke fietser}
```

```
font        162: lmroman12-regular.otf @ 24.0pt

features    analyze=yes, calt=yes, clig=yes, devanagari=yes,
            dummies=yes, extensions=yes, extrafeatures=yes,
            kern=yes, language=dflt, liga=yes, mathkerns=yes,
            mode=node, script=latn, spacekern=yes

step 1      flinke fietser   U+66:f U+6C:l U+69:i U+6E:n
            U+6B:k U+65:e [glue] U+66:f U+69:i U+65:e U+74:t
            U+73:s U+65:e U+72:r
```

```
feature 'liga', type 'gsub_ligature', lookup 's_s_8',
  replacing U+00066 (f) upto U+0006C (l) by ligature
  U+0FB02 (f_l) case 2
```

step 2 ﬂinke fietser U+FB02:ﬂ U+69:i U+6E:n U+6B:k U+65:e [glue] U+66:f U+69:i U+65:e U+74:t U+73:s U+65:e U+72:r

```
feature 'liga', type 'gsub_ligature', lookup 's_s_9',
  replacing U+00066 (f) upto U+00069 (i) by ligature
  U+0FB01 (f_i) case 2
```

step 3 ﬂinke ﬁetser U+FB02:ﬂ U+69:i U+6E:n U+6B:k U+65:e [glue] U+FB01:ﬁ U+65:e U+74:t U+73:s U+65:e U+72:r

```
feature 'kern', type 'gpos_pair', lookup 'p_s_1',
  shifting single U+00065 (e) by -0.648pt
```

result ﬂinke ﬁetser U+FB02:ﬂ U+69:i U+6E:n U+6B:k [kern] U+65:e [glue] U+FB01:ﬁ U+65:e U+74:t U+73:s U+65:e U+72:r

The next example uses Arabtype. Here we apply the following features:

```
\definefontfeature
  [arabtype-default]
  [mode=node,language=dflt,script=arab,
   init=yes,medi=yes,fina=yes,isol=yes,
   ccmp=yes,locl=yes,calt=yes,
   liga=yes,clig=yes,dlig=yes,rlig=yes,
   mark=yes,mkmk=yes,kern=yes,curs=yes]
```

```
\showotfcomposition
  {arabtype*arabtype-default at 48pt}
  {-1}
  {}
```

font 163: arabtype.ttf @ 48.0pt

features analyze=yes, calt=yes, ccmp=yes, clig=yes, curs=yes, devanagari=yes, dlig=yes, dummies=yes, extensions=yes, extrafeatures=yes, fina=yes, init=yes, isol=yes, kern=yes, language=dflt, liga=yes, locl=yes, mark=yes, mathkerns=yes, medi=yes, mkmk=yes, mode=node, rlig=yes, script=arab, spacekern=yes

step 1 الضَّرَّ [+TRT] U+627:ا U+644:ل U+636:ض U+651:ّ U+64E:َ U+631:ر U+651:ّ U+64E:َ

```
feature 'ccmp', type 'gsub_ligature', lookup 's_s_1',
  replacing U+00651 upto U+0064E by ligature U+F0171
  case 1

feature 'ccmp', type 'gsub_ligature', lookup 's_s_1',
  replacing U+00651 upto U+0064E by ligature U+F0171
  case 1
```

step 2 الضَّرَّ [+TRT] U+627:ا U+644:ل U+636:ض U+F0171:َّ U+631:ر U+F0171:َّ

```
feature 'fina', type 'gsub_alternate', lookup 's_s_6',
  replacing U+00631 by alternative 'U+0FEAE' to value
  1, taking 1,
```

step 3 الضَّرَّ [+TRT] U+627:ا U+644:ل U+636:ض U+F0171:َّ U+FEAE:ﺮ U+F0171:َّ

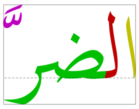
```
feature 'medi', type 'gsub_single', lookup 's_s_7',
  replacing U+00636 by single U+0FEC0
```

step 4 الضَّرَّ [+TRT] U+627:ا U+644:ل U+FEC0:ﻀ U+F0171:َّ U+FEAE:ﺮ U+F0171:َّ

```
feature 'init', type 'gsub_single', lookup 's_s_8',
  replacing U+00644 by single U+0FEDF
```

step 5 الضَّرَّ [+TRT] U+627:ا U+FEDF:ﻟ U+FEC0:ﻀ U+F0171:َّ U+FEAE:ﺮ U+F0171:َّ feature 'liga', type 'gsub_ligature', lookup 's_s_38', replacing U+0FEC0 upto U+0FEAE by ligature U+0FD2C case 2

step 6

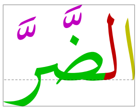


[+TRT] U+627: ا U+FEDF: ﻟ U+FD2C: ﴬ U+F0171: U+F0171:

feature 'mark', type 'gpos_mark2ligature', lookup 'p_s_16', anchor , index 1, bound 1, anchoring mark U+F0171 to baselig U+0FD2C at index 1 => (22.73438pt,0.70313pt)

feature 'mark', type 'gpos_mark2ligature', lookup 'p_s_16', anchor , index 2, bound 2, anchoring mark U+F0171 to baselig U+0FD2C at index 2 => (5.39063pt,-3.75pt)

result



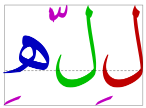
[+TRT] U+627: ا U+FEDF: ﻟ U+FD2C: ﴬ U+F0171: U+F0171:

```
\showotfcomposition
  {arabtype*arabtype-default at 48pt}
  {-1}
  {}
```

font 163: arabtype.ttf @ 48.0pt

features analyze=yes, calt=yes, ccmp=yes, clig=yes, curs=yes, devanagari=yes, dlig=yes, dummies=yes, extensions=yes, extrafeatures=yes, fina=yes, init=yes, isol=yes, kern=yes, language=dflt, liga=yes, locl=yes, mark=yes, mathkerns=yes, medi=yes, mkmk=yes, mode=node, rlig=yes, script=arab, spacekern=yes

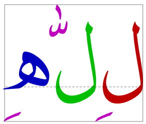
step 1



[+TRT] U+644: ل U+650: U+644: ل U+651: U+670: U+647: ه U+650:

feature 'ccmp', type 'gsub_ligature', lookup 's_s_1', replacing U+00651 upto U+00670 by ligature U+F0174 case 1

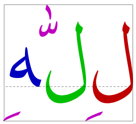
step 2



[+TRT] U+644:ل U+650: U+644:ل U+F0174: U+647:ه U+650:

feature 'fina', type 'gsub_alternate', lookup 's_s_6', replacing U+00647 by alternative 'U+0FEEA' to value 1, taking 1,

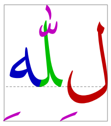
step 3



[+TRT] U+644:ل U+650: U+644:ل U+F0174: U+FEEA:ﻪ U+650:

feature 'medi', type 'gsub_single', lookup 's_s_7', replacing U+00644 by single U+0FEE0

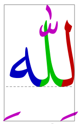
step 4



[+TRT] U+644:ل U+650: U+FEE0:ﻠ U+F0174: U+FEEA:ﻪ U+650:

feature 'init', type 'gsub_single', lookup 's_s_8', replacing U+00644 by single U+0FEDF

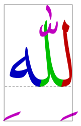
step 5



[+TRT] U+FEDF:ﻟ U+650: U+FEE0:ﻠ U+F0174: U+FEEA:ﻪ U+650:

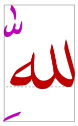
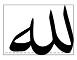
feature 'calt', type 'gsub_contextchain', chain lookup 's_s_29', index -1, replacing single U+0FEDF by U+F0058 (uniFEDF.alt1)

step 6

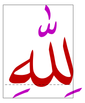
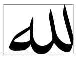


[+TRT] U+F0058:ﻟ U+650: U+FEE0:ﻠ U+F0174: U+FEEA:ﻪ U+650:

feature 'liga', type 'gsub_ligature', lookup 's_s_38', replacing U+F0058 (uniFEDF.alt1) upto U+0FEEA by ligature U+F03E6 (uni064406440647.isol) case 2

step 7  [+TRT] U+F03E6:  U+650: U+F0174: U+650:

```
feature 'mark', type 'gpos_mark2ligature', lookup
  'p_s_16', anchor , index 1, bound 1, anchoring mark
  U+00650 to baselig U+F03E6 (uni064406440647.isol) at
  index 1 => (23.4375pt,8.20313pt)

feature 'mark', type 'gpos_mark2ligature', lookup
  'p_s_16', anchor , index 2, bound 2, anchoring mark
  U+F0174 to baselig U+F03E6 (uni064406440647.isol) at
  index 2 => (13.35938pt,-0.9375pt)

feature 'mark', type 'gpos_mark2ligature', lookup
  'p_s_16', anchor , index 3, bound 3, anchoring mark
  U+00650 to baselig U+F03E6 (uni064406440647.isol) at
  index 3 => (3.98438pt,7.96875pt)
```

result  [+TRT] U+F03E6:  U+650: U+F0174: U+650:

Another arabic example (after all, fonts that support arabic have lots of nice features) is the following. First we define a bunch of feature collections

```
\definefontfeature
  [salt-n]
  [analyze=yes,mode=node,
   language=dflt,script=arab,
   init=yes,medi=yes,fina=yes,isol=yes,
   liga=yes,calt=yes,ccmp=yes,
   kern=yes,curs=yes,mark=yes,mkmk=yes]

\definefontfeature[salt-y][salt-n][salt=yes]
\definefontfeature[salt-1][salt-n][salt=1]
\definefontfeature[salt-2][salt-n][salt=2]
\definefontfeature[salt-3][salt-n][salt=3]
\definefontfeature[salt-r][salt-n][salt=random]
```

Next we show a few traced examples. Watch the reported alternatives.

```
\showotfcomposition{scheherazaderegot*salt-n at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-y at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-1 at 36pt}{-1}{\char"6DD}
```

```
\showotfcomposition{scheherazaderegot*salt-2 at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-3 at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-r at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-r at 36pt}{-1}{\char"6DD}
\showotfcomposition{scheherazaderegot*salt-r at 36pt}{-1}{\char"6DD}
```

```
font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result        [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result        [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result        [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result        [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes
```

```
result      [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result      [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result      [+TRT] U+6DD:

font        1: lt55485.pfb @ 12.0pt

features    autolanguage=position, autoscript=position,
            compose=yes, curs=yes, dummies=yes, kern=yes,
            liga=yes, mark=yes, mkmk=yes, mode=node, script=auto,
            tlig=yes, trep=yes

result      [+TRT] U+6DD:
```

The font that we use here can be downloaded from the website of Sil International.

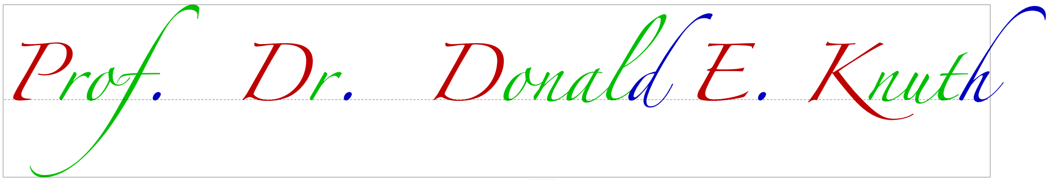
For a Zapfino example we use the following feature set:

```
\definefontfeature
   [zapfino-default]
   [mode=node,language=dflt,script=latn,
    calt=yes,clig=yes,rlig=yes,tlig=yes,
    kern=yes,curs=yes]

\showotfcomposition
  {zapfinoextraltpro*zapfino-default at 48pt}
  {0}
  {Prof. Dr. Donald E. Knuth}
```

```
font        164: zapfinoextraltpro.otf @ 48.0pt
```

features analyze=yes, calt=yes, clig=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, kern=yes, language=dflt, mathkerns=yes, mode=node, rlig=yes, script=latn, spacekern=yes, tlig=yes

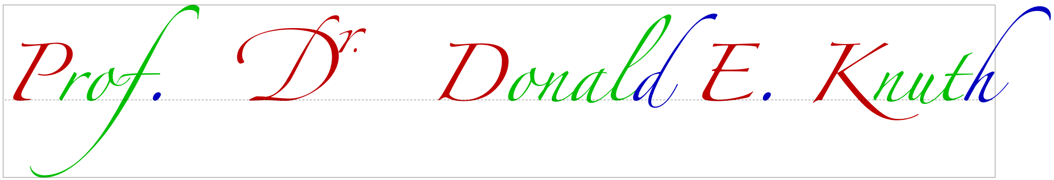
step 1



U+50: P U+72: r U+6F: o U+66: f U+2E: . [glue] U+44: D U+72: r U+2E: . [glue] U+44: D U+6F: o U+6E: n U+61: a U+6C: l U+64: d [glue] U+45: E U+2E: . [glue] U+4B: K U+6E: n U+75: u U+74: t U+68: h

feature 'clig', type 'gsub_ligature', lookup 's_s_22', replacing U+00044 (D) upto U+0002E (period) by ligature U+0E366 (D_r_period) case 2

step 2

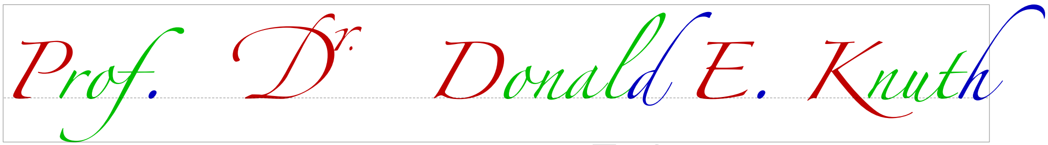


U+50: P U+72: r U+6F: o U+66: f U+2E: . [glue] U+E366: Dr. [glue] U+44: D U+6F: o U+6E: n U+61: a U+6C: l U+64: d [glue] U+45: E U+2E: . [glue] U+4B: K U+6E: n U+75: u U+74: t U+68: h

feature 'calt', type 'gsub_contextchain', chain lookup 's_s_32', index 0, replacing single U+00066 (f) by

```
  U+0E1AC (f.3)

feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_32', index 0, replacing single U+00061 (a) by
  U+0E190 (a.3)
```

step 3



U+50: P U+72: r U+6F: o U+E1AC: f U+2E: . [glue]

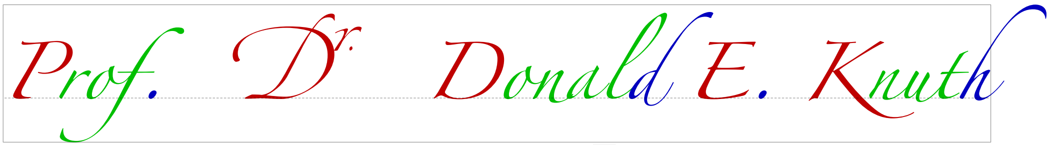
U+E366: Dr [glue] U+44: D U+6F: o U+6E: n

U+E190: a U+6C: l U+64: d [glue] U+45: E U+2E: . [glue]

U+4B: K U+6E: n U+75: u U+74: t U+68: h

```
feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_34', index 0, replacing single U+0006E (n) by
  U+0E1C8 (n.2)
```

step 4



U+50: P U+72: r U+6F: o U+E1AC: f U+2E: . [glue]

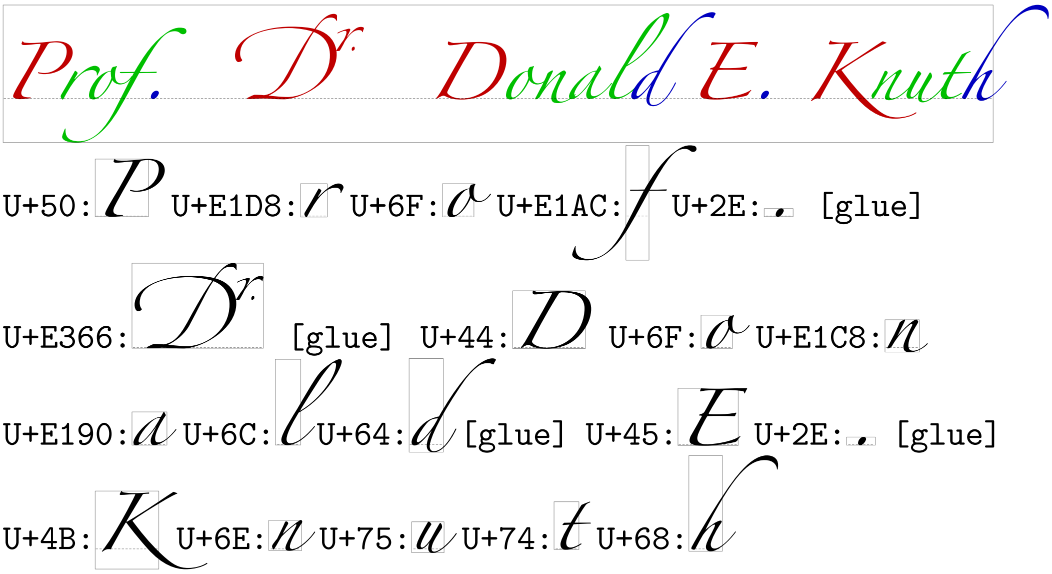
U+E366: Dr [glue] U+44: D U+6F: o U+E1C8: n

U+E190: a U+6C: l U+64: d [glue] U+45: E U+2E: . [glue]

U+4B: K U+6E: n U+75: u U+74: t U+68: h

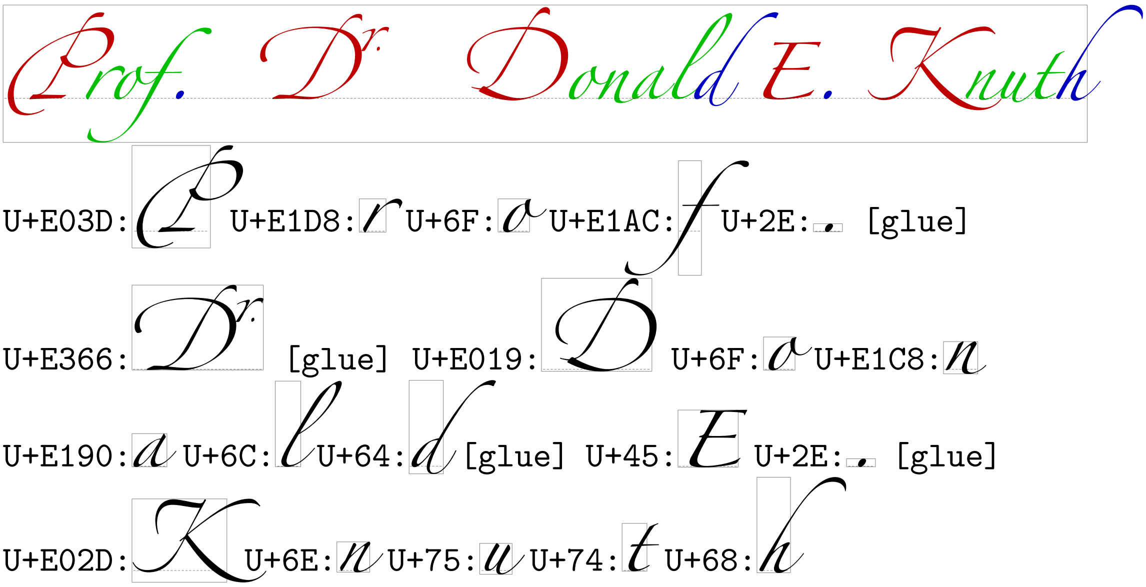
```
feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_36', index 0, replacing single U+00072 (r) by
  U+0E1D8 (r.2)
```

step 5



```
feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_61', index 0, replacing single U+00050 (P) by
  U+0E03D (P.3)

feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_61', index 0, replacing single U+00044 (D) by
  U+0E019 (D.3)

feature 'calt', type 'gsub_contextchain', chain lookup
  's_s_61', index 0, replacing single U+0004B (K) by
  U+0E02D (K.2)
```

result



When dealing with features, we may run into problems due to characters that are in the input stream but have no associated glyph in the font. Although we test for this a user might want to intercept side effect.

```
\checkcharactersinfont
\removemissingcharacters
```

The first command only checks and reports missing characters, while the second one also removes them.

XXX The order of things

Normally the text that makes up a paragraph comes directly from the input stream or macro expansions (think of labels). When TEX has collected enough content to make a paragraph, for instance because a `\par` token signals it TEX will try to create one. The raw material available for making such a paragraph is linked in a list nodes: references to glyphs in a font, kerns (fixed spacing), glue (flexible spacing), penalties (consider them to be directives), whatsits (can be anything, e.g. PDF literals or hyperlinks). The result is a list of horizontal boxes (wrappers with lists that represent 'lines') and this is either wrapped in vertical box of added to the main vertical list that keeps the page stream.

The treatment consists of four activities:

- construction of ligatures (an f plus an i can become fi)
- hyphenation of words that cross a line boundary
- kerning of characters based on information in the font
- breaking the list in lines in the most optimal way

The process of breaking into lines is also influenced by protrusion (like hanging punctuation) and expansion (hz-optimization) but here we will not take these processes into account. There are numerous variables that control the process and the quality.

These activities are rather interwoven and optimized. For instance, in order to hyphenate, ligatures are to be decomposed and/or constructed. Hyphenation happens when needed. Decisions about optimal breakpoints in lines can be influenced by penalties (like: not to many hyphenated words in a row) and permitting extra stretch between words. Because a paragraph can be boxed and unboxed, decomposed and fed into the machinery again, information is kept around. Just imagine the following: you want to measure the width of a word and therefore you box it. In order to get the right dimensions, TEX has to construct the ligatures and add kerns. However, when we unbox that word and feed it into the paragraph builder, potential hyphenation points have to be consulted and at such a point might lay between the characters that resulted in the ligature. You can imagine that adding (and removing) inter-character kerns complicates the process even more.

At the cost of some extra runtime and memory usage, in LUATEX these steps are more isolated. There is a function that builts ligatures, one that kerns characters, and another one that hyphenates all words in a list, not just the ones that are candidate for breaking. The potential breakpoints (called discretionaries) can contain ligature information as well. The linebreak process is also a separate function.

The order in which this happens now is:

- hyphenation of words
- building of ligatures from sequences of glyphs
- kerning of glyphs
- breaking all this into lines

One can discuss endless about the terminology here: are we dealing with characters or with glyphs. When a glyph node is made, it contains a reference to a slot in a font. Because in traditional TEX the number of slots is limited to 256 the relationship between characters in the input and the shape in the font, called glyph, is kind of indirect (the input encoding versus font encoding issue) while in LUATEX we can keep the font in UNICODE encoding if we want. In traditional TEX, hyphenation is based on the font encoding and therefore glyphs, and although in LUATEX this is still the case, there we can more safely talk of characters till we start mapping then to shapes that have no UNICODE point. This is of course macro package dependent but in CONTEXT MKIV we normalize all input to UNICODE exclusively.

The last step is now really isolated and for that reason we can best talk in terms of preparation of the to-be paragraph when we refer to the first three activities. In LUATEX these three are available as functions that operate on a node list. They each have their own callback so we can disable them by replacing the default functions by dummies. Then we can hook in a new function in the two places that matter: `hpack_filter` and `pre_linebreak_filter` and move the preparation to there.

A simple overload is shown below. Because the first node is always a whatsit that holds directional information (and at some point in the future maybe even more paragraph related state info), we can safely assume that `head` does not change. Of course this situation might change when you start adding your own functionality.

```
local function my_preparation(head)
    local tail = node.slide(head) -- also add prev pointers
    tail = lang.hyphenate(head,tail)
    tail = node.ligaturing(head,tail)
    tail = node.kerning(head,tail)
    return head
end

callback.register("pre_linebreak_filter", my_preparation)
callback.register("hpack_filter",         my_preparation)

local dummy = function(head,tail) return tail end

callback.register("hyphenate",  dummy)
callback.register("ligaturing", dummy)
callback.register("kerning",    dummy)
```

It might be clear that the order of actions matter. It might also be clear that you are responsible for that order yourself. There is no pre–cooked mechanism for guarding your actions and there are several reasons for this:

- Each macro package does things its own way so any hard-coded mechanism would be replaced and overloaded anyway. Compare this to the usage of catcodes, font systems, auxiliary files, user interfaces, handling of inserts etc. The combination of callbacks, the three mentioned functions and the availability of LUA makes it possible to implement any system you like.

- Macro packages might want to provide hooks for specialized node list processing, and since there are many places where code can be hooked in, some kind of oversight is needed (real people who keep track of interference of user supplied features, no program can do that).

- User functions can mess up the node list and successive actions then might make the wrong assumptions. In order to guard this, macro packages might add tracing options and again there are too many ways to communicate with users. Debugging and tracing has to be embedded in the bigger system in a natural way.

In CONTEXT MKIV there are already a few places where users can hook code into the task list, but so far we haven't really encouraged that. The interfaces are simply not stable enough yet. On the other hand, there are already quite some node list manipulators at work. The most prominent one is the OPENTYPE feature handler. That one replaces the ligature and kerning functions (at least for some fonts). It also means that we need to keep an eye on possible interferences between CONTEXT MKIV mechanisms and those provided by LUATEX.

For fonts, that is actually quite simple: the LUATEX functions use ligature and kerning information stored in the TFM table, and for OPENTYPE fonts we simply don't provide that information when we define a font, so in that case LUATEX will not ligature and kern. Users can influence this process to some extend by setting the `mode` for a specific instance of a font to `base` or `node`. Because TYPE1 fonts have no features like OPENTYPE such fonts are (at least currently) always are processed in base mode.

Deep down in CONTEXT we call a sequence of actions a 'task'. One such task is 'processors' and the actions discussed so far are in this category. Within this category we have subcategories:

subcategory	**intended usage**
before	experimental (or module) plugins

normalizers	cleanup and preparation handlers
characters	operations on individual characters
words	operations on words
fonts	font related manipulations
lists	manipulations on the list as a whole
after	experimental (or module) plugins

Here 'plugins' are experimental handlers or specialized ones provided in modules that are not part of the kernel. The categories are not that distinctive and only provide a convenient way to group actions.

Examples of normalizers are: checking for missing characters and replacing character references by fallbacks. Character processors are for instance directional analysers (for right to left typesetting), case swapping, and specialized character triggered hyphenation (like compound words). Word processors deal with hyphenation (here we use the default function provided by LuaTeX) and spell checking. The font processors deal with OpenType as well as the ligature building and kerning of other font types. Finally, the list processors are responsible for tasks like special spacing (french punctuation) and kerning (additional inter–character kerning). Of course, this all is rather ConTeXt specific and we expect to add quite some more less trivial handlers the upcoming years.

Many of these handlers are triggered by attributes. Nodes can have many attributes and each can have many values. Traditionally TeX had only a few attributes: language and font, where the first is not even a real attribute and the second is only bound to glyph nodes. In LuaTeX language is also a glyph property. The nice thing about attributes is that they can be set at the TeX end and obey grouping. This makes them for instance perfect for implementing color mechanims. Because attributes are part of the nodes, and not nodes themselves, they don't influence or interfere processing unless one explicitly tests for them and acts accordingly.

In addition to the mentioned task 'processors' we also have a task 'shipouts' and there will be more tasks in future versions of ConTeXt. Again we have subcategories, currently:

subcategory	**intended usage**
before	experimental (or module) plugins
normalizers	cleanup and preparation handlers
finishers	manipulations on the list as a whole
after	experimental (or module) plugins

An example of a normalizer is cleanup of the 'to be shipped out' list. Finishers deal with color, transparency, overprint, negated content (sometimes used in page imposition), special effects effect (like outline fonts) and viewer layers (something PDF). Quite possible hyperlink support will also be handled there but not before the backend code is rewritten.

The previous description is far from complete. For instance, not all handlers use the same interface: some work `head` onwards, some need a `tail` pointer too. Some report back success or failure. So the task handler needs to normalize their usage. Also, some effort goes into optimizing the task in such a way that processing the document is still reasonable fast. Keep in mind that each construction of a box invokes a callback, and there are many boxes used for constructing a page. Even a nilled callback is one, so for a simple one word paragraph four callbacks are triggered: the (nilled) hyphenate, ligature and kern callbacks as well as the one called `pre_linebreak_filter`. The task handler that we plug in the filter callbacks calls many functions and each of them does one of more passes over the node list, and in turn might do many call to functions. You can imagine that we're quite happy that TEX as well as LUA is so efficient.

As I already mentioned, implementing a task handler as well as deciding what actions within tasks to perform in what order is specific for the way a macro package is set up. The following code can serve as a starting point

```
filters = { } -- global namespace

local list = { }

function filters.add(fnc,n)
    if not n or n > #list + 1 then
        table.insert(list,#list+1)
    elseif n < 0 then
        table.insert(list,1)
    else
        table.insert(list,n)
    end
end

function filters.remove(fnc,n)
    if n and n > 0 and n <= #list then
        table.remove(list,n)
    end
end

local function run_filters(head,...)
    local tail = node.slide(head)
    for _, fnc in ipairs(list) do
        head, tail = fnc(head,tail,...)
    end
    return head
end
```

```
local function hyphenation(head,tail)
    return head, tail, lang.hyphenate(head,tail) -- returns done
end
local function ligaturing(head,tail)
    return node.ligaturing(head,tail) -- returns head,tail,done
end
local function kerning(head,tail)
    return node.kerning(head,tail) -- returns head,tail,done
end

filters.add(hyphenation)
filters.add(ligaturing)
filters.add(kerning)

callback.register("pre_linebreak_filter", run_filters)
callback.register("hpack_filter",         run_filters)
```

Although one can inject extra filters by using the `add` function it may be clear that this can be dangerous due to interference. Therefore a slightly more secure variant is the following, where `main` is reserved for macro package actions and the others can be used by add–ons.

```
filters = { } -- global namespace

local list = {
    pre = { }, main = { }, post = { },
}

local order = {
    "pre", "main", "post"
}

local function somewhere(where)
    if not where then
        texio.write_nl("error: invalid filter category")
    elseif not list[where] then
        texio.write_nl(string.format("error: invalid filter category
'%s'",where))
    else
        return list[where]
    end
    return false
end
```

```
function filters.add(where,fnc,n)
    local list = somewhere(where)
    if not list then
        -- error
    elseif not n or n > #list + 1 then
        table.insert(list,#list+1)
    elseif n < 0 then
        table.insert(list,1)
    else
        table.insert(list,n)
    end
end

function filters.remove(where,fnc,n)
    local list = somewhere(where)
    if list and n and n > 0 and n <= #list then
        table.remove(list,n)
    end
end

local function run_filters(head,...)
    local tail = node.slide(head)
    for _, lst in pairs(order) do
        for _, fnc in ipairs(list[lst]) do
            head, tail = fnc(head,tail,...)
        end
    end
    return head
end

filters.add("main",hyphenation)
filters.add("main",ligaturing)
filters.add("main",kerning)

callback.register("pre_linebreak_filter", run_filters)
callback.register("hpack_filter",         run_filters)
```

Of course, ConTeXt users who try to use this code will be punished by loosing much of the functionality already present, simply because we use yet another variant of the above code.

XXXI Unicode math

I assume that the reader is somewhat familiar with math in TEX. Although in ConTEXt we try to support the concepts and symbols used in the TEX community we have our own way of implementing math. The fact that ConTEXt is not used extensively for conventional math journals permits us to rigourously re-implement mechanisms. Of course the user interfaces mostly remain the same.

introduction

The LuaTEX project entered a new stage when end of 2008 and beginning of 2009 math got opened up. Although TEX can handle math pretty good we had a few wishes that we hoped to fulfill in the process. That TEX's math machinery is a rather independent subsystem is reflected in the fact that after parsing there is an intermediate list of so called noads (math elements), which then gets converted into a node list (glyphs, kerns, penalties, glue and more). This conversion can be intercepted by a callback and a macro package can do whatever it likes with the list of noads as long as it returns a proper list.

Of course ConTEXt does support math and that is visible in its code base:

- Due to the fact that we need to be able to switch to alternative styles the font system is quite complex and in ConTEXt MkII math font definitions (and changes) are good for 50% of the time involved. In MkIV we can use a more efficient model.
- Because some usage of ConTEXt demands the mix of several completely different encoded math fonts there is a dedicated math encoding subsystem in MkII. In MkIV we will use Unicode exclusively.
- Some constructs (and symbols) are implemented in a way that we find suboptimal. In the perspective of Unicode in MkIV we aim at all symbols being real characters. This is possible because all important constructs (like roots, accents and delimiters) are supported by the engine.
- In order to fit vertical spacing around math (think for instance of typesetting on a grid) in MkII we have ended up with rather messy and suboptimal code.[6] The expectation is that we can improve that.

In the following sections I will discuss a few of the implementation details of the font related issues in MkIV. Of course a few years from now the actual solutions we implemented might look different but the principles remain the same. Also, as with other

---

[6] This is because spacing before and after formulas has to cooperate with spacing of structural components that surround it.

components of LuaTEX Taco and I worked in parallel on the code and its usage, which made both our tasks easier.

transition

In TEX, math typesetting uses a special concept called families. Each math component (number, letter, symbol, etc) is member of a family. Because we have three sizes (text, script and scriptscript) this results in a family–size matrix of defined fonts. Because the number of glyphs in a font was limited to 256, in practice it meant that we had quite some font definitions. The minimum number of families was 4 (roman, italic, symbol, and extension) but in practice several more could be active (sans, bold, mono-spaced, more symbols, etc.) for specific alphabets or extra symbols (for instance AMS set A and B). The total number of families in traditional TEX is limited to 16, and one easily hits this maximum. In that case, some 16 times 3 fonts are defined for one size of which in practice only a few are really used in the typesetting.

A potential source of confusion is bold math. Bold in math can either mean having some bold letters, or having the whole formula in bold. In practice this means that for a complete bold formula one has to define the whole lot using bold fonts. A complication is that the math symbols (etc) are kind of bound to families and so we end up with either redefining symbols, or reusing the families (which is easier and faster). In any case there is a performance issue involved due to the rather massive switch from normal to bold.

In Unicode all alphabets that make sense as well as all math symbols are part of the definition although unfortunately some alphabets have their letters spread over the Unicode vector and not in a range (like blackboard). This forces all applications that want to support math to implement similar hacks to deal with it.

In MkIV we will assume that we have Unicode aware math fonts, like OpenType. The font that sets the standard is Microsoft Cambria. The upcoming (I'm writing this in January 2009) TEXGyre fonts will be compliant to this standard but they're not yet there and so we have a problem. The way out is to define virtual fonts and now that LuaTEX math is extended to cover all of Unicode as well as provides access to the (intermediate) math lists this has become feasible. This also permits us to test LuaTEX with both Cambria and Latin Modern Virtual Math.

The advantage is that we can stick to just one family for all shapes which simplifies the underlying TEX code enormously. First of all we need to define way less fonts (which is partially compensated by loading them as part of the virtual font) and all math aspects can now be dealt with using the character data tables.

One tricky aspect of the new approach is that the Latin Modern fonts have design sizes, so we have to define several virtual fonts. On the other hand, fonts like Cambria have alternative script and scriptscript shapes which is controlled by the `ssty` feature, a gsub alternate that provides some alternative sizes for a couple of hundred characters that matter.

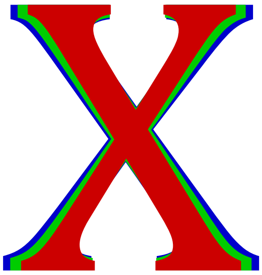
text	`lmmi12 at 12pt`	`cambria at 12pt with ssty=no`
script	`lmmi8 at 8pt`	`cambria at 8pt with ssty=1`
scriptscript	`lmmi6 at 6pt`	`cambria at 6pt with ssty=2`

So Cambria not so much has design sizes but shapes optimized relative to the text variant: in the following example we see text in red, script in green and scriptscript in blue.

```
\definefontfeature[math][analyze=false,script=math,language=dflt]

\definefontfeature[text]        [math][ssty=no]
\definefontfeature[script]      [math][ssty=1]
\definefontfeature[scriptscript][math][ssty=2]
```

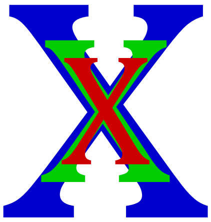
Let us first look at Cambria:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 150pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at 150pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at 150pt]\mkred   X}
\stopoverlay
```



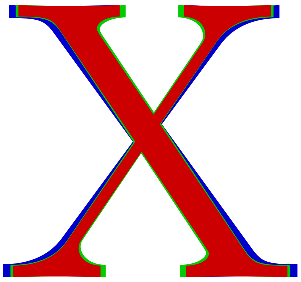
When we compare them scaled down as happens in real script and scriptscript we get:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 120pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at  80pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at  60pt]\mkred   X}
\stopoverlay
```



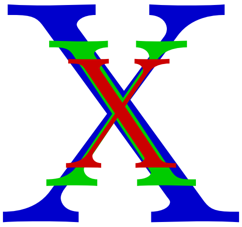
Next we see (scaled) Latin Modern:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 150pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at 150pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at 150pt]\mkred   X}
\stopoverlay
```



In practice we will see:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 120pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at  80pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at  60pt]\mkred   X}
\stopoverlay
```



Both methods probably work out well although you need to keep in mind that the OpenType `ssty` feature is not so much a design size related feature.

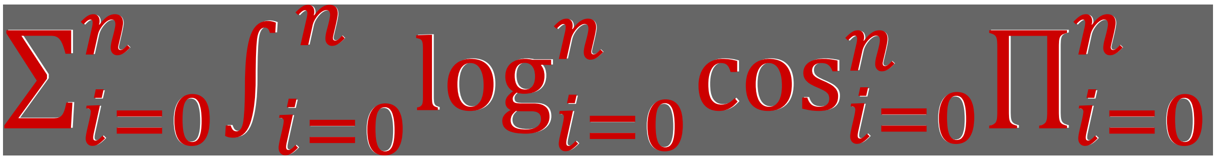
An OpenType font can have a specification for the script and scriptscript size. By default we listen to this specification instead of the one imposed by the bodyfont environment. When you turn on tracing

```
\enabletrackers[otf.math]
```

you will get messages like:

```
asked scriptscript size: 458752, used: 471859.2 (102.86 %)
asked script size: 589824, used: 574095.36 (97.33 %)
```

The differences between the defaults and the font recommendations are not that large so by default we listen to the font specification.



In this overlay the white text is scaled according to the specification in the font, while the red text is scaled according to the bodyfont environment (12/7/5 points).

going virtual

The number of math fonts (used) in the TEX community is relatively small and of those only Latin Modern (which builds upon Computer Modern) has design sizes. This means that the amount of Unicode compliant virtual math fonts that we have to make is not that large. We could have used an already present virtual composition mechanism but instead we made a handy helper function that does a more efficient job. This means that a definition looks (a bit simplified) as follows:

```
mathematics.make_font ( "lmroman10-math", {
  { name="lmroman10-regular", features="virtualmath", main=true },
  { name="lmmi10", vector="tex-mi", skewchar=0x7F },
  { name="lmsy10", vector="tex-sy", skewchar=0x30, parameters=true
} ,
  { name="lmex10", vector="tex-ex", extension=true } ,
  { name="msam10", vector="tex-ma" },
  { name="msbm10", vector="tex-mb" },
  { name="lmroman10-bold", "tex-bf" } ,
  { name="lmmib10", vector="tex-bi", skewchar=0x7F } ,
  { name="lmsans10-regular", vector="tex-ss", optional=true },
  { name="lmmono10-regular", vector="tex-tt", optional=true },
} )
```

For the TEXGyre Pagella it looks this way:

```
mathematics.make_font ( "px-math", {
  { name="texgyrepagella-regular", features="virtualmath", main=true
},
  { name="pxr", vector="tex-mr" } ,
  { name="pxmi", vector="tex-mi", skewchar=0x7F },
  { name="pxsy", vector="tex-sy", skewchar=0x30, parameters=true }
,
  { name="pxex", vector="tex-ex", extension=true } ,
  { name="pxsya", vector="tex-ma" },
  { name="pxsyb", vector="tex-mb" },
} )
```

As you can see, it is possible to add alphabets, given that there is a suitable vector that maps glyph indices onto UNICODES. It is good to know that this function only defines the way such a font is constructed. The actual construction is delayed till the font is needed.

Such a virtual font is used in typescripts (the building blocks of typeface definitions in CONTEXT) as follows:

```
\starttypescript [math] [palatino] [name]
  \definefontsynonym [MathRoman] [pxmath@px-math]
  \loadmapfile[original-youngryu-px.map]
\stoptypescript
```

If you're familiar with the way fonts are defined in CONTEXT, you will notice that we no longer need to define MathItalic, MathSymbol and additional symbol fonts. Of course users don't have to deal with these issues themselves. The `@` triggers the virtual font builder.

You can imagine that in MKII switching to another font style or size involves initializing (or at least checking) involves some 30 to 40 font definitions when it comes to math (the number of used families times 3, the number o fmath sizes.). And even if we take into account that fonts are loaded only once, this checking and enabling takes time. Keep in mind that in CONTEXT we can have several math font sets active in one document which comes at a price.

In MKIV we use one family (at three sizes). Of course we need to load the font (and more than one in the case of virtual variants) but when switching bodyfont sizes we only need to enable one (already defined) math font. And that really saves time. This is one of the areas where we gain back time that we loose elsewhere by extending core functionality using LUA (like OPENTYPE support).

dimensions

By setting font related dimensions you can control the way TEX positions math elements relative to each other. Math fonts have a few more dimensions than regular text fonts. But OPENTYPE math fonts like Cambria have quite some more. There is a nice booklet published by Microsoft, 'Mathematical Typesetting', where dealing with math is discussed in the perspective of their word processor and TEX. In the booklet some of the parameters are discussed and since many of them are rather special it makes no sense (yet) to elaborate on them here.[7] Figuring out their meaning was quite a challenge.

I am the first to admit that the current code in MKIV that deals with math parameters is somewhat messy. There are several reasons for this:

---

[7] Googling on 'Ulrich Vieth', 'TeX' and 'conferences' might give you some hits on articles on these matters.

- We can pass parameters as `MathConstants` table in the TFM table that we pass to the core engine.
- We can use some named parameters, like `x_height` and pass those in the `parameters` table.
- We can use the traditional font dimension numbers in the `parameters` table, but since they overlap for symbol and extensible fonts, that is asking for troubles.

Because in MkIV we create virtual fonts at run-time and use just one family, we fill the `MathConstants` table for traditional fonts as well. Future versions may use the upcoming mechanisms of font parameter sets at the macro level. These can be defined for each of the sizes (display, text, script and scriptscript, and the last three in cramped form as well) but since a font only carries one set, we currently use a compromise.

tracing

One of the nice aspects of the opened up math machinery is that it permits us to get a more detailed look at what happens. It also fits nicely in the way we always want to visualize things in ConTeXt using color, although most users are probably unaware of many such features because they don't need them as I do.

```
\enabletrackers[math.analyzing]
\ruledhbox{$a = \sqrt{b^2 + \sin{c} - {1 \over \gamma}}$}
\disabletrackers[math.analyzing]
```

$$a = \sqrt{b^2 + \sin c - \frac{1}{\gamma}}$$

This tracker option colors characters depending on their nature and the fact that they are remapped. The tracker also was handy during development of LuaTeX especially for checking if attributes migrated right in constructed symbols.

For over a year I had been using a partial Unicode math implementation in some projects but for serious math the vectors needed to be completed. In order to help the 'math department' of the ConTeXt development team (Aditya Mahajan, Mojca Miklavec, Taco Hoekwater and myself) we have some extra tracing options, like

```
\showmathfontcharacters[list=0x0007B]
```

```
U+0007B: { { left curly bracket
  width: 393216, height: 589824, depth: 196608, italic: 0
  mathclass: open, mathname: lbrace
  next: U+F0647 { => U+F065D { => U+F0673 { => U+F0689 { =>
    U+F069F { => U+F06B5 { => U+F06D1 {
```

variants: U+023A7 ⎧ => U+F06D3 ⎪ => U+023A8 ⎨ => U+F06D3 ⎪ =>
U+023A9 ⎩

The simple variant with no arguments would have extended this document with many pages of such descriptions.

Another handy command (defined in module `fnt-25`) is the following:

```
\ShowCompleteFont{name:cambria}{9pt}{1}
\ShowCompleteFont{dummy@lmroman10-math}{10pt}{1}
```

This will for instance for Cambria generate between 50 and 100 pages of character tables.

If you look at the following samples you can imagine how coloring the characters and replacements helped figuring out the alphabets We use the following input (stored in a buffer):

```
$abc \bf abc \bi abc$
$\mathscript abcdefghijklmnopqrstuvwxyz %
  1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathfraktur abcdefghijklmnopqrstuvwxyz %
  1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathblackboard abcdefghijklmnopqrstuvwxyz %
  1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathscript abc IRZ \mathfraktur abc IRZ %
  \mathblackboard abc IRZ \ss abc IRZ 123$
```

For testing Cambria we say:

```
\usetypescript[cambria]
\switchtobodyfont[cambria,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

And we get:

*abc***abc*****abc***
𝒶𝒷𝒸𝒹ℯ𝒻ℊ𝒽𝒾𝒿𝓀𝓁𝓂𝓃ℴ𝓅𝓆𝓇𝓈𝓉𝓊𝓋𝓌𝓍𝓎𝓏1234567890𝒜ℬ𝒞𝒟ℰℱ𝒢ℋℐ𝒥𝒦ℒℳ𝒩𝒪𝒫𝒬ℛ𝒮𝒯𝒰𝒱𝒲𝒳𝒴𝒵
𝔞𝔟𝔠𝔡𝔢𝔣𝔤𝔥𝔦𝔧𝔨𝔩𝔪𝔫𝔬𝔭𝔮𝔯𝔰𝔱𝔲𝔳𝔴𝔵𝔶𝔷1234567890𝔄𝔅ℭ𝔇𝔈𝔉𝔊ℌℑ𝔍𝔎𝔏𝔐𝔑𝔒𝔓𝔔ℜ𝔖𝔗𝔘𝔙𝔚𝔛𝔜ℨ
abcdefghijklmnopqrstuvwxyz1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
𝒶𝒷𝒸ℐℛ𝒵𝔞𝔟𝔠ℑℜℨabcIRZabcIRZ123

For the virtualized Latin Modern we say:

```
\usetypescript[modern]
```

```
\switchtobodyfont[modern,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

This gives:

$abc\mathbf{abc}\boldsymbol{abc}$
abcdefghijklmnopqrstuvwxyz1234567890$\mathscr{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$
$\mathfrak{abcdefghijklmnopqrstuvwxyz}1234567890\mathfrak{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$
$\mathbb{abcdefghijklmnopqrstuvwxyz1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ}$
abc$\mathscr{IRZ}\mathfrak{abcIRZ}\mathbb{abcIRZ}\mathsf{abcIRZ123}$

These two samples demonstrate that Cambria has a rather complete repertoire of shapes which is no surprise because it is a recent font that also serves as a showcase for UNICODE and OPENTYPE driven math.

Commands like `\mathscript` sets an attribute. When we post-process the noad list and encounter this attribute, we remap the characters to the desired variant. Of course this happens selectively. So, a capital A (`0x0041`) becomes a capital script A (`0x1D49C`). Of course this solution is rather ConTEXt specific and there are other ways to achieve the same goal (like using more families and switching family).

special cases

Because we now are operating in the UNICODE domain, we run into problems if we keep defining some of the math symbols in the traditional TEX way. Even with the AMS fonts available we still end up with some characters that are represented by combining others. Take for instance $\neq$ which is composed of two characters. Because in MkIV we want to have all characters in their pure form we use a virtual replacement for them. In MkIV speak it looks like this:

```
local function negate(main,unicode,basecode)
    local characters = main.characters
    local basechar = characters[basecode]
    local ht, wd = basechar.height, basechar.width
    characters[unicode] = {
        width    = wd,
        height   = ht,
        depth    = basechar.depth,
        italic   = basechar.italic,
        kerns    = basechar.kerns,
        commands = {
            { "slot", 1, basecode },
            { "push" },
            { "down",    ht/5},
            { "right", - wd/2},
            { "slot", 1, 0x2215 },
            { "pop" },
        }
    }
end
```

In case you're curious, there are indeed kerns, in this case the kerns with the Greek Delta.

Another thing we need to handle is positioning of accents on top of slanted (italic) shapes. For this TeX uses a special character in its fonts (set with `\skewchar`). Any character can have in its kerning table a kern towards this special character. From this kern we can calculate the `top_accent` variable that we can pass for each character. This variable lives at the same level as `width`, `height`, `depth` and `italic` and is calculated as: $w/2+k$, so it defines the horizontal anchor. A nice side effect is that (in the ConTeXt font management subsystem) this saves us passing information associated with specific fonts such as the skew character.

A couple of concepts are unique to TeX, like having `\hat` and `\widehat` where the wide one has sizes. In OpenType and Unicode we don't have this distinction so we need special trickery to simulate this. We do so by adding extra code points in a private Unicode space which in return results in them being defined automatically and the relevant first size variant being used for `\hat`. For some users this might still be too wide but at least it's better than a wrongly positioned ascii variant. In the future we might use this private space for similar cases.

Arrows, horizontal extenders and radicals also fall in the category 'troublesome' if only because they use special dimensions to get the desired effect. Fortunately OpenType math is modeled after TeX, so in LuaTeX we introduce a couple of new constructs to deal with this. One such simplification at the macro level is in the definition of `\root`. Here we use the new `\Uroot` primitive. The placement related parameters are those used by traditional TeX, but when they are available the OpenType parameters are applied. The simplified plain definitions are now:

```
\def\rootradical{\Uroot 0 "221A }

\def\root#1\of{\rootradical{#1}}

\def\sqrt{\rootradical{}}
```

The successive sizes of the root will be taken from the font in the same way as traditional TeX does it. In that sense LuaTeX is no doing anything differently, it only has more parameters to control the process. The definition of `\sqrt` in ConTeXt permits an optional first argument that sets the degree.

U+0221A: √ √ square root
width: 655098, height: 31457, depth: 754975, italic: 0
mathclass: root, mathname: rootradical
mathclass: radical, mathname: surdradical
mathclass: ordinary, mathname: surd
next: U+F078E √ => U+F078F √ => U+F0790 √ => U+F0791 √
variants: U+F078D ┌ => U+F078C │ => U+023B7 ⎷

Note that we've collected all characters in family 0 (simply because that is what TEX defaults characters to) and that we use the formal UNICODE slots. When we use the Latin Modern fonts we just remap traditional slots to the right ones.

Another neat trick is used when users choose among the bigger variants of some characters. The traditional approach is to create a box of a certain size and create a fake delimited variant which is then used.

```
\definemathcommand [big]  {\choosemathbig\plusone  }
\definemathcommand [Big]  {\choosemathbig\plustwo  }
\definemathcommand [bigg] {\choosemathbig\plusthree}
\definemathcommand [Bigg] {\choosemathbig\plusfour }
```

Of course this can become a primitive operation and we might decide to add such a primitive later on so we won't bother you with more details.

Attributes are also used to make live easier for authors who have to enter lots of pairs. Compare:

```
\setupmathematics[autopunctuation=no]

$ (a,b) = (1.20,3.40) $
```

$$(a, b) = (1.20, 3.40)$$

with:

```
\setupmathematics[autopunctuation=yes]

$ (a,b) = (1.20,3.40) $
```

$$(a,b) = (1.20,3.40)$$

So we don't need to use this any more:

```
$ (a{,}b) = (1{.}20{,}3{.}40) $
```

Features like this are implemented on top of an experimental math manipulation framework that is part of MkIV. When the math font system is stable we will rework the rest of math support and implement additional manipulating frameworks.

control

As with all other character related issues, in MkIV everything is driven by a character table (consider it a database). Quite some effort went into getting that one right and although by now math is represented well, more data will be added in due time.

In MkIV we no longer have huge lists of TEX definitions for math related symbols. Everything is initialized using the mentioned table: normal symbols, delimiters, radicals, whether or not with name. Take for instance the square root:

U+0221A: √ √ square root
  width: 655098, height: 31457, depth: 754975, italic: 0
  mathclass: root, mathname: rootradical
  mathclass: radical, mathname: surdradical
  mathclass: ordinary, mathname: surd
  next: U+F078E √ => U+F078F √ => U+F0790 √ => U+F0791 √
  variants: U+F078D => U+F078C => U+023B7 ⎷

Its entry is:

```
[0x221A] = {
    adobename = "radical",
    category = "sm",
    cjkwd = "a",
    description = "SQUARE ROOT",
    direction = "on",
    linebreak = "ai",
    mathclass = "radical",
    mathname = "surd",
    unicodeslot = 0x221A,
}
```

The fraction symbol also comes in sizes. This symbol is not to be confused with the negation symbol `0x2215`, which in TEX is known as `\not`).

U+02044: ⁄ ⁄ fraction slash

width: 393216, height: 589824, depth: 196608, italic: 0
mathclass: ordinary, mathname: slash
mathclass: close, mathname: solidus

next: U+F0727 / => U+F072E / => U+F0735 / => U+F073C / =>

U+F0743 / => U+F074A / => U+F0751 /

```
[0x2044] = {
    adobename = "fraction",
    category = "sm",
    contextname = "textfraction",
    description = "FRACTION SLASH",
    direction = "cs",
    linebreak = "is",
    mathspec = {
        { class = "binary", name = "slash" },
        { class = "close", name = "solidus" },
    },
    unicodeslot = 0x2044,
}
```

However, since most users don't have this symbol visualized in their word processor, they expect the same behaviour from the regular slash. This is why we find a reference to the real symbol in its definition.

U+0002F: / / solidus
width: 393216, height: 589824, depth: 196608, italic: 0
mathclass: middle, mathname: no name
mathclass: ordinary, mathname: no name

next: U+F0725 / => U+F072C / => U+F0733 / => U+F073A / =>

U+F0741 / => U+F0748 / => U+F074F /

The definition is:

```
[0x002F] = {
    adobename = "slash",
    category = "po",
    cjkwd = "na",
    contextname = "textslash",
    description = "SOLIDUS",
    direction = "cs",
    linebreak = "sy",
    mathsymbol = 0x2044,
    unicodeslot = 0x002F,
}
```

One problem left is that currently we have only one class per character (apart from the delimiter and radical usage which have their own definitions). Future releases of ConTEXt will provide support for math dictionaries (as in OpenMath and MathML 3). At that point we will also have a `mathdict` entry.

There is another issue with character mappings, one that will seldom reveal itself to the user, but might confuse macro writers when they see an error message.

In traditional TEX, and therefore also in the Latin Modern fonts, a chain from small to large character goes in two steps: the normal size is taken from one family and the larger variants from another. The larger variant then has a pointer to an even larger one and so on, until there is no larger variant or an extensible recipe is found. The default family is number 0. It is for this reason that some of the definition primitives expect a small and large family part.

However, in order to support OpenType in LuaTEX the alternative method no longer assumes this split. After all, we no longer have a situation where the 256 limit forces us to take the smaller variant from one font and the larger sequence from another (so we need two family–slot pairs where each family eventually resolves to a font).

It is for that reason that the new `\U...` primitives expect only one family specification: the small symbol, which then has a pointer to a larger variant when applicable. However deep down in the engine, there is still support for the multiple family solution (after all, we don't want to drop compatibility). As a result, in error messages you can still find references (defaulting to 0) to large specifications, even if you don't use them. In that case you can simply ignore the large symbol (0,0), since it is not used when the small symbol provides a link.

extensibles

In TEX fences can be told to become larger automatically. In traditional TEX a character can have a linked list of next larger shapes ending in a description of how to compose even larger variants.

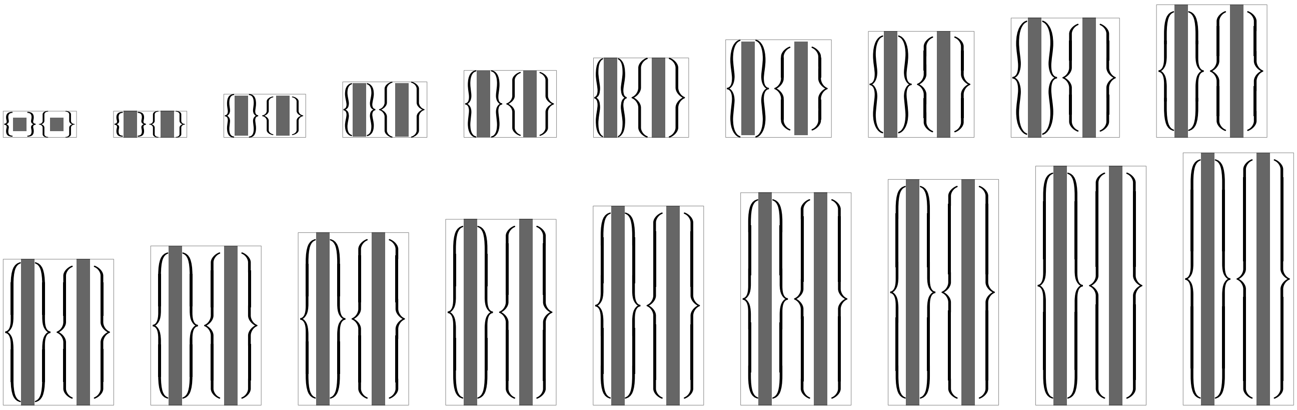
A parenthesis in Cambria has the following list:

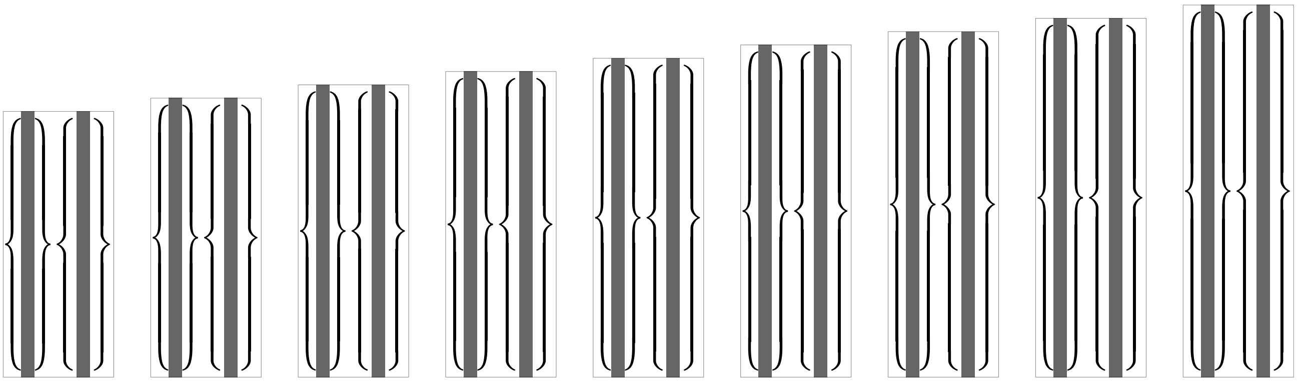
```
U+00028: ( ( left parenthesis
   width: 272000, height: 462400, depth: 144640, italic: 0
   mathclass: open, mathname: lparent
   next: U+F0571 ( => U+F0988 ( => U+F0572 ( => U+F098E ( => U+F0573 ( => U+F0994 ( =>
      U+F0574 (
   variants: U+0239B ⎛ => U+0239C ⎜ => U+0239D ⎝
```

In Latin Modern we have:

```
U+00028: ( ( left parenthesis
   width: 254935, height: 490209, depth: 162529, italic: 0
   mathclass: open, mathname: lparent
   next: U+F0643 ( => U+F0659 ( => U+F066F ( => U+F0685 ( => U+F069B ( => U+F06B1 (
      => U+F06C7 (
   variants: U+0239B ⎛ => U+0239C ⎜ => U+0239D ⎝
```

Of course LuaTeX is downward compatible with respect to this feature, but the internal representation is now closer to what OpenType math provides (which is not that far from how TeX works simply because it's inspired by TeX). Because Cambria has different parameters we get slightly different results. In the following list of pairs, you see Cambria on the left and Latin Modern on the right. Both start with stepwise larger shapes, followed by a more gradual growth. The thresholds for a next step are driven by parameters set in the OpenType font or by TeX's default.





In traditional TEX horizontal extensibles are not really present. Accents are chosen from a linked list of variants and don't have an extensible specification. This is because most such accents grow in two dimensions and the only extensible like accents are rules and braces. However, in UNICODE we have a few more and also because of symmetry we decided to add horizontal extensibles too. Take:

```
$ \overbrace {a+1} \underbrace {b+2} \doublebrace {c+3} $ \par
$ \overparent{a+1} \underparent{b+2} \doubleparent{c+3} $ \par
```

This gives:

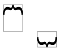
$$\overbrace{a+1}\,\underbrace{b+2}\,\overbrace{\underbrace{c+3}}$$

$$\overparen{a+1}\,\underparen{b+2}\,\overparen{\underparen{c+3}}$$

Contrary to Cambria, Latin Modern Math, which is just like Computer Modern Math, has no ready overbrace glyphs. Keep in mind that in that we're dealing with fonts that have only 256 slots and that the traditional font mechanism has the same limitation. For this reason, the (extensible) braces are traditionally made from snippets as is demonstrated below.

```
\hbox\bgroup
  \ruledhbox{\getglyph{lmex10}{\char"7A}}
  \ruledhbox{\getglyph{lmex10}{\char"7B}}
  \ruledhbox{\getglyph{lmex10}{\char"7C}}
  \ruledhbox{\getglyph{lmex10}{\char"7D}}
  \ruledhbox{\getglyph{lmex10}{\char"7A\char"7D\char"7C\char"7B}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DE}}
  \ruledhbox{\getglyph{lmex10}{\char"7C\char"7B\char"7A\char"7D}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DF}}
\egroup
```

This gives:



The four snippets have the height and depth of the rule that will connect them. Since we want a single interface for all fonts we no longer will use macro based solutions. First of all fonts like Cambria don't have the snippets, and using active character trickery (so that we can adapt the meaning to the font) has no preference either. This leaves virtual glyphs.

It took us a bit of experimenting to get the right virtual definition because it is a multi–step process:

- The right Unicode character (`0x23DE`) points to a character that has no glyph itself but only horizontal extensibles.
- The snippets that make up the extensible don't have the right dimensions (as they define the size of the connecting rule), so we need to make them virtual themselves and give them a size that matches LuaTeX's expectations.
- Each virtual snippet contains a reference to the physical snippet and moves it up or down as well as fixes its size.
- The second and fifth snippet are actually not real glyphs but rules. The dimensions are derived from the snippets and it is shifted up or down too.

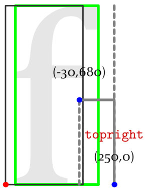
You might wonder if this is worth the trouble. Well, it is if you take into account that all upcoming math fonts will be organized like Cambria.

math kerning

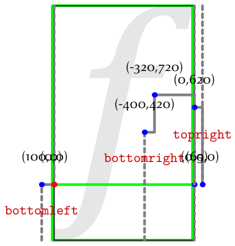
While reading Microsofts orange booklet, it became clear that OpenType provides advanced kerning possibilities and we decided to put it on the agenda for LuaTeX.

It is possible to define a ladder–like boundary for each corner of a character where the ladder more or less follows the shape of a character. In theory this means that when we attach a superscript to a base character we can use two such ladders to determine the optimal spacing between them.

Let's have a look at a few characters, the upright f and its italic cousin.



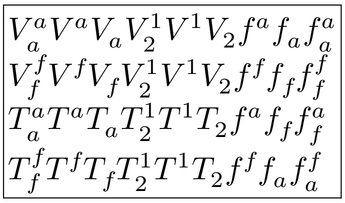
U+00066



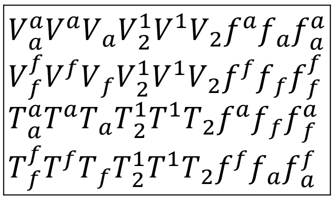
0x1D453

The ladders on the right can be used to position a super or subscript, that is, they are positioned in the normal way but the ladder, as well as the boundingbox and/or left ladders of the scripts can be used to fine tune the positioning.

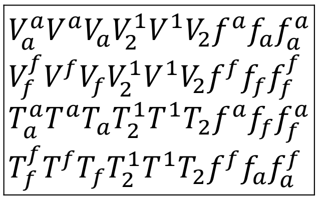
Should we use this information? I made this visualizer for checking some Arabic fonts anchoring and cursive features and then it made sense to add some of the information related to math as well.[8] The orange booklet shows quite advanced ladders, and when looking at the 3500 shapes in Cambria, it quickly becomes clear that in practice there is not that much detail in the specification. Nevertheless, because without this feature the result is not acceptable LuaTeX gracefully supports it.



latin modern



cambria
without kerning



cambria with kerning

faking glyphs

A previous section already discussed virtual shapes. In the process of replacing all shapes that lack in Latin Modern and are composed from snippets instead we ran into the dots. As they are a nice demonstration of something that, although somewhat of a hack, survived 30 years without problems we show the definition used in ConTeXt MkII:

```
\def\PLAINldots{\ldotp\ldotp\ldotp}
\def\PLAINcdots{\cdotp\cdotp\cdotp}

\def\PLAINvdots
  {\vbox{\forgetall\baselineskip.4\bodyfontsize\lineskiplimit\zeropoint\kern

\def\PLAINddots
  {\mkern1mu%
   \raise.7\bodyfontsize\ruledvbox{\kern.7\bodyfontsize\hbox{.}}%
   \mkern2mu%
   \raise.4\bodyfontsize\relax\ruledhbox{.}%
   \mkern2mu%
   \raise.1\bodyfontsize\ruledhbox{.}%
   \mkern1mu}
```

This permitted us to say:

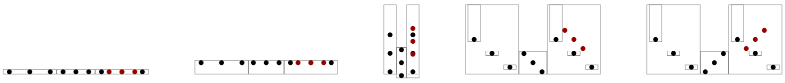
---

[8] Taco extended the visualizer for his presentation at Bachotek 2009 so you might run into variants.

```
\definemathcommand [ldots] [inner]   {\PLAINldots}
\definemathcommand [cdots] [inner]   {\PLAINcdots}
\definemathcommand [vdots] [nothing] {\PLAINvdots}
\definemathcommand [ddots] [inner]   {\PLAINddots}
```

However, in MkIV we use virtual shapes instead.

The following lines show the virtual shapes in red. In each triplet we see the original, the virtual and the overlaid character.



As you can see here, the virtual variants are rather close to the originals. At 12pt there are no real differences but (somehow) at other sizes we get slightly different results but it is hardly visible. Watch the special spacing above the shapes. It is probably needed for getting the spacing right in matrices (where they are used).

XXXII User code

Previous versions of LuaTeX had multiple Lua instances but in practice this was not that useful and therefore we decided to remove that feature and stick to one instance. One reason is that all activities take place in the zero instance anyway and other instance could not access variables defined there. Another reason was that every `\directlua` call is in fact a function call (and as such a closure) and LuaTeX catches errors nicely.

The formal `\directlua` primitive originally can be called in two ways:

```
\directlua <instance> {lua code}
\directlua name {some text} <instance> {lua code}
```

The optional text is then part of the error message when one is issued. The new approach is that the number is used for the error message in case no `name` is specified. The exact string is set in Lua. This means that in principle the command is backward compatible. Old usage will basically ignore the number and use the one and only instance, while new usage will use the number for an eventual message:

```
\directlua <message id> {lua code}
\directlua name {some text} <message id> {lua code}
```

In the second case the id is ignored. The advantage of the first call is that it saves tokens at the TeX end and can be configured at the Lua end. In ConTeXt MkIV we have adapted the code that invokes multiple instances by compatible code that provides a modest form of isolation. We don't want to enforce too many constraints, first of all because users will often use high level interfaces anyway, and also because we assume that users have no bad intentions.

The main Lua instance in ConTeXt is accessible by:[9]

```
\startluacode
global.tex.print("lua")
\stopluacode
```

This gives: 'lua'.

However, sometimes you don't want user code to interfere too much with the main code but still provide access to useful data. This is why we also provide:

```
\startusercode
global.tex.print("user 1")
```

---

[9] Note 2016: you can of course also use `context("lua")` here.

```
global.tex.print("user 2")
if characters then
    global.tex.print("access")
else
    global.tex.print("no access")
end
global.tex.print(global.characters.data[0xA9].contextname)
\stopusercode
```

This gives: 'global.tex.print("user 1")global.tex.print("user 2")if characters then   global.tex.print("access")else   global.tex.print("no access")endglobal.tex.print(global.characters.data[0xA9].contextname)'.

If you're writing a module, you might want to reserve a private namespace. This is done with:

```
\definenamedlua[mymodule][my interesting module]
```

Now we can say:

```
\startmymodulecode
help = { "help" }
global.tex.print(help[1])
\stopmymodulecode
```

This gives: 'help'. The information is remembered:

```
\startmymodulecode
global.tex.print(help[1])
\stopmymodulecode
```

Indeed we get: 'help'.

Just to check the isolation we try:

```
\startusercode
global.tex.print(help and help[1] or "no help")
\stopusercode
```

As expected this gives: 'global.tex.print(help and help[1] or "no help")' but when we do the following we will get an error message:

```
\startusercode
global.tex.print(help[1])
\stopusercode
```

```
! LuaTeX error <private user instance>:2: attempt to index global
'help' (a nil value)
stack traceback:
        <private user instance>:2: in main chunk.
<inserted text> ...userdata")
global.tex.print(help[1])
}
```

An even more isolated variant is:

```
\startisolatedcode
help = { "help" }
global.tex.print(help and help[1] or "no help")
\stopisolatedcode
```

We get: ‘help = { "help" }global.tex.print(help and help[1] or "no help")’, while

```
\startisolatedcode
global.tex.print(help and help[1] or "no help")
\stopisolatedcode
```

gives: ‘global.tex.print(help and help[1] or "no help")’.

You can get access to the global data of other named code blocks by using the `global` prefix. At that level you have also access to the instances, but this time we append `data`, so `user` has a table `userdata`:

For convenience we have made `tex` as well as some Lua tables directly accessible within an instance. However, we recommend not to extend these yourself (even if we do it in the core of MkIV).

XXXIII Just plain

running

For testing basic LuaTeX functionality it makes sense to have a minimal system, and traditionally plain TeX has been the most natural candidate. It is for this reason that it had been on the agenda for a while to provide basic OpenType font support for plain TeX as well. Although the MkIV node mode subsystem is not yet perfect, the time was right to start experimenting with a subset of the MkIV code.

Using plain roughly comes down to the following. First you need to generate a format:

```
luatex --ini --fmt=luatex.fmt luatex-plain.tex
```

This format has to be moved to a place where it can be found by the kpse library. Since this can differ per distribution there is no clear recipe for it, but for TeXLive some path ending in `web2c/luatex` is probably the right spot. After that you can run

```
luatex luatex-test.tex
```

This file lives under `generic/context`. When it is run it is quite likely that you will get an error message because the font name database cannot be found. You can generate one with the following command (which assumes that you have ConTeXt installed):

```
mtxrun --usekpse --script fonts --names
```

The resulting file `luatex-fonts-names.lua` has to be placed somewhere in your TeX tree so that it can be found anytime. Beware: the `--usekpse` flag is only used outside ConTeXt and provides very limited functionality, just enough for this task. Again this is a distribution specific issue so we will not dwell upon it here.

The way fonts are defined is modelled after XeTeX, as it makes no sense to support the somewhat more fancy ConTeXt way of doing things. Keep in mind that although ConTeXt MkIV does support the XeTeX syntax too, the preferred way there is to use a more symbolic feature definition approach.

As this is an experimental setup, it might not always work out as expected. Around LuaTeX version 0.50 we expect the code to be more or less okay.

implementation

The `luatex-fonts.lua` file is the first in a series of basic functionality enhancements for LuaTeX derived from the ConTeXt MkIV code base. Please don't pollute the `luatex-*`

namespace with code not coming from the ConTEXt development team as we may add more files.

This file implements a basic font system for a bare LuaTEX system. By default LuaTEX only knows about the classic TFM fonts but it can read other font formats and pass them to Lua. With some glue code one can then construct a suitable TFM representation that LuaTEX can work with. For more advanced font support a bit more code is needed that needs to be hooked into the callback mechanism.

This file is currently rather simple: it just loads the Lua file with the same name. An example of a `luatex.tex` file that is just the plain TEX format:

```
\catcode`\{=1 % left brace is begin-group character
\catcode`\}=2 % right brace is end-group character

\input plain

\everyjob\expandafter{\the\everyjob\input luatex-fonts\relax}

\dump
```

We could load the Lua file in `\everyjob` but maybe some day we will need more here.

When defining a font, in addition to the XƎTEX way, you can use two prefixes. A `file:` prefix forces a file search, while a `name:` prefix will result in consulting the names database. The font definitions shown in figure 1 are all valid.

```
\font\testa=file:lmroman10-regular at 12pt
\font\testb=file:lmroman12-regular:+liga; at 24pt
\font\testc=file:lmroman12-regular:mode=node;+liga; at 24pt
\font\testd=name:lmroman10bold at 12pt
\font\testh=cmr10
\font\testi=ptmr8t
\font\teste=[lmroman12-regular]:+liga at 30pt
\font\testf=[lmroman12-regular] at 40pt
\font\testj=adobesongstd-light % cid font
\font\testk=cambria(math) {\mathtest 123}
\font\testl=file:IranNastaliq.ttf:mode=node;script=arab;\
    language=dflt;+calt;+ccmp;+init;+isol;+medi;+fina;+liga;\
    +rlig;+kern;+mark;+mkmk at 14pt
```

You can load maths fonts but as Plain TEX is set up for Computer Modern (and as we don't adapt Plain TEX) loading Cambria does not give you support for its math features automatically.

If you want access by name you need to generate a font database, using:

```
mtxrun --script font --names
```

and put the resulting file in a spot where LuaTEX can find it.

remarks

The code loaded in `luatex-fonts.lua` does not come out of thin air, but is mostly shared with ConTEXt; however, in that macro package we go beyond what is provided in the plain variant. When using this code you need to keep a few things in mind:

- This subsystem will be extended, improved etc. at about the same pace as ConTEXt MkIV. However, because ConTEXt provides a rather high level of integration not all features will be supported in the same quality. Use ConTEXt if you want more goodies.

- There is no official API yet, which means that using functions implemented here is at your own risk, in the sense that names and namespaces might change. There will be a minimal API defined once LuaTEX version 1.0 is out. Instead of patching the files it's better to overload functions if needed.

- The modules are not stripped too much, which makes it possible to benefit from improvements in the code that take place in the perspective of ConTEXt development. They might be split a bit more in due time so the baseline might become smaller.

- The code is maintained and tested by the ConTEXt development team. As such it might be better suited for this macro package and integration in other systems might demand some additional wrapping. The plain version discussed here is the benchmark and should be treated as a kind of black box.

- Problems can be reported to the team but as we use ConTEXt MkIV as our baseline, you'd better check if the problem is a general ConTEXt problem too.

- The more high level support for features that is provided in ConTEXt is not part of the code loaded here as it makes no sense elsewhere. Some experimental features are not part of this code either but some might show up later.

- Math font support will be added but only in its basic form once the Latin Modern and TEX Gyre math fonts are available. Currently traditional and OpenType math fonts can be loaded.

- At this moment the more nifty speedups are not enabled because they work in tandem with the alternative file handling that ConTEXt uses. Maybe around LuaTEX 1.0 we will bring some speedup into this code too (if it pays off at all).

- The code defines a few global tables. If this code is used in a larger perspective then you can best make sure that no conflicts occur. The ConTEXt package expects users to work in their own namespace (`userdata`, `thirddata`, `moduledata` or `document`). We give ourselves the freedom to use any table at the global level but will not use tables that are named after macro packages. Later, ConTEXt might operate in a more controlled namespace but it has a low priority.

- There is some tracing code present but this is not enabled and not supported as it integrates quite tightly into ConTEXt. In case of problems you can use ConTEXt for tracking down problems.

- Patching the original code in distributions is dangerous as it might fix your problem but introduce new ones for ConTEXt. So, best keep the original code as it is and overload functions and callbacks when needed. This is trivial in Lua.

- Attributes are (automatically) taken from the range 127–255 so you'd best not use these yourself. Don't count on an attribute number staying the same and don't mess with these attributes.

If this all sounds a bit strict, keep in mind that it makes no sense for us to maintain multiple code bases and we happen to use ConTEXt.

advanced features

The latest versions now also support font extending, slanting, protrusion and expansion. Here are a few examples:

```
\pdfprotrudechars2 \pdfadjustspacing2

\font\testa=file:lmroman12-regular:+liga;extend=1.5          at 12pt
\font\testb=file:lmroman12-regular:+liga;slant=0.8           at 12pt
\font\testc=file:lmroman12-regular:+liga;protrusion=default at 12pt
\font\testd=file:lmroman12-regular:+liga;expansion=default  at 12pt
```

The extend and slant options are similar to those used in map files. The extend is limited to 10 and the slant to 1.

In the protrusion and expansion specification the keyword `default` is an entry in a definition table. You can find an example at the end of `font-dum.lua`.

A setup for expansion looks as follows:

```
fonts.expansions.setups['default'] = {
    stretch = 2, shrink = 2, step = .5, factor = 1,
    [byte('A')] = 0.5, [byte('B')] = 0.7,
    ...........
    [byte('8')] = 0.7, [byte('9')] = 0.7,
}
```

The stretch, shrink and steps become font properties and characters gets a value assigned. In pseudo code it looks like:

```
chr(A).expansion_factor = 0.5 * factor
```

The protrusion table has left and right protrusion factors for each relevant character.

```
fonts.protrusions.setups['default'] = {
    factor = 1, left = 1, right = 1,

    [0x002C] = { 0, 1 }, -- comma
    [0x002E] = { 0, 1 }, -- period
    [0x003A] = { 0, 1 }, -- colon
    ........
    [0x061B] = { 0, 1 }, -- arabic semicolon
    [0x06D4] = { 0, 1 }, -- arabic full stop
}
```

So, the comma will stick out in the right margin:

```
chr(comma).right_protruding = right * 1 * factor
```

As we prefer measures relative to the width (precentages) we actualy use:

```
chr(comma).right_protruding = right * 1 * factor * (width/quad)
```

You can add additional tables and access them by keyword in the font specification.

The model used in the plain variant is a simplification of the ConTEXt model so ConTEXt users should not take this as starting point.

XXXIV Halfway

introduction

We are about halfway into the LuaTeX project now. At the time of writing this document we are only a few days away from version 0.40 (the BachoTeX cq. TeXLive version) and around euroTeX 2009 we will release version 0.50. Starting with version 0.30 (which we released around the conference of the Korean TeX User group meeting) all one-decimal releases are supported and usable for (controlled) production work. We have always stated that all interfaces may change until they are documented to be stable, and we expect to document the first stable parts in version 0.50. Currently we plan to release version 1.00 sometime in 2012, 30 years after TeX82, with 0.60 and 0.70 in 2010, 0.80 and 0.90 in 2011. But of course it might turn out different.

In this update we assume that the reader knows what LuaTeX is and what it does.

design principles

We started this project because we wanted an extensible engine. We chose Lua as the glue language. We do not regret this choice as it permitted us to open up TeX's internals reasonably well. There have been a few extensions to TeX itself, and there will be a few more, but none of them are fundamental in the sense that they influence typesetting. Extending TeX in that area is up to the macro package writer, who can use the Lua language combined with TeX macros. In a similar fashion we made some decisions about Lua libraries that are included. What we have now is what you will get. Future versions of LuaTeX will have the ability to load additional libraries but these will not be part of the core distribution. There is simply too much choice and we do not want to enter endless discussions about what is best. More flexibility would also add a burden on maintenance that we do not want. Portability has always been a virtue of TeX and we want to keep it that way.

lua scripting

Before 0.40 there could be multiple instances of the Lua interpreter active at the same time, but we have now decided to limit the number of instances to just one. The reason is simple: sharing all functionality among multiple Lua interpreter instances does more bad than good and Lua has enough possibilities to create namespaces anyway. The new limit also simplifies the internal source code, which is a good thing. While the `\directlua` command is now sort of frozen, we might extend the functionality of `\latelua`, especially in relation to what is possible in the backend. Both commands still accept a number but this now refers to an index in a user–definable name table that will be shown when an error occurs.

input and output

The current LuaTeX release permits multiple instances of kpse which can be handy if you mix, for instance, a macro package and mplib, as both have their own 'progname' (and engine) namespace. However, right from the start it has been possible to bring most input under Lua control and one can overload the usual kpse mechanisms. This is what we do in ConTeXt (and probably only there).

Logging, etc., is also under Lua control. There is no support for writing to TeX's opened output channels except for the log and the terminal. We are investigating limited write control to numbered channels but this has a very low priority.

Reading from zip files and sockets has been available for a while now.

Among the first things that have been implemented is a mechanism for managing category codes (`\catcode`) although this is not really needed for practical usage as we aim at full compatibility. It just makes printing back to TeX from Lua a bit more comfortable.

interface to tex

Registers can always be accessed from Lua by number and (when defined at the TeX end) also by name. When writing to a register grouping is honored. Most internal registers can be accessed (mostly read-only). Box registers can be manipulated but users need to be aware of potential memory management issues.

There will be provisions to use the primitives related to setting codes (lowercase codes and such). Some of this functionality will be available in version 0.50.

fonts

The internal font model has been extended to the full Unicode range. There are readers for OpenType, Type1, and traditional TeX fonts. Users can create virtual fonts on the fly and have complete control over what goes into TeX. Font specific features can either be mapped onto the traditional ligature and kerning mechanisms or be implemented in Lua.

We use code from FontForge that has been stripped to get a smaller code base. Using the FontForge code has the advantage that we get a similar view on the fonts in LuaTeX as in this editor which makes debugging easier and developing fonts more convenient.

The interface is already rather stable but some of the keys in loaded tables might change. Almost all of the font interface will be stable in version 0.50.

tokens

It is possible to intercept tokenization. Once intercepted, a token table can be manipulated before being piped back into LuaTeX. We still support Omega's translation processes but that might become obsolete at some point.

Future versions of LuaTeX might use Lua's so-called 'user data' concept but the interface will mostly be the same. Therefore this subsystem will not be frozen yet in version 0.50.

nodes

Users have access to the node lists in various stages. This interface has already been quite stable for some time but some cleanup might still take place. Currently the node memory maintenance is still explicit, but eventually we will make releasing unused nodes automatic.

We have plans for keeping more extensive information within a paragraph (initial whatsit) so that one can build alternative paragraph builders in Lua. There will be a vertical packer (in addition to the horizontal packer) and we will open up the page builder (inserts etc.). The basic interface will be stable in version 0.50.

attributes

This new kid on the block is now available for most subsystems but we might change some of its default behaviour. As of 0.40 you can also use negative values for attributes. The original idea of using negative values for special purposes has been abandoned as we consider a secondary (faster and more efficient) limited variant. The basic principles will be stable around version 0.50, but we reserve the freedom to change some aspects of attributes until we reach version 1.00.

hyphenation

In LuaTeX we have clearly separated hyphenation, ligature building and kerning. Managing patterns as well as hyphenation is reimplemented from scratch but uses the same principles as traditional TeX. Patterns can be loaded at run time and exceptions are quite efficient now. There are a few extensions, like embedded discretionaries in exceptions and pre- as well as posthyphens.

On the agenda is fixing some 'hyphenchar' related issues and future releases might deal with compound words as well. There are some known limitations that we hope to have solved in version 0.50.

images

Image handling is part of the backend. This part of the PDFTEX code has been rewritten and can now be controlled from LUA. There are already a few more options than in PDFTEX (simple transformations). The image code will also be integrated in the virtual font handler.

paragraph building

The paragraph builder has been rewritten in C (soon to be converted back to CWEB). There is a callback related to the builder so it is possible to overload the default line breaker by one written in LUA.

There are no further short-term revisions on the agenda, apart from writing an advanced (third order) Arabic routine for the Oriental TEX project.

Future releases may provide a bit more control over `\parshape`s and multiple paragraph shapes.

metapost

The closely related MPLIB project has resulted in a METAPOST library that is included in LUATEX. There can be multiple instances active at the same time and METAPOST processing is very fast. Conversion to PDF is to be done with LUA.

On the to-do list is a bit more interoperability (pre- and postscript tables) and this will make it into release 0.50 (maybe even in version 0.40 already).

mathematics

Version 0.50 will have a stable version of UNICODE math support. Math is backward compatible but provides solutions for dealing with OPENTYPE math fonts. We provide math lists in their intermediate form (noads) so that it is possible to manipulate math in great detail.

The relevant math parameters are reorganized according to what OPENTYPE math provides (we use the Cambria font as our reference). Parameters are grouped by style. Future versions of LUATEX will build upon this base to provide a simple mechanism for switching style sets and font families in-formula.

There are new primitives for placing accents (top and bottom variants and extensible characters), creating radicals, and making delimiters. Math characters are permitted in text mode.

There will be an additional alignment mechanism analogous to what MathML provides. Expect more.

page building

Not much work has been done on opening up the page builder although we do have access to the intermediate lists. This is unlikely to happen before 0.50.

going cweb

After releasing version 0.50 around EuroTEX 2009 there will be a period of relative silence. Apart from bug fixes and (private) experiments there will be no release for a while. At the time of the 0.50 release the LuaTEX source code will probably be in plain C completely. After that is done, we will concentrate hard on consolidating and upgrading the code base back into cweb.

cleanup

Cleanup of code is a continuous process. Cleanup is needed because we deal with a merge of traditional TEX, ε-TEX extensions, pdfTEX functionality and some Omega (Aleph) code.

Compatibility is a prerequisite, with the exception of logging and rather special ligature reconstruction code.

We also use the opportunity to slowly move away from all the global variables that are used in the Pascal version.

alignments

We do have some ideas about opening up alignments, but it has a low priority and it will not happen before the 0.50 release.

error handling

Once all code is converted to cweb, we will look into error handling and recovery. It has no high priority as it is easier to deal with after the conversion to cweb.

backend

The backend code will be rewritten stepwise. The image related code has already been redone, and currently everything related to positioning and directions is redesigned and made more consistent. Some bugs in the ALEPH code (inherited from OMEGA) have been removed and we are trying to come up with a consistent way of dealing with directions. Conceptually this is somewhat messy because much directionality is delegated to the backend.

We are experimenting with positioning (preroll) and better literal injection. Currently we still use the somewhat fuzzy PDFTEX methods that evolved over time (direct, page and normal injection) but we will come up with a clearer model.

Accuracy of the output (PDF) will be improved and character extension (hz) will be done more efficiently. Experimental code seems to work okay. This will become available from release 0.40 and onwards and further cleanup will take place when the CWEB code is there, as much of the PDF backend code is already C.

context mkiv

When we started with LUATEX we decided to use a branch of CONTEXT for testing as it involves quite drastic changes, many rewrites, a tight connection with binary versions, etc.

As a result for some time we now have two versions of CONTEXT: MKII and MKIV, where the former targets PDFTEX and XETEX, and the latter exclusively uses LUATEX. Although the user interface is downward compatible the code base starts to diverge more and more. Therefore at the last CONTEXT meeting it was decided to freeze the current version of MKII and only apply bug fixes and an occasional simple extension.

This policy change opened the road to rather drastic splitting of the code, also because full compatibility between MKII and MKIV is not required. Around LUATEX version 0.40 the new, currently still experimental, document structure related code will be merged into the regular MKIV version. This might have some impact as it opens up new possibilities.

the future

In the future, MKIV will try to create (more) clearly separated layers of functionality so that it will become possible to make subsets of CONTEXT for special purposes. This is done under the name METATEX. Think of layering like:

- IO, catcodes, callback management, helpers
- input regimes, characters, filtering
- nodes, attributes and noads
- user interface
- languages, scripts, fonts and math

- spacing, par building and page construction
- XML, graphics, MetaPost, job management, and structure (huge impact)
- modules, styles, specific features
- tools

fonts

At this moment MkIV is already quite capable of dealing with OpenType fonts. The driving force behind this is the Oriental TeX project which brings along some very complex and feature rich Arabic font technology. Much time has gone into reverse engineering the specification and behaviour of how these fonts behave in Uniscribe (which we use as our reference for Arabic).

Dealing with the huge CJK fonts is less a font issue and more a matter of node list processing. Around the annual meeting of the Korean User Group we got much of the machinery working, thanks to discussions on the spot and on the mailing list.

math

Between LuaTeX versions 0.30 and 0.40 the math machinery was opened up (stage one). In order to test this new functionality, MkIV's math subsystem (that was then already partially Unicode aware) had to be adapted.

First of all Unicode permits us to use only one math family and so MkIV now does that. The implementation uses Microsoft's Cambria Math font as a benchmark. It creates virtual fonts from the other (old and new) math fonts so they appear to match up to Cambria Math. Because the TeX Gyre math project is not yet up to speed MkIV currently uses virtual variants of these fonts that are created at run time. The missing pieces in for instance Latin Modern and friends are compensated for by means of virtual characters.

Because it is now possible to parse the intermediate noad lists MkIV can do some manipulations before the formula is typeset. This is for instance used for alphabet remapping, forcing sizes, and spacing around punctuation.

Although MkIV already supports most of the math that users expect there is still room for improvement once there is even more control over the machinery. This is possible because MkIV is not bound to downward compatibility.

As with all other LuaTeX related MkIV code, it is expected that we will have to rewrite most of the current code a few times as we proceed, so MkIV math support is not yet stable either. We can take such drastic measures because MkIV is still experimental and because users are willing to do frequent synchronous updating of macros and engine. In the process we hope to get away from all ad–hoc boxing and kerning and whatever solutions for creating constructs, by using the new accent, delimiter, and radical primitives.

tracing and testing

Whenever possible we add tracing and visualization features to ConTEXt because the progress reports and articles need them. Recent extensions concerned tracing math and tracing OpenType processing.

The OpenType tracing options are a great help in stepwise reaching the goals of the Oriental TEX project. This project gave the LuaTEX project its initial boost and aims at high quality right-to-left typesetting. In the process complex (test) fonts are made which, combined with the tracing mentioned, help us to reveal the secrets of OpenType.

XXXV Where do we stand

In the previous chapter we discussed the state of LuaTeX in the beginning of 2009, the prelude to version 0.50. We consider the release of the 0.50 version to be a really important, both for LuaTeX and for MkIV so here I will reflect on the state around this release. I will do this from the perspective of processing documents because useability is an important measure.

There are several reasons why LuaTeX 0.50 is an important release, both for LuaTeX and for MkIV. Let's start with LuaTeX.

- Apart from a couple of bug fixes, the current version is pretty usable and stable. Details of what we've reached so far have been presented previously.
- The code base has been converted from Pascal to C, and as a result the source tree has become simpler (being cweb compliant happens around 0.60). This transition also opens up the possibility to start looking into some of the more tricky internals, like page building.
- Most of the front end has been opened up and the new backend code is getting into shape. As the backend was partly already done in C the moment has come to do a real cleanup. Keep in mind that we started with pdfTeX and that much of its extra functionality is rather interwoven with traditional TeX code.

If we look at ConTeXt, we've also reached a crucial point in the upgrade.

- The code base is now divided into MkII and MkIV. This permits us not only to reimplement bits and pieces (something that was already in progress) but also to clean up the code (only MkIV).
- If you kept up with the development you already know the kind of tasks we can (and do) delegate to Lua. Just to mention a few: file handling, font loading and OpenType processing, casing and some spacing issues, everything related to graphics and MetaPost, language support, color and other attributes, input regimes, xml, multi-pass data, etc.
- Recently all backend related code was moved to Lua and the code dealing with hyperlinks, widgets and alike is now mostly moved away from TeX. The related cleanup was possible because we no longer have to deal with a mix of dvi drivers too.
- Everything related to structure (which includes numbering and multi-pass data like tables of contents and registers) is now delegated to Lua. We move around way more information and will extend these mechanisms in the near future.

Tracing on Taco's machine has shown that when processing the LuaTEX reference manual the engine spends about 10% of the time on getting tokens, 15% on macro expansion, and some 50% on Lua (callback interfacing included). Especially the time spent by Lua differs per document and garbage collections seems to be a bottleneck here. So, let's wrap up how LuaTEX performs around the time of 0.50.

We use three documents for testing (intermediate) LuaTEX binaries: the reference manual, the history document 'mk', and the revised metafun manual. The reference manual has a MetaPost graphic on each page which is positioned using the ConTEXt background layering mechanism. This mechanism is active only when backgrounds are defined and has some performance consequences for the page builder. However, most time is spent on constructing the tables (tabulate) and because these can contain paragraphs that can run over multiple pages, constructing a table takes a few analysis passes per table plus some so-called vsplitting. We load some fonts (including narrow variants) but for the rest this document is not that complex. Of course colors are used as well as hyperlinks.

The report at the end of the runs looks as follows:

```
input load time            - 0.109 seconds
stored bytecode data       - 184 modules, 45 tables, 229 chunks
node list callback tasks   - 4 unique tasks, 4 created, 20980 calls
cleaned up reserved nodes  - 29 nodes, 10 lists of 1427
node memory usage          - 19 glue_spec, 2 dir
h-node processing time     - 0.312 seconds including kernel
attribute processing time  - 1.154 seconds
used backend               - pdf (backend for directly generating pdf output)
loaded patterns            - en:us:pat:exc:2
jobdata time               - 0.078 seconds saving, 0.047 seconds loading
callbacks                  - direct: 86692, indirect: 13364, total: 100056
interactive elements       - 178 references, 356 destinations
v-node processing time     - 0.062 seconds
loaded fonts               - 43 files: ....
fonts load time            - 1.030 seconds
metapost processing time   - 0.281 seconds, loading: 0.016 seconds,
                             execution: 0.156 seconds, n: 161
result saved in file       - luatexref-t.pdf
luatex banner              - this is luatex, version beta-0.42.0
control sequences          - 31880 of 147189
current memory usage       - 106 MB (ctx: 108 MB)
runtime                    - 12.433 seconds, 164 processed pages,
                             164 shipped pages, 13.191 pages/second
```

The runtime is influenced by the fact that some startup time and font loading takes place. The more pages your document has, the less the runtime is influenced by this.

More demanding is the 'mk' document (figure ??fig.mk). Here we have many fonts, including some really huge cjk and Arabic ones (and these are loaded at several sizes and with different features). The reported font load time is large but this is partly due to the fact that on my machine for some reason passing the tables to TEX involved a lot of page-faults (we think that the cpu cache is the culprit). Older versions of LUATEX didn't have that performance penalty, so probably half of the reported font loading time is kind of wasted.

The hnode processing time refers mostly to OPENTYPE font processing and attribute processing time has to do with backend issues (like injecting color directives). The more features you enable, the larger these numbers get. The METAPOST font loading refers to the punk font instances.

```
input load time            - 0.125 seconds
stored bytecode data       - 184 modules, 45 tables, 229 chunks
node list callback tasks   - 4 unique tasks, 4 created, 24295 calls
cleaned up reserved nodes - 116 nodes, 29 lists of 1411
node memory usage          - 21 attribute, 23 glue_spec, 7 attribute_list,
                             7 local_par, 2 dir
h-node processing time     - 1.763 seconds including kernel
attribute processing time - 2.231 seconds
used backend               - pdf (backend for directly generating pdf output)
loaded patterns            - en:us:pat:exc:2 en-gb:gb:pat:exc:3 nl:nl:pat:exc:4
language load time         - 0.094 seconds, n=4
jobdata time               - 0.062 seconds saving, 0.031 seconds loading
callbacks                  - direct: 98199, indirect: 20257, total: 118456
xml load time              - 0.000 seconds, lpath calls: 46, cached calls: 31
v-node processing time     - 0.234 seconds
loaded fonts               - 69 files: ....
fonts load time            - 28.205 seconds
metapost processing time   - 0.421 seconds, loading: 0.016 seconds,
                             execution: 0.203 seconds, n: 65
graphics processing time   - 0.125 seconds including tex, n=7
result saved in file       - mk.pdf
metapost font generation   - 0 glyphs, 0.000 seconds runtime
metapost font loading      - 0.187 seconds, 40 instances,
                             213.904 instances/second
luatex banner              - this is luatex, version beta-0.42.0
control sequences          - 34449 of 147189
current memory usage       - 454 MB (ctx: 465 MB)
runtime                    - 50.326 seconds, 316 processed pages,
                             316 shipped pages, 6.279 pages/second
```

Looking at the Metafun manual one might expect that one needs even more time per page but this is not true. We use OPENTYPE fonts in base mode as we don't use fancy font features (base mode uses traditional TEX methods). Most interesting here is the time involved in processing METAPOST graphics. There are a lot of them (1772) and in addition we have 7 calls to independent CONTEXT runs that take one third of the total runtime. About half of the runtime involves graphics.

```
input load time            - 0.109 seconds
stored bytecode data       - 184 modules, 45 tables, 229 chunks
```

```
node list callback tasks  - 4 unique tasks, 4 created, 33510 calls
cleaned up reserved nodes - 39 nodes, 93 lists of 1432
node memory usage         - 249 attribute, 19 glue_spec, 82 attribute_list,
                            85 local_par, 2 dir
h-node processing time    - 0.562 seconds including kernel
attribute processing time - 2.512 seconds
used backend              - pdf (backend for directly generating pdf output)
loaded patterns           - en:us:pat:exc:2
jobdata time              - 0.094 seconds saving, 0.031 seconds loading
callbacks                 - direct: 143950, indirect: 28492, total: 172442
interactive elements      - 214 references, 371 destinations
v-node processing time    - 0.250 seconds
loaded fonts              - 45 files: l.....
fonts load time           - 1.794 seconds
metapost processing time  - 5.585 seconds, loading: 0.047 seconds,
                            execution: 2.371 seconds, n: 1772,
                            external: 15.475 seconds (7 calls)
mps conversion time       - 0.000 seconds, 1 conversions
graphics processing time  - 0.499 seconds including tex, n=74
result saved in file      - metafun.pdf
luatex banner             - this is luatex, version beta-0.42.0
control sequences         - 32587 of 147189
current memory usage      - 113 MB (ctx: 115 MB)
runtime                   - 43.368 seconds, 362 processed pages,
                            362 shipped pages, 8.347 pages/second
```

By now it will be clear that processing a document takes a bit of time. However, keep in mind that these documents are a bit atypical. Although . . . thee average ConTEXt document probably uses color (including color spaces that involve resource management), and has multiple layers, which involves some testing of the about 30 areas that make up the page. And there is the user interface that comes with a price.

It might be good to say a bit more about fonts. In ConTEXt we use symbolic names and often a chain of them, so the abstract `SerifBold` resolves to `MyNiceFontSerif-Bold` which in turn resolves to `mnfs-bold.otf`. As XƎTEX introduced lookup by internal (or system) fontname instead of filename, MkII also provides that method but MkIV adds some heuristics to it. Users can specify font sizes in traditional TEX units but also relative to the body font. All this involves a bit of expansion (resolving the chain) and parsing (of the specification). At each of the levels of name abstraction we can have associated parameters, like features, fallbacks and more. Although these mechanisms are quite optimized this still comes at a performance price.

Also, in the default MkIV font setup we use a couple more font variants (as they are available in Latin Modern). We've kept definitions sort of dynamic so you can change them and combine them in many ways. Definitions are collected in typescripts which are filtered. We support multiple mixed font sets which takes a bit of time to define but switching is generally fast. Compared to MkII the model lacks the (font) encoding and case handling code (here we gain speed) but it now offers fallback fonts (replaced ranges within fonts) and dynamic OpenType font feature switching. When used we might lose a bit of processing speed although fewer definitions are needed which gets us some back. The font subsystem is anyway a factor in the performance, if only because more complex scripts or font features demand extensive node list parsing.

Processing the TEXbook with LuaTEX on Taco's machine takes some 3.5 seconds in pdfTEX and 5.5 seconds in LuaTEX. This is because LuaTEX internally is Unicode and has a larger memory space. The few seconds more runtime are consistent with this. One of the reasons that The TEX Book processes fast is that the font system is not that complex and has hardly any overhead, and an efficient output routine is used. The format file is small and the macro set is optimal for the task. The coding is rather low level so to say (no layers of interfacing). Anyway, 100 pages per second is not bad at all and we don't come close with ConTEXt and the kind of documents that we produce there.

This made me curious as to how fast really dumb documents could be processed. It does not make sense to compare plain TEX and ConTEXt because they do different things. Instead I decided to look at differences in engines and compare runs with different numbers of pages. That way we get an idea of how startup time influences overall performance. We look at pdfTEX, which is basically an 8-bit system, XƎTEX, which uses external libraries and is Unicode, and LuaTEX which is also Unicode, but stays closer to traditional TEX but has to check for callbacks.

In our measurement we use a really simple test document as we only want to see how the baseline performs. As not much content is processed, we focus on loading (startup), the output routine and page building, and some basic pdf generation. After all, it's often a quick and dirty test that gives users their first impression. When looking at the times you need to keep in mind that XƎTEX pipes to dvipdfmx and can benefit from multiple cpu cores. All systems have different memory management and garbage collection might influence performance (as demonstrated in an earlier chapter of the 'mk' document we can trace in detail how the runtime is distributed). As terminal output is a significant slowdown for TEX we run in batchmode. The test is as follows:

```
\starttext
    \dorecurse{2000}{test\page}
\stoptext
```

On my laptop (Dell M90 with 2.3Ghz T76000 Core 2 and 4MB memory running Vista) I get the following results. The test script ran each test set 5 times and we show the fastest run so we kind of avoid interference with other processes that take time. In practice runtime differs quite a bit for similar runs, depending on the system load. The time is in seconds and between parentheses the number of pages per seconds is mentioned.

engine	30	300	2000	10000
xetex	1.81 (16)	2.45 (122)	6.97 (286)	29.20 (342)

pdftex	1.28 (23)	2.07 (144)	6.96 (287)	30.94 (323)
luatex	1.48 (20)	2.36 (127)	7.85 (254)	34.34 (291)

The next table shows the same test but this time on a 2.5Ghz E5420 quad core server with 16GB memory running Linux, but with 6 virtual machines idling in the background. All binaries are 64 bit.

engine	30	300	2000	10000
xetex	0.92 (32)	1.89 (158)	8.74 (228)	42.19 (237)
pdftex	0.49 (61)	1.14 (262)	5.23 (382)	24.66 (405)
luatex	1.07 (27)	1.99 (150)	8.32 (240)	38.22 (261)

A test demonstrated that for LuaTeX the 30 and 300 page runs take 70% more runtime with 32 bit binaries (recent binaries for these engines are available on the ConTeXt wiki `contextgarden.net`).

When you compare both tables it will be clear that it is non-trivial to come to conclusions about performances. But one thing is clear: LuaTeX with ConTeXt MkIV is not performing that badly compared to its cousins. The Unicode engines perform about the same and pdfTeX beats them significantly. Okay, I have to admit that in the meantime some cleanup of code in MkIV has happened and the LuaTeX runs benefit from this, but on the other hand, the other engines are not hindered by callbacks. As I expect to use MkII less frequently optimizing the older code makes no sense.

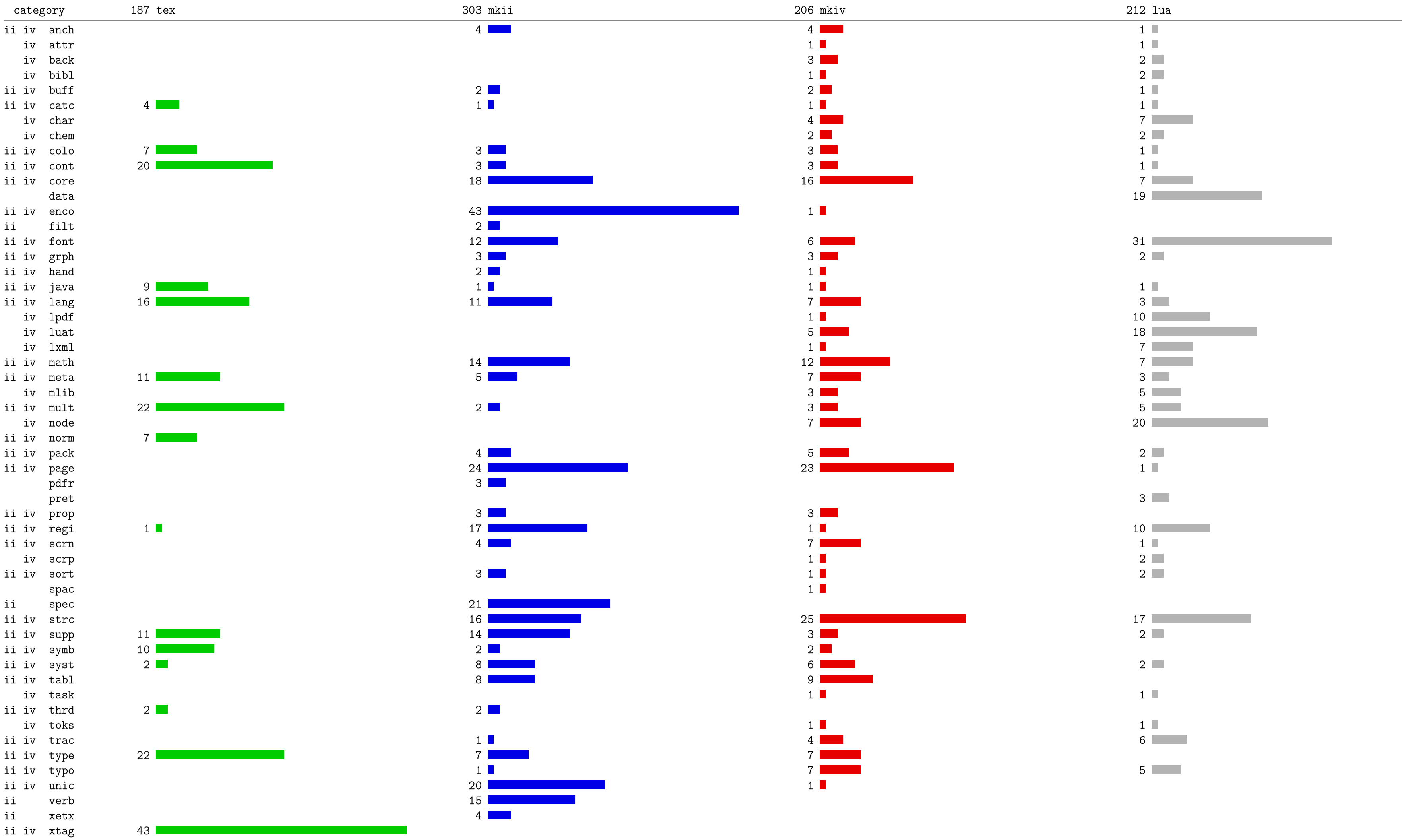
There is not much chance of LuaTeX itself becoming faster, although a few days before writing this Taco managed to speed up font inclusion in the backend code significantly (we're talking about half a second to a second for the three documents used here). On the contrary, when we open up more mechanisms and have upgraded backend code it might actually be a bit slower. On the other hand, I expect to be able to clean up some more ConTeXt code, although we already got rid of some subsystems (like the rather flexible (mixed) font encoding, where each language could have multiple hyphenation patters, etc.). Also, although initial loading of math fonts might take a bit more time (as long as we use virtual Latin Modern math), font switching is more efficient now due to fewer families. But speedups in the ConTeXt code might be compensated for by more advanced mechanisms that call out to Lua. You will be surprised by how much speed can be improved by proper document encoding and proper styles. I can try to gain a couple more pages per second by more efficient code, but a user's style that does an inefficient massive font switch for some 10 words per page easily compensates for that.

When processing this 10 page chapter in an editor (Scite) it takes some 2.7 seconds between hitting the processing key and the result showing up in Acrobat. I can live with that, especially when I keep in mind that my next computer will be faster.

This is where we stand now. The three reports shown before give you an impression of the impact of LuaTeX on ConTeXt. To what extent is this reflected in the code base? We end this chapter with showing four tables. The first table shows the number of files that make up the core of ConTeXt (modules are excluded). The second table shows the accumulated size of these files (comments and spacing stripped). The third and fourth table show the same information in a different way, just to give you a better impression of the relative number of files and sizes. The four character tags represent the file groups, so the files have names like `node-ini.mkiv`, `font-otf.lua` and `supp-box.tex`.

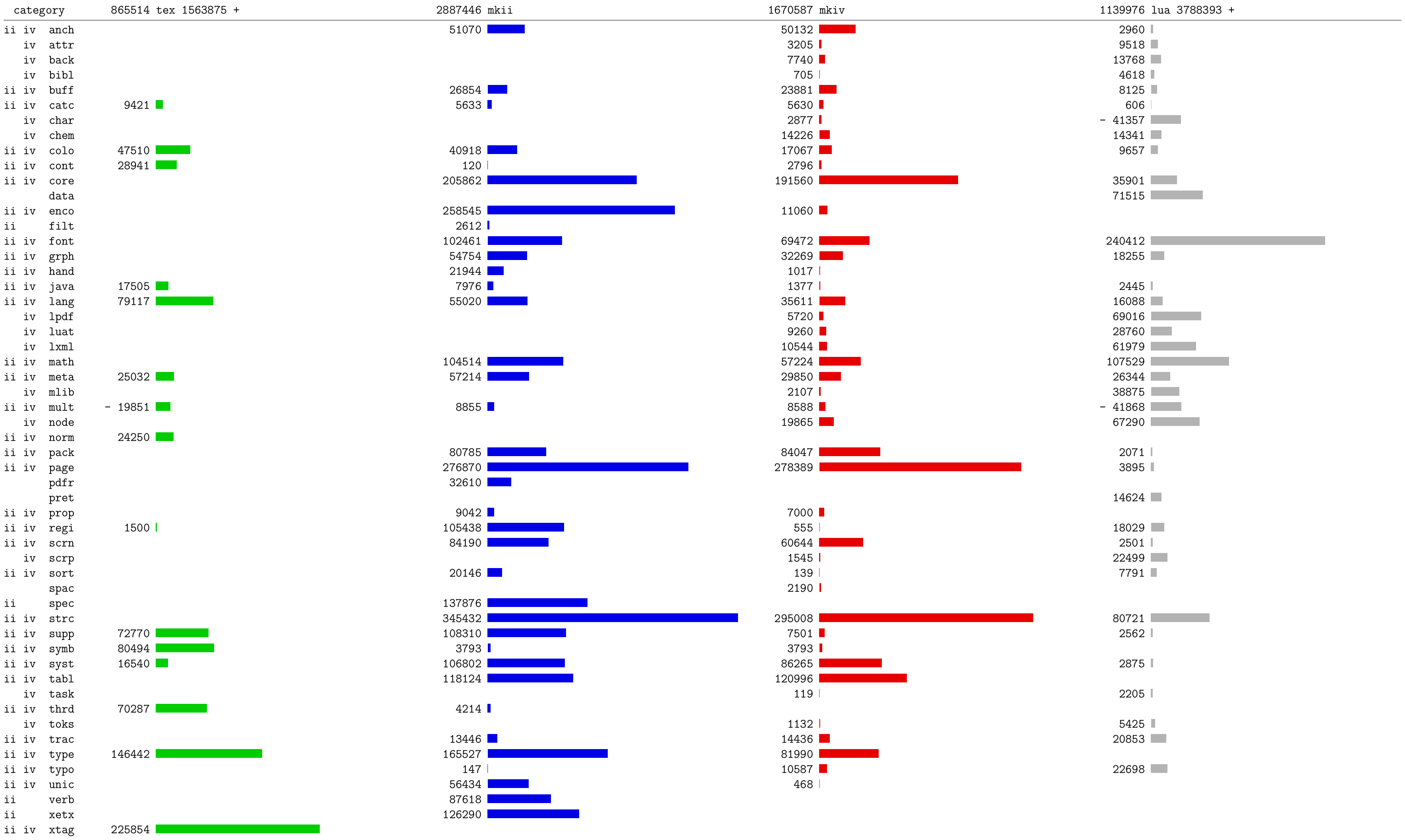
Eventually most MkII files (with the `mkii` suffix) and MkIV files (with suffix `mkiv`) will differ and the number of files with the `tex` suffix will be fewer. Because they are and will be mostly downward compatible, styles and modules will be shared as much as possible.

July 19, 2009 - The number of files used in ConTeXt (modules and styles are excluded).



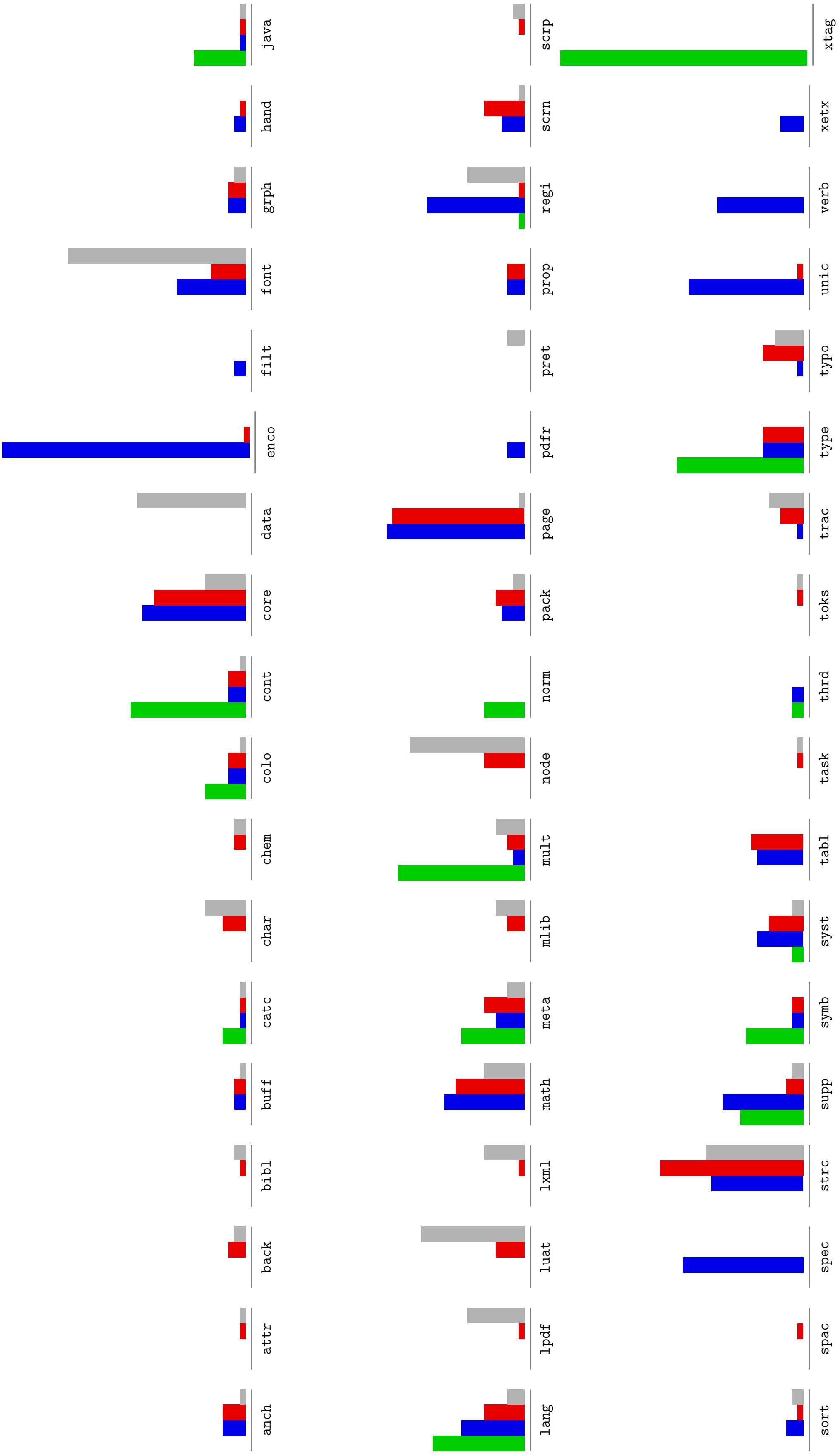
category		187 tex	303 mkii	206 mkiv	212 lua
ii iv	anch		4	4	1
iv	attr			1	1
iv	back			3	2
iv	bibl			1	2
ii iv	buff		2	2	1
ii iv	catc	4	1	1	1
iv	char			4	7
iv	chem			2	2
ii iv	colo	7	3	3	1
ii iv	cont	20	3	3	1
ii iv	core		18	16	7
	data				19
ii iv	enco		43	1	
ii	filt		2		
ii iv	font		12	6	31
ii iv	grph		3	3	2
ii iv	hand		2	1	
ii iv	java	9	1	1	1
ii iv	lang	16	11	7	3
iv	lpdf			1	10
iv	luat			5	18
iv	lxml			1	7
ii iv	math		14	12	7
ii iv	meta	11	5	7	3
iv	mlib			3	5
ii iv	mult	22	2	3	5
iv	node			7	20
ii iv	norm	7			
ii iv	pack		4	5	2
ii iv	page		24	23	1
	pdfr		3		
	pret				3
ii iv	prop		3	3	
ii iv	regi	1	17	1	10
ii iv	scrn		4	7	1
iv	scrp			1	2
ii iv	sort		3	1	2
	spac			1	
ii	spec		21		
ii iv	strc		16	25	17
ii iv	supp	11	14	3	2
ii iv	symb	10	2	2	
ii iv	syst	2	8	6	2
ii iv	tabl		8	9	
iv	task			1	1
ii iv	thrd	2	2		
iv	toks			1	1
ii iv	trac		1	4	6
ii iv	type	22	7	7	
ii iv	typo		1	7	5
ii iv	unic		20	1	
ii	verb		15		
ii	xetx		4		
ii iv	xtag	43			

July 19, 2009 - The size of (core) files used in ConTeXt (- indicates exclusion of large data files; + indicates inclusion of large data files; comment and spaces removed.)

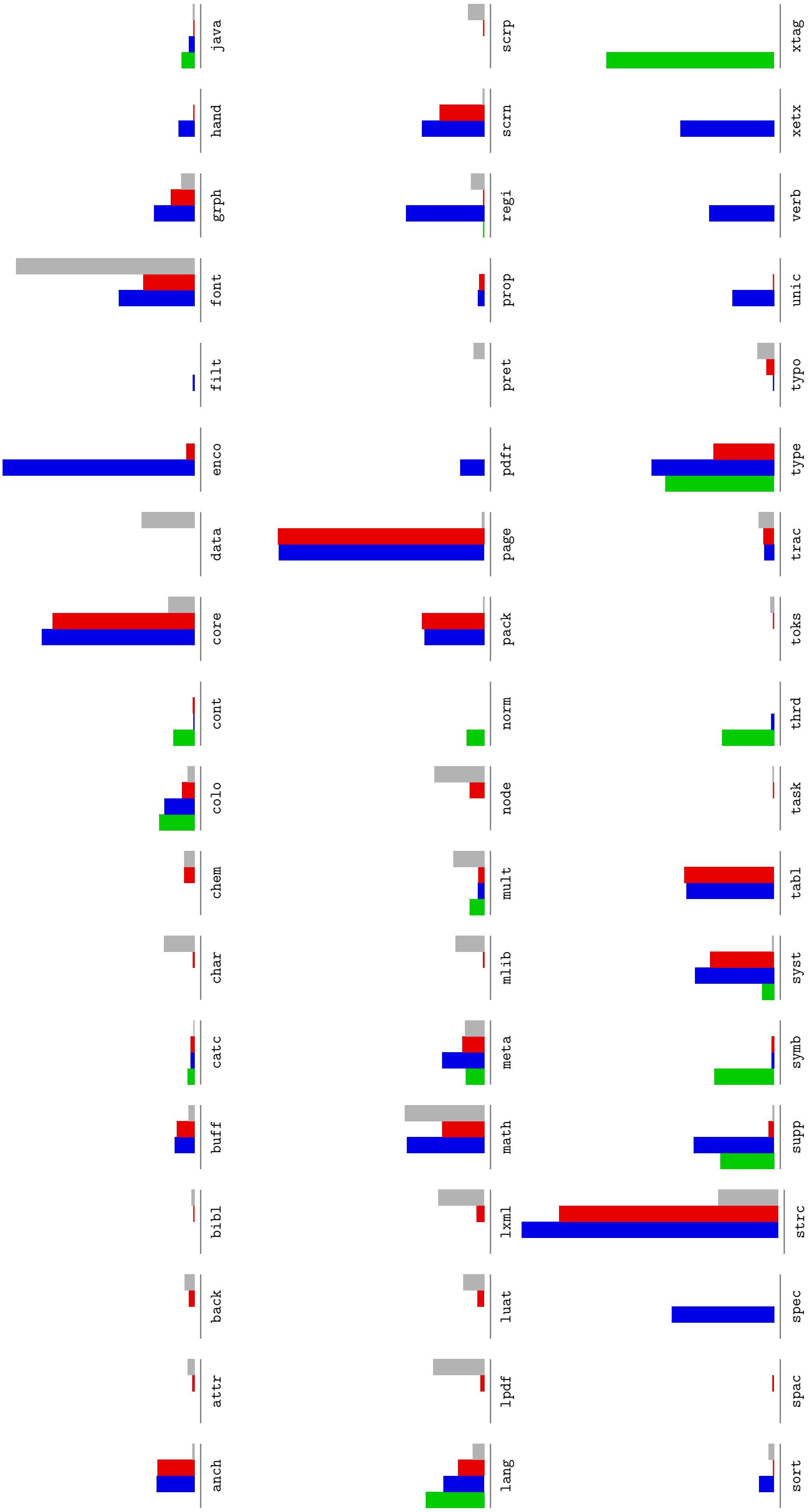


	category	865514 tex 1563875 +	2887446 mkii	1670587 mkiv	1139976 lua 3788393 +
ii iv	anch		51070	50132	2960
iv	attr			3205	9518
iv	back			7740	13768
iv	bibl			705	4618
ii iv	buff		26854	23881	8125
ii iv	catc	9421	5633	5630	606
iv	char			2877	- 41357
iv	chem			14226	14341
ii iv	colo	47510	40918	17067	9657
ii iv	cont	28941	120	2796	
ii iv	core		205862	191560	35901
	data				71515
ii iv	enco		258545	11060	
ii	filt		2612		
ii iv	font		102461	69472	240412
ii iv	grph		54754	32269	18255
ii iv	hand		21944	1017	
ii iv	java	17505	7976	1377	2445
ii iv	lang	79117	55020	35611	16088
iv	lpdf			5720	69016
iv	luat			9260	28760
iv	lxml			10544	61979
ii iv	math		104514	57224	107529
ii iv	meta	25032	57214	29850	26344
iv	mlib			2107	38875
ii iv	mult	- 19851	8855	8588	- 41868
iv	node			19865	67290
ii iv	norm	24250			
ii iv	pack		80785	84047	2071
ii iv	page		276870	278389	3895
	pdfr		32610		
	pret				14624
ii iv	prop		9042	7000	
ii iv	regi	1500	105438	555	18029
ii iv	scrn		84190	60644	2501
iv	scrp			1545	22499
ii iv	sort		20146	139	7791
	spac			2190	
ii	spec		137876		
ii iv	strc		345432	295008	80721
ii iv	supp	72770	108310	7501	2562
ii iv	symb	80494	3793	3793	
ii iv	syst	16540	106802	86265	2875
ii iv	tabl		118124	120996	
iv	task			119	2205
ii iv	thrd	70287	4214		
iv	toks			1132	5425
ii iv	trac		13446	14436	20853
ii iv	type	146442	165527	81990	
ii iv	typo		147	10587	22698
ii iv	unic		56434	468	
ii	verb		87618		
ii	xetx		126290		
ii iv	xtag	225854			

July 19, 2009 - The relative number of files used in ConTeXt (tex, mkii, mkiv, lua).



July 19, 2009 - The relative size of files used in ConTeXt (tex, mkii, mkiv, lua).


anch
attr
back
bibl
buff
catc
char
chem
colo
cont
core
data
enco
filt
font
grph
hand
java
lang
lpdf
luat
lxml
math
meta
mlib
mult
node
norm
pack
page
pdfr
pret
prop
regi
scrn
scrp
sort
spac
spec
strc
supp
symb
syst
tabl
task
thrd
toks
trac
type
typo
unic
verb
xetx
xtag